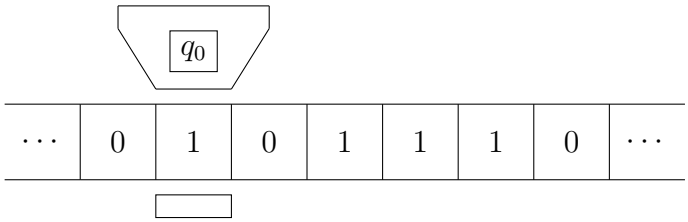
Notes on Computability Theory

Tyler Arant



Compiled on January 4, 2026

Contents

I Defining Computability **5**

1 Preliminaries **7**
1.1 Natural numbers, n-tuples, and relations 7
1.2 Logical connectives 8
1.3 Characteristic functions 9
1.4 Partial functions 10
1.5 Important definition schemes 12
1.6 Additional exercises 16

2 Recursive partial functions and decidability **18**
2.1 Recursive partial functions 18
2.2 Recursive relations 22
2.3 Sequence coding 29
2.4 Additional exercises 35

3 The unlimited register machine **40**
3.1 How the URM works 40
3.2 URM-computable functions 45
3.3 Recursive partial functions are URM-computable 47
3.4 URM-computable partial functions are recursive 50
3.5 Effective enumerations and the s-m-n theorem 56
3.6 Kleene's second recursion theorem 61
3.7 Additional exercises 65

4 Turing machines **67**
4.1 Turing computable partial function 67
4.2 The Church-Turing Thesis 73
4.3 Additional exercises 75

II Exploring the Uncomputable **77**

5 Undecidable relations **79**
5.1 The halting problem 79
5.2 Wang tilings 82
5.3 Index sets and Rice's Theorem 90
5.4 Additional exercises 92

6 Semirecursive relations **94**
6.1 Basic properties 94
6.2 Recursively enumerable sets 100
6.3 An effective enumeration of r.e. sets 103
6.4 Semirecursive-selection and reduction 106
6.5 Additional exercises 109

7 Computable reductions **113**
7.1 m-reductions 113
7.2 Creative and simple sets 118
7.3 1-reductions 121
7.4 Additional exercises 126

8 Relative computability **128**
8.1 α-recursive partial functions 129
8.2 α-URM-computable functions 131
8.3 α-semirecursive relations 134
8.4 Turing reducibility 136
8.5 Computation relative to finite sequences 140
8.6 Additional exercises 145

9 The arithmetical hierarchy **147**
9.1 Closure properties 151
9.2 Universal relations 154
9.3 Classifying relations in the arithmetical hierarchy 156
9.4 Turing jumps and the arithmetical hierarchy 159
9.5 Additional exercises 161

Preface

These notes have been developed as I have taught Math 114C at UCLA several times, and are intended as an undergraduate introduction to computability theory. This presentation of the subject takes seriously the first important job of computability theory: to carefully define the formal, mathematical notion of computable (partial) function and justify that we have correctly captured what we *ought* to mean when we say a (partial) function can be computed. After that task is done, the notes move on to the other (more active) aim of computability theory: to understand and map out the realm of the uncomputable. This is a vast (and beautiful) mathematical world, and these notes will only really scratch the surface.

I would like to acknowledge Nigel J. Cutland and his textbook, *Computability: an introduction to recursive function theory*. His textbook was my starting point when first teaching the subject, and it has certainly influenced many parts of these notes. In particular, Cutland's book convinced me of the advantages of using unlimited register machines as a model of computation.

Many people have helped improve these notes over the years with corrections and insightful comments. Andrew Marks provided much appreciated help in the early versions of these notes. Also, I want to thank the many Math 114C students who have helped discover many typos and suggested countless ways to improve the presentation.

Lastly, I must thank and acknowledge Yiannis Moschovakis, who has had the single largest impact on both my perspective on the subject and my passion for it. Anyone familiar with Yiannis' work will see its influence all over these pages. I hope I have done it justice.

Part I
Defining Computability

Given two positive integers n, m, can you "compute" their greatest common divisor? Given a polynomial $f(x)$ with integer coefficients, can you "compute" the derivative $f'(x)$? Given a positive integer n, can you "decide" whether or not n is prime?

From our previous math classes, we know that the answer to all these questions is "yes". But, what do we really mean when we say we can "compute" a quantity or "decide" a mathematical problem? E.g., what precisely do we mean when we say something like "we have an algorithm to compute the product of two integers"?

The notion of an "algorithm" or "effective procedure" is surprisingly difficult to define in general, but we usually "know one when we see one". Some characteristics that we look for in an algorithm are:

(1) The list of instructions is finite.

(2) The procedure is never vague and it always clearly specifies the next step to take.

(3) intuition or insight ion the part of the user is not required; the user just needs to follow the directions.

(4) The algorithm may require the use of "scratch work" and a place to store the information obtained along the way.

(5) When it gives an answer, the procedure will terminate in finitely many steps of computation.

In our modern context, we have a nice way of summarizing all these desired properties: the algorithm can be implemented by a computer.

How can we determine if we can "compute" a function or "decide" a problem with an effective procedure? A positive answer to this type of question can be demonstrated by describing the algorithm and rigorously proving that it always gives the right answer. A negative answer is more difficult to justify. For a negative answer, we can't rely on something like "we haven't thought of an algorithm yet, so there probably isn't one". Instead, we need *computability theory*; in particular, we need formal mathematical definitions of *computable function* and *decidable problem*.

We will quickly describe an example of an undecidable problem. A *Diophantine equation* is a polynomial equation with finitely many variables and integer coefficients. For example,

$$2x + y = 3, \qquad x^2 + y^2 = z^2, \qquad x - 3y^2 = 1,$$

are all Diophantine equations.

An *integer solution* for a Diophantine equation is an assignment of each variable to an integer value so that the equation holds when these values are substituted into the equation. For example, $x = 3, y = 4$ and $z = 5$ is an integer solution to

$$x^2 + y^2 = z^2.$$

(Integer solutions to this Diophantine equation are called *Pythagorean triples*.)

Integer solutions to Diophantine equations have interested mathematicians for thousands of years. At the turn of the 20th century, David Hilbert devoted one of his famous problems to them:

Hilbert's 10th problem: Devise an effective procedure which, given any Diophantine equation, will determine in a finite number of operations whether or not the Diophantine equation has an integer solution.

As mathematicians began to strongly suspect that Hilbert's 10th problem had a negative answer (that there is no such effective procedure), some began to realize that it would be difficult to justify and prove that type of answer to Hilbert's problem. If they wanted they statement "determining whether a given Diophantine equation is undecidable" to be a mathematical theorem, then that statement would have to be translated into the language of formal mathematics.

This will also be our first set of goals: to give rigorous, mathematical definitions of computable functions and decidable problems, and to justify that our mathematical definitions of these concepts captures the philosophical and intuitive notions of computability. In Part I of these notes, we will map out the territory of the computable and the decidable; in Part II, we will try to understand the world of the uncomputable and the undecidable.

1 Preliminaries

In this chapter, we will familiarize ourselves with the basic types of objects that we will work with. Our study of (classical) computable theory will focus on natural numbers, relations on natural numbers, and functions that operate on natural numbers.

1.1 Natural numbers, n-tuples, and relations

The set of natural numbers is

$$\mathbb{N} =_{\text{df}} \{0, 1, 2, \dots\}.$$

If $n > 0$ is a positive natural number and X is a set, then

$$X^n =_{\text{df}} \{(x_0, \dots, x_{n-1}) : x_0, \dots, x_{n-1} \in X\},$$

is the set of all n-tuples of elements of X.[1] Most important for our purposes are the sets $\mathbb{N}^n$ of n-tuples of natural numbers. We will often use notation like $\vec{x}, \vec{y}, \vec{m}$, etc., to denote elements of $\mathbb{N}^n$.

For any natural numbers $0 < i < n$, we define the *projection function* $\pi_i^n : \mathbb{N}^n \to \mathbb{N}$ by

$$\pi_i^n(x_0, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_{n-1}) =_{\text{df}} x_i \qquad [x_0, \dots, x_{n-1} \in \mathbb{N}].$$

So, we have an infinite family of projection functions π_i^n, where n tells us the length of the input tuples and i tells us which coordinate is being projected. For example,

$$\pi_0^2(8, 6) = 8, \qquad \pi_1^3(4, 7, 5) = 7, \qquad \pi_0^1(3) = 3.$$

Note that $\pi_0^1 : \mathbb{N} \to \mathbb{N}$ is just the identity function on $\mathbb{N}$.

Let $n > 0$ be a positive natural number. An *n-ary relation on* $\mathbb{N}$ is a subset $R \subseteq \mathbb{N}^n$. R is a set, but in the context of thinking of R as a relation we use the notation

$$R(x_0, \dots, x_{n-1}) \iff_{\text{df}} (x_0, \dots, x_{n-1}) \in R \qquad [(x_0, \dots, x_{n-1}) \in \mathbb{N}^n]$$

When $(x_0, \dots, x_{n-1}) \in R$, we also say that $R(x_0, \dots, x_{n-1})$ *holds*. We also

[1] By convention, we will always index things starting at 0, as in $(x_0, \dots, x_{n-1})$.

have notation for when an n-tuple is not in the relation:

$$\neg R(x_0, \ldots, x_{n-1}) \quad \iff_{\text{df}} \quad (x_0, \ldots, x_{n-1}) \notin R \qquad [(x_0, \ldots, x_{n-1}) \in \mathbb{N}^n].$$

When $(x_0, \ldots, x_{n-1}) \notin R$, we can also say that $R(x_0, \ldots, x_{n-1})$ *does not hold*. Note that $\neg R$ is also an n-ary relation on $\mathbb{N}$; as a set, it is just the complement of R, $\mathbb{N}^n \setminus R$.

Example 1.1. Consider the divisibility relation $R \subseteq \mathbb{N}^2$ on the natural numbers. We can define this relation by writing

$$R(x, y) \quad \iff_{\text{df}} \quad x \text{ divides } y \qquad [x, y \in \mathbb{N}].$$

Of course, one could also use set notation to introduce this relation,

$$R =_{\text{df}} \{(x, y) \in \mathbb{N}^2 : \ x \text{ divides } y\}.$$

Clearly, $R(2, 4)$ holds. On the other hand, $R(4, 2)$ does not hold. We could also express this by writing $\neg R(4, 2)$.

In the case that $n = 1$, R is a subset of $\mathbb{N}^1 = \mathbb{N}$ and is called a *unary relation*.[2] We often still use the notation $R(x)$ in place of $x \in R$, but both mean the same thing. Similarly, $\neg R(x)$ and $x \notin R$ both mean the same thing.

Example 1.2. Primality P is a unary relation on $\mathbb{N}$,

$$P(x) \quad \iff_{\text{df}} \quad x \text{ is prime} \qquad [x \in \mathbb{N}].$$

Of course, you can also think of P as simply the set of prime numbers, $P =_{\text{df}} \{x \in \mathbb{N} : x \text{ is prime}\}$.

1.2 Logical connectives

We have seen that if R is an n-ary relation, then $\neg R$ is also an n-ary relation. $\neg$ denotes logical negation, i.e., it means "not". $\neg R$ holds exactly when R does not hold.

We can form new relations using other logical connectives. For example, if R and Q are n-ary relations, then we can think of the relation that holds exactly when R *and* Q both hold. This relation is denoted $R \& Q$ and is called

[2]Note that we will always conflate the set of 1-tuples $\mathbb{N}^1$ with the set of natural numbers $\mathbb{N}$. This is mathematically harmless.

the *conjunction* of R and Q. It is, of course, defined by

$$(R \,\&\, Q)(\vec{x}) \quad \Longleftrightarrow_{\text{df}} \quad R(\vec{x}) \,\&\, Q(\vec{x}).$$

Thinking of R and Q as sets, the conjunction of R and Q is really the same as the *intersection* of R and Q, $R \,\&\, Q = R \cap Q$. These are two ways of thinking about the same concept, one relational and one set theoretic, but we will often focus on the relation way of thinking.

The symbol for the logical connective "or" is $\vee$. We can form a new relation from R and Q by taking the *disjunction* of R and Q,

$$(R \vee Q)(\vec{x}) \quad \Longleftrightarrow_{\text{df}} \quad R(\vec{x}) \vee Q(\vec{x}) \iff R(\vec{x}) \text{ or } Q(\vec{x}).$$

Note that is an "inclusive or", meaning that $(R \vee Q)(\vec{x})$ does indeed hold in the case that both $R(\vec{x})$ and $Q(\vec{x})$ both hold. Thinking of R and Q as sets, the disjunction of R and Q is the same as the *union* of R and Q, $R \vee Q = R \cup Q$.

Exercise 1.3 (DeMorgan's Laws)**.** Let R and Q be n-ary relations on $\mathbb{N}$. Prove that

$$\neg(R \vee Q) = (\neg R) \,\&\, (\neg Q), \qquad \neg(R \,\&\, Q) = (\neg R) \vee (\neg Q).$$

These are equalities of relations, and they are proved by showing that the left-hand-side relation holds at $\vec{x}$ if and only if the right-hand-side relation holds at $\vec{x}$.

1.3 Characteristic functions

Definition 1.4. Let R be an n-ary relation on $\mathbb{N}$. The *characteristic function of* R is the function $\text{Char}_R : \mathbb{N}^n \to \{0, 1\}$ defined by

$$\text{Char}_R(\vec{x}) =_{\text{df}} \begin{cases} 1 & \text{if } R(\vec{x}) \\ 0 & \text{if } \neg R(\vec{x}). \end{cases}$$

Example 1.5. Let $R_{=0}$ be the "equals zero" relation,

$$R_{=0}(x) \quad \Longleftrightarrow_{\text{df}} \quad x = 0.$$

The characteristic function of $R_{=0}$ is

$$\text{Char}_{R_{=0}}(x) = \begin{cases} 1 & \text{if } x = 0; \\ 0 & \text{otherwise.} \end{cases}$$

Often, we can express characteristic functions of more complicated relations in terms of the characteristic functions of their constituent parts.

Example 1.6. If R is an n-ary relation on $\mathbb{N}$, then

$$\mathrm{Char}_{\neg R}(\vec{x}) = 1 - \mathrm{Char}_R(\vec{x}).$$

This is easily verified as follows. The functions on both sides are $\{0,1\}$-valued, so we only need to check that they both evaluate to 1 on the exact same inputs:

$$\mathrm{Char}_{\neg R}(\vec{x}) = 1 \iff \neg R(\vec{x}) \iff \mathrm{Char}_R(\vec{x}) = 0 \iff 1 - \mathrm{Char}_R(\vec{x}) = 1 - 0 = 1.$$

Exercise 1.7. Let R and Q be n-ary relations on $\mathbb{N}$. Show that

$$\mathrm{Char}_{R\&Q}(\vec{x}) = \mathrm{Char}_R(\vec{x}) \cdot \mathrm{Char}_Q(\vec{x})$$

and that

$$\mathrm{Char}_{R\vee Q}(\vec{x}) = \min(1, \mathrm{Char}_R(\vec{x}) + \mathrm{Char}_Q(\vec{x})).$$

1.4 Partial functions

Algorithms can hang on certain inputs, i.e., the algorithm keeps executing instructions without ever reaching a state that tells it to halt and return an output. To accommodate this aspect of algorithms, computability theory makes extensive use of partial functions.

Consider a function $f : X \to Y$. X is the domain of the function, which means that for every $x \in X$, $f(x)$ is defined and equal to a unique element of the codomain Y. A *partial function* on a set X is one that does not need to be defined at all elements of X.

Definition 1.8. Let X and Y be sets. A *partial function* f from X to Y is a function whose domain is a subset $D \subseteq X$ and whose codomain is Y. We use the notation

$$f : X \rightharpoonup Y$$

to mean that f is a partial function from X to Y. Note that for a partial function, the arrow used is "$\rightharpoonup$" rather than the usual "$\to$".

When $x \in X$ is not in the domain of f, then we write $f(x)\uparrow$ and say that f is *not defined at* x or that f *diverges at* x. When $x \in X$ is in the domain of f, we write $f(x)\downarrow$ and say that f is *defined at* x or that f *converges at* x. If m is the value of f at x, then we write $f(x)\downarrow = m$.

When f is defined on all of X, we can emphasize this by calling it a *total function* from X to Y. Note that the total functions $X \to Y$ are a particular type of partial function $X \rightharpoonup Y$; i.e., when we consider an arbitrary partial function $f : X \rightharpoonup Y$, we allow the possibility that f is, in fact, total.

Two partial function $f, g : X \rightharpoonup Y$ are equal just in case they have the same domain and they agree in value on every point in their common domain. Written out carefully, this means

$$f = g \iff_{\text{df}} (\forall x \in X)\Big[[f(x)\!\uparrow \,\&\, g(x)\!\uparrow] \vee (\exists y \in Y)[f(x)\!\downarrow = y \,\&\, g(x)\!\downarrow = y]\Big].$$

We can also talk about a partial function g *extending* a partial function f, written $f \subseteq g$. This means that wherever f is defined, g is also defined and takes the same value as f. The formal definition is

$$f \subseteq g \iff_{\text{df}} (\forall x \in X)(\forall y \in Y)[f(x)\!\downarrow = y \to g(x)\!\downarrow = y].$$

Our interest will be in partial functions $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$, but we give some familiar examples from calculus class.

Example 1.9. Division d can be viewed as a partial function from $\mathbb{R}^2$ to $\mathbb{R}$, $d : \mathbb{R}^2 \rightharpoonup \mathbb{R}$. Where it is defined, $d(x, y) = x/y$. Clearly, $d(x, 0)\!\uparrow$ for all $x \in \mathbb{R}$. We could write the definition of d as

$$d(x, y) = \begin{cases} x/y & \text{if } y \neq 0 \\ \uparrow & \text{if } y = 0. \end{cases}$$

The second case just says that $d(x, y)\!\uparrow$ when $y = 0$.

Example 1.10. Consider the partial function $f : \mathbb{R} \rightharpoonup \mathbb{R}$ which on its domain is given by

$$f(x) = \sum_{n=0}^{\infty} x^n.$$

Recall from calculus that $f(x)\!\downarrow$ if and only if $|x| < 1$. Using the sum of a convergent geometric series, we also have that

$$f(x) = \begin{cases} \frac{1}{1-x} & \text{if } |x| < 1 \\ \uparrow & \text{if } |x| \geq 1. \end{cases}$$

Note that the partial function $1/(1 - x)$ extends the partial function f.

1.5 Important definition schemes

There are many ways to computably define a new function from existing computable functions. We refer to these as "definition schemes" and we will study the most important ones here. As we will see, each of these definition schemes also applies to partial functions.

§ **Definition by substitution**. Let $h : \mathbb{N}^k \to \mathbb{N}$ and let $g_0, \ldots, g_{k-1}$ be functions $\mathbb{N}^n \to \mathbb{N}$. The function $f : \mathbb{N}^n \to \mathbb{N}$ defined by substitution from $h, g_0, \ldots, g_{k-1}$ is the function

$$f(\vec{x}) = h(g_0(\vec{x}), \ldots, g_{k-1}(\vec{x})).$$

Example 1.11. Let $h : \mathbb{N}^4 \to \mathbb{N}$. Then, the function $f : \mathbb{N}^3 \to \mathbb{N}$ given by

$$f(x_0, x_1, x_2) = h(x_2, x_0, 7, x_2)$$

is defined by substitution from h, projection functions, and the constant function $c_7(x_0, x_1, x_2) \equiv 7$. Indeed,

$$\begin{aligned} f(x_0, x_1, x_2) &= h(x_2, x_0, 7, x_2) \\ &= h(\pi^3_2(x_0, x_1, x_2), \pi^3_0(x_0, x_1, x_2), c_7(x_0, x_1, x_2), \pi^3_2(x_0, x_1, x_2)). \end{aligned}$$

Definition by substitution can also be applied to partial functions, but we have to think carefully about when a function defined by substitution from partial functions converges. Let $h : \mathbb{N}^k \rightharpoonup \mathbb{N}$ and $g_0, \ldots, g_{k-1} : \mathbb{N}^n \rightharpoonup \mathbb{N}$ be partial functions. When we introduce the function

$$f(\vec{x}) = h(g_0(\vec{x}), \ldots, g_{k-1}(\vec{x}))$$

we mean that f is the partial function such that

$$\begin{aligned} f(\vec{x})\!\downarrow = y \iff h(g_0(\vec{x}), \ldots, g_{k-1}(\vec{x}))\!\downarrow = y \iff \\ (\exists m_0, \ldots, m_{k-1})\big[g_0(\vec{x})\!\downarrow = m_0 \mathbin{\&} \cdots \mathbin{\&} g_{k-1}(\vec{x})\!\downarrow = m_{k-1} \\ \mathbin{\&} h(m_0, \ldots, m_{k-1})\!\downarrow = y\big]. \end{aligned}$$

Example 1.12. Suppose $h(x_0, x_1) = x_0 + x_1$ and $g_0, g_1 : \mathbb{N} \rightharpoonup \mathbb{N}$ are partial functions. The partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ defined by substitution from h, g_0, g_1 is

$$f(x) = g_0(x) + g_1(x).$$

We have $f(x)\!\downarrow$ if and only if $g_0(x)\!\downarrow$ and $g_1(x)\!\downarrow$. In other words, domain(f) =

$\text{domain}(g_0) \cap \text{domain}(g_1)$.

§ **Definition by primitive recursion**. Let $n > 0$. A function $f : \mathbb{N}^{n+1} \to \mathbb{N}$ is defined via *primitive recursion* from functions $g : \mathbb{N}^n \to \mathbb{N}$ and $h : \mathbb{N}^{n+2} \to \mathbb{N}$ if f satisfies the equations

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

for all $\vec{x} \in \mathbb{N}^n$ and $y \in \mathbb{N}$. These equations are sometimes called the *recursive equations* for f. Values of f are defined via previous values of f. For example, if we fix $\vec{x} \in \mathbb{N}^n$, then the values $f(\vec{x}, 0), f(\vec{x}, 1), f(\vec{x}, 2), \ldots$ can be computed in order as follows:

$$f(\vec{x}, 0) = g(\vec{x}), \quad f(\vec{x}, 1) = h(\vec{x}, f(\vec{x}, 0), 0), \quad f(\vec{x}, 2) = h(\vec{x}, f(\vec{x}, 1), 1), \quad \ldots$$

In this context, $\vec{x}$ is referred to as the *parameter of the recursion*.

Many examples of recursion that you are already familiar with do not have parameters (i.e., $n = 0$.) When $n = 0$, we replace the function $g(\vec{x})$ with a constant m_0 and the recursive equations become

$$f(0) = m_0$$
$$f(y+1) = h(f(y), y)$$

Example 1.13. The factorial function $f(y) = y!$ is defined via primitive recursion from the constant 1 and the function $h(t, y) = t \cdot (y + 1)$:

$$f(0) = 1$$
$$f(y+1) = h(f(y), y).$$

We can check by induction that this gives the right values of the factorial function. Clearly, $f(0) = 1 = 0!$. For the inductive step, assume that $f(y) = y!$. Then we have

$$f(y+1) = h(f(y), y) = f(y) \cdot (y+1) = y! \cdot (y+1) = (y+1)!,$$

as desired.

Note that when we specify h, we have to define the values $h(t, y)$ for any $t \in \mathbb{N}$. We cannot define just the values $h(f(y), y)$ since we haven't defined f yet! We will use h to define f via recursion, so its definition must make sense without reference to f.

Next, we give an example where we do need parameters in the recursion. We let $s : \mathbb{N} \to \mathbb{N}$ denote the successor function, $s(x) = x + 1$.

Example 1.14. Addition $A : \mathbb{N}^2 \to \mathbb{N}$, $A(x, y) = x + y$, can be defined via primitive recursion (with parameters) from the identity function π^1_0 on $\mathbb{N}$ and the function $h(x, t, y) = s(t)$. The recursive equations are

$$\begin{aligned} A(x, 0) &= \pi^1_0(x) = x \\ A(x, y + 1) &= h(x, A(x, y), y) = s(A(x, y)). \end{aligned}$$

Exercise 1.15. Use induction to show that the recursive equations above do, in fact, define addition on the natural numbers.

Exercise 1.16. Show how multiplication on the natural numbers can be defined via primitive recursion with "simpler" functions. *Hint.* Consider the equality $x \cdot (y + 1) = (x \cdot y) + x$.

Example 1.17. Exponentiation on the natural numbers can be defined via primitive recursion from multiplication. Indeed, consider first the identity

$$x^{y+1} = x^y \cdot x.$$

Now define $h(x, t, y) = t \cdot x$. One can easily check by induction that exponentiation is defined by the recursive equations

$$\begin{aligned} x^0 &= 1 \\ x^{y+1} &= h(x, x^y, y) = x^y \cdot x. \end{aligned}$$

Similar to definition by substitution, we can apply definition by primitive recursion to partial functions. If $g : \mathbb{N}^n \rightharpoonup \mathbb{N}$ and $h : \mathbb{N}^{n+2} \rightharpoonup \mathbb{N}$ are partial functions and f is defined via primitive recursion from g and h, then we can inductively define when f converges as follows:

$$\begin{aligned} f(\vec{x}, 0)\!\downarrow = z &\iff g(\vec{x})\!\downarrow = z, \\ f(\vec{x}, y + 1)\!\downarrow = z &\iff (\exists m)\big[f(\vec{x}, y)\!\downarrow = m \,\&\, h(\vec{x}, m, y)\!\downarrow = z\big]. \end{aligned}$$

Note that, in order for $f(\vec{x}, y + 1)$ to converge, all the previous values $f(\vec{x}, i)$, $i \leq y$, must converge.

§ **Definition by minimization**. Given a partial function $g : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$, we can define a new partial function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ via *minimization* as follows.

For $\vec{x} \in \mathbb{N}^n$, we have

$$f(\vec{x})\!\downarrow = y \iff (\forall i < y)\big[g(\vec{x}, i)\!\downarrow \,\&\, g(\vec{x}, i) \neq 0 \,\&\, g(\vec{x}, y)\!\downarrow = 0\big].$$

In notation, we also write this definition as

$$f(\vec{x}) = \mu y\big[g(\vec{x}, y) = 0\big].$$

μ is the Greek letter for m and stands for "minimization". We read $\mu y[\cdots]$ as "the least y such that $[\cdots]$".

Note that, even if g is total, the function f defined via minimization from g may still be a partial function. This is because there may be $\vec{x}$ such that $g(\vec{x}, y) \neq 0$ for all $y \in \mathbb{N}$.

If g is partial, then there are other ways for us to have $f(\vec{x})\!\uparrow$. If $g(\vec{x}, y) = 0$ but there is $i < y$ with $g(\vec{x}, i)\!\uparrow$, then we will have $f(\vec{x})\!\uparrow$. Intuitively the idea is that when evaluating $f(\vec{x})$, we have to start searching for zeros of g one at a time: we check if $g(\vec{x}, 0)\!\downarrow = 0$, then we check if $g(\vec{x}, 1)\!\downarrow = 0$, etc., until we find the least y with $g(\vec{x}, y)\!\downarrow = 0$. However, if during one of these checks, the value $g(\vec{x}, i)\!\uparrow$, then our computation hangs and we spend the rest of time trying to compute something that will never give an output. In this case, $f(\vec{x})$ diverges.

Exercise 1.18. Suppose $g : \mathbb{N}^2 \to \mathbb{N}$ satisfies

$$g(5, 0)\!\downarrow = 1, \quad g(5, 1)\!\uparrow, \quad g(5, 2)\!\downarrow = 0.$$

Let $f(x) = \mu y[g(x, y) = 0]$. Do we have $f(5)\!\downarrow$?

Example 1.19. Let $P \subseteq \mathbb{N}$ be the set of prime numbers. Define $g : \mathbb{N}^2 \to \mathbb{N}$ by

$$g(x, y) = \begin{cases} 0 & \text{if } y \geq x \text{ and } y \in P \\ 1 & \text{otherwise.} \end{cases}$$

Let $f(x)$ be the function defined via minimization from g,

$$f(x) = \mu y[g(x, y) = 0]$$

Then,

$$f(0) = \mu y[g(0, y) = 0] = \text{ (least } y \text{ such that } y \geq 0 \text{ and } y \in P) = 2.$$

Similarly, $f(1) = f(2) = 2$. Then,

$$f(3) = \mu y[g(3, y) = 0] = \text{ (least } y \text{ such that } y \geq 3 \text{ and } y \in P) = 3.$$

One can easily check that $f(4) = f(5) = 5$ and $f(6) = 7$. What is $f(8)$?

1.6 Additional exercises

Exercise 1.20. (a) For subsets $A, B \subseteq \mathbb{N}$, prove that $A = B$ if and only if $\text{Char}_A = \text{Char}_B$.

(b) Prove that for any total function $f : \mathbb{N} \to \{0, 1\}$, there exists a unique $A \subseteq \mathbb{N}$ such that $f = \text{Char}_A$.

Note that, together, these two facts show that the correspondence

$$A \mapsto \text{Char}_A$$

is a bijection between the power set of $\mathbb{N}$, $\mathcal{P}(\mathbb{N}) =_{\text{df}} \{A : A \subseteq \mathbb{N}\}$, and the set $2^{\mathbb{N}} =_{\text{df}} \{f : \ f \text{ is a function from } \mathbb{N} \text{ to } \{0, 1\}\}$.

Exercise 1.21. Let $h : \mathbb{N}^4 \to \mathbb{N}$ be a function. Define $f : \mathbb{N}^3 \to \mathbb{N}$ by

$$f(x_0, x_1, x_2) = h(x_0 + x_2, x_1, x_1, x_0).$$

Find functions $g_0, g_1, g_2, g_3 : \mathbb{N}^3 \to \mathbb{N}$ so that f is defined via substitution from h, g_0, g_1, g_2, g_3.

Exercise 1.22. For each pair of partial functions $f, g : \mathbb{N} \rightharpoonup \mathbb{N}$ below, determine a formula for the composition function $f \circ g : \mathbb{N} \rightharpoonup \mathbb{N}$. Show work that justifies your answers.

(a) $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is any partial function and g is the unique unary partial function with empty domain.

(b) $f(x) = \begin{cases} 1 & \text{if } x \geq 20 \\ \uparrow & \text{otherwise,} \end{cases} \qquad g(x) = \begin{cases} 2x & \text{if } x \text{ is even} \\ \uparrow & \text{otherwise.} \end{cases}$

(c) $f(x) = \begin{cases} x & \text{if } x \text{ is a multiple of 3} \\ \uparrow & \text{otherwise,} \end{cases} \qquad g(x) = \begin{cases} x & \text{if } x \text{ is a multiple of 5} \\ \uparrow & \text{otherwise.} \end{cases}$

Exercise 1.23. Let $f : \mathbb{N} \rightharpoonup \mathbb{N}$ be a partial function. For $x, y \in \mathbb{N}$, we introduce the notation

$$f^0(x) = x \quad \text{and} \quad f^y(x) = (\underbrace{f \circ f \circ \cdots \circ f}_{y \text{ times}})(x) \quad \text{for } y > 0.$$

Show how to define the partial function $F : \mathbb{N}^2 \rightharpoonup \mathbb{N}$,

$$F(x, y) = f^y(x),$$

via primitive recursion. Then, give conditions for which we have $F(x, y)\downarrow$.

Exercise 1.24. Define $g : \mathbb{N}^2 \to \mathbb{N}$ by

$$g(x, y) = \begin{cases} 0 & \text{if } y \geq 2 \text{ and } y \text{ divides } x \\ 1 & \text{otherwise.} \end{cases}$$

Let $f : \mathbb{N} \rightharpoonup \mathbb{N}$ be the partial function defined via minimization from g,

$$f(x) = \mu y[g(x, y) = 0].$$

Compute $f(1)$, $f(4)$, $f(7)$, and $f(15)$. Show work that justifies your answers. What are these values of f if the "1" in the definition of g is replaced with "$\uparrow$"?

Exercise 1.25. Let $g : \mathbb{N}^2 \rightharpoonup \mathbb{N}$ be the partial function defined by

$$g(x_0, x_1) = \begin{cases} 1 & \text{it } x_0 = x_1 \\ \uparrow & \text{otherwise.} \end{cases}$$

Let $c_0^2(x_0, x_1) \equiv 0$ be the zero function.

True or false: for all $x_0, x_1 \in \mathbb{N}$,

$$g(x_0, x_1) = 1 + \big(c_0^2(x_0, x_1) \cdot \mu y[x_0 = x_1 \ \& \ x_0 = y]\big).$$

Justify your answer. *Hint:* think carefully about when minimization converges.

2 Recursive partial functions and decidability

Recall that we are searching for a formal definition of computable partial function. The notion of a *recursive partial function* will be our first candidate definition. Later, we will study other candidate definitions, but we will ultimately see that all these definitions are (non-trivially) equivalent. We will then say that the recursive partial functions are exactly the *computable* partial functions. In the literature, the terms "recursive" and "computable" are interchangeable, but initially we will keep them separate in our minds until we are convinced that the recursive partial functions do precisely capture the notion of computability.

2.1 Recursive partial functions

Recall the successor function $s : \mathbb{N} \to \mathbb{N}$, $s(x) = x + 1$, and the projection functions $\pi_i^n : \mathbb{N}^n \to \mathbb{N}$, $i < n$, $\pi_i^n(\vec{x}) = x_i$. We also give names to the constant functions. For any $n, k \in \mathbb{N}$ with $n > 0$, let $c_k^n : \mathbb{N}^n \to \mathbb{N}$ be the constant function $c_k^n(\vec{x}) = k$ for all $\vec{x} \in \mathbb{N}^n$.

Consider a collection $\mathcal{C}$ of partial functions on the natural numbers. Think of $\mathcal{C}$ as containing functions of all possible arities. We say that $\mathcal{C}$ is *recursively closed* if

(i) The successor function s, all the projection functions π_i^n, $i < n$, and all the constant functions c_k^n are in $\mathcal{C}$.

(ii) $\mathcal{C}$ *is closed under definition by substitution.* If $h : \mathbb{N}^k \rightharpoonup \mathbb{N}$ and $g_0, \ldots, g_{k-1} : \mathbb{N}^n \rightharpoonup \mathbb{N}$ are all in $\mathcal{C}$, and $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is defined via substitution from $h, g_0, \ldots, g_{k-1}$, then f is also in $\mathcal{C}$.

(iii) $\mathcal{C}$ *is closed under definition by primitive recursion.* If $g : \mathbb{N}^n \rightharpoonup \mathbb{N}$ and $h : \mathbb{N}^{n+2} \rightharpoonup \mathbb{N}$ are in $\mathcal{C}$ and $f : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ is defined via primitive recursion from g and h, then f is also in $\mathcal{C}$.

(iv) $\mathcal{C}$ *is closed under definition by minimization.* If $g : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ is in $\mathcal{C}$ and $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is defined via minimization from g, then f is also in $\mathcal{C}$.

Definition 2.1. A partial function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is *recursive* if it belongs to every recursively closed collection of partial functions.

Let $\mathcal{R}$ denote the set of all recursive partial functions. Then, $\mathcal{R}$ is the intersection of all recursively closed collections of partial functions,

$$\mathcal{R} = \bigcap_{\mathcal{C} \text{ is recursively closed}} \mathcal{C}.$$

We also express this by saying the $\mathcal{R}$ is the smallest recursively closed class of partial functions.

Here is a less formal, but more intuitive, way of stating the definition:

Informal definition of recursive partial function.

(a) The successor function s, all the projection functions π_i^n, $i < n$, and all the constant functions c_k^n are recursive.

(b) If $h : \mathbb{N}^k \rightharpoonup \mathbb{N}$ and $g_0, \ldots, g_{k-1} : \mathbb{N}^n \rightharpoonup \mathbb{N}$ are all recursive, and $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is defined via substitution from $h, g_0, \ldots, g_{k-1}$, then f is also recursive.

(c) If $g : \mathbb{N}^n \rightharpoonup \mathbb{N}$ and $h : \mathbb{N}^{n+2} \rightharpoonup \mathbb{N}$ are recursive and $f : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ is defined via primitive recursion from g and h, then f is also recursive.

(d) If $g : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ is recursive and $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is defined via minimization from g, then f is also recursive.

(e) No partial function is recursive except by virtue of (a)-(d).

Our basic examples of recursive functions come from clause (a) above. Note that in particular, the identity function on $\mathbb{N}$, $\pi_0^1 : \mathbb{N} \to \mathbb{N}$, $\pi_0^1(x) = x$, is recursive. The next theorem provides us with many more important examples.

Theorem 2.2. The following functions are recursive:

(i) The predecessor function $\mathrm{Pred} : \mathbb{N} \to \mathbb{N}$ defined by

$$\mathrm{Pred}(0) = 0 \quad \text{and} \quad \mathrm{Pred}(x+1) = x \text{ for } x = 0, 1, \ldots.$$

(ii) Addition $+ : \mathbb{N}^2 \to \mathbb{N}$.

(iii) Multiplication $\cdot : \mathbb{N}^2 \to \mathbb{N}$.

(iv) Exponentiation $f : \mathbb{N}^2 \to \mathbb{N}$, $f(x, y) = x^y$.

Proof. (i) The predecessor function is defined via primitive recursion from 0 and $\pi_1^2(t, y) = y$ since

$$\begin{aligned} \mathrm{Pred}(0) &= 0 \\ \mathrm{Pred}(y+1) &= \pi_1^2(\mathrm{Pred}(y), y) = y \end{aligned}$$

Since π_1^2 is recursive, it follows that Pred is recursive.

(ii) We have already seen that addition can be defined via primitive recursion from π_0^1 and $h(x,t,y) = s(t)$. Note that h is recursive since it is defined via substitution from the recursive functions s and π_1^3:

$$h(x,t,y) = s(\pi_1^3(x,t,y)).$$

(iii) and (iv) are proved similarly to (ii), using the primitive recursive definitions we discussed before. □

Exercise 2.3. Fill in the details of the proofs of (iii) and (iv) above. Carefully justify why the functions in the primitive recursive definitions are recursive.

We have seen that the basic arithmetic operations of addition, multiplication, and exponentiation are recursive. What about subtraction? Since we are only considering functions whose codomain is $\mathbb{N}$, we can't consider the usual subtraction function; for example, $3 - 5 = -2 \notin \mathbb{N}$. However, we can consider *truncated subtraction*. We denote this operation by $\dot{-}$ and it is defined by

$$x \dot{-} y = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{if } x < y \end{cases}$$

Whenever regular subtraction would return a negative integer, truncated subtraction just returns 0. This way, truncated subtraction only takes values in $\mathbb{N}$.

Theorem 2.4. Truncated subtraction $x \dot{-} y$ is recursive.

Proof. This follows by verifying the recursive equations

$$\begin{aligned} x \dot{-} 0 &= x \\ x \dot{-} (y+1) &= \text{Pred}(x \dot{-} y) \end{aligned}$$

and noting that the identity function and Pred are recursive functions. □

Exercise 2.5. Let $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ be a recursive partial function. Carefully explain why the partial function $g : \mathbb{N}^n \rightharpoonup \mathbb{N}$ defined by

$$g(\vec{x}) = 1 \dot{-} f(\vec{x})$$

is recursive.

We will continue showing that many common and important functions are recursive. The following lemma will be useful.

Lemma 2.6. There is a recursive total function $f : \mathbb{N} \to \mathbb{N}$ such that for any even $x \in \mathbb{N}$, $f(x) = x/2$. (Here, we don't particularly care what f does to odd numbers.)

Proof. We define such an f via minimization:

$$f(x) = \mu y[x \dot{-} (2 \cdot y) = 0].$$

Since the function $x \dot{-} (2 \cdot y)$ is recursive, so is f. It is easy to check that f has the desired property. □

Exercise 2.7. For the f defined in the proof for Lemma 2.6, what is $f(3)$? $f(5)$? In general, what is $f(x)$ where $x \in \mathbb{N}$ is odd?

Exercise 2.8. Show that there is a recursive total $g : \mathbb{N} \to \mathbb{N}$ such that $g(x) = x/3$ for any $x \in \mathbb{N}$ which is divisible by 3. Describe what your function g does on numbers which are not divisible by 3.

Theorem 2.9. The following functions are recursive:

(i) The distance function $|x - y|$.

(ii) The max function $\max(x, y)$.

(iii) The min function $\min(x, y)$.

Proof. We sketch the proofs of (i) and (ii). (i) follows by the identity

$$|x - y| = (x \dot{-} y) + (y \dot{-} x) \qquad [x, y \in \mathbb{N}].$$

For (ii), use the equality

$$\max(x, y) = \frac{x + y + |x - y|}{2}$$

along with Lemma 2.6.

The proof of (iii) is left as an exercise. □

Exercise 2.10. Prove that $\min(x, y)$ is recursive. Show all the details of the proof (unlike the sketch for $\max(x, y)$ done above).

We end this section with a nice example of definition via primitive recursion. In this case, we show that we can computably add or multiply the values of a computable partial function.

Theorem 2.11. Let $f : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ be a recursive partial function.

(a) Define $g : \mathbb{N}^n \rightharpoonup \mathbb{N}$ by setting $g(\vec{x}, 0) = 0$ and

$$g(\vec{x}, y+1)\downarrow \iff f(\vec{x}, 0)\downarrow \,\&\, f(\vec{x}, 1)\downarrow \,\&\, \cdots \,\&\, f(\vec{x}, y)\downarrow$$
$$g(\vec{x}, y+1)\downarrow \implies g(\vec{x}, y+1) = \sum_{i=0}^{y} f(\vec{x}, i).$$

Then, g is also recursive.

(b) Define $h : \mathbb{N}^n \rightharpoonup \mathbb{N}$ by setting $h(\vec{x}, 0) = 1$ and

$$h(\vec{x}, y+1)\downarrow \iff f(\vec{x}, 0)\downarrow \,\&\, f(\vec{x}, 1)\downarrow \,\&\, \cdots \,\&\, f(\vec{x}, y)\downarrow$$
$$h(\vec{x}, y+1)\downarrow \implies h(\vec{x}, y+1) = \prod_{i=0}^{y} f(\vec{x}, i) = f(\vec{x}, 0) \cdot f(\vec{x}, 1) \cdot \cdots \cdot f(\vec{x}, y).$$

Then, h is also recursive.

Proof. Define $u(\vec{x}, t, y) = t + f(\vec{x}, y)$. This is recursive since it is defined via a substitution from the recursive function f and $+$. Then it is easy to show by induction that g is defined via the primitive recursion

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

Thus, g is recursive.

The proof of (b) is left as an exercise. □

Exercise 2.12. Prove (b) of Theorem 2.11.

2.2 Recursive relations

Recall that for a relation $R \subseteq \mathbb{N}^n$, its characteristic function $\mathrm{Char}_R : \mathbb{N} \to \mathbb{N}$ is the total function defined by

$$\mathrm{Char}_R(\vec{x}) = \begin{cases} 1 & \text{if } R(\vec{x}) \\ 0 & \text{if } \neg R(\vec{x}). \end{cases}$$

Definition 2.13. An n-ary relation $R \subseteq \mathbb{N}^n$ is *recursive* if its characteristic function Char_R is recursive.

Remark. Just as we are thinking of the recursive partial functions as a candidate for which partial functions we should consider to be *computable*, we think of the recursive relations as our candidate for which relations we should consider the be *decidable*.

Example 2.14. The "equals zero" relation,

$$R_{=0}(x) \quad \iff_{\text{df}} \quad x = 0$$

is a recursive unary relation since

$$\text{Char}_{R_{=0}}(x) = 1 \dot{-} x$$

is recursive.

To build up more examples of (more interesting) recursive relations, we we establish various *closure properties* for the class of recursive relations. For instance, the next theorem tells us that we can form new recursive relations using total recursive substitution into relations we already know are recursive.

Theorem 2.15 (Recursive relations closed under substitution by recursive total functions)**.** If $R(x_0, \ldots, x_{k-1})$ is a k-ary recursive relation and $f_0, \ldots, f_{k-1} : \mathbb{N}^n \to \mathbb{N}$ are recursive <u>total</u> functions, then the relation

$$Q(\vec{y}) \quad \iff_{\text{df}} \quad R(f_0(\vec{y}), \ldots, f_{k-1}(\vec{y})) \qquad [\vec{y} \in \mathbb{N}^n]$$

is a recursive n-ary relation.

Proof. Since R is recursive, $\text{Char}_R : \mathbb{N}^k \to \mathbb{N}$ is recursive. Then,

$$h(\vec{y}) =_{\text{df}} \text{Char}_R(f_0(\vec{y}), \ldots, f_{k-1}(\vec{y}))$$

is also recursive since it is defined via substitution from recursive functions. The last step is to check that $\text{Char}_Q(\vec{y}) = h(\vec{y})$ for all $\vec{y} \in \mathbb{N}^k$. (This last step is where we use that the f_i are total.) □

Remark. Recursive relations are not closed under substitutions by recursive partial functions. We will see this later, when we have methods to prove that relations are not recursive.

Exercise 2.16. Finish the proof of Theorem 2.15 by checking that $\text{Char}_Q = h$.

Exercise 2.17. Let $R(x_0, x_1, x_2)$ be a recursive 3-ary relation. Prove that the relation

$$Q(x_0, x_1, x_2, x_3) \quad \iff_{\text{df}} \quad R(x_0 + x_2, x_3, x_1)$$

is recursive.

We can also form new recursive relations using logical connectives.

Theorem 2.18. Let R and Q be recursive n-ary relations. Then:

(i) (Closure under $\neg$) The relation $\neg R$ is recursive.

(ii) (Closure under &) The relation

$$(R \,\&\, Q)(\vec{x}) \quad \Longleftrightarrow_{\text{df}} \quad R(\vec{x}) \,\&\, Q(\vec{x})$$

is recursive.

(iii) (Closure under $\vee$) The relation

$$(R \vee Q)(\vec{x}) \quad \Longleftrightarrow_{\text{df}} \quad R(\vec{x}) \vee Q(\vec{x})$$

is recursive.

(iv) (Closure under $\to$) The relation

$$(R \to Q)(\vec{x}) \quad \Longleftrightarrow_{\text{df}} \quad R(\vec{x}) \to Q(\vec{x}) \iff (\neg R(\vec{x})) \vee Q(\vec{x}).$$

is recursive.

Proof. The proofs of (i)-(iii) use equalities of characteristic functions we have discussed previously,

$$\text{Char}_{\neg R}(\vec{x}) = 1 \dot{-} \text{Char}_R(\vec{x}), \quad \text{Char}_{R\&Q}(\vec{x}) = \text{Char}_R(\vec{x}) \cdot \text{Char}_Q(\vec{x}),$$

$$\text{Char}_{R\vee Q}(\vec{x}) = \min(1, \text{Char}_R(\vec{x}) + \text{Char}_Q(\vec{x})).$$

Note that we are, of course, also using that $+$, $\cdot$, $\dot{-}$, min are recursive. (iv) immediately follows from (i) and (iii). □

We can now use these basic facts about recursive relations to establish that some important relations are recursive.

Proposition 2.19. The following relations are recursive.

(i) The equality relation, $\quad R_{=}(x, y) \quad \Longleftrightarrow_{\text{df}} \quad x = y.$

(ii) The "not equal" relation, $\quad R_{\neq}(x, y) \quad \Longleftrightarrow_{\text{df}} \quad x \neq y.$

(iii) The non-strict inequality relations, $R_{\leq}(x,y) \iff_{\text{df}} x \leq y$ and $R_{\geq}(x,y) \iff_{\text{df}} x \geq y$.

(iv) The strict inequality relations, $R_{<}(x,y) \iff_{\text{df}} x < y$ and $R_{>}(x,y) \iff_{\text{df}} x > y$.

Proof. (i) follows from $\text{Char}_{R_=}(x,y) = 1 \dot{-} |x-y|$. Note that we could also prove this by noting

$$R_=(x,y) \iff |x-y| = 0 \iff R_{=0}(|x-y|)$$

and using the closure of recursive relations under total recursive substitution.

For (iii), use the equivalence

$$R_{\leq}(x,y) \iff x \dot{-} y = 0,$$

along with closure under total recursive substitution. For $R_{\geq}$, one could again use closure under total recursive substitution:

$$R_{\geq}(x,y) \iff R_{\leq}(y,x) \iff R_{\leq}(\pi^2_1(x,y), \pi^2_0(x,y)).$$

(There are other ways to prove this of course, but it's good to get used to using closure under total recursive substitution.)

For (ii) and (iv), just use closure under negation. For instance, the equivalence

$$R_{<}(x,y) \iff y \not\leq x \iff \neg R_{\leq}(y,x).$$

along with closure under negation shows that $R_<$ is recursive. □

We can use recursive relations in minimizations to define recursive partial functions. If $R(\vec{x}, y)$ is relation, then the partial function

$$f(\vec{x}) = \mu y[R(\vec{x}, y)]$$

is defined by

$$f(\vec{x})\downarrow = z \iff R(\vec{x}, z) \,\&\, (\forall y < z)[\neg R(\vec{x}, z)],$$

i.e., $f(\vec{x})$ is the least y such that $R(\vec{x}, y)$ holds. Of course, $f(\vec{x}) \uparrow$ if there are no y with $R(\vec{x}, y)$.

Theorem 2.20. If $R(\vec{x}, y)$ is a recursive $(n+1)$-ary relation, then

$$f(\vec{x}) = \mu y[R(\vec{x}, y)]$$

is a recursive partial function.

Proof. Just establish that

$$f(\vec{x}) = \mu y[1 \dot{-} \text{Char}_R(\vec{x}, y) = 0].$$

The details are left to the reader. □

The following is a useful characterization of total recursive functions.

Theorem 2.21. Let $f : \mathbb{N}^n \to \mathbb{N}$ be a <u>total</u> function. Then, f is recursive if and only if its graph relation,

$$G_f(\vec{x}, y) \iff_{\text{df}} f(\vec{x}) = y$$

is recursive.

Proof. The ($\Rightarrow$) direction just uses that G_f can be written as a total recursive substitution into the equality relation.

($\Leftarrow$) Assume G_f is recursive. Then,

$$f(\vec{x}) = \mu y[G_f(\vec{x}, y)],$$

which shows f is recursive by Theorem 2.20. □

Remark. Intuitive, the definition $f(\vec{x}) = \mu y[G_f(\vec{x}, y)]$ says that, as long at G_f is recursive, we can compute $f(\vec{x})$ by doing a "table look up". Just start checking one by one which of $G_f(\vec{x}, 0), G_f(\vec{x}, 1), \ldots$ holds.

Remark. If you replace the word "total" in Theorem 2.21 with "partial", the resulting statement is *not* true. In other words, it is not true that a partial function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is recursive if and only if its graph G_f is recursive. We will see later how to properly extend Theorem 2.21 to partial functions.

The class of recursive relations is closed under some other logical operations. Notably, they are closed under *bounded quantification*

Theorem 2.22 (Closure under bounded quantification)**.** Let $R(\vec{x}, y)$ be a recursive $(n+1)$-ary relation. Then:

(i) The relation

$$Q(\vec{x}, y) \iff_{\text{df}} (\exists i \leq y) R(\vec{x}, i)$$

is recursive.

(ii) The relation

$$P(\vec{x}, y) \iff_{\text{df}} (\forall i \leq y) R(\vec{x}, i)$$

is recursive.

Proof. To prove (ii), first note that

$$\text{Char}_P(\vec{x}, y) = \prod_{i=0}^{y} \text{Char}_R(\vec{x}, i)$$

Then, use Theorem 2.11. The proof of (i) is similar, and it is left as an exercise. □

Exercise 2.23. Prove (i) of Theorem 2.22.

There are many variations of bounded quantification which also preserve decidability, like in the following exercises.

Exercise 2.24. Let $R(\vec{x}, y)$ be a recursive $(n+1)$-ary relation. Prove that the relations

$$Q(\vec{x}, y) \iff_{\text{df}} (\exists i < y) R(\vec{x}, i), \qquad P(\vec{x}, y) \iff_{\text{df}} (\forall i < y) R(\vec{x}, i)$$

are both recursive.

Exercise 2.25. Let $R(\vec{x}, i)$ be a recursive $(n+1)$-ary relation. Let $f : \mathbb{N}^m \to \mathbb{N}$ be a recursive total function. Prove that the $(n+m)$-ary relations

$$Q(\vec{x}, \vec{y}) \iff_{\text{df}} (\exists i \leq f(\vec{y})) R(\vec{x}, i), \qquad P(\vec{x}, \vec{y}) \iff_{\text{df}} (\forall i \leq f(\vec{y})) R(\vec{x}, i)$$

are both recursive.

Bounded quantification helps us prove many important relations are recursive.

Theorem 2.26. The following relations are recursive:

(i) The "divides" relation,

$$x|y \iff_{\text{df}} x \text{ divides } y \iff_{\text{df}} y = i \cdot x \text{ for some } i \in \mathbb{N}.$$

(ii) The unary relation of primality,

$$P(x) \iff_{\text{df}} x \text{ is prime.}$$

Proof. (i) If there is $i \in \mathbb{N}$ with $y = i \cdot x$, then it must be that $i \leq y$. Thus,

$$x|y \iff (\exists i \leq y)[i \cdot x = y],$$

which shows the relation is recursive by Theorem 2.22.

The proof of (ii) is left as an exercise. □

Exercise 2.27. Prove (ii) of Theorem 2.26.

We end the section with a useful way to define a recursive function from recursive cases.

Theorem 2.28. Let $R(\vec{x})$ be a recursive relation and let $f, g : \mathbb{N}^n \to \mathbb{N}$ be recursive total functions. Then,

$$h(\vec{x}) = \begin{cases} f(\vec{x}) & \text{if } R(\vec{x}) \\ g(\vec{x}) & \text{if } \neg R(\vec{x}). \end{cases}$$

is recursive.

Proof. Just check the equality

$$h(\vec{x}) = f(\vec{x}) \cdot \text{Char}_R(\vec{x}) + g(\vec{x}) \cdot \text{Char}_{\neg R}(\vec{x}),$$

which defines h from substitutions of recursive (total) functions. □

Theorem 2.28 uses recursive *total* functions f and g. Later, in Theorem 3.21, we will strengthen the statement to work with recursive partial functions f and g. We don't quite have the machinery to prove the stronger statement now, but it is worth thinking about why the above proof does not work when f and g are partial functions.

Exercise 2.29. Let R and Q be recursive n-ary relations and let f_1, f_2, f_3 be n-ary total recursive functions. Show that

$$h(\vec{x}) = \begin{cases} f_1(\vec{x}) & \text{if } R(\vec{x}) \\ f_2(\vec{x}) & \text{if } \neg R(\vec{x}) \text{ and } Q(\vec{x}) \\ f_3(\vec{x}) & \text{if } \neg R(\vec{x}) \text{ and } \neg Q(\vec{x}). \end{cases}$$

is recursive.

2.3 Sequence coding

We know that we can recursively minimize over natural numbers. However, in the sequel there will be many instances in which we would like to search over all sequences of natural numbers, of any finite length. To implement that sort of search, it will be helpful to assign numerical codes to every finite sequence.

Let $\mathbb{N}^*$ be the set of all finite sequences of natural numbers. For example, $(1, 4, 0), (4, 11), (22)$ are all elements of $\mathbb{N}^*$. By convention, $\emptyset$, considered as the empty sequence, is also an element in $\mathbb{N}^*$.

We will assign to each sequence in $\mathbb{N}^*$ a single number, which will be called the *code* of the sequence. We want to do this in a way so that natural operations on sequences are recursive in the codes.

The key to our coding will be the fact that every number has a unique prime factorization. To assign these codes recursively, we want to first establish that we can list all the prime numbers recursively.

Lemma 2.30. Let $p_0 < p_1 < p_2 < \cdots$ list all the prime numbers in increasing order. Then, the function $f : \mathbb{N} \to \mathbb{N}$ defined by $f(n) = p_n$ is recursive.

Proof. Let $P(x)$ be the unary relation of primality. The function f satisfies the recursive equations

$$\begin{aligned} f(0) &= 2 \\ f(n+1) &= \mu y[y > f(n) \,\&\, P(y)]. \end{aligned}$$

The relation

$$R(t, y) \quad \iff_{\text{df}} \quad y > t \,\&\, P(y)$$

is recursive since it is just the conjugation of two recursive relations. Thus, f is defined via a primitive recursion from the constant 2 and the recursive function

$$h(t, n) = \mu y[R(t, y)] = \mu y[y > t \,\&\, P(y)].$$

Thus, f is recursive. □

The above lemma can easily be generalized to enumerate any infinite recursive unary relation:

Exercise 2.31. Let R be a recursive unary relation such that $R(x)$ for infinitely many $x \in \mathbb{N}$. Let $a_0 < a_1 < \cdots$ list all the elements in R in increasing order. Prove that $g(n) = a_n$ is a recursive function.

Definition 2.32. We define a function $\langle \cdot \rangle : \mathbb{N}^* \to \mathbb{N}$ by $\langle \emptyset \rangle = 1$ and

$$\langle x_0, x_1, \ldots, x_{n-1} \rangle = p_0^{1+x_0} p_1^{1+x_1} \cdots p_{n-1}^{1+x_{n-1}}$$

and define the set

$$\text{Seq} = \{u \in \mathbb{N} : u = 1 \text{ or } u = \langle x_0, x_1, \ldots, x_{n-1}\rangle \text{ for some } x_0, x_1, \ldots, x_{n-1} \in \mathbb{N}\}.$$

Elements of Seq are called *sequence codes*. We often consider Seq as a unary relation and write $\text{Seq}(u)$ when u is a sequence code.

One might wonder why we add 1 to the exponents in our coding. If we did not do this, we would have the following problem: $\langle 0\rangle = 2^0 = 1$ and $\langle 0, 0\rangle = 2^0 \cdot 3^0 = 1$. So, different sequences would receive the same code! Adding 1 in the exponents avoids this issue.

Given a number, one can decide whether or not that number is the code of a sequence.

Proposition 2.33. $\text{Seq} \subseteq \mathbb{N}$ is recursive.

Proof. First, convince yourself that u is code exactly when $u = 1$ or else $u > 1$ and whenever a prime p divides u, all the previous primes also divide u. (This is because of the "plus 1" convention of our definition.) Then, the result follows from the equivalence

$$\begin{aligned}\text{Seq}(u) \iff\ & [u = 1] \\ & \vee \left[u > 1 \mathbin{\&} (\forall p \leq u)(\forall q \leq u)(p, q \text{ prime } \mathbin{\&}\ q < p \mathbin{\&}\ p|u) \to q|u\right]\end{aligned}$$

along with the closure properties of recursive relations. □

Exercise 2.34. Establish the equivalence in the proof of Proposition 2.33.

Given a code u for a sequence, the following propositions show that we compute all the basic information about the sequence it codes, namely, the length of the sequence and the values of its terms.

Proposition 2.35. There is a recursive function $\text{lh} : \mathbb{N} \to \mathbb{N}$ such that

$$\text{lh}(u) = n \ \text{ if } \text{Seq}(u) \ \text{ and } \ u = \langle x_0, \ldots, x_{n-1}\rangle \text{ for some } x_0, \ldots, x_{n-1} \in \mathbb{N}$$

and $\text{lh}(u) = 0$ if $u = 1$ or $\neg\text{Seq}(u)$.

Proof. First, verify the equivalence

$$\begin{aligned}\text{lh}(u) = n \iff\ & [(\neg\text{Seq}(u) \vee u = 1) \mathbin{\&}\ n = 0] \\ & \vee [\text{Seq}(u) \mathbin{\&}\ p_{n-1}|u \mathbin{\&}\ p_n \not| u].\end{aligned}$$

Thus, the graph of lh is recursive by the closure properties of recursive relations. By Theorem 2.21, lh is recursive since its graph is recursive. □

Proposition 2.36. There is a recursive function $\text{proj} : \mathbb{N}^2 \to \mathbb{N}$ such that for any $x_0, x_1, \ldots, x_{n-1} \in \mathbb{N}$

$$\text{proj}(\langle x_0, x_1, \ldots, x_{n-1}\rangle, i) = \begin{cases} x_i & \text{if } i < n \\ 0 & \text{otherwise} \end{cases}$$

and $\text{proj}(u, i) = 0$ when $u = 1$ or $u \notin \text{Seq}$.

Proof. Check that the graph of proj satisfies the equivalence

$$\begin{aligned} \text{proj}(u, i) = x \iff & [(\neg\text{Seq}(u) \vee (\text{Seq}(u) \;\&\; \text{lh}(u) \leq i)) \;\&\; x = 0] \\ & \vee [i < \text{lh}(u) \;\&\; (p_i^{1+x} | u) \;\&\; (p_i^{1+(x+1)} \not| u)] \end{aligned}$$

and use closure properties to show that the graph of proj is recursive. Then use Theorem 2.21. □

Typically, we denote the values of the function proj using the notation

$$(u)_i =_{\text{df}} \text{proj}(u, i).$$

We also have notation for repeated use of proj. We define

$$(u)_{i,j} =_{\text{df}} ((u)_i)_j = \text{proj}(\text{proj}(u, i), j),$$

and similarly for $(u)_{i,j,k}$, etc. For instance, if $u = \langle 0, \langle 7, 2, 0\rangle, 1, 1\rangle$, then we have $(u)_{1,0} = 7$.

Proposition 2.37. There is a recursive function $\text{append} : \mathbb{N}^2 \to \mathbb{N}$ such that for any $x_0, x_1, \ldots, x_{n-1}, y \in \mathbb{N}$ we have

$$\text{append}(\langle x_0, x_1, \ldots, x_{n-1}\rangle, y) = \langle x_0, x_1, \ldots, x_{n-1}, y\rangle$$

Moreover, there are recursive functions $\text{res} : \mathbb{N}^2 \to \mathbb{N}$ and $* : \mathbb{N}^2 \to \mathbb{N}$ such that for any $x_0, x_1, \ldots, x_{n-1}$, $y_0, y_1, \ldots, y_{m-1} \in \mathbb{N}$ we have

$$\text{res}(\langle x_0, x_1, \ldots, x_{n-1}\rangle, i) = \langle x_0, x_1, \ldots, x_{i-1}\rangle \qquad \text{for } i < n$$

$$\langle x_0, x_1, \ldots, x_{n-1}\rangle * \langle y_0, y_1, \ldots, y_{m-1}\rangle = \langle x_0, x_1, \ldots, x_{n-1}, y_0, y_1, \ldots, y_{m-1}\rangle$$

and $u * v = 0$ if either $u \notin \text{Seq}$ or $v \notin \text{Seq}$.

Proof. First we define

$$\text{append}(u, y) = u \cdot p_{\text{lh}(u)}^{1+y},$$

which is recursive since it is the composition of recursive functions.

Next, we define function $h : \mathbb{N}^3 \to \mathbb{N}$ by the recursion

$$\begin{aligned} h(u, v, 0) &= u \\ h(u, v, y+1) &= \text{append}(h(u, v, y), (v)_y). \end{aligned}$$

Since h is defined by primitive recursion from recursive functions, h is recursive. Next, check by induction on i that for any $x_0, x_1, \ldots, x_{n-1}, y_0, y_1, \ldots, y_{m-1} \in \mathbb{N}$,

$$h(\langle x_0, x_1, \ldots, x_{n-1}\rangle, \langle y_0, y_1, \ldots, y_{m-1}\rangle, i) = \langle x_0, x_1, \ldots, x_{n-1}, y_0, y_1, \ldots, y_{i-1}\rangle.$$

Finally, define

$$\text{res}(u, i) = h(\langle\emptyset\rangle, u, i), \qquad u * v = h(u, v, \text{lh}(v)).$$

The verification that these definitions give the desired functions is left as an exercise. □

Exercise 2.38. Finish the proof of Proposition 2.37.

For restrictions, we typically use the notation

$$u \restriction i =_{\text{df}} \text{res}(u, i)$$

We will see that sequence coding has many applications. For now, we will end the section by examining just two of them.

§ **Semirecursive graphs**. We have previously said that a total function f is recursive if and only if its graph relation G_f is recursive. We will use sequence codes to strengthen one direction of this equivalence, but first a definition.

Definition 2.39. An n-ary relation $P(\vec{x})$ is *semirecursive* if it satisfies an equivalence of the form

$$P(\vec{x}) \iff (\exists y)\ R(\vec{x}, y)$$

for some recursive relation $R \subseteq \mathbb{N}^{n+1}$.

Remark. Such a P as in the definition is called *semirecursive*, because there is a computable procedure that will correctly tell you when a tuple $\vec{x}$ is in the relation P: namely, given $\vec{x}$, you start searching for a y with $R(\vec{x}, y)$ and return "yes" when you find such a y. However, when $\neg P(\vec{x})$, this search will never terminate, so the procedure will not give an answer.

We will study semirecursive relations in earnest later, but for now we will establish the following using sequence codes.

Theorem 2.40. Let $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ be a partial function. If the graph relation G_f,

$$G_f(\vec{x}, y) \iff f(\vec{x})\!\downarrow = y$$

is semirecursive, then f is recursive.

Proof. Let $R(\vec{x}, y, z)$ be a recursive relation satisfying the equivalence

$$G_f(\vec{x}, y) \iff (\exists z)\ R(\vec{x}, y, z).$$

Then, check that

$$f(\vec{x}) = \big(\mu u[R(\vec{x}, (u)_0, (u)_1)]\big)_0.$$

The details are left as an exercise. □

It is often convenient to show that a partial (or total) function is recursive by simply writing down its graph relation and seeing that it is semirecursive. As mentioned previously, we will later see that the previous theorem can actually be strengthened to an equivalence.

The following fact is proved similarly, using sequence codes:

Exercise 2.41. Let $R(\vec{x}, y_0, y_1 \ldots, y_{k-1})$ be recursive. Prove that the relation

$$P(\vec{x}) \quad \iff_{\text{df}} \quad (\exists y_0)(\exists y_1)\cdots(\exists y_{k-1})\ R(\vec{x}, y_0, y_1 \ldots, y_{k-1})$$

is semirecursive.

§ **Course-of-values recursion**. Consider the Fibonacci sequence $(a_n)_{n=0}^{\infty}$ defined by the recursive equations

$$a_0 = a_1 = 0$$
$$a_{n+1} = a_{n-1} + a_n$$

Intuitively speaking, this sequence is defined recursively, but the recursive equations are different than all of the recursive definitions we have looked at previously. The next value a_{n+1} is defined in terms of the previous two values, a_{n-1} and a_n. This is a more general type of recursion, which we will now discuss.

$f : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ is defined by course-of-values recursion from $g : \mathbb{N}^n \rightharpoonup \mathbb{N}$ and $h^* : \mathbb{N}^* \rightharpoonup \mathbb{N}$ if it satisfies the recursive equations

$$\begin{aligned} f(\vec{x}, 0) &= g(\vec{x}) \\ f(\vec{x}, y+1) &= h^*(\vec{x}, f(\vec{x}, 0), \ldots, f(\vec{x}, y), y) \end{aligned}$$

These recursive equations are different than usual since h^* is defined on $\mathbb{N}^*$ rather than $\mathbb{N}^{n+2}$. However, we can use sequence codes to replace h^* by a function $h : \mathbb{N}^{n+2} \rightharpoonup \mathbb{N}$,

$$f(\vec{x}, y+1) = h(\vec{x}, \langle f(\vec{x}, 0), \ldots, f(\vec{x}, y)\rangle, y)$$

This is already much better, since now we can talk about whether h is recursive.

Theorem 2.42. Suppose $f : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ is defined by course-of-values recursion from $g : \mathbb{N}^n \rightharpoonup \mathbb{N}$ and $h : \mathbb{N}^{n+2} \to \mathbb{N}$,

$$\begin{aligned} f(\vec{x}, 0) &= g(\vec{x}) \\ f(\vec{x}, y+1) &= h(\vec{x}, \langle f(\vec{x}, 0), \ldots, f(\vec{x}, y)\rangle, y) \end{aligned}$$

If g and h are recursive, then f is also recursive.

Proof. First, we show that $F : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ defined by

$$F(\vec{x}, y) = \langle f(\vec{x}, 0), \ldots, f(\vec{x}, y)\rangle$$

is recursive. Then it will follow that $f(\vec{x}, y) = (F(\vec{x}, y))_y$ is clearly recursive.

We can see that F is recursive since it can be defined via the primitive recursion

$$\begin{aligned} F(\vec{x}, 0) &= \langle f(\vec{x}, 0)\rangle \\ F(\vec{x}, y+1) &= F(\vec{x}, y) * \langle h(\vec{x}, F(\vec{x}, y), y)\rangle. \end{aligned}$$

We can verify that, with this definition,

$$F(\vec{x}, y) = \langle f(\vec{x}, 0), \ldots, f(\vec{x}, y)\rangle \qquad (*)$$

by a simple induction on y. The base case, $n = 0$, is clearly true. For the

inductive step, assume $(*)$ holds. Then,

$$\begin{aligned} F(\vec{x}, y+1) &= F(\vec{x}, y) * \langle h(\vec{x}, F(\vec{x}, y), y)\rangle \\ &= \langle f(\vec{x}, 0), \ldots, f(\vec{x}, y)\rangle * \langle h(\vec{x}, \langle f(\vec{x}, 0), \ldots, f(\vec{x}, y)\rangle, y)\rangle \\ &= \langle f(\vec{x}, 0), \ldots, f(\vec{x}, y)\rangle * \langle f(\vec{x}, y+1)\rangle \\ &= \langle f(\vec{x}, 0), \ldots, f(\vec{x}, y), f(\vec{x}, y+1)\rangle, \end{aligned}$$

as desired. □

Exercise 2.43. Suppose $f : \mathbb{N} \to \mathbb{N}$ is defined by the recursive equations

$$\begin{aligned} f(0) &= x_0 \\ f(1) &= x_1 \\ f(y+2) &= h(f(y), f(y+1), y). \end{aligned}$$

Prove that if h is recursive, then f is recursive. Conclude that the Fibonacci sequence is recursive. *Hint.* Show that the function $F : \mathbb{N} \to \mathbb{N}$ defined by

$$\begin{aligned} F(0) &= \langle f(0), f(1)\rangle \\ F(y+1) &= \langle f(0), f(1), \ldots, f(y+1), f(y+2)\rangle \end{aligned}$$

is recursive.

2.4 Additional exercises

Exercise 2.44. Let $g : \mathbb{N}^{n+1} \to \mathbb{N}$ be a recursive total function. Define $f : \mathbb{N}^{n+1} \to \mathbb{N}$ by

$$f(\vec{x}, y) = \max\{g(\vec{x}, 0), g(\vec{x}, 1), \ldots, g(\vec{x}, y)\}. \qquad (\vec{x} \in \mathbb{N}^n,\ y \in \mathbb{N})$$

Prove that f is recursive.

Exercise 2.45. We define the *exponential tower function* $f : \mathbb{N}^2 \to \mathbb{N}$ by

$$f(x,0) = x, \quad f(x,1) = x^x, \quad f(x,2) = x^{x^x}, \quad \ldots \quad f(x,n) = \underbrace{x^{x^{\cdot^{\cdot^{\cdot^x}}}}}_{n+1 \text{ copies of } x}, \quad \ldots.$$

Prove that f is recursive.

Exercise 2.46. Suppose $f : \mathbb{N} \to \mathbb{N}$ is total function which is injective, i.e., for any $x, y \in \mathbb{N}$, $f(x) = f(y) \implies x = y$. Let $g : \mathbb{N} \rightharpoonup \mathbb{N}$ be the partial

inverse of f, i.e., the partial function defined by

$$g(y)\downarrow \iff (\exists x)\ f(x) = y,$$
$$g(y)\downarrow \implies g(y) = \text{the unique } x \text{ s.t. } f(x) = y.$$

Prove that if f is recursive, then g is recursive.

Exercise 2.47. Prove that the following relations are recursive.

(a) The unary relation of evenness,

$$E(x) \quad \iff_{\text{df}} \quad x \text{ is even.}$$

(b) The unary relation of primality,

$$P(x) \quad \iff_{\text{df}} \quad x \text{ is prime.}$$

(c) The unary relation

$$R(x) \quad \iff_{\text{df}} \quad x \text{ is a power of a prime.}$$

Exercise 2.48. Let $R(\vec{x}, y)$ be a recursive $(n+1)$-ary relation. Define a total function $f : \mathbb{N}^{n+1} \to \mathbb{N}$ by

$$f(\vec{x}, y) =_{\text{df}} \text{ the number of elements in the set } \{z \in \mathbb{N} : z \leq y \ \&\ R(\vec{x}, z)\}.$$

Prove that f is recursive.

Exercise 2.49. Let $f : \mathbb{N} \rightharpoonup \mathbb{N}$ be a recursive partial function, and let range(f) be the range of f.

(a) Prove that range(f) is semirecursive.

(b) Prove that if f is total, then range(f) is recursive.

Exercise 2.50. Recall that for $x, y > 0$, there exists unique natural numbers q and r such that $r < x$ and $y = q \cdot x + r$. q is called the *quotient* and r the *remainder* when we divide y by x. We introduce the notation

$$\text{qt}(x, y) = q, \qquad \text{rm}(x, y) = r \qquad [x, y > 0].$$

If x or y is 0, we set $\text{qt}(x, y) = 0$ and $\text{rm}(x, y) = 0$. Prove that The quotient function qt and the remainder function rm are recursive.

Exercise 2.51. We say that a sequence code $u = \langle x_0, x_1, \ldots, x_{n-1} \rangle$ is *increasing* if

$$x_0 < x_1 < \cdots < x_{n-1}.$$

Prove that the unary relation

$$R(u) \quad \iff_{\text{df}} \quad u \text{ is an increasing sequence code}$$

is recursive.

Exercise 2.52. Let $R \subseteq \mathbb{N}$ be the unary relation defined by

$$R(u) \quad \iff_{\text{df}} \quad u \text{ is the code of sequence, all of whose entries are odd.}$$

Show that R is recursive.

Exercise 2.53. Let R be an n-ary relation. Define the unary relation R' by

$$R'(u) \quad \iff_{\text{df}} \quad u = \langle \vec{x} \rangle \text{ for some } \vec{x} \in \mathbb{N} \text{ with } R(\vec{x})$$

Prove that R is recursive if and only if R' is recursive.

Exercise 2.54. (a) Let $g : \mathbb{N}^2 \to \mathbb{N}$ and $h : \mathbb{N} \to \mathbb{N}$ be recursive total functions. Show that the function $f : \mathbb{N} \to \mathbb{N}$ defined by

$$f(x) = \sum_{i=0}^{h(x)} g(x, i)$$

is recursive.

(b) Prove that there is a recursive total function $f : \mathbb{N} \to \mathbb{N}$ such that for any positive length sequence code $\langle x_0, x_1, \ldots, x_{n-1} \rangle$,

$$f(\langle x_0, x_1, \ldots, x_{n-1} \rangle) = x_0 + x_1 + \cdots + x_{n-1}.$$

(It doesn't matter to what values f converges to on the code of the empty sequence and on non-sequence codes, but it should always converge.)

Exercise 2.55. Partial functions $f_0, f_1 : \mathbb{N}^{k+1} \rightharpoonup \mathbb{N}$ are define by *simultaneous recursion* from $g_0, g_1 : \mathbb{N}^k \rightharpoonup \mathbb{N}$ and $h_0, h_1 : \mathbb{N}^{k+3} \rightharpoonup \mathbb{N}$ if they satisfy the equations

$$\begin{cases} f_0(\vec{x}, 0) = g_0(\vec{x}) \\ f_0(\vec{x}, y+1) = h_0(\vec{x}, y, f_0(\vec{x}, y), f_1(\vec{x}, y)), \end{cases}$$

$$\begin{cases} f_1(\vec{x}, 0) = g_1(\vec{x}) \\ f_1(\vec{x}, y+1) = h_1(\vec{x}, y, f_0(\vec{x}, y), f_1(\vec{x}, y)), \end{cases}$$

Prove that f_0 and f_1 are defined by simultaneous recursion from recursive partial functions g_0, g_1 and h_0, h_1, then f_0 and f_1 are both recursive.

Hint. Consider the function $F(\vec{x}, y) = \langle f_0(\vec{x}, y), f_1(\vec{x}, y) \rangle$

As we have seen with course-of-values and simultaneous recursions, many different types of recursions can be rewritten as standard primitive recursions. This, however, is not true for every type of recursion. Famously, the Ackermann function is a recursive function, but you cannot define it only using substitutions and primitive recursions. (A minimization is absolutely needed somewhere.) For this reason, the Ackermann function is said to be recursive but not primitive recursive. We will not prove this full statement, but the following exercises will establish that the Ackermann function is a recursive total function.

The Ackermann function $A(m, n)$ is defined by the following recursive equations:

$$\begin{aligned} A(0, n) &= n + 1 \\ A(m+1, 0) &= A(m, 1) \\ A(m+1, n+1) &= A(m, A(m+1, n)). \end{aligned}$$

Exercise 2.56. (a) Compute $A(1, 1)$.

(b) Prove that $A(m, n) \downarrow$ for every $m, n \in \mathbb{N}$.

Hint: This will require a *double induction* on both m and n. The "outer" in induction is on m and is used to prove the following statement for every m: $(\forall n)\ A(m, n) \downarrow$. In the inductive step of this induction, you will assume $A(m, n) \downarrow$ for every n. But, then you will prove $A(m+1, n) \downarrow$ for every n by *another induction*, this time on n.

An *Ackermann computation* is a sequence of elements of $\mathbb{N}^3$,

$$(m_0, n_0, y_0), \ldots, (m_{k-1}, n_{k-1}, y_{k-1})$$

such that for every $i < k$, one of the following holds:

(1) $m_i = 0$ and $y_i = n_i + 1$

(2) there exists m such that $m_i = m + 1$, $n_i = 0$, and there is $j < i$ such that $m_j = m$, $n_j = 1$ and $y_j = y_i$.

(3) there exists m, n such that $m_i = m + 1$, $n_i = n + 1$, there exists $j < i$ such that $m_j = m + 1$, $n_j = n$, and there exists $\ell < i$ such that $m_\ell = m$, $n_\ell = y_j$, and $y_\ell = y_i$. Written out as tuples, this means

$$\begin{aligned}(m_i, n_i, y_i) &= (m + 1, n + 1, y_i)\\ (m_j, n_j, y_j) &= (m + 1, n, y_j)\\ (m_\ell, n_\ell, y_\ell) &= (m, y_j, y_i).\end{aligned}$$

The idea is that if a tuple (m, n, y) shows up in an Ackermann computation, that is a "proof" or a "certificate" which establishes that $A(m, n) = y$. Each of (1)-(3) corresponds to one of the recursive equations in the definition of the Ackermann function (in the same order).

In the next problem, you may use the following fact without justification: if $\vec{x}_0, \ldots, \vec{x}_{k-1}$ and $\vec{z}_0, \ldots, \vec{z}_{\ell-1}$ are both Ackermann computations, then their concatenation

$$\vec{x}_0, \ldots, \vec{x}_{k-1}, \vec{z}_0, \ldots, \vec{z}_{\ell-1}$$

as also an Ackermann computation. (This is not difficult to prove, so you should think a bit about why it is true.)

Exercise 2.57. (a) Prove that $A(m, n) = y$ if and only if there is an Ackermann computation

$$(m_0, n_0, y_0), \ldots, (m_{k-1}, n_{k-1}, y_{k-1})$$

such that $(m_{k-1}, n_{k-1}, y_{k-1}) = (m, n, y)$.

Hint: This requires two inductive proofs, one for each direction. For proving the left-to-right direction, the induction should be a double induction on m and n, similar to the previous problem. For proving the right-to-left direction, use an induction on the length k of the Ackermann computation.

(b) Prove that the Ackermann function is recursive.

Hint: Use part (a) and Theorem 2.40. Also, note that you can code the 3-tuples and sequences of 3-tuples with sequence codes.

3 The unlimited register machine

In the introduction, we said that we wanted to give a formal, rigorous definition of computable partial functions. We can think of the recursive partial functions as one potential candidate definition. We will study a few more candidate definitions, and eventually see that all these definitions are actually equivalent.

The recursive partial functions seem intuitively computable, but their definition is a bit abstract. Our next "model of computation" will be more closely tied to our modern notion of computability which comes from our experience with computers.

We will introduce a theoretical machine called the *unlimited register machine* (URM). We will describe how the URM works and then we will study the partial functions that the machine can compute, the so called *URM-computable partial functions*.

3.1 How the URM works

The URM has infinitely many *registers*, labeled $R_0, R_1, R_2, \ldots$. Each register R_i contains at any given moment a number r_i. The machine is visualized as follows:

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	
r_0	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	$\cdots$

The contents of the machine can be altered by the machine implementing certain instructions. There are four types of instructions: zero instructions, successor instructions, transfer instructions, and jump instructions.

A *URM-program* (or simply, a *program*) is a finite, indexed list of instructions,

$$I_0, I_1, I_2, \ldots, I_{n-1},$$

where each I_k is an instruction.

(1) Zero instructions. For each $n \in \mathbb{N}$ there is a *zero instruction*, denoted $Z(n)$. $Z(n)$ instructs the machine to change the contents of register R_n to 0.

For example,

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	
9	1	0	2	4	1	0	11	2	$\cdots$

$$\Downarrow Z(4)$$

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	
9	1	0	2	0	1	0	11	2	$\cdots$

(2) Successor instructions. For each $n \in \mathbb{N}$ there is a *successor instruction* $S(n)$. $S(n)$ instructs the machine to add 1 to the contents of register R_n.

For example,

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	
9	1	0	2	4	1	0	11	2	$\cdots$

$$\Downarrow S(1)$$

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	
9	2	0	2	4	1	0	11	2	$\cdots$

(3) Transfer instructions. For each $m, n \in \mathbb{N}$ there is a *transfer instruction* $T(m, n)$. $T(m, n)$ instructs the machine to replace the contents of register R_n with the contents of R_m.

For example,

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	
9	1	0	2	4	1	0	11	2	$\cdots$

$$\Downarrow\ T(0,4)$$

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	
9	1	0	2	9	1	0	11	2	$\cdots$

(4) Jump instructions. For each $m, n, q \in \mathbb{N}$, there is a *jump instruction* $J(m, n, q)$. The effect of $J(m, n, q)$ is conditional. If $r_m = r_n$, go to the qth instruction in the program. Otherwise, when $r_m \neq r_n$, go to the next instruction in the list. When executing $J(m, n, q)$, the contents of the register are never changed; $J(m, n, q)$ only determines what instruction we execute next.

For example,

R_0	R_1	R_2	R_3	R_4	R_5	
9	1	0	2	4	1	$\cdots$

next: $I_k = J(0, 1, 7)$

$$\Downarrow\ J(0,1,7)$$

R_0	R_1	R_2	R_3	R_4	R_5	
9	1	0	2	4	1	$\cdots$

next: I_{k+1}

Since R_0 and R_1 have different contents, we simply go to the next instruction I_{k+1}.

For another example,

R_0	R_1	R_2	R_3	R_4	R_5	
9	1	0	2	4	1	$\cdots$

next: $I_k = J(1, 5, 2)$

$\Downarrow \; J(1, 5, 2)$

R_0	R_1	R_2	R_3	R_4	R_5	
9	1	0	2	4	1	$\cdots$

next: I_2

Since R_1 and R_5 have the same content (1), we jump to instruction I_2.

Note that an when executing a jump instruction of the form $J(n, n, q)$, we will always jump to instruction I_q since, trivially, R_n has the same content as itself, R_n.

Example 3.1. We will illustrate how a URM executes computations under a program with an example.

Consider the program P:

I_0 $J(0, 1, 5)$
I_1 $S(1)$
I_2 $S(2)$
I_3 $J(0, 1, 5)$
I_4 $J(0, 0, 1)$
I_5 $T(2, 0)$

We will consider the computation by the URM under program P with initial configuration

R_0	R_1	R_2	R_3	R_4	R_5	
6	4	0	0	0	0	$\cdots$

We start with the initial configuration, and apply the first instruction I_0 in the program. Then, we continue, following all the instructions in the program:

R_0	R_1	R_2	R_3	R_4	R_5	
6	4	0	0	0	0	$\cdots$

next:$I_0 = J(0, 1, 5)$

R_0	R_1	R_2	R_3	R_4	R_5	
6	4	0	0	0	0	$\cdots$

next:$I_1 = S(1)$

R_0	R_1	R_2	R_3	R_4	R_5	
6	5	0	0	0	0	$\cdots$

next:$I_2 = S(2)$

R_0	R_1	R_2	R_3	R_4	R_5	
6	5	1	0	0	0	$\cdots$

next:$I_3 = J(0, 1, 5)$

R_0	R_1	R_2	R_3	R_4	R_5	
6	5	1	0	0	0	$\cdots$

next:$I_4 = J(0, 0, 1)$

R_0	R_1	R_2	R_3	R_4	R_5	
6	5	1	0	0	0	$\cdots$

next:$I_1 = S(1)$

R_0	R_1	R_2	R_3	R_4	R_5	
6	6	1	0	0	0	$\cdots$

next:$I_2 = S(2)$

R_0	R_1	R_2	R_3	R_4	R_5	
6	6	2	0	0	0	$\cdots$

next:$I_3 = J(0, 1, 5)$

R_0	R_1	R_2	R_3	R_4	R_5	
6	6	2	0	0	0	$\cdots$

next:$I_5 = T(2, 0)$

R_0	R_1	R_2	R_3	R_4	R_5	
2	6	2	0	0	0	$44\cdot$

no next instruction

The computation halts here because there is no next instruction to follow.

In general, we say that a program P computation *terminates* or *halts* when the program instructs the machine to follow an instruction that does not exist, i.e., the program tells the machine to execute the nth instruction, but P has fewer than n instructions. This can happen in two ways:

(1) The machine executes a jump instruction $J(m, n, q)$ when the machine has contents $r_m = r_n$ and there is no instruction I_q; or

(2) The machine executes its last instruction I_s, there is no jump, and the next instruction is I_{s+1}, which does not exist.

Our example computation halted due to case (2).

Example 3.2. There are computations which never halt; for example, the program

I_0 $S(0)$

I_1 $J(0, 0, 0)$

will not halt for any initial state of the registers.

Exercise 3.3. Show that the program in the previous example never halts.

3.2 URM-computable functions

If at the beginning of a computation, the machine is in state

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	
a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	$\cdots$

then we say the initial configuration is $a_0, a_1, a_2, \ldots$

Let P be a program and let $a_0, a_1, a_2, \ldots$ be an infinite sequence in $\mathbb{N}$. We write $P(a_0, a_1, a_2, \ldots)$ to denote the computation under P with initial configuration $a_0, a_1, a_2, \ldots$. We use the notation

$$P(a_0, a_1, a_2, \ldots)\downarrow \iff \text{the computation } P(a_0, a_1, a_2, \ldots) \text{ halts,}$$

and

$$P(a_0, a_1, a_2, \ldots)\uparrow \iff \text{the computation } P(a_0, a_1, a_2, \ldots) \text{ does not halt.}$$

Usually, our initial configurations will have infinitely many zeros, so we introduce the following useful notation. Let $a_0, \ldots, a_{n-1}$ be a finite sequence

in $\mathbb{N}$. We write $P(a_0, \ldots, a_{n-1})$ for the computation $P(a_0, \ldots, a_{n-1}, 0, 0, \ldots)$. $P(a_0, \ldots, a_{n-1})\downarrow$ means that $P(a_0, \ldots, a_{n-1}, 0, 0, \ldots)\downarrow$, and $P(a_0, \ldots, a_{n-1})\uparrow$ means that $P(a_0, \ldots, a_{n-1}, 0, 0, \ldots)\uparrow$.

Definition 3.4. Let P be a program and let n be a positive integer. We define a partial function $f_P^{(n)} : \mathbb{N}^n \rightharpoonup \mathbb{N}$ as follows: for $a_0, \ldots, a_{n-1} \in \mathbb{N}$,

$$f_P^{(n)}(a_0, \ldots, a_{n-1})\downarrow = y \iff P(a_0, \ldots, a_{n-1})\downarrow \text{ and } y \text{ is the content of } R_0 \text{ when the computation halts.}$$

Definition 3.5. A partial function $g : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is URM-*computable* if there is a program P such that $g = f_P^{(n)}$. When $g = f_P^{(n)}$, we say that the program P *computes* g.

Example 3.6. The successor function $s : \mathbb{N} \to \mathbb{N}$ is URM-computable. Indeed, let P be the program with just one instruction, $I_0 = S(0)$. Then, for any $x \in \mathbb{N}$, $f_P^{(1)}(x) = s(x)$ since

R_0	R_1	R_2	R_3	R_4	R_5	
x	0	0	0	0	0	$\cdots$

next:$I_0 = S(0)$

R_0	R_1	R_2	R_3	R_4	R_5	
$x+1$	0	0	0	0	0	$\cdots$

no next instruction

Note that this same program P defines functions of different arities. For example, $f_P^{(3)}$ is the function $f_P^{(3)}(x, y, z) = x + 1$. Indeed, for any $x, y, z \in \mathbb{N}$,

R_0	R_1	R_2	R_3	R_4	R_5	
x	y	z	0	0	0	$\cdots$

next:$I_0 = S(0)$

R_0	R_1	R_2	R_3	R_4	R_5	
$x+1$	y	z	0	0	0	$\cdots$

no next instruction

Exercise 3.7. Two different programs P_0, P_1 can compute the same n-ary partial function, i.e., $f_{P_0}^{(n)} = f_{P_1}^{(n)}$.

Show that the program Q given by $I_0 = J(0,0,1)$, $I_1 = S(0)$ also defines the successor function. Then, give yet a third program which computes the successor function.

Example 3.8. The projection function $\pi_1^3 : \mathbb{N}^3 \to \mathbb{N}$ is URM-computable. Indeed, it is easy to show that the program P given by $I_0 = T(1,0)$ computes π_1^3.

Exercise 3.9. Show that every projection function π_i^n, $i < n$, is URM-computable.

Example 3.10. The constant functions $c_k^n : \mathbb{N}^n \to \mathbb{N}$, $c_k^n(\vec{x}) = k$, are all URM-computable. For instance, c_3^1 is computed by the program P given by the instruction $I_0 = Z(0)$, $I_1 = S(0)$, $I_2 = S(0)$, $I_3 = S(0)$. This same program also computes c_3^n for any integer $n > 0$.

3.3 Recursive partial functions are URM-computable

Our next big goal is to demonstrate that the URM-computable functions are exactly the recursive partial functions:

Theorem 3.11. A partial function is URM-computable if and only if it is recursive.

In this section, we will discuss why all recursive partial functions are URM-computable. Later, we will discuss the other direction, that URM-computable partial functions are all recursive.

Recall that the class $\mathcal{R}$ of recursive partial functions is the smallest class of partial functions which is recursively closed. So, if we show that the class $\mathcal{U}$ of URM-computable partial functions is recursively closed, it follows that $\mathcal{R} \subseteq \mathcal{U}$, i.e., the class of recursive partial functions is contained in the class of URM-computable partial functions.

Theorem 3.12. The class $\mathcal{U}$ of URM-computable partial functions is recursively closed, i.e.,

(i) $\mathcal{U}$ contains the successor function s, all the projection functions π_i^n, $i < n$, and all the constant functions c_k^n.

(ii) $\mathcal{U}$ is closed under definition by substitution.

(iii) $\mathcal{U}$ is closed under definition by primitive recursion.

(iv) $\mathcal{U}$ is closed under definition by minimization.

We have already show in the previous section that many of the functions in (i) are all URM-computable. The proofs of (ii)-(iv) require a lot of URM programming, which involves some technicalities. We will not prove the theorem in full details, but we will sketch the main ideas of the proof of (ii) by examining a special case.

Consider URM-computable partial functions $f : \mathbb{N}^2 \rightharpoonup \mathbb{N}$, $g_0 : \mathbb{N} \rightharpoonup \mathbb{N}$, $g_1 : \mathbb{N} \rightharpoonup \mathbb{N}$. We sketch a proof that the composition function

$$h(x) = f(g_0(x), g_1(x))$$

is also URM-computable. Let P_f, P_0, and P_1 be URM-programs that compute f, g_0, and g_1, respectively. We define a new program P_h as follows:

(1) The first instruction in P_h should transfer the content of R_0 (which is the input x) into a register that is "out of the way". This instruction stores the input x for use later. We want to store it in a register that we will not need for the steps that follow. A register that has an index larger than any index mentioned in any of P_f, P_0, P_1 will do. After this instruction, the register will look like

R_0	R_1	R_2	R_3	R_4	R_5		R_k	R_{k+1}	
x	0	0	0	0	0	$\cdots$	x	0	$\cdots$

(2) Next, we put the instructions from P_0. This will cause the program to try to compute $g_0(x)$. Assuming this computation converges, we will have:

R_0	R_1	R_2	R_3	R_4	R_5		R_k	R_{k+1}	
$g_0(x)$	$*$	$*$	$*$	$*$	$*$	$\cdots$	x	0	$\cdots$

The $*$ symbol indicates that we don't know exactly what those registers will contain. Indeed, the computation of $g_0(x)$ may have left "scratch work" in many of them.

(3) The next instruction should transfer the content of the first register to another register that is "out of the way". This instruction stores the

result of $g_0(x)$ (assuming it converged) so that we can remember it later. After this instruction is implemented, the machine will look like:

R_0	R_1	R_2	R_3	R_4	R_5		R_k	R_{k+1}	
$g_0(x)$	$*$	$*$	$*$	$*$	$*$	$\cdots$	x	$g_0(x)$	$\cdots$

(4) Consider the registers that are mentioned by P_1. The next instructions should reset all of these registers to $x, 0, 0, \ldots, 0$. We can load x into the first register from the register we stored x in during (1). This way, we are ready to compute $g_1(x)$, and the machine will be in the following state:

R_0	R_1	R_2	R_3	R_4	R_5		R_k	R_{k+1}	
x	0	0	0	0	0	$\cdots$	x	$g_0(x)$	$\cdots$

(5) The next set of instructions should be P_1, so that we try to compute $g_1(x)$ next. Assuming this computation converges, we will have:

R_0	R_1	R_2	R_3	R_4	R_5		R_k	R_{k+1}	
$g_1(x)$	$*$	$*$	$*$	$*$	$*$	$\cdots$	x	$g_0(x)$	$\cdots$

(6) Next, we should load the contents of R_0 (which will be $g_1(x)$, assuming the computation converged) into R_1. Next, transfer the stored value $g_0(x)$ from (3) into R_0. Then, zero out all the other registers, so that the contents of the registers are

R_0	R_1	R_2	R_3	R_4	R_5		R_k	R_{k+1}	
$g_0(x)$	$g_1(x)$	0	0	0	0	$\cdots$	0	0	$\cdots$

(7) Finally, the next instructions should be the program P_f. This way, we will compute $f(g_0(x), g_1(x))$.

There are some technicalities to be addressed. In steps (2), (5), and (7), it turns out we cannot simply use the programs P_0, P_1, and P_f as they are.

This is because the original indexing of the the instructions in these programs does not correspond to the indexing of the instructions in P_h. E.g., the third instruction of P_0 is no longer the third instruction of P_h. This will mean that the jump instructions in P_0, P_1, P_f must be updated so that they correctly reference the instruction as they are indexed in P_h.

The other technicality has to do with how the programs P_0, P_1, P_f halt. These are now "subroutines" in a larger program, so when these subroutines halt, we want to make sure we move on to the correct next step. In other words, when we place these subroutines in P_h, we want to change these programs so that they always halt by referencing the next instruction we want to execute in P_h.

As we can see, the proof that all recursive partial functions are URM-computable is somewhat technical, but the above sketch gives us a sense of how to do the necessary programing with URM's. In the next section, we will sketch the proof of another technical result: that all URM-computable functions are recursive.

3.4 URM-computable partial functions are recursive

We are able to show that recursive partial functions are URM-computable by programming URM's and showing they can execute definitions by substitution, primitive recursion, and minimization.

To show that URM-computable functions are recursive takes a different type of proof. A priori, the recursive partial functions only talk about numbers and their arithmetic, but somehow we have to show that the recursive partial functions can do everything a URM can do. To do this, we will encode the information of a URM-computation into arithmetic. This process is called *arithmetization*, and it is used throughout mathematical logic, most famously by Godel is his incompleteness theorems.

Throughout, we will use the following notational convention. If $*$ is some object, $\#*$ will denote the code of $*$.

We assign numbers to the four different types of instructions:

$$\#Z =_{\text{df}} 0, \quad \#S =_{\text{df}} 1, \quad \#T =_{\text{df}} 2, \quad \#J = 3.$$

Next, we assign codes for instructions:

$$\begin{aligned}\#Z(n) &=_{\text{df}} \langle \#Z, n\rangle\\ \#S(n) &=_{\text{df}} \langle \#S, n\rangle\\ \#T(n,m) &=_{\text{df}} \langle \#T, n, m\rangle\\ \#J(n,m,q) &=_{\text{df}} \langle \#J, n, m, q\rangle\end{aligned}$$

Having assigned codes to instructions, we can now assign codes to programs. If $P = I_0, \ldots, I_{s-1}$ is a program, then

$$\#P =_{\text{df}} \langle \#I_0, \ldots, \#I_{s-1}\rangle.$$

Example 3.13. Consider the program $P = Z(2), T(2,1)$. Then,

$$\#Z(2) = \langle \#Z, 2\rangle = \langle 0, 2\rangle = 2^{0+1}3^{2+1} = 54,$$

$$\#T(2,1) = \langle \#T, 2, 1\rangle = \langle 2, 2, 1\rangle = 2^3 3^3 5^2 = 5400,$$

and

$$\#P = \langle \#Z(2), \#T(2,1)\rangle = \langle 54, 5400\rangle = 2^{1+54}3^{1+5400}.$$

Of course, these codes are quite immense and not suitable for practical purposes, but the coding is still useful theoretically.

The following lemma will be useful:

Lemma 3.14. The following subsets of $\mathbb{N}$ are recursive:

$$\begin{aligned}IC &= \text{set of instruction codes}\\ PC &= \text{set of program codes}\end{aligned}$$

Proof. First, establish the equivalence

$$\begin{aligned}IC(u) \iff \text{Seq}(u) \,\&\, \big[&(\text{lh}(u) = 2 \,\&\, (u)_0 = \#Z)\\ &\vee (\text{lh}(u) = 2 \,\&\, (u)_0 = \#S) \vee (\text{lh}(u) = 3 \,\&\, (u)_0 = \#T)\\ &\vee (\text{lh}(u) = 4 \,\&\, (u)_0 = \#J)\big],\end{aligned}$$

then use the closure properties of recursive relations.

The proof that PC is recursive is left as an exercise. □

Exercise 3.15. Show that PC is recursive. *Hint.* u is in PC if and only if it codes a sequence, each of whose entries is in IC.

Next, we will code the URM in various states while it runs a program. For a program P, let $\rho(P)$ denote the largest index mentioned by program P. The program only effects registers R_i with $i \leq \rho(P)$. In the context of running program P, these registers are called the *working registers*.

Definition 3.16. Let $P = I_0, \ldots, I_{s-1}$ be a program, let n be a positive integer, and let $m = \max(\rho(P), n-1)$. A code u is a *state code for* $f_P^{(n)}$ if either (1) there is $t < s$ such that $u = \langle t, \#I_t, \langle r_0, \ldots, r_m \rangle \rangle$ for some $r_0, \ldots, r_m \in \mathbb{N}$; or (2) there is $t \geq s$ such that $u = \langle t, 0, \langle r_0, \ldots, r_m \rangle \rangle$.

The idea behind this definition is that if we are running program $P = I_0, \ldots, I_{s-1}$ to compute the n-ary function $f_P^{(n)}$ and the working registers $R_0, \ldots, R_m$, $m = \max(\rho(P), n-1)$ have content $r_0, \ldots, r_m$, and the next instruction to be executed by the machine is I_t with $t < s$, then the code for this state is $\langle t, \#I_t, \langle r_0, \ldots, r_m \rangle \rangle$. If, on the other hand, $t \geq s$ so that the next instruction I_t does not exists, then the code for this state is $\langle t, 0, \langle r_0, \ldots, r_m \rangle \rangle$; this type of code in particular is called a *halting state code*.

We define the following recursive functions to help us remember what information state codes are storing:

$$\mathrm{InsNum}(u) =_{\mathrm{df}} (u)_0, \qquad \mathrm{Ins}(u) =_{\mathrm{df}} (u)_1, \qquad \mathrm{Reg}(u,i) =_{\mathrm{df}} (u)_{2,i}$$

If $u = \langle t, \#I_t, \langle r_1, \ldots, r_k \rangle \rangle$, then $\mathrm{InsNum}(u) = t$, the index of the next instruction, $\mathrm{Ins}(u) = \#I_t$, the code for the next instruction (or 0 if it's a halting instruction), and $\mathrm{Reg}(u,i) = r_i$, the content of register i.

Remark. With this type of arithmetization, it is very common to conflate objects with their numerical codes. For instance, if e is the code of a program P, then when we say something like "the first instruction of e" we really mean "the first instruction of P (the program coded by e)". Or, if u is a state code, then we might say "in state u, register R_0 has content 4" to mean "in the state *coded by* u, the register R_0 has content 4". This type of conflation is ultimately harmless, but one has to get used to it.

Lemma 3.17. There is a recursive relation $\mathrm{StateCode} \subseteq \mathbb{N}^3$ such that

$$\mathrm{StateCode}(n,e,u) \iff_{\mathrm{df}} \begin{array}{l} e \text{ is the code of program } P \\ \text{and } u \text{ is a state code for } f_P^{(n)}. \end{array}$$

Proof. Show that the relation satisfies the equivalence

$$\begin{aligned} \mathrm{StateCode}(n,e,u) \iff\ & n > 0 \ \&\ \mathrm{PC}(e) \ \&\ \mathrm{Seq}(u) \ \&\ \mathrm{lh}(u) = 3 \ \&\ \mathrm{Seq}((u)_2) \\ & \&\ \mathrm{lh}((u)_2) = \max(\rho(e), n-1) \ \&\ \mathrm{Ins}(u) = (e)_{\mathrm{InsNum}(u)} \end{aligned}$$

and then use the usual closure properties of recursive relations. One technicality is that we have to show that there is a recursive function which will compute $\rho(e)$, but this is not too difficult to show. Note that if u is a halting state, then $(e)_{\text{InsNum}(u)} = (e)_t = 0 = \text{Ins}(u)$ since $t \geq \text{lh}(e)$. □

We define the following recursive relation,

$$\text{HaltState}(n, e, u) \iff \text{StateCode}(n, e, u) \;\&\; \text{Ins}(u) = 0,$$

which holds if and only if u is a halting state code for $f_P^{(n)}$, where P is the program coded by e.

The next lemma says that we can, in a recursive manner, check whether a change from one state to another is done according to the rules by which URM's run.

Lemma 3.18. There is a recursive relation CmpSt such that

$$\begin{aligned}\text{CmpSt}(n, e, u, v) \iff\ & e \text{ is a code for program } P,\ u, v \text{ are state codes for } f_P^{(n)},\\ & \text{and } v \text{ is the code for the state that results when applying}\\ & \text{the instruction of } u \text{ to the register content of } u.\end{aligned}$$

Proof. We need to account for all the different types of instructions that u could have, and for each one make sure that we can arithmatize them in a recursive way.

Consider, for example, the case where the state code u has a zero instruction $Z(j)$. Then, to check if v is the code for the state which occurs after $Z(j)$ is executed, we need to check that the jth register of v contains 0, and the rest of the registers of v are identical to the registers in u. Finally, we need to make sure that the instruction that v contains is the instruction whose index is the successor of the index for the instruction of u. We can write all of this using conjunctions of recursive relations

$$\begin{aligned}\text{Ins}(u) = \#Z(j) \,\&\, \text{Reg}(v, j) = 0 \,\&\;\; (\forall i < \ell)(i \neq j \to \text{Reg}(u, i) = \text{Reg}(v, i))\\ \&\, \text{InsNum}(v) = \text{InsNum}(u) + 1,\end{aligned}$$

where ℓ is the number of registers. Of course, we will need similar clauses for the other types of instructions.

The proof is finished by by verifying that the following equivalence is valid and using (many, many) applications of our closure properties for recursive

relations:

$$
\begin{aligned}
\mathrm{CmpSt}(n,e,u,v) \iff\ & \mathrm{StateCode}(n,e,u)\ \&\ \neg\mathrm{HaltState}(n,e,u)\\
& \&\ \mathrm{StateCode}(n,e,v)\ \ [\text{set } \ell = \mathrm{lh}((u)_2) = \mathrm{lh}((v)_2)]\\
& \&\ \exists j,k,q \le u \text{ s.t}\\
& \Big[\Big(\mathrm{Ins}(u) = \#Z(j)\ \&\ \mathrm{Reg}(v,j) = 0\\
& \qquad\qquad \&\ (\forall i < \ell)(i \neq j \to \mathrm{Reg}(u,i) = \mathrm{Reg}(v,i))\\
& \qquad\qquad \&\ \mathrm{InsNum}(v) = \mathrm{InsNum}(u) + 1\Big)\\
& \vee \Big(\mathrm{Ins}(u) = \#S(j)\ \&\ \mathrm{Reg}(v,j) = \mathrm{Reg}(u,j) + 1\\
& \qquad\qquad \&\ (\forall i < \ell)(i \neq j \to \mathrm{Reg}(u,i) = \mathrm{Reg}(v,i))\\
& \qquad\qquad \&\ \mathrm{InsNum}(v) = \mathrm{InsNum}(u) + 1\Big)\\
& \vee \Big(\mathrm{Ins}(u) = \#T(j,k)\ \&\ \mathrm{Reg}(v,k) = \mathrm{Reg}(u,j)\\
& \qquad\qquad \&\ (\forall i < \ell)(i \neq k \to \mathrm{Reg}(u,i) = \mathrm{Reg}(v,i))\\
& \qquad\qquad \&\ \mathrm{InsNum}(v) = \mathrm{InsNum}(u) + 1\Big)\\
& \vee \Big(\mathrm{Ins}(u) = \#J(j,k,q)\ \&\ (\forall i < \ell)\mathrm{Reg}(u,i) = \mathrm{Reg}(v,i)\\
& \qquad\qquad \&\Big\{(\mathrm{Reg}(u,j) = \mathrm{Reg}(u,k)\ \&\ \mathrm{InsNum}(v) = q)\\
& \qquad\qquad\quad \vee (\mathrm{Reg}(u,j) \neq \mathrm{Reg}(u,k)\\
& \qquad\qquad\qquad\quad \&\ \mathrm{InsNum}(v) = \mathrm{InsNum}(u) + 1)\Big\}\Big)\Big]
\end{aligned}
$$

□

Theorem 3.19 (Kleene T-predicate)**.** For each positive integer n, there is a recursive $(n+2)$-ary relation $T_n(e,\vec{x},y)$ and a total recursive function $U : \mathbb{N} \to \mathbb{N}$ which satisfy the following condition: for every program P with code $e = \#P$ and every $\vec{x} \in \mathbb{N}^n$,

$$f_P^{(n)}(\vec{x}) = U(\mu y[T_n(e,\vec{x},y)])$$

Proof. Recall the recursive relation $\mathrm{HaltState}(n,e,u)$ which holds if and only if u is a halting state code for $f_P^{(n)}$, where P is the program coded by e. We

will also define, for each positive integer n, a recursive $(n+2)$-ary relation,

$$\begin{aligned}\text{InitialState}_n(e,u,\vec{x}) \iff & \text{StateCode}(n,e,u) \;\&\; \text{InsNum}(u)=0 \\ & \;\&\; \text{Reg}(u,0)=x_0 \;\&\; \cdots \;\&\; \text{Reg}(u,n-1)=x_{n-1} \\ & \;\&\; (\forall i<\text{lh}((u)_2))[i\geq n \to \text{Reg}(u,i)=0]\end{aligned}$$

which holds if and only if u is an state code for the initial configuration of the machine which computes $f_P^{(n)}$ (with $e=\#P$) with input $\vec{x}$; such a u is of the form $u=\langle 0,\#I_0,\langle x_0,\ldots,x_{n-1},0,\ldots,0\rangle\rangle$.

Now, we can define the Kleene T-predicate as follows:

$$\begin{aligned}T_n(e,\vec{x},y) \iff & \text{PC}(e) \;\&\; \text{Seq}(y) \;\&\; (\forall i<\text{lh}(y)-2)\text{CmpSt}(n,e,(y)_i,(y)_{i+1}) \\ & \;\&\; \text{InitialState}_n(e,(y)_0,\vec{x}) \;\&\; \text{HaltState}(n,e,(y)_{\text{lh}(y)-1})\end{aligned}$$

Thus, $T_n(e,\vec{x},y)$ if and only if y is a code of a halting computation under program P (with $e=\#P$) with the registers initially configured with $x_0,\ldots,x_{n-1},0,0,\ldots$. For such a y, $(y)_{\text{lh}(y)-1}$ is a halting state code of the form

$$(y)_{\text{lh}(y)-1}=\langle t,0,\langle r_0,\ldots,r_m\rangle\rangle,$$

where $(y)_{\text{lh}(y)-1,2,0}=r_0=f_P^{(n)}(\vec{x})$. So, we define the recursive function $U:\mathbb{N}\to\mathbb{N}$ by

$$U(y)=(y)_{\text{lh}(y)-1,2,0}$$

Thus, for a program P with code e,

$$f_P^{(n)}(\vec{x})=U(\mu y T_n(e,\vec{x},y))$$

as desired. □

Corollary 3.20. Every URM-computable partial function is recursive.

Proof. We have seen that every URM-computable partial function $f:\mathbb{N}^n\rightharpoonup\mathbb{N}$ is of the form

$$f(\vec{x})=U(\mu y[T_n(e,\vec{x},y)]),$$

where T_n is recursive and U is recursive. □

We have now seen that the recursive partial functions are precisely the URM-computable functions. This is a remarkable fact, since the classes are defined in very different ways. Both notions are giving equivalent definitions of a very special class of partial functions, and, in fact, there are many other ways of defining this class, e.g., using Turing machines, as will be done in Chapter 4.

The equivalence of these definitions, each capturing different intuitive features of the philosophical notion of "computability", gives credence to the idea that the recursive partial functions (i.e., the URM-computable partial functions) really are the partial functions that we *ought* to consider as precisely the partial functions which are computable.

Note that similar remarks apply to which relations we *ought* to consider to be *decidable*. We define a relation $R \subseteq \mathbb{N}^n$ to be *URM-decidable* if its characteristic function Char_R is URM-computable. Of course, then, R is URM-decidable if and only if R is recursive.

We will discuss these ideas more later, in Section 4.2 on the Church-Turing Thesis. For now, we just introduce the following terminological conventions, which reflect the idea that we have correctly

(1) We say a partial function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is *computable* if it is recursive (equivalently, if it is URM-computable).

(2) We say that a relation $R \subseteq \mathbb{N}^n$ is *decidable* or *computable* if it is recursive (equivalently, if it is URM-decidable).

In the sequel, we will often still use the old terminology *recursive partial function* and *recursive relation*.

3.5 Effective enumerations and the s-m-n theorem

The existence of the Kleene T-predicate from the previous section has many important consequences. For each $n > 0$ and $e \in \mathbb{N}$, define $\varphi_e^{(n)} : \mathbb{N}^n \rightharpoonup \mathbb{N}$ by

$$\varphi_e^{(n)}(\vec{x}) =_{\mathrm{df}} U(\mu y[T_n(e, \vec{x}, y)]),$$

Our work above shows that, for a fixed n,

$$\varphi_0^{(n)}, \varphi_1^{(n)}, \varphi_2^{(n)}, \ldots$$

lists (or enumerates) all of the n-ary computable partial functions. However, this list has a special property: it is an *effective enumeration* in the sense that the partial function $\psi_U^{(n)} : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ defined by

$$\psi_U^{(n)}(e, \vec{x}) =_{\mathrm{df}} \varphi_e^{(n)}(\vec{x}) = U(\mu y[T_n(e, \vec{x}, y)])$$

is computable.

Since $\psi_U^{(n)}$ is computable, it is of course URM-computable. Let P be a

program that computes it. Then, for every $e \in \mathbb{N}$ and $\vec{x} \in \mathbb{N}^n$,

$$f_P^{(n+1)}(e, \vec{x}) = \varphi_e^{(n)}(\vec{x})$$

So, this single program P has the ability to compute any n-ary computable partial function. For this reason, the machine running this program is called the *universal machine for n-ary computable partial functions*.

When $f = \varphi_e^{(n)}$, we will refer to e as a *code for f*, or sometimes (conflating codes with the coded objects) we will call e a *program for f*. Recall that each n-ary computable partial function f has infinitely many codes. In particular, the enumeration

$$\varphi_0^{(n)}, \varphi_1^{(n)}, \varphi_2^{(n)}, \ldots$$

has many, many repetitions.

We pause to remark that by convention, we use the notation

$$\varphi_e =_{\text{df}} \varphi_e^{(1)},$$

in other words, when φ_e has no superscript, the arity it understood to be unary.

We can now use the effective enumeration to prove the promised strengthened version of Theorem 2.28.

Theorem 3.21. Let f and g be n-ary computable *partial* functions, and let $R(\vec{x})$ be an n-ary decidable relation. Then, the partial function $h : \mathbb{N}^n \rightharpoonup \mathbb{N}$ defined by

$$h(\vec{x}) = \begin{cases} f(\vec{x}) & \text{if } R(\vec{x}) \\ g(\vec{x}) & \text{if } \neg R(\vec{x}) \end{cases}$$

is computable.

Proof. Fix codes e_f and e_g for f and g respectively. Define

$$u(\vec{x}) = e_f \cdot \text{Char}_R(\vec{x}) + e_g \cdot \text{Char}_{\neg R}(\vec{x}).$$

Note that u is a computable total function and that $u(\vec{x}) = e_f$ when $R(\vec{x})$ holds and $u(\vec{x}) = e_g$ when $\neg R(\vec{x})$ holds. Then, check that

$$h(\vec{x}) = \varphi_{u(\vec{x})}(\vec{x}),$$

and use that h is computable since the universal n-ary computable partial function is computable. □

Note the following particular (and important) special case of this theorem: if f is a computable partial n-ary function and R is a decidable n-ary relation, then the partial function

$$h(\vec{x}) = \begin{cases} f(\vec{x}) & \text{if } R(\vec{x}) \\ \uparrow & \text{otherwise} \end{cases}$$

is computable.

The effective enumeration $\{\varphi_e^{(n)}\}$ has another important property. Fix $m, n > 0$, and consider a computable partial function $f : \mathbb{N}^{m+n} \rightharpoonup \mathbb{N}$. Suppose, however, you only care about values of this function when the first m variables are some fixed $\vec{x}$. In other words, you care about the function

$$g : \mathbb{N}^n \rightharpoonup \mathbb{N}, \qquad g(\vec{y}) = f(\vec{x}, \vec{y}).$$

It is easy to see that g is computable (using closure under computable substitutions), but can we easily find a program to compute g? Moreover, can we compute a code for a program for g given just the fixed entries $\vec{x}$? The s-m-n theorem says that we can.

Theorem 3.22 (s-m-n theorem)**.** Let $f : \mathbb{N}^{m+n} \rightharpoonup \mathbb{N}$ be a computable partial function. Then, there is a computable total function $u : \mathbb{N}^m \to \mathbb{N}$ such that

$$\varphi^{(n)}_{u(\vec{x})}(\vec{y}) = f(\vec{x}, \vec{y}),$$

for any $\vec{x} \in \mathbb{N}^m$ and $\vec{y} \in \mathbb{N}^n$. Moreover, there is always such a u which is injective.

Remark. The fact that the function u in the theorem can be assumed to be injective is often irrelevant; however, it is an important technical detail in some arguments. For example, it will be useful in Section 7.3, where we pay careful attention to injectivity.

Again, we will avoid getting bogged down in the technicalities of URM programming; in fact, we will only sketch the proof of a special case where $m = n = 1$.

Let $f : \mathbb{N}^{1+1} \rightharpoonup \mathbb{N}$ be a computable partial function. We want a computable total $u : \mathbb{N} \to \mathbb{N}$ such that for all $x, y \in \mathbb{N}$,

$$\varphi_{u(x)}(y) = f(x, y).$$

(Here, we are using the convention that $\varphi_e =_{\text{df}} \varphi_e^{(1)}$.) Consider, for the mo-

ment, fixing $x \in \mathbb{N}$ and defining $g : \mathbb{N} \rightharpoonup \mathbb{N}$ by

$$g(y) = f(x, y) \qquad [x, y \in \mathbb{N}].$$

First, we describe a URM-program that computes the function g. Then, we will argue that we can compute a code of this program with a computable function u which uses the same procedure for any $x \in \mathbb{N}$.

The program which computes g is composed of the following pieces, in order:

(1) On input y, the machine will start in state

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	
y	0	0	0	0	0	0	0	0	$\cdots$

The first "subprogram" E_1 is just the instructions that move y to R_2, and then zeros out R_0, so that we have

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	
0	y	0	0	0	0	0	0	0	$\cdots$

(2) Next, we need to load x into R_0. This is done by a subprogram E_2 which has x-many $S(0)$ instructions in a row. After this is executed, we will have

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	
x	y	0	0	0	0	0	0	0	$\cdots$

and we are ready to compute f on x and y.

(3) The next subprogram E_3 is just the program which computes f, where we adjust the indexing of the jump instructions. The amount we have to adjust the indexing is determined by how many instruction are in the subprograms from (1) and (2). If this is done properly, and the computation

converges, then we will end in state

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	
$f(x,y)$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$\cdots$

as desired.

So, our program is $E_1E_2E_3$, put together one after another. Roughly, our function $u : \mathbb{N} \to \mathbb{N}$ should be

$$u(x) = \#E_1 * \#E_2 * \#E_3,$$

however, the programs E_2 and E_3 both depend on x, so we should write them as $E_2(x)$ and $E_3(x)$ to emphasize their dependence on x. The program E_1 does not depend on x, and so $\#E_1 = c$ for some constant c not depending on x.

For E_2 and E_3, we need to show that there are computable total functions $v, w : \mathbb{N} \to \mathbb{N}$ such that for any $x \in \mathbb{N}$,

$$v(x) = \#E_2(x) \qquad w(x) = \#E_3(x).$$

When $x = 0$, $v(x)$ should just be the code of the empty program since we don't need to load anything into R_0. For $x > 0$, $v(x) = \langle S(0), S(0), \ldots, S(0)\rangle$, where there are x-many $S(0)$. We can easily define this $v(x)$ using a primitive recursion.

Exercise 3.23. Show that $v(x)$ is computable.

Note that v is clearly an injective function. Since $u(x) = \#E_1 * v(x) * u(x)$, it follows that the function u is also injective.

$w(x)$ is a bit more technical to define, since we have to be careful about counting instructions and what indices to update. We omit these details.

Once we have $v(x)$ and $w(x)$ and we know they are computable, then the proof is done since

$$u(x) = c * v(x) * w(x)$$

is injective and computable and gives the code of a program which computes g.

Now that we have sketched a proof of the s-m-n theorem, we will give a simple example of how to use it.

Example 3.24. We will prove that there is a computable total function $u : \mathbb{N} \to \mathbb{N}$ such that

$$\varphi_{u(x)}(y)\downarrow = x + y$$

for every x and y. (In other words, u is a computable procedure which, given x, returns a code for the "add x" function.)

First define $f : \mathbb{N}^2 \to \mathbb{N}$ by

$$f(x, y) = x + y.$$

Clearly f is computable, so by the s-m-n theorem there is a computable total $u : \mathbb{N} \to \mathbb{N}$ such that

$$\varphi_{u(x)}(y) = f(x, y) = x + y$$

for every $x, y \in \mathbb{N}$.

Exercise 3.25. Prove the following stronger version of the s-m-n theorem:

For every $m, n \geq 1$, there is a computable total $(m+1)$-ary function $s_n^m(e, \vec{x})$ such that

$$\varphi_e^{(m+n)}(\vec{x}, \vec{y}) = \varphi_{s_n^m(e,\vec{x})}^{(n)}(\vec{y}).$$

for all $\vec{x} \in \mathbb{N}^m$ and $\vec{y} \in \mathbb{N}^n$.

3.6 Kleene's second recursion theorem

In this section, we will establish a very important consequence of the s-m-n theorem.

Theorem 3.26 (Kleene's second recursion theorem)**.** If $f(e, \vec{x})$ is an $(n + 1)$-ary computable partial function, then there exists $e^* \in \mathbb{N}$ such that

$$\varphi_{e^*}^{(n)}(\vec{x}) = f(e^*, \vec{x}).$$

Before we give the proof, we comment that, although the proof is elementary, it is a bit mysterious.

Proof. First, define the partial function $h : \mathbb{N}^{n+2} \rightharpoonup \mathbb{N}$ by

$$h(z, e, \vec{x}) = \varphi_z^{(n+1)}(e, \vec{x}).$$

h is a computable partial function (since there are universal machines). By the s-m-n theorem, we have a computable total $S : \mathbb{N}^2 \to \mathbb{N}$ such that

$$\varphi_{S(z,e)}(\vec{x}) = h(z, e, \vec{x}) = \varphi_z^{(n+1)}(e, \vec{x})$$

for all $z, e \in \mathbb{N}$ and $\vec{x} \in \mathbb{N}^n$. Define $g : \mathbb{N}^{(n+1)} \rightharpoonup \mathbb{N}$ by

$$g(e, \vec{x}) = f(S(e, e), \vec{x}).$$

g is computable, so we can find e_0 with $\varphi_{e_0}^{(n+1)} = g$. Define $e^* = S(e_0, e_0)$. Then,

$$\varphi_{e^*}^{(n)}(\vec{x}) = \varphi_{S(e_0,e_0)}^{(n)}(\vec{x}) = \varphi_{e_0}^{(n+1)}(e_0, \vec{x}) = g(e_0, \vec{x}) = f(S(e_0, e_0), \vec{x}) = f(e^*, \vec{x}),$$

as desired. □

The following is an immediate consequence of the second recursion theorem.

Corollary 3.27 (Roger's fixed point theorem)**.** If $f : \mathbb{N} \to \mathbb{N}$ is a computable total function, then for any $n > 0$ there is $e^* \in \mathbb{N}$ such that $\varphi_{e^*}^{(n)} = \varphi_{f(e^*)}^{(n)}$.

The proof is left as an exercise. Actually, the second recursion theorem and Roger's fixed point theorem easily imply each other. In some sense, they contain the same mathematical content. It is useful as an exercise to prove the second recursion theorem from Roger's theorem.

Exercise 3.28. Prove Roger's fixed point theorem.

Also, assuming Roger's fixed point theorem, prove the second recursion theorem.

The second recursion theorem has many deep and important applications, but for now we will only give two simple applications.

(1) We will use the second recursion theorem to prove that there is an $x^* \in \mathbb{N}$ such that

$$\varphi_{x^*}(y)\!\downarrow = x^* + y$$

for all $y \in \mathbb{N}$.[3] The function

$$f(x, y) = x + y$$

is computable. Thus, by the second recursion theorem there exists $x^* \in \mathbb{N}$ such that

$$\varphi_{x^*}(y) = f(x^*, y) = x^* + y$$

[3]It is worth reflecting on why it is so strange that such an x^* should exist. x^* codes URM instructions, so you might think it just has x^*-many $S(0)$ instructions. However, when you take the code of those instructions, you would necessarily get a number much larger than x^*. In fact, for this reason the program can't directly reference the number x^* at all. Instead the instructions must cleverly reference x^* *indirectly*; this is why the proof of the second recursion theorem is so mysterious.

for all $y \in \mathbb{N}$.

Exercise 3.29. Prove that there is $x^* \in \mathbb{N}$ such that

$$\varphi_{x^*}(y) = x^* \cdot y$$

for all $y \in \mathbb{N}$.

(2) Our second application is more difficult. We will see that the second recursion theorem shows us that very general types of recursion are computable.

Recall that in Exercise 2.57 we showed that the Ackermann function is computable. That was quite a bit of work, and now we will use the second recursion theorem to get a much easier proof that Ackermann function is computable.

Recall the Ackermann function $A : \mathbb{N}^2 \to \mathbb{N}$, which is the function defined by the recursive equations

$$\begin{aligned} A(0, n) &= n + 1 \\ A(m + 1, 0) &= A(m, 1) \\ A(m + 1, n + 1) &= A(m, A(m + 1, n)) \end{aligned}$$

By Exercise 2.56, A is total, hence it is the unique function which satisfies these equations.

Define the function

$$f(e, m, n) = \begin{cases} n + 1 & \text{if } m = 0 \\ \varphi_e^{(2)}(m \dot{-} 1, 1) & \text{if } m > 0 \text{ and } n = 0 \\ \varphi_e^{(2)}(m \dot{-} 1, \varphi_e^{(2)}(m, n \dot{-} 1)) & \text{if } m, n > 0, \end{cases}$$

which is computable by Theorem 3.21. By the second recursion theorem, there is e^* such that $\varphi_{e^*}^{(2)}(m, n) = f(e^*, m, n)$ for all $m, n \in \mathbb{N}$. Thus, φ_{e^*} satisfies the recursive equations which uniquely define A. Thus, $A = \varphi_{e^*}^{(2)}$, which shows that A is computable.

§ **The second recursion theorem with parameters**. The second recursion theorem can be strengthened to accommodate parameters. Consider a computable partial function $f(e, y, \vec{x})$. If we fix the y value to be, say, 0, we can apply the second recursion theorem to get a fixed e_0 satisfying

$$\varphi_{e_0}^{(n)}(\vec{x}) = f(e_0, 0, \vec{x}).$$

Similarly, we could set y to be $1, 2, \ldots$ and get $e_1, e_2, \ldots$ with

$$\varphi_{e_1}^{(n)}(\vec{x}) = f(e_1, 1, \vec{x}), \quad \varphi_{e_2}^{(n)}(\vec{x}) = f(e_2, 2, \vec{x}), \quad \ldots$$

Naturally, as computability theorists, we ask: is there a computable procedure which, given some value $y \in \mathbb{N}$, computes e_y. The second recursion theorem with parameters tells us that the answer is "yes".

Theorem 3.30 (Second recursion theorem with parameters)**.** Let $f = f(e, \vec{y}, \vec{x}) : \mathbb{N}^{1+m+n} \rightharpoonup \mathbb{N}$ be a computable partial function. Then, there is a computable total function $u : \mathbb{N}^m \to \mathbb{N}$ such that

$$\varphi_{u(\vec{y})}^{(n)}(\vec{x}) = f(u(\vec{y}), \vec{y}, \vec{x})$$

for all $\vec{y} \in \mathbb{N}^m$ and all $\vec{x} \in \mathbb{N}^n$. Moreover, there is always such a u which is injective.

Exercise 3.31. Prove Theorem 3.30. *Hint.* You only need to slightly modify the proof of second recursion theorem without parameters. The injectivity of u will follow from the injectivity we may assume fro s-m-n functions.

Often, we only need the version without parameters, but the stronger version with parameters is sometimes necessary. The injectivity condition is sometimes relevant, but not always. We end this section with an example and some exercises.

Example 3.32. Let $g : \mathbb{N} \to \mathbb{N}$ be a computable total function and let $A \subseteq \mathbb{N}$ be decidable. We will show that there is a computable total $u : \mathbb{N} \to \mathbb{N}$ such that

$$\mathrm{Dom}(\varphi_{u(y)}) = \begin{cases} \{g(u(y))\} & \text{if } y \in A \\ \emptyset & \text{otherwise.} \end{cases}$$

First, define a function $f(e, y, x)$ by

$$f(e, y, x) =_{\mathrm{df}} \begin{cases} 0 & \text{if } x = g(e) \text{ and } y \in A \\ \uparrow & \text{otherwise.} \end{cases}$$

Clearly, f is computable. By the second recursion theorem with parameters, there is a computable total $u : \mathbb{N} \to \mathbb{N}$ satisfying

$$\varphi_{u(y)} = f(u(y), y, x)$$

for all $y \in \mathbb{N}$. It is easy to verify that u has the desired properties.

Exercise 3.33. Let $g : \mathbb{N} \to \mathbb{N}$ be a computable total function and let $A \subseteq \mathbb{N}$ be decidable. Prove that there is a computable total $u : \mathbb{N} \to \mathbb{N}$ such that

$$\text{range}(\varphi_{u(y)}) = \begin{cases} \{g(u(y))\} & \text{if } y \in A \\ \emptyset & \text{otherwise.} \end{cases}$$

Exercise 3.34. Let $A \subseteq \mathbb{N}$ be decidable. Prove that there is a computable total $u : \mathbb{N} \to \mathbb{N}$ such that

$$\varphi_{u(y)}(x) = \begin{cases} \varphi_x(u(y)) & \text{if } y \in A \\ \uparrow & \text{otherwise.} \end{cases}$$

3.7 Additional exercises

Exercise 3.35. Let P be the URM-program

$$T(0,1), S(0), S(2), J(1,2,5), J(0,0,1).$$

(a) Carry out the computation $P(2,0,0,\dots)$.

(b) Determine a formula for the partial function $f_P^{(1)}$.

Exercise 3.36. Let P be the URM-program $S(1), S(1), J(0,1,4), J(0,0,0)$. Let $f_P^{(1)}$ be the unary function computed by P. Determine for which $x \in \mathbb{N}$ we have $f_P^{(1)}(x)\downarrow$. Justify your answer.

Exercise 3.37. Determine whether each of the following statements about URM-programs is true or false. If the statement is true, then provide a sketch of a proof justifying its truth. If the statement is false, provide a counterexample that justifies its falsity.

(a) If the last instruction of P is $J(0,0,0)$, then $f_P^{(n)}$ never halts on any input.

(b) For any program P, any positive integers n, k, and any $x_0, \dots, x_{n-1}, y \in \mathbb{N}$,

$$f_P^{(n+k)}(x_0, \dots, x_{n-1}, 0, \dots, 0)\downarrow = y \iff f_P^{(n)}(x_0, \dots, x_{n-1})\downarrow = y.$$

(c) For any program P, there is a program $P' \neq P$ such that $f_P^{(1)} = f_{P'}^{(1)}$.

Exercise 3.38. Let P be the URM program $J(0,1,3), S(0), J(0,0,0)$. Does $f_P^{(2)}(1,3)\downarrow$? Justify your answer.

The next problem contains applications of the s-m-n theorem and the second recursion theorem. It is not really a significant application, but it demonstrates the sort of fascinating oddities that the second recursion theorem can produce.

Exercise 3.39. (a) For any $e \in \mathbb{N}$, let W_e be the domain of $\varphi_e^{(1)}$. Show that there is a computable total function $u : \mathbb{N} \to \mathbb{N}$ such that

$$W_{u(x)} = \{y \in \mathbb{N} : x \text{ divides } y\}.$$

Hint. Use part (a) and the s-m-n theorem.

(b) Prove that there is $e^* \in \mathbb{N}$ such that

$$W_{e^*} = \{y \in \mathbb{N} : y \text{ divides } y\}.$$

Exercise 3.40. For any $e \in \mathbb{N}$, let R_e denote the range of $\varphi_e^{(1)}$,

$$R_e = \{y \in \mathbb{N} : (\exists x)\ \varphi_e^{(1)}(x)\downarrow = y\}$$

(a) Prove that there is a computable total function $S : \mathbb{N} \to \mathbb{N}$ such that for every $x \in \mathbb{N}$,

$$R_{S(x)} = \{y \in \mathbb{N} : y \text{ is a power of } x\}.$$

(b) Prove that there is $x^* \in \mathbb{N}$ such that

$$R_{x^*} = \{y \in \mathbb{N} : y \text{ is a power of } x^*\}.$$

Exercise 3.41. Prove that for any computable partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$, there is an *injective* computable total function $u : \mathbb{N} \to \mathbb{N}$ such that

$$f(x)\downarrow \implies \varphi_{u(x)} = \varphi_{f(x)}$$

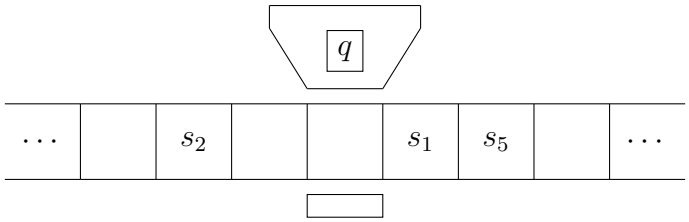
for every $x \in \mathbb{N}$. *Hint.* Use the s-m-n theorem, including the part that the function it provides may be assumed to be injective.

4 Turing machines

In this chapter, we will study yet another model of computation: the Turing machine. This concept was invented by Alan Turing and is one of the most commonly used models of computation. Also, we will discuss the Church-Turing Thesis, a philosophical (well-supported) claim about the nature of computable functions.

4.1 Turing computable partial function

A Turing machine M is composed of several parts and looks like

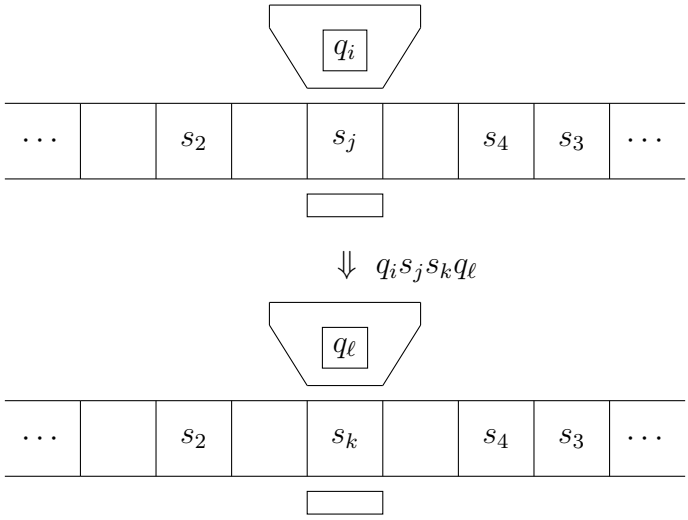


The tape is infinite in both direction, segmented into infinitely many squares. Each square can be blank or contain a symbol from the alphabet of the machine. The alphabet is a fixed, finite collection of symbols $s_0, \ldots, s_{n-1}$. Sometimes, We denote being blank by the symbol B. The machine has a *reading head* which at any given time is *reading* a particular square. Also, at any given time the head is in a *state* q. There are finitely many possible states $\{q_0, \ldots, q_{m-1}\}$.

The machine is capable of three operations:

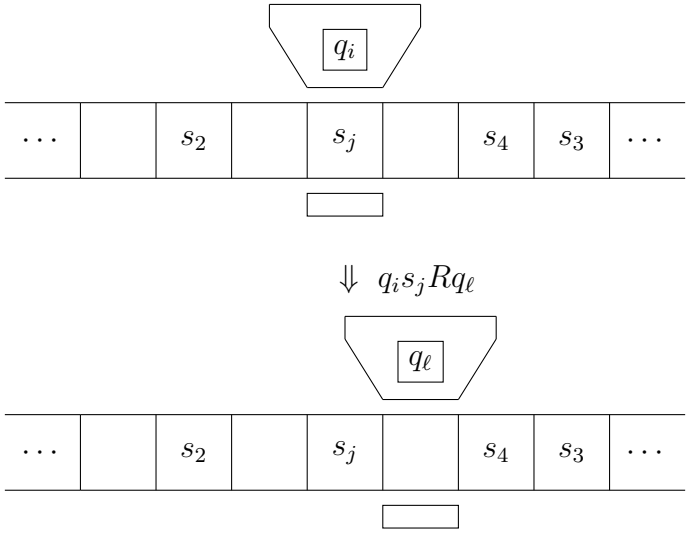
(1) If the machine is in state q_i and reading s_j, then the machine can erase the symbol s_j, write the symbol s_k, and change to state q_ℓ.

This instruction is abbreviated by $q_i s_j s_k q_\ell$, and its execution looks like:



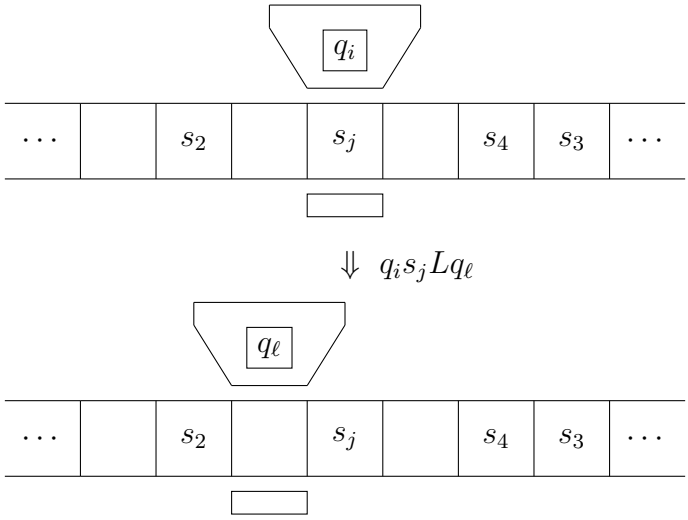
(2) If the machine is in state q_i and reading s_j, then the head can move one square to the *right* and change to state q_ℓ.

This instruction is abbreviated by $q_i s_j R q_\ell$, and its execution looks like:



(3) If the machine is in state q_i and reading s_j, then the head can move one square to the *left* and change to state q_ℓ.

This instruction is abbreviated by $q_i s_j L q_\ell$, and its execution looks like:

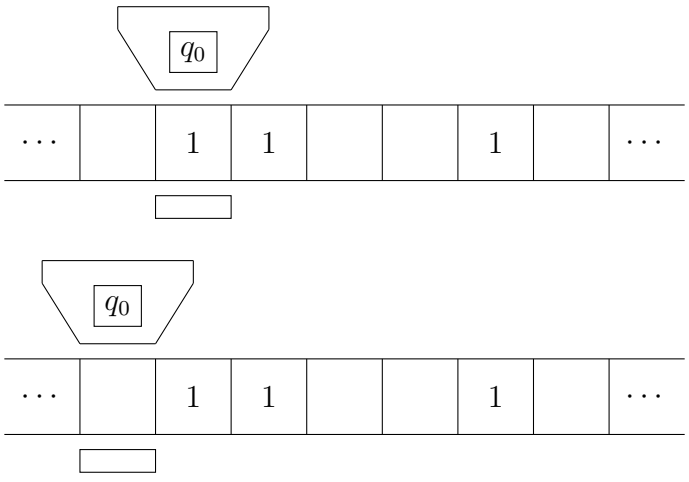


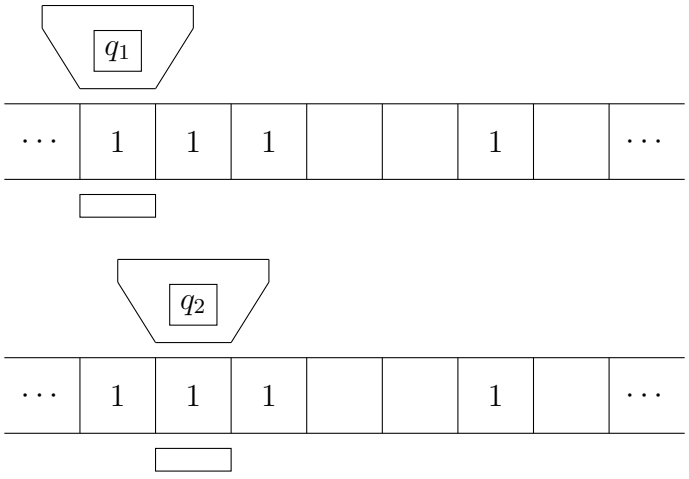
Definition 4.1. A *Turing machine program* is a finite list of instructions such that for any pair $q_i s_j$ (where q_i is a state and s_j is a symbol), there is at most one instruction of the form $q_i s_j \alpha q_\ell$.

Example 4.2. Consider a Turing machine with alphabet $\{1\}$ (and blank). Let P be the program

$q_0 1 L q_0$
$q_0 B 1 q_1$
$q_1 1 R q_2$.

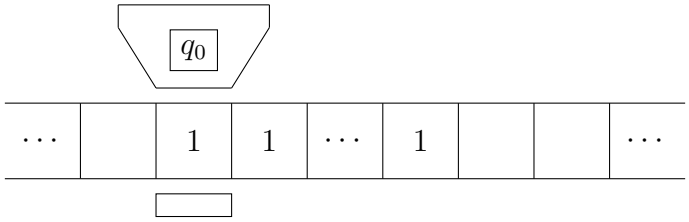
Then, we compute with initial state





Definition 4.3. If the Turing machine is running program P, in state q_i, and reading symbol s_j, and there is no instruction of the form $q_i s_j \alpha q_\ell$, then we say the machine is in a *halting state*.

Suppose our alphabet consists of only the symbol '1'. Let P be a Turing machine program. The partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ computed by P is defined as follows. To compute $f(x)$, start with initial state



with $x + 1$ many '1's; then, run program P, and if the machine halts on this input then,

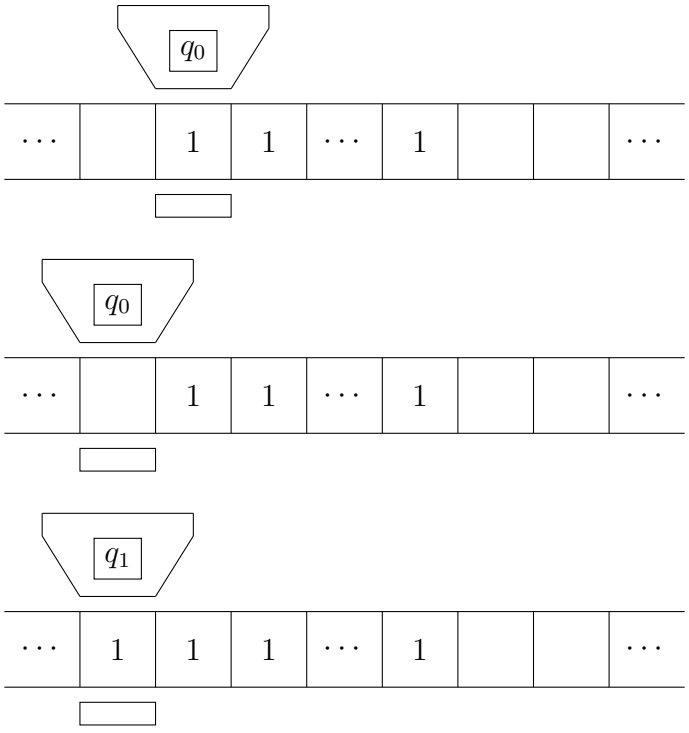
$$f(x)\downarrow = \text{the number of '1's on the tape in the halting state}$$

Definition 4.4. We say a partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is *Turing computable* if there is a Turing machine and a program which computes it.

Example 4.5. The function $x + 2$ is Turing computable. Indeed, it is computed by the Turing machine program

$q_0 1 L q_1$
$q_1 B 1 q_1$

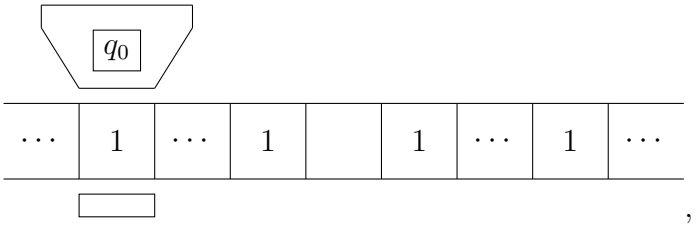
To see that this program works, just consider the computation:



For impute x, we would have started with $x + 1$ many '1's. The program added a single 1 to the left and then halted. So, we end with $x + 2$ many '1's on the tape, so the output is $x + 2$.

Exercise 4.6. Show that the predecessor function is Turing computable.

The definition of Turing computability generalizes to n-ary partial functions. For example, for a function $f : \mathbb{N}^2 \rightharpoonup \mathbb{N}$ on input $(x, y) \in \mathbb{N}$, start the machine in state



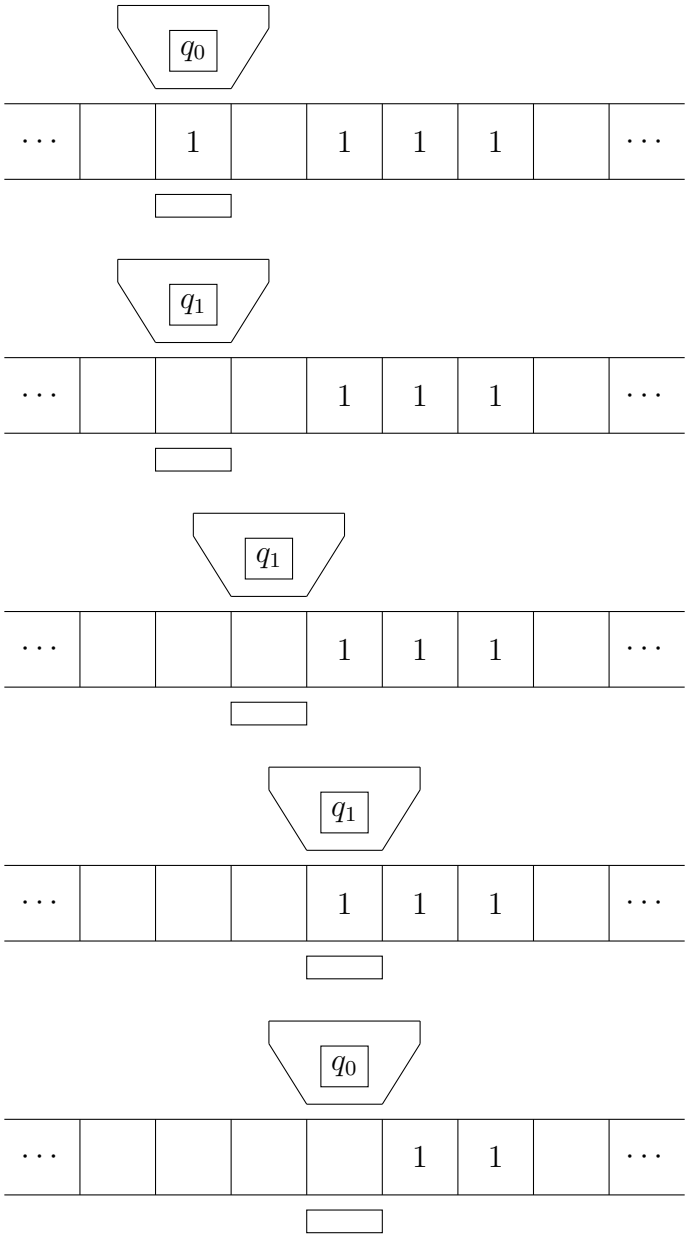
,

where the first string has $(x + 1)$-many '1's and the second string has $(y + 1)$-many '1's.

Example 4.7. The function $f(x, y) = x + y$ is Turing computable. Indeed, it is computed by the program

$$q_0 1 B q_1, \quad q_1 B R q_1, \quad q_1 1 B q_0$$

To convince ourselves that this works, we consider the computation of $0+2$:



Since the tape ends with two '1's, the output is 2.

Exercise 4.8. Verify that the program works for $2 + 0$ and $1 + 1$.

4.2 The Church-Turing Thesis

We have defined three classes of partial functions:

(1) Recursive partial functions. We denote this class by $\mathcal{R}$.

(2) URM-computable partial functions. We denote this class by $\mathcal{U}$.

(3) Turing computable partial functions. We denote this class by $\mathcal{T}$.

We have sketched the proof that $\mathcal{R} = \mathcal{U}$, but it turns out that $\mathcal{T}$ is also the same class of functions.

Theorem 4.9. A partial function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is recursive iff f is URM-computable iff f is Turing computable. In other words, $\mathcal{R} = \mathcal{U} = \mathcal{T}$.

We will not prove or sketch the proof that the Turing computable partial functions are the same as the recursive partial functions. The method of the proof is the same as the method we used to prove $\mathcal{R} = \mathcal{U}$.

Even though these definitions are all equivalent, they are all still useful in different contexts. For example,

(1) The "recursive partial function" definition is perhaps the most mathematically elegant. It can be used to develop the theory without programming any machines. Also, it more easily generalizes to "higher recursion theories" on objects more complicated than the natural numbers.

(2) The definitions through theoretical machines connect the theory more closely to our intuitive notion of computability and to computer science. The way register machines operate is fairly close to low level computation in actual computers, and the programming of register machines is perhaps more intuitive than Turing machine programming.

(3) Turing machines are often used in *complexity theory*, where the complexity of algorithms is studied. It is relatively natural to count how many steps a Turing machines takes to execute an algorithm. Also, Turing machines operate very well in binary, which more closely models actual computers.

There are yet still other ways to define the class of computable partial functions, including Church's lambda calculus and other types of theoretical machines like multi-tape Turing machines. The ideas of the lambda calculus have been very influential in computer science.

In the early 20th century, many mathematicians were searching for the proper way to rigorously define the intuitive notation of an "effectively computable" function. Alonzo Church developed his lambda calculus, Godel discovered the notion of recursive function (which was later extensively studied by Stephen Kleene), and Alan Turing developed the notion of the Turing machine.

It is an amazing fact that all these seemingly different notions ended up defining the same class of functions. This, in part, gave rise to the following important philosophical claim:

Church-Turing Thesis. The intuitively and informally defined class of effectively computable partial functions is exactly the class of Turing computable partial functions.

The Church-Turing Thesis is **not** a theorem of mathematics; however, it is not a "definition by stipulation", where we merely agree by convention. It is a philosophical claim with strong evidence that has stood for nearly a century. There have been no serious challenges to it, despite the efforts of many.

Even though it has a philosophical nature, the thesis does make falsifiable claims. If we prove a function f is not Turing computable, then according to the Thesis no one will ever devise an algorithm that can compute f.

We will summarize some of the main supporting arguments for the Church-Turing thesis:

Main arguments supporting Church-Turing Thesis:

(1) Turing's original analysis in his famous paper, *On Computable Numbers, with an Application to the Entscheidungsproblem* (1936). This is a classic argument that has only been further supported by our modern experience with digital computing.

(2) The fact that the class of recursive partial functions has so many seemingly different, but equivalent, definitions demonstrates that recursive partial functions form a very natural and, in some sense, "unavoidable" class of functions.

(3) The great wealth of recursive partial functions, and the very strong closure properties of this class developed by computability theory.

(4) The experience of more than 80 years with digital computing, and the failure to come up with any plausible counterexamples to the thesis.

We end the section with some remarks on so-called "proofs by Church-Turing thesis". Even though the Church-Turing thesis is not a theorem, it is often used (informally) in proofs to justify that a function is computable. It is indeed customary to claim that "f is computable, since we have given intuitive instructions for computing it", but what is always meant by this is "f is computable, but I do not want to take the time to prove this in detail because it is boring and routine (to someone who is experienced in the subject and who has understood the justification of the Church-Turing thesis)".

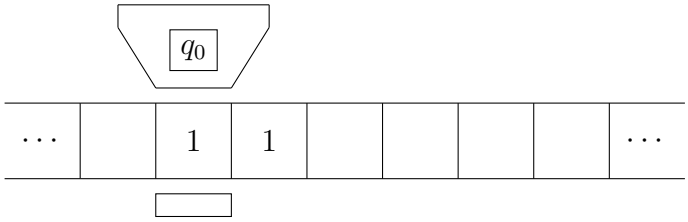
However, since the reader is not a seasoned computability theorist, we will avoid this type of argument. It is important to practice formally proving that functions are computable, and an introductory class is the right venue to get this experience. There are many applications of computability that are far too complicated and unintuitive to rely only on Church-Turing, e.g., finite and infinite injury priority arguments, effective transfinite recursion, etc.

4.3 Additional exercises

Exercise 4.10. Consider a Turing machine with state space $\{q_0, q_1, q_2, q_3\}$ and alphabet $\{1\}$ (and blank, of course, denoted by B). Let P be the following Turing machine program:

$q_0 1 R q_0$
$q_0 B 1 q_1$
$q_1 1 R q_2$
$q_2 B 1 q_3$

(a) Carry out, showing all the steps, the computation of the Turing machine starting with initial state



(b) What unary function $f : \mathbb{N} \to \mathbb{N}$ does this program compute? Justify your answer with a reasoned argument.

Exercise 4.11. For each of the following functions, write a Turing machine program (for a machine with a finite state space and with alphabet consisting of 1 and blank B) that computes the given function.

(a) $x \dot{-} 1$.

(b) The zero function $c_0^1 : \mathbb{N} \to \mathbb{N}$, $c_0^1(x) = 0$.

(c) The projection function $\pi_1^2 : \mathbb{N} \to \mathbb{N}$, $\pi_1^2(x_0, x_1) = x_1$.

You do not have to write a formal proof that your program computes the function, but give a reasoned argument for why the program works.

Part II
Exploring the Uncomputable

In this second part, we will turn our attention to partial functions which are not computable and relations that are not decidable. We have several goals in mind:

(1) Develop methods to formally prove partial functions are not computable and relations are not decidable. We will begin pursuing this goal in Section 5.

(2) Understand relations which are, in some sense, not too far from being computable. What we have in mind are the semirecursive relations and they will be studied in Section 6.

(3) Develop tools to compare uncomputable objects to each other. If two objects are uncomputable, can we compare how complicated they are relative to each other? There are many notions that allow us to make these sorts of comparisons (many-one reducibility and Turing reducibility being notable examples) and these notions will be studied in Sections 7.1 and 8.

(4) Finally, in Section 9 we will study the arithmetical hierarchy of relations; these relations get quite far from being computable, but are definable in the language of arithmetic. This definability connects them closely to the theory of (relative) computability.

Before we dive in, we first take a moment to summarize the important properties of computable partial functions which we have so far established.

For every integer $n > 0$, there is an enumeration (with repititions)

$$\varphi_0^{(n)}, \varphi_1^{(n)}, \varphi_2^{(n)}, \dots$$

of n-ary partial functions such that:

(1) the $\varphi_e^{(n)} : \mathbb{N}^n \rightharpoonup \mathbb{N}$ are exactly the n-ary computable partial functions;

(2) (existence of universal machine) the partial function $\psi_U^{(n)} : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ defined by
$$\psi_U^{(n)}(e, \vec{x}) = \varphi_e^{(n)}(\vec{x})$$
is computable; and

(3) (s-m-n theorem) for every computable partial $f : \mathbb{N}^{m+n} \rightharpoonup \mathbb{N}$, there is a computable total $S : \mathbb{N}^m \to \mathbb{N}$ such that

$$f(\vec{x}, \vec{y}) = \varphi^{(n)}_{S(\vec{x})}(\vec{y})$$

for all $\vec{x} \in \mathbb{N}^m$ and $\vec{y} \in \mathbb{N}^n$.

The condition in item (2) is described informally in many ways. As indicated above, sometimes people describe (2) as saying that there is a universal machine. Alternatively, people say that (2) means that $\varphi^{(n)}_0, \varphi^{(n)}_1, \ldots$ is an *effective* enumeration. Finally, people express (2) by saying that the sequence $\{\varphi^{(n)}_e\}$ is computable *uniformly in* e.

5 Undecidable relations

In this chapter, we will formally prove the uncomputability of functions and relations. In particular, we will study some famous and important undecidable problems: the halting problem, index problems, and the Wang tiling problem.

5.1 The halting problem

The most famous (and perhaps the most fundamental) undecidable problem is the halting problem. The halting problem actually comes in various forms.

The *(diagonal) halting problem* is the unary relation

$$K(e) \iff_{\text{df}} \varphi_e(e)\downarrow$$

The halting problem will be our first example of an *undecidable* relation. Of course, this will also give us our first example of a non-computable (total) function, namely the characteristic function of K.

Theorem 5.1. The halting problem K is undecidable, i.e., the characteristic function Char_K of K is not computable.

Proof. Suppose towards a contradiction that K is decidable. Then, define a total function $f : \mathbb{N} \to \mathbb{N}$ by

$$f(e) = \begin{cases} 1 \dot{-} \varphi_e(e) & \text{if } K(e) \\ 0 & \text{if } \neg K(e). \end{cases}$$

Since $K(e)$ is decidable, then f is computable by Theorem 3.21. It is also easy to check that f is total.

Since f is computable, there exists $e_0 \in \mathbb{N}$ such that $f = \varphi_{e_0}$. Since f is total, $K(e_0)$ holds, so that $\varphi_{e_0}(e_0)\downarrow$. By the definition of f, we have

$$\varphi_{e_0}(e_0) = f(e_0) = 1 \dot{-} \varphi_{e_0}(e_0),$$

which is a contradiction. □

The halting problem is the most fundamental of the undecidable problems. It actually comes in several different variants and below we will discuss why they are all undecidable.

(1) **The full halting problem is undecidable.** The relation K,

$$K(e) \iff_{\text{df}} \varphi_e(e)\downarrow$$

is the *diagonal* halting problem because the computable partial function with code e is given input e. In the theorem above, we worked with the diagonal halting problem because it is the version on which the method of proof works best. In fact, the method of proof is called the *diagonalization method*, and it is useful throughout computability theory. For example, one can use diagonalization for the following exercise.

Exercise 5.2. Prove that there is no effective enumeration of the total computable unary functions, i.e., prove that there does not exist a sequence

$$\theta_0, \theta_1, \ldots, \qquad \theta_e : \mathbb{N} \to \mathbb{N}$$

such that (1) every computable unary total $f : \mathbb{N} \to \mathbb{N}$ is equal to some θ_e; and (2) the function $g(e, x) = \theta_e(x)$ is computable. *Hint.* Use a strategy similar to the one used in the proof that the halting problem is undecidable.

Naturally, one wonders if the *full* halting problem,

$$H(e, x) \quad \iff_{\text{df}} \quad \varphi_e(x)\!\downarrow$$

is undecidable. Indeed, it is undecidable, and this fact easily follows from the undecidability of K. In fact, K is a special case of H,

$$K(e) \iff H(e, e).$$

Thus, if we had a decision procedure for H, we would immediately have a decision procedure for K: if you want to know if $K(e)$ holds, just run your decision procedure for $H(e, e)$. Thus, if H is decidable, then K must also be decidable. Since K is not decidable, we conclude that H is also undecidable.

(2) **The halting problems for higher arities are undecidable.** The relations K and H above only concern unary computable partial functions. However, if we allow the arity to be any $n \geq 1$, the corresponding haling problem,

$$H^{(n)}(e, \vec{x}) \quad \iff_{\text{df}} \quad \varphi_e^{(n)}(\vec{x})$$

is undecidable.

To prove this, we can use facts about our particular definition of $\varphi_e^{(n)}$. For any e, $\varphi_e^{(n)}$ is, by definition, the URM-computable partial function computed by the URM-program with code e. Recall that for any $x \in \mathbb{N}$,

$$\varphi_e(x) = \varphi_e^{(n)}(x, 0, \ldots, 0)$$

because both URM-computations start with the initial state $x, 0, 0, \ldots$ and run the program coded by e. Thus, H (the full halting problem for unary computable partial functions) is a special case of $H^{(n)}$,

$$H(e,x) \iff \varphi_e(x)\downarrow \iff \varphi_e^{(n)}(x,0,\ldots,0)\downarrow \iff H^{(n)}(x,0,\ldots,0).$$

Thus, using similar reasoning as above, if $H^{(n)}$ is decidable, then H is decidable. We conclude that $H^{(n)}$ is undecidable.

(3) **Different effective enumerations have undecidable halting problems.** This last item addresses a foundational concern, but will not have practical importance going forward. For this discussion, we will focus our attention on unary partial functions, but this is not necessary. Recall that we used codes of URM-programs to define our effective enumeration

$$\varphi_0, \varphi_1, \ldots, \varphi_e, \ldots$$

of computable partial functions. This was done in a way so that this enumeration has a computable universal function and satisfies the s-m-n theorem. This particular enumeration is not the only way to do this, however. For instance, we could have coded Turing machine programs and defined ψ_e to be the unary partial function computed by the Turing machine program with code e. If done carefully, this we yield an enumeration

$$\psi_0, \psi_1, \ldots, \psi_e, \ldots$$

of all unary computable partial function which also has a computable universal machine and satisfies the s-m-n theorem. But, if one examines the proof of Theorem 5.1, it is not difficult to see that these properties are the only ones needed to make the proof of undecidability of the halting problem work. Thus, the halting problem for this enumeration,

$$\widetilde{K}(e) \quad \iff_{\text{df}} \quad \psi_e(e)\downarrow$$

is undecidable.

Generally, the particularities of our effective enumeration φ_e do not affect the computability theory we develop. Going forward, we will stick with our φ_e and not worry about other effective enumerations like the ψ_e described above.

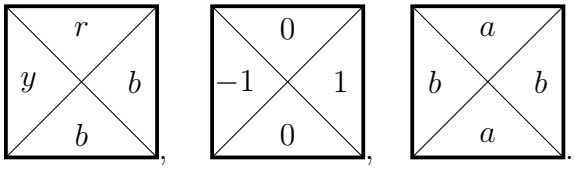
In items (1) and (2) above, we showed problems are undecidable by *reducing* the (diagonal) halting problem K to them, i.e., showing that K is a special case of the problem. This type of method will be much expanded upon in Chapter

??. However, we will see another example of this method in the next section. We will see how to prove a natural combinatorial problem is undecidable by showing that, if we could decide the combinatorial problem, then we could decide the halting problem.

5.2 Wang tilings

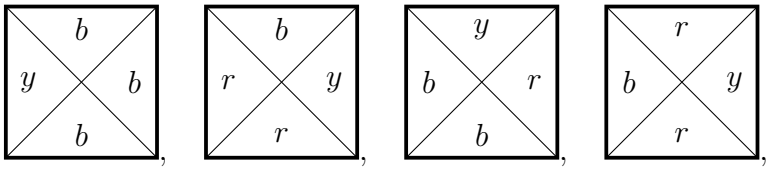
In this section, we will give another famous example of an undecidable problem. The problem is combinatorial in nature and has to do with deciding if we can cover certain regions with tiles, called Wang tiles, according to particular rules. Unlike the halting problem, the Wang tiling problem, on its face, is not about computability theory; thus, it is an example of a "natural" mathematical problem that happens to be undecidable.

A *Wang tile* is a square with a label given to each of its four sides. Each of the following are examples of Wang tiles:

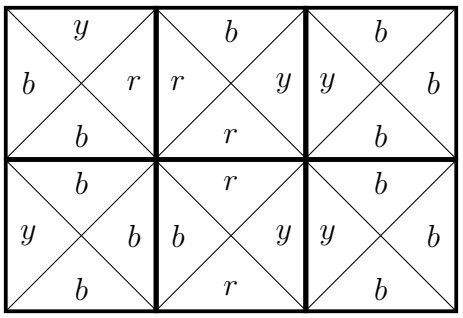


Let T be a set of Wang tiles, let A be a rectangular region in the plane. A *valid tiling using tiles from* T of the region A is a placement of copies of tiles from T in a way that the tiles cover exacly area A, sides of the tiles line up precisely so that the tiles form a rectangular grid, and any two adjacent tiles have the same labels on the sides at which they touch. Note that by "rectangular region" we include regions with infinite area like the whole plane $\mathbb{R}^2$ and the upper-right quadrant $\{(x, y) \in \mathbb{R}^2 : x, y \geq 0\}$. If there exists a valid tiling of A using tiles from T, then we say T *admits a valid tiling of* A.

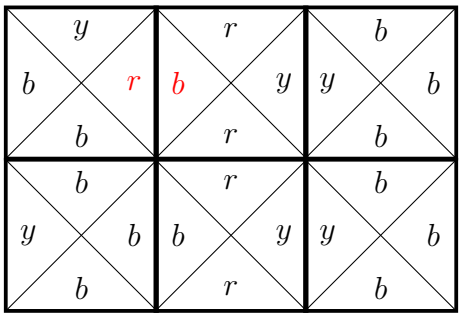
For example, if T consists exactly of the tiles



then



is a valid tiling of a 2×3 rectangular region. On the other hand,



is not a valid tiling, since the labels highlighted in red do not match.

Wang tiling problem. Fix a rectangular region A in the plane. Given a finite tile set T, does T admit a valid tiling of A?

A slight variant of this problem that we will focus on is the following:

Tiling with restricted origin problem. Fix a rectangular region A in the plane. Given a finite tile set T and a distinguished tile $t^* \in T$, does T admit a valid tiling of A which uses t^* at least once?

We will show that the restricted origin problem is undecidable for the region $A = \mathbb{R}^2$. This result can then be used to show that Wang tiling problem (without a restricted origin) is also undecidable for $A = \mathbb{R}^2$ using some interesting tiling tricks, but we will focus here on the restricted origin problem.

We will show that the restricted origin problem is undecidable by showing that, if we could decide it, then we could also decide the following variant of the halting problem:

Blank tape halting problem. The problem of deciding whether a Turing machine program with code e halts when run on a tape of all blanks is undecidable.

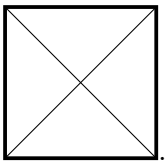
This halting problem is undecidable, and one can use methods outlined in the previous section to prove its undecidability. Thus, once we reduce this halting problem to the restricted origin tiling problem, we will have proved that the restricted origina tiling problem is undecidable.

Let P be a program for a Turing machine with alphabet $\{1\}$ and state space $\{q_0, \ldots, q_{n-1}\}$, where q_0 is the initial state. We will *computably* define a finite tiling set T_P with a distinguished tile t^* such that

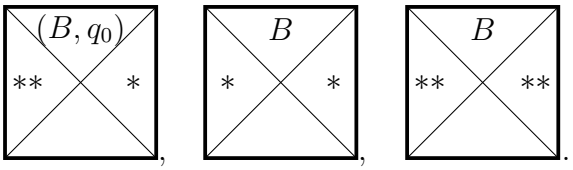
$$P \text{ halts when run on an initial tape of all blanks } \iff T_P \text{ does not admit a valid tiling of } \mathbb{R}^2 \text{ with origin } t^*. \quad (5.1)$$

Thus, we cannot decide whether tiling with a given origin admit valid tilings of $\mathbb{R}^2$, since otherwise we would be able to decide if Turing programs halt on an initial blank tape. There is some work to be done with codes and showing that the function $P \mapsto T_P$ is computable (in the codes), but they are straightforward and we will choose to focus on the more conceputally difficult aspects.

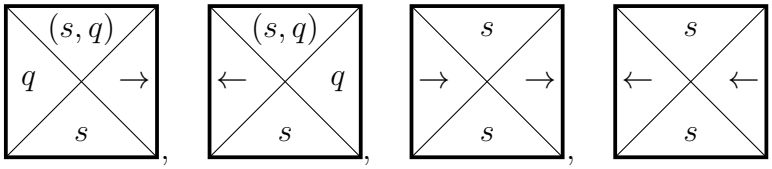
We form the finite tiling set T_P as follows. First put in the "filler tile"



Next, put in the *starting tiles*

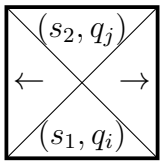


Note that here, "B" is a label for Wang tiles distinct from the blank label for Wang tiles. For any $s \in \{1, B\}$ and state q, we place the *transition tiles*

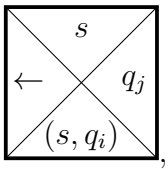


into T_P. The remaining tiles are called *action tiles*, and which of them appear

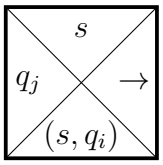
in T_P depend on the instructions found in P. For instructions of the form $q_i s_1 s_2 q_j$ in P, we place



into T_P. For an instruction $q_i s D q_j$, we place into T_P the tiles

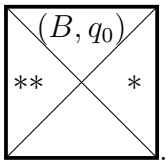


, if $D = R$,



, if $D = L$.

The distinguished tile t^* is the starting tile

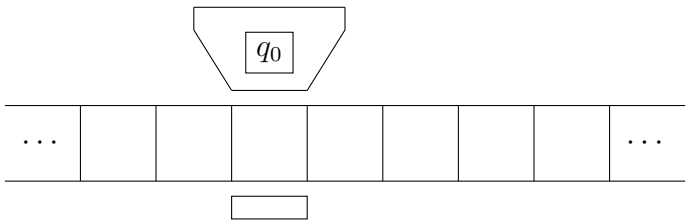


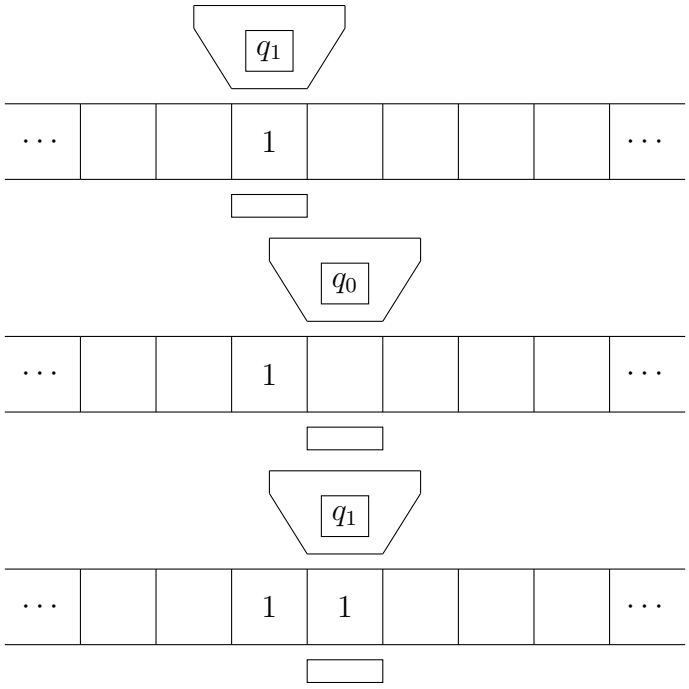
.

We say that an infinite row of tiles *codes a machine state* if their adjacent sides have matching labels and the top labels are all labeled with elements of $\{1, B\}$, except exactly one top label is of the form (s, q) for some $s \in \{1, B\}$ and state q. We will see in the following example how we can use tilings from T_P to code Turing machines computations.

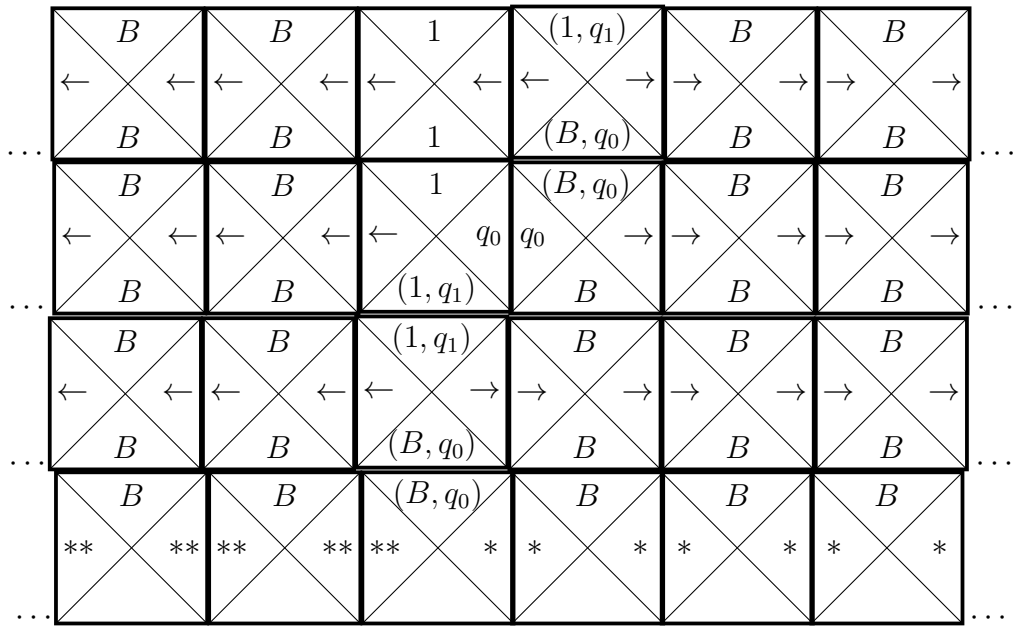
Example 5.3. Consider the program P:
$q_0 B 1 q_1$
$q_1 1 R q_0$

When started on a tape of all blanks, the first four steps of the computation using program P are:





Next, we will see that we can encode these machine states and the transitions between them using the tiles from T_P:



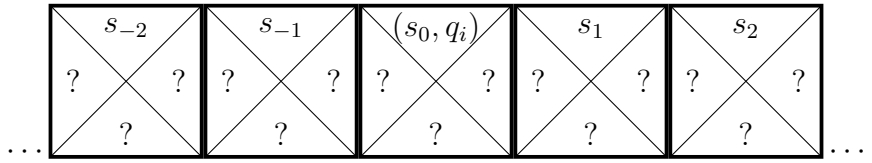
Note that each row of this tiling codes a machine state from our compu-

tation on the previous page: the bottom row codes the initial state of the machine, the second-to-bottom row codes the second state of the machine, etc.

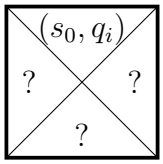
The idea here is that if this computation does not halt, then the computation has infinitely many steps and we can continue the above tiling indefinitely. On the other hand, if the computation were to halt, then we could not continue the tiling in a valid manner.

Lemma 5.4. Suppose we have an infinite row of tiles which codes a machine state. Then, there is a way to validly extend the tiling one row up exactly when there is an instruction in P that can be applied to the coded machine state. Moreover, this extension is unique and the new row codes the machine state that results from executing the instruction.

Proof. Suppose our row is of the form



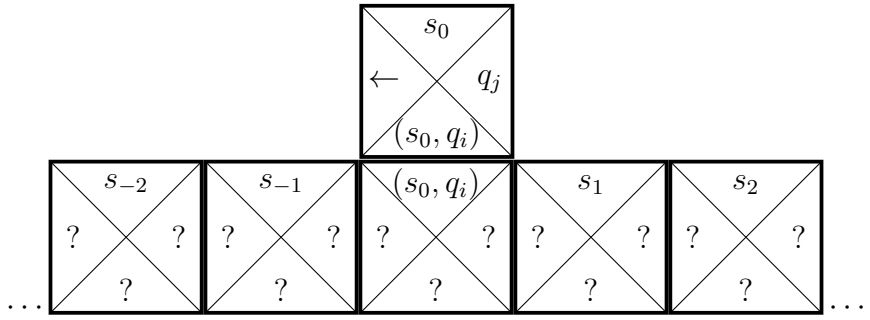
The only possible tiles that can go above the head tile



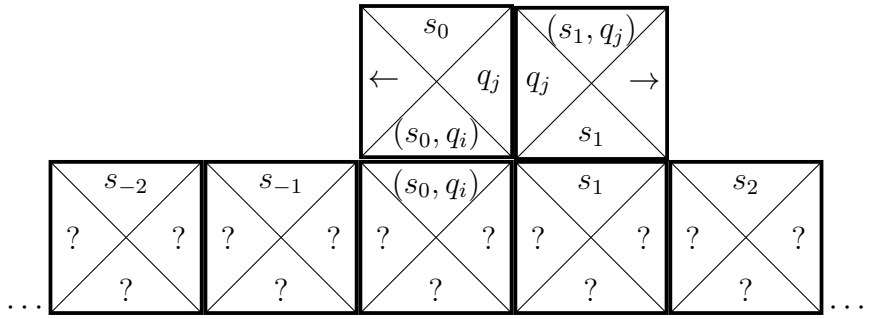
are action tiles. There is an action tile with bottom label (s_0, q_i) if and only if P has an executable instruction for the coded machine state.

Suppose there is a valid instruction. Since P has no conflicting instructions, there is exactly one such instruction. We must take cases on what kind of instruction.

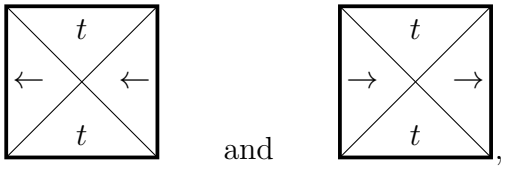
Case 1, the instruction is of the form $q_i s_0 R q_j$, so we can extend the tiling with an action tile as follows



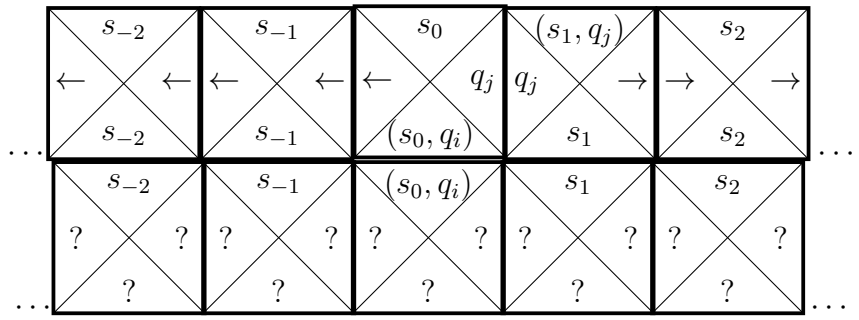
For the space to the right of our new action tile, we must place a tile whose left label is q_j and whose bottom label is s_1. But a quick look at our tile set shows that there is only one such tile. So the only way to extend the tiling is



Finally, the only tiles with with $\leftarrow$ as their right labels and the only tiles with $\rightarrow$ as their left label are the transition tiles



respectively. Thus, an easy induction shows that the unique way to extend the tiling to the infinite row is

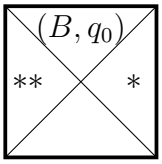


It is clear that this new row codes the machine state that results from applying the instruction.

The other cases are handled similarly. □

Exercise 5.5. Finish the proof of the lemma in the cases where the next instruction is of the form $q_i s_0 L q_j$, $q_i s_0 t q_j$.

Now, we are ready to establish the reduction (5.1). Consider a program P which does not halt when started with a tape of all blanks. We will show that T_P with restricted origin tile

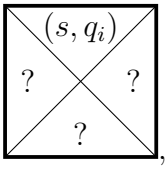


admits a valid tiling of the plane. Starting from the origin tile, there is clearly only one way to tile the row with the origin since only the starting tiles have $*$ and $**$ as labels. We can use the filler tiles to validly tile everything below the origin row.

Now, the origin row codes the machine state starting with all blanks on the tape. Since the program does not halt on this state, our lemma implies that we can validly tile the next row up, and that new row will code the next machine state that results from the program. Continuing inductively, we can always validly tile the next row up since the computation never halts. Thus, we have a valid tiling of the entire plane.

Next, consider a program P which does halt. We show that we cannot tile the whole plane using T_P and its restricted origin. From the origin tile, again we are forced to tile that row so that it codes the initial state of all blanks. Continuing with the lemma as before, the best we can do is to apply instructions from the program and get rows of tiles that code the machines

states that result. Thus, eventually we will get to a row that codes a halting state. So, that row's "head tile" is of the form



,

where there is no valid instruction for (s, q_i). But this implies there is not tile with bottom label (s, q_i), and so there is no way to continue the tiling. Thus, we have established the reduction 5.1.

5.3 Index sets and Rice's Theorem

Index sets provide a rich source of examples of undecidable relations. To explain what an index set is, we start with describing an example. Define the set $\text{Tot} \subseteq \mathbb{N}$ by

$$e \in \text{Tot} \quad \iff_{\text{df}} \quad \varphi_e \text{ is total.}$$

Tot is called an index set because for any $e, e' \in \mathbb{N}$, if $\varphi_e = \varphi_{e'}$, then e, e' are either both in Tot or both outside of Tot. In this way, Tot is really describing a property about the computable partial function coded by e, rather than describing anything about e as just a number.

Definition 5.6. An *index set* is a set $A \subseteq \mathbb{N}$ such that for any $e, e' \in \mathbb{N}$,

$$\text{if } \varphi_e = \varphi_{e'}, \text{ then } \Big(e \in A \iff e' \in A\Big)$$

We can think of index sets in a different way. Start with a set $\mathcal{A}$ of unary computable partial functions. (The members of $\mathcal{A}$ are actual partial functions – not their codes.) Then, define

$$\mathcal{A}_{\text{codes}} =_{\text{df}} \{e \in \mathbb{N} : \varphi_e \in \mathcal{A}\}$$

Then $\mathcal{A}_{\text{codes}}$ is an index set; in fact, every index set can be obtained this way, which is established in the following exercise.

Exercise 5.7. Let $A \subseteq \mathbb{N}$ be an index set. Define $\mathcal{A} =_{\text{df}} \{\varphi_e : e \in A\}$. Prove that $\mathcal{A}_{\text{codes}} = A$.

Decidable relations, by definition, are always on natural numbers, but it is natural to ask about the decidability of determining whether a particular

computable partial function has a given property, e.g., whether a particular partial function is total. One way to incorporate such a question into the subject of computability theory is to ask whether the corresponding index set is decidable.

Tot is an index set. There are many other examples. We give a few:

(1) $R(e) \iff_{\text{df}} \varphi_e = \emptyset \iff (\forall x)\ \varphi_e(x)\!\uparrow$

(2) $Q(e) \iff_{\text{df}} \varphi_e$ is onto $\mathbb{N}$

(3) $P(e) \iff_{\text{df}} \varphi_e$ is constant on its domain

An important non-example is the diagonal halting problem,

$$K(e) \iff_{\text{df}} \varphi_e(e)\!\downarrow$$

Intuitively, K fails to be an index set because the relation uses e not only as a program code, but also e is used as an input into that program. Proving K is not an index set takes a little bit of work, but it is a useful exercise.

Exercise 5.8. Prove K is not an index set.

Hint. Use the second recursion theorem to create a φ_{e^*} which only converges on e^*. Then, use the fact that any computable partial function has many different programs that compute it.

It turns out that almost no index sets are decidable. The only ones that are decidable are the trivial ones, $\emptyset$ and $\mathbb{N}$.

Theorem 5.9 (Rice's theorem)**.** Let A be an index set with $\emptyset \subsetneq A \subsetneq \mathbb{N}$. Then A is undecidable.

Proof. Assume towards a contradiction that A is decidable. Since $\emptyset \subsetneq A \subsetneq \mathbb{N}$, we can pick e_0, e_1 with $e_0 \in A$ and $e_1 \notin A$. Let $f = \varphi_{e_0}$ and $g = \varphi_{e_1}$. Now, define

$$h(e, x) =_{\text{df}} \begin{cases} g(x) & \text{if } e \in A \\ f(x) & \text{if } e \notin A, \end{cases}$$

which is computable by Theorem 3.21. By the second recursion theorem, there is e^* with $\varphi_{e^*}(x) = h(e^*, x)$. If $e^* \in A$, then $\varphi_{e^*} = g = \varphi_{e_1}$. But since $e_1 \notin A$ and A is an index set, it follows that $e^* \notin A$, which is a contradiction. We get a similar contradiction if $e^* \notin A$. □

This immediately tells us that all sorts of problems are undecidable, including examples (1)-(3) above.

We can strengthen Rice's Theorem in the following exercise.

Exercise 5.10. Say an n-ary relation $R(\vec{e})$ is an *index relation* if for all $\vec{e}, \vec{d} \in \mathbb{N}^n$,

$$[(\forall i < n)\ e_i = d_i] \implies [R(\vec{e}) \iff R(\vec{d})].$$

Prove that the only decidable n-ary index relations are $R = \emptyset$ and $R = \mathbb{N}^n$.

Hint. Start with the $n = 2$ case, $R(x, y)$. Assume $R(x, y)$ is a decidable binary index relation. For any $a, b \in \mathbb{N}$, consider the relations

$$R_a(y) \iff_{\text{df}} R(a, y), \qquad R^b(x) \iff_{\text{df}} R(x, b).$$

Each R_a, R^b are (unary) index relations, and they are decidable because R is decidable. Thus, by Rice's Theorem each one is either $\emptyset$ or $\mathbb{N}$.

5.4 Additional exercises

Exercise 5.11. Let $R(\vec{x})$ and $Q(\vec{x})$ be two undecidable n-ary relations. Determine whether each of the following statements are true or false. If the statement is true, give a proof. If the statement is false, provide a counterexample.

(a) The relation $\neg R(\vec{x})$ is undecidable.

(b) The conjunction relation $R(\vec{x}) \,\&\, Q(\vec{x})$ is undecidable.

(c) The disjunction relation $R(\vec{x}) \vee Q(\vec{x})$ is undecidable.

(d) If R is binary (and undecidable), then the quantified relation

$$P(x) \iff_{\text{df}} (\exists y) R(x, y)$$

is undecidable.

Exercise 5.12. Prove that the following problems are undecidable.

(a) $P(x) \iff_{\text{df}} \varphi_x(0)\!\downarrow$

(b) $R(x) \iff_{\text{df}} (\varphi_e)$ is infinite

(c) Fix some computable partial function g. Then, define

$$Q(x) \iff_{\text{df}} \varphi_x = g$$

(d) The relation Q in part (c), except now g is some uncomputable partial function.

Exercise 5.13. Prove that there is no total computable function $f : \mathbb{N}^2 \to \mathbb{N}$ such that for all $e, x \in \mathbb{N}$,

$$\varphi_e(x)\downarrow \implies (\exists y < f(e,x))\ T_1(e,x,y),$$

where T_1 is the Kleene T predicate for unary functions.

Exercise 5.14. Consider the Turing program P consisting of the one instruction q_0BRq_0. Consider the computation under this program with an initial tape of all blanks.

(a) Explain why the computation never halts.

(b) Draw a picture of the Wang tiling of the plane which simulates the Turing machine computation using tiles from the tile set T_P defined in the previous section.

Exercise 5.15. A sequence of sets $A_0, A_1, \ldots$ is called *uniformly decidable* if the binary relation

$$R(n,x) \quad \iff_{\text{df}} \quad x \in A_n$$

is decidable.

Prove that there does **not** exist a uniformly decidable sequence $A_0, A_1, \ldots$ such that $\{A_i : i \in \mathbb{N}\}$ is the collection of all the decidable subsets of $\mathbb{N}$.

Exercise 5.16. Let $f_0, f_1, f_2, \ldots$ be an enumeration, perhaps with repetitions, of the recursive total unary functions whose ranges are all infinite.

Prove that the binary function $g : \mathbb{N}^2 \to \mathbb{N}$,

$$g(n,x) =_{\text{df}} f_n(x)$$

is not recursive.

6 Semirecursive relations

Recall that an n-ary relation R is *decidable* if and only if its characteristic function Char_R is computable. When R is decidable, we also say that R is *computable* or that R is *recursive*.

We previously defined the semirecursive relations and saw one important application: if a graph relation

$$G_f(\vec{x}, y) \iff f(\vec{x})\!\downarrow = y$$

is semirecursive, then f is recursive. Semirecursive relations have many other uses which we will explore in this chapter.

6.1 Basic properties

Definition 6.1. An n-ary relation R on $\mathbb{N}$ is *semirecursive* (or *semidecidable* or *semicomputable*) if there exists a recursive $(n+1)$-ary relation R_0 such that

$$R(\vec{x}) \iff (\exists y) R_0(\vec{x}, y)$$

for every $\vec{x} \in \mathbb{N}^n$.

In the definition above, for $\vec{x}$ with $R(\vec{x})$, any y such that $R_0(\vec{x}, y)$ holds is called a *witness* that $\vec{x}$ is in relation R. Since R_0 is decidable, we have a decision procedure to check if y witnesses that $\vec{x}$ is in R, i.e., to check if $R_0(\vec{x}, y)$ holds. If we want to know if $\vec{x}$ is in R, then we can start a search through all y's for a witness; importantly, this is not a decision procedure for membership in R. Of course, if there is a witness, i.e., $R(\vec{x})$ holds, we will eventually find a the witness and be able to return a definite "yes"; however, if $R(\vec{x})$ does not hold, during our search we will never know for certain in finite time if there are no witnesses or if we just haven't found one yet. Thus, when $R(\vec{x})$ does not hold we will never be in a position to return the answer "no".

Example 6.2. We will show that the halting problem

$$K(e) \iff_{\text{df}} \varphi_e(e)\!\downarrow$$

is semirecursive. Recall that

$$\varphi_e(e) = U(\mu y[T_1(e, e, y)]),$$

where T_1 is the Kleene T-predicate and U is a total recursive function. Since

U is total, $\varphi_e(e)\!\downarrow$ if and only if $\mu y[T_1(e,e,y)]\!\downarrow$. Thus,

$$K(e) \iff (\exists y) T_1(e,e,y),$$

which shows that K is semirecursive since T_1 is recursive.

The previous example shows that not all semirecursive relations are recursive. However, the converse is true:

Proposition 6.3. Every recursive relation is semirecursive.

Proof. Let R be a recursive n-ary relation. Define

$$R_0(\vec{x}, y) \quad \iff_{\text{df}} \quad R(\vec{x}),$$

which is recursive since recursive relations are closed under substitution by total recursive functions. Thus,

$$R(\vec{x}) \iff (\exists y) R_0(\vec{x}, y),$$

which shows that R is semirecursive. □

The next two theorems establish some basic closure properties for semirecursive relations.

Theorem 6.4. Let R and Q be semirecursive n-ary relations. Then,

(i) $R \,\&\, Q$ is semirecursive.

(ii) $R \vee Q$ is semirecursive.

Proof. Let R_0 and Q_0 be recursive relations with

$$R(\vec{x}) \iff (\exists y) R_0(\vec{x}, y), \qquad Q(\vec{x}) \iff (\exists y) Q_0(\vec{x}, y).$$

To prove (i), we first establish the equivalences

$$\begin{aligned} R(\vec{x}) \,\&\, Q(\vec{x}) &\iff \big((\exists y) R_0(\vec{x}, y)\big) \,\&\, \big((\exists y) Q_0(\vec{x}, y)\big) \\ &\iff (\exists u)[R_0(\vec{x}, (u)_0) \,\&\, Q_0(\vec{x}, (u)_1)] \end{aligned}$$

The first $\iff$ is trivial, so we consider the second $\iff$. First, we show that the $\Rightarrow$ direction holds. Assume the left-hand-side,

$$\big((\exists y) R_0(\vec{x}, y)\big) \,\&\, \big((\exists y) Q_0(\vec{x}, y)\big),$$

holds. Then there is y_0 and y_1 such that $R_0(\vec{x}, y_0)$ and $Q_0(\vec{x}, y_1)$ both hold. (Note that it is not necessarily the case that there is a single y such that $R_0(\vec{x}, y)$ and $Q_0(\vec{x}, y)$ hold simultaneously.) Now define $u = \langle y_0, y_1 \rangle$. Then, this u clearly witnesses that the right-hand-side,

$$(\exists u)[R_0(\vec{x}, (u)_0) \,\&\, Q_0(\vec{x}, (u)_1)],$$

holds. The $\Leftarrow$ direction of the second equivalence is easier, and we leave the details to the reader.

Now, we can define

$$P(\vec{x}, u) \iff_{\text{df}} R_0(\vec{x}, (u)_0) \,\&\, Q_0(\vec{x}, (u)_1),$$

which is recursive since recursive relations are closed under total recursive substitutions and &. Thus,

$$(R \,\&\, Q)(\vec{x}) \iff (\exists u) P(\vec{x}, u),$$

which shows that $R \,\&\, Q$ is semirecursive.

The proof of (ii) is left as an exercise. □

Exercise 6.5. Prove (ii) of Theorem 6.4.

Theorem 6.6. Let R be a $(n+1)$-ary semirecursive relation. Then, the following relations are also semirecursive:

(i) $P(\vec{x}) \iff_{\text{df}} (\exists y) R(\vec{x}, y)$

(ii) $Q_0(\vec{x}, y) \iff_{\text{df}} (\exists i \leq y) R(\vec{x}, i)$

(iii) $Q_1(\vec{x}, y) \iff_{\text{df}} (\forall i \leq y) R(\vec{x}, i)$.

Proof. Let R_0 be a recursive relation satisfying

$$R(\vec{x}, y) \iff (\exists t) R_0(\vec{x}, y, t).$$

To prove that P is semirecursive, it is enough to establish the equivalences

$$\begin{aligned} P(\vec{x}) &\iff_{\text{df}} (\exists y) R(\vec{x}, y) \iff (\exists y)(\exists t) R_0(\vec{x}, y, t) \\ &\iff (\exists u) R_0(\vec{x}, (u)_0, (u)_1). \end{aligned}$$

The first two equivalences are trivial, and the third equivalence is proved very similarly to the proof of (i) in Theorem 6.4.

To prove (iii) is semirecursive, we must establish the equivalences

$$\begin{aligned} Q_1(\vec{x}, y) \quad &\iff_{\text{df}} \quad (\forall i \leq y) R(\vec{x}, i) \iff (\forall i \leq y)(\exists t) R_0(\vec{x}, i, t) \\ &\iff (\exists u)(\forall i \leq y) R_0(\vec{x}, i, (u)_i). \end{aligned}$$

Note that the relation $(\forall i \leq y) R_0(\vec{x}, i, (u)_i)$ is recursive since recursive relations are closed under total recursive substitution and bounded quantification. The details are left as an exercise.

The proof of (ii) is also an exercise. □

Exercise 6.7. Finish the proof of Theorem 6.6. This consists of proving the equivalence

$$(\forall i \leq y)(\exists t) R_0(\vec{x}, i, t) \iff (\exists u)(\forall i \leq y) R_0(\vec{x}, i, (u)_i).$$

and proving (ii).

Recall that previously we showed that for a <u>total</u> function $f : \mathbb{N}^n \to \mathbb{N}$, f is recursive if and only if its graph relation

$$G_f(\vec{x}, y) \quad \iff_{\text{df}} \quad f(x) = y$$

is recursive. We next prove the theorem that correctly generalizes this fact for partial functions, the important difference being that the graph relation is now semirecursive.

Theorem 6.8. Let $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ be a partial function and let

$$G_f(\vec{x}, w) \quad \iff_{\text{df}} \quad f(\vec{x}){\downarrow} = w$$

be its graph relation. Then, f is recursive if and only if G_f is semirecursive.

Proof. We already proved the right-to-left direction in Theorem 2.40, so we only have to prove the left-to-right direction. Assume f is recursive. Then, there is e such that

$$f(\vec{x}) = U(\mu y[T_n(e, \vec{x}, y)]),$$

where T_n is recursive and U is a total recursive function. Thus,

$$G_f(\vec{x}, w) \iff (\exists y)\Big[T_n(e, \vec{x}, y) \,\&\, (\forall i < y)[\neg T_n(e, \vec{x}, i)] \,\&\, U(y) = w\Big],$$

which shows that G_f is semirecursive. Note that we are using that the graph of U is recursive since U is a total recursive function. □

This theorem has many important applications which we will now explore. Recall that we proved that recursive relations are closed under substitution of *total* recursive functions. Semirecursive relations have the following *stronger* closure property:

Theorem 6.9 (Semirecursive relations are closed under partial recursive substitution)**.** Let R be a semirecursive k-ary relation and let $f_0, \ldots, f_{k-1} : \mathbb{N}^n \rightharpoonup \mathbb{N}$ be recursive partial functions. Then, the n-ary relation

$$Q(\vec{x}) \iff_{\text{df}} f_0(\vec{x})\downarrow \,\&\, \cdots \,\&\, f_{k-1}(\vec{x})\downarrow \,\&\, R(f_0(\vec{x}), \ldots, f_{k-1}(\vec{x}))$$

is semirecursive.

Proof. For $i = 0, \ldots, k-1$, let G_i be the graph of f_i. Note that G_i is a semirecursive relation since f_i is computable. Then, Q satisfies the equivalence

$$\begin{aligned} Q(\vec{x}) \iff (\exists y_0) \ldots (\exists y_{k-1})[G_0(\vec{x}, y_0) \,\&\, G_1(\vec{x}, y_1) \,\&\, \cdots \,\&\, G_{k-1}(\vec{x}, y_{k-1}) \\ \&\, R(y_0, y_1, \ldots, y_{k-1})] \end{aligned}$$

Using the closure of semirecursive relations under & and existential quantification over $\mathbb{N}$, we conclude that Q is semirecursive. □

The next theorem provides us with an important new criterion for establishing that a set is recursive.

Theorem 6.10 (Kleene's theorem)**.** A relation R is recursive if and only if both R and $\neg R$ are semirecursive.

Proof. The ($\Rightarrow$) direction is easy. It just uses that recursive relations are closed under $\neg$ and that recursive relations are semirecursive.

For the ($\Leftarrow$) direction, suppose R and $\neg R$ are semirecursive. To show that R is recursive, we must show that Char_R is a recursive total function. To show Char_R is recursive, it is enough to show that its graph is semirecursive; but this is shown by the equivalence

$$\text{Char}_R(\vec{x}) = y \iff [y = 1 \,\&\, R(\vec{x})] \vee [y = 0 \,\&\, \neg R(\vec{x})].$$

and the closure properties of semirecursive relations. □

Example 6.11. The complement of the halting problem, $\neg K$, is not semirecursive, since otherwise Kleene's theorem would imply that K is recursive.

The next theorem gives us an equivalent definition of semirecursive relation.

Theorem 6.12. An n-ary relation $R \subseteq \mathbb{N}^n$ is semirecursive if and only if it is the domain of a computable partial function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$.

Proof. ($\Rightarrow$) Assume R is semirecursive. Pick a recursive R_0 such that

$$R(\vec{x}) \iff (\exists y) R_0(\vec{x}, y).$$

Then, define $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ by

$$f(\vec{x}) = \mu y[R_0(\vec{x}, y)].$$

It is then easy to check that f is computable and its domain is all $\vec{x}$ such that $R(\vec{x})$ holds.

($\Leftarrow$) Assume R is the domain of a computable $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$. Since f is computable, its graph G_f is semirecursive. Then,

$$R(\vec{x}) \iff f(\vec{x})\downarrow \iff (\exists y) f(\vec{x})\downarrow = y \iff (\exists y) G_f(\vec{x}, y).$$

Since G_f is semirecursive and semirecursive relations are closed under existential quantification over $\mathbb{N}$, we conclude that R is semirecursive. □

The next exercise establishes yet another equivalent definition of semirecursive relation.

Exercise 6.13. Prove that an n-ary relation R is semirecursive if and only if its *partial characteristic function* $\mathrm{Char}^p_R : \mathbb{N}^n \rightharpoonup \mathbb{N}$,

$$\mathrm{Char}^p_A(\vec{x}) = \begin{cases} 1 & \text{if } R(\vec{x}) \\ \uparrow & \text{if } \neg R(\vec{x}). \end{cases}$$

is computable.

The following is an easy consequence of the previous exercise, but we will use it many times.

Exercise 6.14. Let $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ be recursive, and let R be a semirecursive n-ary relation. Prove that the partial function $g : \mathbb{N}^n \rightharpoonup \mathbb{N}$ defined by

$$g(\vec{x}) = \begin{cases} f(\vec{x}) & \text{if } R(\vec{x}) \text{ holds and } f(\vec{x})\downarrow, \\ \uparrow & \text{otherwise.} \end{cases}$$

is recursive.

It is worth describing the intuition behind why the function g in Exercise 6.14 is computable. First, let R be the domain of some computable partial function h. To compute $g(\vec{x})$, begin by trying to compute $h(\vec{x})$. If the $h(\vec{x})$ computation ever halts, then start computing $f(\vec{x})$. If that computation halts, then return the value $f(\vec{x})$. Note that if $h(\vec{x})$ never halts (so that $R(\vec{x})$ does not hold), then the whole computation will diverge; you cannot return some other value in the case $\neg R(\vec{x})$ since you do not know in finite time if $\neg R(\vec{x})$ holds.

6.2 Recursively enumerable sets

This section is concerned with subsets of $\mathbb{N}$. These are the same as unary relations on $\mathbb{N}$, but in this context it is more traditional to think of them as sets.

A subset $A \subseteq \mathbb{N}$ which is semirecursive (as a unary relation) is also called *recursively enumerable* or *computably enumerable*. "Recursively enumerable" is often abbreviated by "r.e.", and "computably enumerable" is often abbreviated by "c.e." The reason for this terminology is the next theorem.

Theorem 6.15. A subset $A \subseteq \mathbb{N}$ is recursively enumerable (i.e., semirecursive) if and only if $A = \emptyset$ or there is a recursive total $f : \mathbb{N} \to \mathbb{N}$ whose image is A,

$$f[\mathbb{N}] =_{\text{df}} \{f(x) : x \in \mathbb{N}\} = A.$$

Proof. ($\Rightarrow$) Assume $A \subseteq \mathbb{N}$ is recursively enumerable and nonempty. If A is finite, $A = \{x_0, x_1, \ldots, x_{k-1}\}$, then it is easy to define a recursive f whose image is A. Just take $f(i) = x_i$ for $i = 0, \ldots, k-1$, and $f(j) = x_{k-1}$ for $j \geq k$. It is easy to check (!) that this f is recursive by showing its graph is recursive.

Now, assume A is infinite. Choose a recursive R such that

$$x \in A \iff (\exists y) R(x, y).$$

Define $f : \mathbb{N} \to \mathbb{N}$ by

$$f(x) = \big(\mu y[y \geq x \;\&\; R((y)_0, (y)_1)\big)_0,$$

which is clearly recursive. Then, check (!) that f is total and that $f[\mathbb{N}] = A$.

We pause to remark on the intuition of the above definition of f. Recall that we can computably enumerate infinite recursive sets in increasing order. As we will see in an exercise below, we cannot always computably enumerate an infinite semirecursive set A *in increasing order*. However, the relation R

is recursive, and hence we can computably enumerate (sequence codes for) its elements in increasing order:

$$\langle a_0, b_0\rangle, \langle a_1, b_1\rangle, \langle a_2, b_2\rangle, \ldots$$

Note that each a_i is an element of A and the corresponding b_i is a witness to the fact that a_i is an element of A. If we then project all of these pairs to their first coordinates,

$$a_0, a_1, a_2 \ldots$$

then we have computably enumerated all elements of A. Note, importantly, that these a_i need not be listed in increasing order and there may be repetitions, since an element of A may have many many witnesses to it being an elements of A.

($\Leftarrow$) Assume now that $A = \emptyset$ or that A is the range of a recursive total function. The case where $A = \emptyset$ is trivial, so assume there is a recursive total $f : \mathbb{N} \to \mathbb{N}$ with $f[\mathbb{N}] = A$. Then, the equivalence

$$y \in A \iff (\exists x)[f(x) = y] \iff (\exists x)G_f(x, y)$$

shows that A is recursively enumerable, using that G_f is semirecursive and the closure of semirecursive relations under $\exists$. □

Exercise 6.16. Give the details of the two "check (!)"'s in the proof above.

We make a few remarks about this theorem.

(1) The proof can easily be adapted to establish the following fact:

Theorem 6.17. $A \subseteq \mathbb{N}$ is recursively enumerable if and only if it is the image of a recursive partial function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$.

Note that no special case has to be mentioned for $A = \emptyset$, since $\emptyset$ is the image of the partial function which diverges on every input.

(2) For infinite recursively enumerable sets, we have the following strengthening of the theorem:

Theorem 6.18. Let A be an infinite recursively enumerable set. Then, there is a recursive total injective $f : \mathbb{N} \to \mathbb{N}$ whose image is A.

When $f : \mathbb{N} \to \mathbb{N}$ has image A, we say that f *enumerates* A. If f is also injective, we say that f *enumerates A without repetitions.*

Exercise 6.19. Prove Theorem 6.18. *Hint.* Start with a recursive total function $g : \mathbb{N} \to \mathbb{N}$ which enumerates A, perhaps with repetitions. Then use a primitive recursion to adjust g to eliminate repetitions.

If A is an infinite r.e. set, then there is a recursive $f : \mathbb{N} \to \mathbb{N}$ which enumerates A without repetitions. However, the following exercise establishes that in general this f cannot enumerate A in increasing order, $f(0) < f(1) < f(2) < \cdots$.

Exercise 6.20. Let $A \subseteq \mathbb{N}$ and let $f : \mathbb{N} \to \mathbb{N}$ be a recursive function with $f[\mathbb{N}] = A$. Show that if f is a strictly increasing function, then A is recursive.

Hint. When f is strictly increasing, the search for a x with $f(x) = y$ becomes a bounded search.

We can use this exercise and Theorem 6.18 to prove the following fact.

Theorem 6.21. If $A \subseteq \mathbb{N}$ is an infinite r.e. set, then it contains an infinite recursive subset $B \subseteq A$.

Proof. Let $f : \mathbb{N} \to \mathbb{N}$ be a total recursive function which enumerates A without repetitions. Then, define $g : \mathbb{N} \to \mathbb{N}$ by the recursion

$$\begin{aligned} g(0) &= f(0) \\ g(x+1) &= f\big(\mu y[f(y) > g(x)]\big). \end{aligned}$$

It is easy to check that g is recursive and total. Moreover, a simple induction on x shows that $g(x+1) > g(x)$ for all $x \in \mathbb{N}$, i.e., g is strictly increasing. Then, g enumerates an infinite subset B of $A = f[\mathbb{N}]$ *in increasing order*. By Exercise 6.20, B is recursive. □

We put down for the record a theorem which summarizes all the equivalent ways of defining recursively enumerable sets.

Theorem 6.22. Let $A \subseteq \mathbb{N}$. Then, the following are all equivalent.

(i) A is semirecursive, i.e., there is a recursive relation $R(x, y)$ such that

$$x \in A \iff (\exists y) R(x, y).$$

(ii) A is the domain of a recursive partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$.

(iii) A is the image of a recursive partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$.

(iv) $A = \emptyset$ or there is a recursive total $f : \mathbb{N} \to \mathbb{N}$ which enumerates A.

(v) A is finite or there is a recursive injective total $f : \mathbb{N} \to \mathbb{N}$ which enumerates A without repetitions.

(vi) The *partial characteristic function* $\mathrm{Char}_A^p : \mathbb{N} \rightharpoonup \mathbb{N}$,

$$\mathrm{Char}_A^p(x) =_{\mathrm{df}} \begin{cases} 1 & \text{if } x \in A \\ \uparrow & \text{if } x \notin A \end{cases}$$

is recursive.

If A satisfies one (equivalently, all) of these conditions, it is said to be recursively enumerable.

The last exercise points out that if R is an n-ary semirecursive relation, then we can always recursively enumerate sequence codes for the tuples on which R holds.

Exercise 6.23. Let $R \subseteq \mathbb{N}^n$. Prove that R is semirecursive if and only if

$$\{\langle x_0, \ldots, x_{n-1}\rangle : R(x_0, \ldots, x_{n-1})\}$$

is recursively enumerable.

6.3 An effective enumeration of r.e. sets

Recall that an n-ary relation R is semirecursive if and only if it is the domain of a recursive partial function $\varphi_e^{(n)} : \mathbb{N}^n \rightharpoonup \mathbb{N}$. This is a very important representation for semirecursive relations, and so we introduce the following (very standard) notation,

$$W_e^{(n)} =_{\mathrm{df}} \mathrm{domain}(\varphi_e^{(n)}).$$

Thus, we have an enumeration

$$W_0^{(n)}, W_1^{(n)}, W_2^{(n)}, \ldots$$

of all the semirecursive n-ary relations. Importantly, this enumeration is also *effective* in the sense that the relation

$$V_U^{(n)}(e, \vec{x}) \quad \iff_{\mathrm{df}} \quad W_e^{(n)}(\vec{x}) \qquad [\vec{x} \in \mathbb{N}^n]$$

is semirecursive. $V_U^{(n)}$ is semirecursive since is is the domain of the universal n-ary partial function $\psi_U^{(n)} : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$,

$$V_U^{(n)}(e, \vec{x}) \iff \varphi_e^{(n)}(\vec{x})\downarrow \iff \psi_U^{(n)}(e, \vec{x})\downarrow.$$

In the case that $n = 1$, we use the notation

$$W_0, W_1, W_2, \ldots, \qquad V_U(e, x) \iff_{\text{df}} x \in W_e.$$

Thus, the W_e are an effective enumeration of the r.e. subsets of $\mathbb{N}$. We call e a *code* of the r.e. set W_e, and the set V_U is called the *universal relation for r.e. sets*. Since two different recursive partial functions can have the same domain, it is clear that any r.e. set has many different codes.

The r.e. subsets of $\mathbb{N}$ are exactly the ranges of recursive partial functions. Thus, we have another effective enumeration of the r.e. sets:

$$E_0, E_1, \ldots, \quad \text{where } E_e =_{\text{df}} \text{domain}(\varphi_e).$$

As usual, by *effective* enumeration we mean that

$$x \in E_e \iff \varphi_e(x)\!\downarrow$$

is semirecursive (as a relation of both e and x). Generally, the W_e sets are used more often than the E_e sets.

§ **s-m-n and second recursion theorems for relations**. The s-m-n theorem and the second recursion theorem are important tools for the theory of recursive partial functions. They imply (easily) versions which are stated in terms of semirecursive relations.

For the s-m-n theorem, we start with a semirecursve $(m + n)$-ary relation $R(\vec{x}, \vec{y})$. If we fix a tuple $\vec{x}$, then the relation

$$Q(\vec{y}) \iff_{\text{df}} R(\vec{x}, \vec{y})$$

is also semirecursive. The s-m-n theorem tell us that, given the tuple $\vec{x}$, we can compute a code for Q, i.e., compute an e such that $Q = W_e^{(n)}$.

Theorem 6.24 (s-m-n for semirecursive relations)**.** Let $R(\vec{x}, \vec{y})$ be a semirecursive $(m+n)$-ary relation. Then, there is a recursive total $S : \mathbb{N}^m \to \mathbb{N}$ such that

$$R(\vec{x}, \vec{y}) \iff W^{(n)}_{S(\vec{x})}(\vec{y})$$

for all $\vec{x} \in \mathbb{N}^m$ and $\vec{y} \in \mathbb{N}^n$.

Proof. Since R is semirecursive, it is the domain of some $(m+n)$-ary recursive partial function f. By the s-m-n theorem, there is a recursive total $S : \mathbb{N}^m \to \mathbb{N}$ such that

$$f(\vec{x}, \vec{y}) = \varphi^{(n)}_{S(\vec{x})}(\vec{y})$$

for all $\vec{x} \in \mathbb{N}^m$ and $\vec{y} \in \mathbb{N}^n$. It is easily checked that S satisfies the desired condition. $\square$

Theorem 6.25 (Second recursion theorem for semirecursive relations)**.** Let $R(e, \vec{x})$ be a semirecursive $(n+1)$-ary relation. Then, there exists $e^* \in \mathbb{N}$ such that

$$R(e^*, \vec{x}) \iff W_{e^*}^{(n)}(\vec{x})$$

for all $\vec{x} \in \mathbb{N}^n$.

Exercise 6.26. Prove the second recursion theorem for semirecursive relations. *Hint:* It easily follows from the original second recursion theorem.

§ **Effective operations on r.e. sets**. Recall that the semirecursive relations are closed under $\vee$. When discussing r.e. sets, we usually think of closer under $\vee$ as closure under union, $\cup$, since we are thinking of them as sets rather than relations.

So, if W_{e_1} and W_{e_2} are r.e. sets, then $W_{e_1} \cup W_{e_2}$ is also r.e., hence

$$W_{e_1} \cup W_{e_2} = W_z$$

for some $z \in \mathbb{N}$. But what is this code z? Is it computably determined by the codes e_1, e_2? The answer is "yes"; more formally, we are saying that there is a recursive total function $S : \mathbb{N}^2 \to \mathbb{N}$ such that

$$W_{e_1} \cup W_{e_2} = W_{S(e_1,e_2)}$$

for all $e_1, e_2 \in \mathbb{N}$. Because this computable S exists, we say that the union operation is *effective in the codes*. We now prove that this f exists. The key tool in proofs like these is the s-m-n theorem.

Proposition 6.27. The union operation on r.e. sets is effective in the codes, i.e., there is a recursive total function $S : \mathbb{N}^2 \to \mathbb{N}$ such that

$$W_{e_1} \cup W_{e_2} = W_{S(e_1,e_2)}$$

for all $e_1, e_2 \in \mathbb{N}$.

Proof. Define $R(e_1, e_2, x)$ by

$$R(e_1, e_2, x) \iff_{\text{df}} (x \in W_{e_1}) \vee (x \in W_{e_2}).$$

R is semirecursive by closure properties and the fact that the universal unary semirecursive relation is semirecursive. By the s-m-n theorem for relations, there is a recursive total $S : \mathbb{N}^2 \to \mathbb{N}$ such that

$$R(e_1, e_2, x) \iff W_{S(e_1,e_2)}(x).$$

for all $e_1, e_2, x \in \mathbb{N}$. Then check easily that $W_{e_1} \cup W_{e_2} = W_{S(e_1,e_2)}$. □

Although we framed this discussion in terms of r.e. sets, $\vee$ is also an effective operation for semirecursive n-ary relations:

Exercise 6.28. Let $n > 1$. Formulate and prove the proposition that $\vee$ is an operation on semirecursive n-ary relations which is effective in the codes.

Set intersection $\cap$ is the "set equivalent" of logical conjunction & for relations. Intersection is also effective in the codes for r.e. sets:

Exercise 6.29. Formulate and prove the proposition that $\cap$ is an operation on r.e. sets which is effective in the codes.

6.4 Semirecursive-selection and reduction

In this section, we explore a few more structural properties of the class of semirecursive relations, the first of which is a selection property.

Consider a semirecursive relation $R(\vec{x}, y)$. For any $\vec{x}$, let

$$R_{\vec{x}} =_{\text{df}} \{y \in \mathbb{N} : R(\vec{x}, y)\}.$$

Suppose whenever $R_{\vec{x}} \neq \emptyset$ you would like to select a member of $R_{\vec{x}}$. But we don't want to do this arbitrarily; instead, we would like to do this *computably.* In other words, we would like a computable partial function f such that whenever $R_{\vec{x}}$ is nonempty, $f(\vec{x})$ converges and returns a member of $R_{\vec{x}}$. The next theorem says that this can indeed always be done.

Theorem 6.30 (Semirecursive-selection property)**.** Let $R(\vec{x}, y)$ be a semirecursive relation. Then, there exists a recursive partial function $f : \mathbb{N}^n \to \mathbb{N}$ such that

$$\begin{aligned} f(\vec{x})\downarrow &\iff (\exists y)R(\vec{x}, y) \\ f(\vec{x})\downarrow &\implies R(\vec{x}, f(\vec{x})). \end{aligned}$$

f is called a *recursive selection function* for $R(\vec{x}, y)$.

Proof. Let R_0 be a recursive relation satisfying

$$R(\vec{x}, y) \iff (\exists t)\ R_0(\vec{x}, y, t).$$

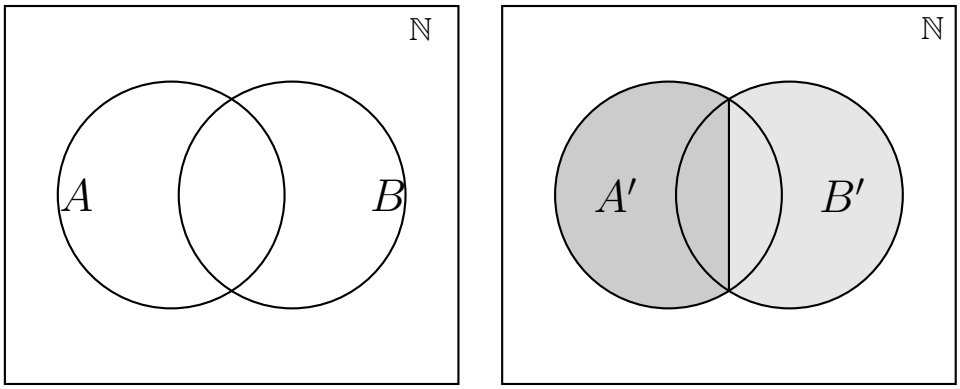
We describe the unituitive idea first: given a tuple $\vec{x}$, start a search search through all $\langle y, t\rangle$ until you first find such a pair with t witnessing that y is in $R_{\vec{x}}$. Our selection function will pick this y.

More formally, define

$$f(\vec{x}) = \big(\mu u[R_0(\vec{x}, (u)_0, (u)_1)]\big)_0.$$

The details of checking that this f has the desired properties are left as an exercise. □

The next property we will consider is the reduction property. Suppose A and B are r.e. sets with overlap, $A \cap B \neq \emptyset$. We would like to effectively divide up the overlap between A and B to get two new sets $A' \subseteq A$ and $B' \subseteq B$ which are disjoint but have the same union as $A \cup B$. This is depicted below, with the original sets A, B in the left picture and the reduction sets A', B' in the right.



Formally, let $A, B \subseteq \mathbb{N}$. A pair of sets $A', B' \subseteq \mathbb{N}$ *reduces* A and B if it satisfies all the following conditions:

(i) $A' \subseteq A$ and $B' \subseteq B$;

(ii) $A' \cap B' = \emptyset$;

(iii) $A' \cup B' = A \cup B$.

We can use the semirecursive-selection principle to show that any pair of r.e. sets can be reduced by a pair of r.e. sets. We say that the collection of r.e. sets has the *reduction property*.

Theorem 6.31. If $A, B \subseteq \mathbb{N}$ are r.e., then there is a pair of r.e. sets $A', B' \subseteq \mathbb{N}$ which reduce A and B.

Proof. The intuitive idea is that we will give every element of A a 0 label and every element of B a 1 label. We'd like to send everyone with a 0 label to A' and everyone with a 1 label to B'. Of course, there is a problem: the elements of $A \cap B$ will each receive two labels, 0 and 1. So, for each element of $A \cap B$ we want to choose a label to keep and a label to throw away. We must do this computably, so we use semirecursive-selection.

Formally, define a binary relation $R(x, n)$ by

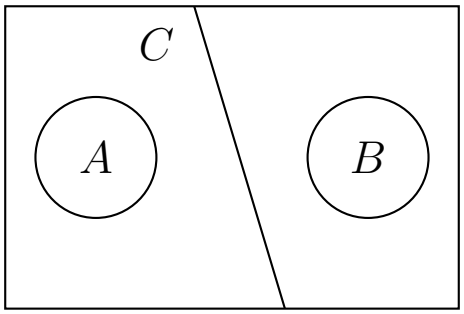
$$R(x, n) \quad \iff_{\text{df}} \quad [x \in A \,\&\, x = 0] \vee [x \in B \,\&\, x = 1].$$

Let $f : \mathbb{N} \rightharpoonup \mathbb{N}$ be a recursive selection function for R. Then, define

$$A' =_{\text{df}} \{x : f(x)\!\downarrow = 0\}, \qquad B' =_{\text{df}} \{x : f(x)\!\downarrow = 1\}.$$

It is straight-forward to check that A', B' are r.e. sets which reduce A and B. □

Let $A, B \subseteq \mathbb{N}$ be disjoint sets. A set $C \subseteq \mathbb{N}$ is said to *separate* A from B if $A \subseteq C$ and $B \subseteq \neg C$. We also say C *separates* A and B if either C separates A from B or C separates B from A.



A set $A \subseteq \mathbb{N}$ is called *co-recursively enumerable* (or *co-r.e.*) if its complement $\neg A$ is r.e. Any pair of disjoint co-r.e. sets can be separated by a recursive set; because of this, we say that the collection of co-r.e. sets has the *separation property*.

Theorem 6.32. If A and B are disjoint co-r.e. sets, then there is a recursive C which separates them.

Exercise 6.33. Prove Theorem 6.32. *Hint.* Consider the complements $\neg A$ and $\neg B$, which are both r.e. Then apply the reduction property to $\neg A$ and $\neg B$.

The r.e. sets do not have the separation property:

Theorem 6.34. There exist recursively inseparable disjoint r.e. sets A and B, i.e., $A \cap B = \emptyset$ and there is no recursive C which separates A and B.

Proof. Define sets

$$A =_{\text{df}} \{e : \varphi_e(e)\downarrow = 0\}, \qquad B =_{\text{df}} \{e : \varphi_e(e)\downarrow = 1\}.$$

A and B are clearly disjoint. Suppose towards a contradiction that there is a recursive $C \subseteq \mathbb{N}$ such that $A \subseteq C$ and $B \subseteq \neg C$. Define a total function $f : \mathbb{N} \to \mathbb{N}$ by

$$f(e) = \begin{cases} 1 & \text{if } x \in C \\ 0 & \text{if } x \notin C. \end{cases}$$

f is recursive since it is defined by recursive cases. Thus, there is e_0 such that $f = \varphi_{e_0}$. Since $f = \varphi_{e_0}$ is total and is $\{0,1\}$-valued, we must have either $e_0 \in A$ or $e^* \in B$. If $e_0 \in A$, then by definition of A we have $\varphi_{e_0}(e_0)\downarrow = 0$. On the other hand, by the definition of f we have $f(e_0) = 1$ since $e_0 \in A \subseteq C$. But this is a contradiction since we have

$$0 = \varphi_{e_0}(e_0) = f(e_0) = 1$$

It is easy to show that the assumption $e_0 \in B$ leads to the same sort of contradiction. □

6.5 Additional exercises

Exercise 6.35. Prove that the following relations are semirecursive.

(a) The unary relation

$$R(x) \iff_{\text{df}} E_x \neq \emptyset,$$

where $E_x = \text{range}(\varphi_x)$.

(b) The unary relation $Q(n) \iff_{\text{df}}$ n is a Fermat number $\iff_{\text{df}}$ there are integers $x, y, z > 0$ such that $x^n + y^n = z^n$.

Exercise 6.36. Determine which of the following sets are recursive, and which are r.e. but not recursive.

(a) $\{x \in \mathbb{N} : \varphi_x \text{ is not injective}\}$.

(b) $\{x \in \mathbb{N} : x \text{ is a perfect square}\}$.

(c) $\{x \in \mathbb{N} : x \in E_x\}$

Exercise 6.37. Determine whether the following relations are recursive, semirecursive and not decidable, or not semirecursive. Prove your answers.

(a) $R(e) \iff_{\text{df}} W_e \neq \emptyset$.

(b) $Q(e) \iff_{\text{df}} \varphi_e(n)\downarrow = n$ for some n.

(c) $P(e, x) \iff_{\text{df}} \varphi_e(x)\uparrow$

(d) $G_f(x, y) \iff_{\text{df}} f(x)\downarrow = y$,
where $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is a fixed recursive *partial* function whose domain is a recursive set.

Hint: Don't forget about Rice's Theorem and Kleene's Theorem!

Exercise 6.38. Give an example of a semirecursive $R \subseteq \mathbb{N}^2$ such that the partial function $f(x) = \mu y[R(x, y)]$ is **not** recursive. *Hint.* Recall that we cannot computably enumerate in increasing order an r.e. set which is not recursive.

Exercise 6.39. Show that there is no effective enumeration of all the decidable unary relations, i.e., show that there does not exist a sequence

$$R_0, R_1, R_2, \ldots, \qquad R_i \subseteq \mathbb{N},$$

such that (1) each R_i is decidable and every decidable unary relation $P \subseteq \mathbb{N}$ is equal to at least one of the R_i; and (2) the binary relation

$$Q(i, x) \iff_{\text{df}} R_i(x)$$

is decidable.

Exercise 6.40. Prove that minimization is an effective operation on recursive partial functions, i.e., for any integer $n > 0$, prove that there is a recursive total function $u : \mathbb{N} \to \mathbb{N}$ such that for all $e \in \mathbb{N}$ and $\vec{x} \in \mathbb{N}^n$

$$\varphi^{(n)}_{u(e)}(\vec{x}) = \mu y[\varphi^{(n+1)}_e(\vec{x}, y) = 0].$$

Exercise 6.41. Prove that if $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is a recursive partial function and $A \subseteq \mathbb{N}$ is r.e., then the preimage

$$f^{-1}[A] =_{\text{df}} \{x \in \mathbb{N} : f(x)\downarrow \in A\}$$

is also r.e. Moreover, show that this is effective in the codes in the following sense: there is a recursive total $s : \mathbb{N} \to \mathbb{N}$ such that for any $e \in \mathbb{N}$,

$$W_{s(e)} = f^{-1}[W_e].$$

Exercise 6.42. Let $f : \mathbb{N} \rightharpoonup \mathbb{N}$ be a recursive partial function. For a set $A \subseteq \mathbb{N}$, we use the notation

$$f[A] =_{\text{df}} \{y \in \mathbb{N} : f(x)\!\downarrow = y \text{ for some } x \in A\}.$$

(a) Prove that if $A \subseteq \mathbb{N}$ is r.e., then $f[A]$ is also r.e.

(b) Prove that part (a) is true effectively in the codes, i.e., prove that there is a recursive total $u : \mathbb{N} \to \mathbb{N}$ such that

$$W_{u(e)} = f[W_e] \quad \text{for all } e \in \mathbb{N}.$$

Exercise 6.43. Suppose $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is a recursive partial function. Prove that there is a recursive partial function $g : \mathbb{N} \rightharpoonup \mathbb{N}$ such that for every $y \in \mathbb{N}$,

$$\begin{aligned} g(y)\!\downarrow &\iff y \in \text{range}(f) \\ g(y)\!\downarrow &\implies f(g(y))\!\downarrow = y. \end{aligned}$$

Exercise 6.44. Show that the semirecursive-selection principle holds uniformly in the codes, i.e., prove that for any $n > 0$, there is a recursive total $u : \mathbb{N} \to \mathbb{N}$ such that for every $e \in \mathbb{N}$, the n-ary recursive partial function $\varphi^{(n)}_{u(e)}$ satisfies:

$$\begin{aligned} \varphi^{(n)}_{u(e)}(\vec{x})\!\downarrow &\iff (\exists y)\ W^{(n+1)}_e(\vec{x}, y) \\ \varphi^{(n)}_{u(e)}(\vec{x})\!\downarrow &\implies W^{(n+1)}_e(\vec{x}, \varphi^{(n)}_{u(e)}(\vec{x})) \end{aligned}$$

Disjoint sets $A, B \subseteq \mathbb{N}$ are called *effectively recursively inseparable* if there is a computable total function $f : \mathbb{N}^2 \to \mathbb{N}$ such that for any r.e. sets W_a, W_b,

$$[A \subseteq W_a \ \&\ B \subseteq W_b \ \&\ W_a \cap W_b = \emptyset] \implies f(a, b) \notin (W_a \cup W_b).$$

To make sense of this condition, think of W_a as a potential recursive separator of A and B. Convince yourself that W_a would be a recursive separator if $W_b = \mathbb{N} \setminus W_a$, i.e., when $W_a \cup W_b = \mathbb{N}$. If A and B are recursively inseparable, then $W_a \cup W_b = \mathbb{N}$ must fail, and in fact the recursive function f effectively produces a witness $f(a, b)$ to the fact that $W_a \cup W_b$ is not all of $\mathbb{N}$.

Exercise 6.45. (a) Prove that effectively recursively inseparable sets are recursively inseparable, i.e., there is no recursive set C such that $A \subseteq C$ and $B \subseteq \mathbb{N} \setminus C$.

(b) Prove that the sets

$$K_0 = \{x \in \mathbb{N} : \varphi_x(x)\downarrow = 0\} \quad \text{and} \quad K_1 = \{x \in \mathbb{N} : \varphi_x(x)\downarrow = 1\}$$

are effectively recursively inseparable. *Hint.* Construct a computable total function $f : \mathbb{N}^2 \to \mathbb{N}$ such that *if* W_a *and* W_b *are disjoint*, then

$$\varphi_{f(a,b)}(x) = \begin{cases} 1 & \text{if } x \in W_a \\ 0 & \text{if } x \in W_b \\ \uparrow & \text{otherwise.} \end{cases}$$

7 Computable reductions

While exploring the world of non-computable objects, we will be interested in comparing the computational power of one object to another. We aim to say that an object A is simpler than an object B because, in some sense, A can be *reduced* to B. Computability theory is full of different notions to do this type of comparison. In this chapter, we will study two of them: m-reductions and 1-reductions. In later chapters, we will add one more: Turing reductions.

When studying these reducibility notions, we will focus on one type of object: subsets of $\mathbb{N}$. This simplifies definitions and notation, and, importantly, we don't lose any generality by ignoring n-ary relations on $\mathbb{N}$. This is because, relative to each of our reducibility notions, an n-ary relation R is equivalent to its set of codes,

$$\widetilde{R} := \{u \in \mathbb{N} : \mathrm{Seq}(u) \ \ \& \ \ \mathrm{lh}(u) = n \ \ \& \ \ R((u)_0, \ldots, (u)_{n-1})\}.$$

So, in place of talking about reductions involving $R \subseteq \mathbb{N}^n$, we can always instead talk about reductions involving its coded version $\widetilde{R} \subseteq \mathbb{N}$.

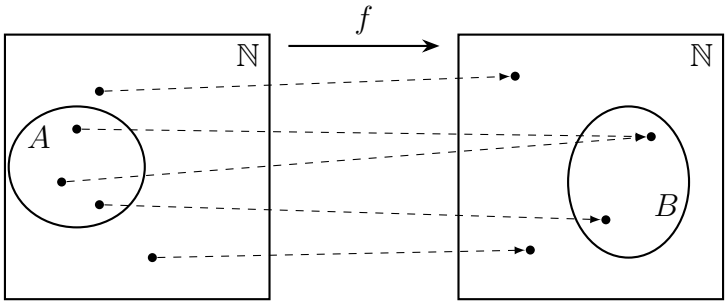
7.1 m-reductions

Our first notion of reducibility will come from m-reductions:

Definition 7.1. Let $A, B \subseteq \mathbb{N}$. An *$m$-reduction* from A to B is a computable total function $f : \mathbb{N} \to \mathbb{N}$ such that, for all $n \in \mathbb{N}$,

$$n \in A \iff f(n) \in B.$$

We say that A is *m-reducible* to B, and write $A \leq_m B$, if there exists an m-reduction f from A to B.



We make a several remarks about this definition:

(1) An m-reduction is not required to be one-to-one (as depicted in the above picture). In fact, the "m" in m-reduction stands for "many-one". When an m-reduction happens to be one-to-one, it is called a 1-reduction; we will study 1-reductions in Section 7.3. For now, just remember that, while an m-reduction can be one-to-one, it is not required to be.

(2) By definition, an m-reduction f from A to B takes elements of A to elements of B, and takes elements of $\neg A$ to elements of $\neg B$. Note that this means that f is also an m-reduction from $\neg A$ to $\neg B$. In fact, for any $A, B \subseteq \mathbb{N}$,

$$A \leq_m B \iff \neg A \leq_m \neg B.$$

(3) In what sense does $A \leq_m B$ mean that A is simpler than B in computational content? An m-reduction f from A to B reduces the problem of membership in A to the problem of membership in B. If you want to know if some number n is in A, you can compute $f(n)$ and ask if $f(n)$ is a member of B. If $f(n) \in B$, then you know $n \in A$, and if $f(n) \notin B$, then you know $n \notin A$. In other words, if you somehow have full knowledge of membership in B, then you also would have full knowledge of membership of A. Not that it is important here that the m-reduction f is computable and total because it means you don't need any special knowledge to evaluate $f(n)$– just use the algorithm that computes f.

Now, we will look at several examples.

Example 7.2. For any set $A \subseteq \mathbb{N}$, we have $A \leq_m A$ since the identity function on $\mathbb{N}$ is an m-reduction from A to A.

Example 7.3. Let $A := \{0, 2, 4, 6, \dots\}$ be the set of even numbers, and let $B := \{0\}$. Define $f : \mathbb{N} \to \mathbb{N}$ by

$$f(x) = \begin{cases} 0 & \text{if } x \text{ is even} \\ x & \text{if } x \text{ is odd} \end{cases}$$

It is readily verified that f is an m-reduction form A to B. Of course, A and B are very different sets since A is infinite and B is finite. However, from a computability theory perspective they are similar because they are both computable subsets of $\mathbb{N}$. In fact, it is an easy exercise to show that we also have $B \leq_m A$.

The previous example can be generalized as follows. Here and going forward, we will call a subset A of $\mathbb{N}$ *nontrivial* if $A \neq \emptyset$ and $A \neq \mathbb{N}$.

Proposition 7.4. Let A be a recursive subset of $\mathbb{N}$. Then, $A \leq_m B$ for any nontrivial subset B of $\mathbb{N}$.

Proof. Since B is nontrivial, we can pick a $b \in B$ and $c \notin B$. Now, define $f : \mathbb{N} \to \mathbb{N}$ by

$$f(x) = \begin{cases} b & \text{if } x \in A \\ c & \text{if } x \notin A \end{cases}$$

It is easy to see that f is an m-reduction from A to B. □

From the perspective of m-reductions, recursive sets are quite simple. They are m-reducible to almost every other subset B. The following exercise will clarify why B in the previous proposition needs to be nontrivial.

Exercise 7.5. Prove that if $A \leq_m \emptyset$, then $A = \emptyset$. Similarly, if $A \leq_m \mathbb{N}$, then $A = \mathbb{N}$.

Among the r.e. subsets, there are those that are maximally complicated with respect to m-reductions:

Definition 7.6. An r.e. set $B \subseteq \mathbb{N}$ is *m-complete* if $A \leq_m B$ for every r.e. set A.

Theorem 7.7. The (diagonal) halting problem K is m-complete.

Proof. Let $A \subseteq \mathbb{N}$ be r.e. Pick a recursive relation R satisfying

$$x \in A \iff (\exists y)\ R(x, y).$$

Define $g : \mathbb{N}^2 \rightharpoonup \mathbb{N}$ by

$$g(x, e) = \mu y[R(x, y)],$$

which is clearly recursive. By the s-m-n theorem, there is a recursive $f : \mathbb{N} \to \mathbb{N}$ such that

$$\varphi_{f(x)}(e) = g(x, e) \qquad \text{for all } x, e \in \mathbb{N}.$$

We show that f is an m-reduction from A to K. Consider the case where $x \in A$. Then there is witness y such that $R(x, y)$, so that $g(\vec{x}, e)\downarrow$ for every $e \in \mathbb{N}$. In particular, $\varphi_{f(x)}(f(x)) = g(x, f(x))\downarrow$, hence $f(x) \in K$.

Now, consider the case where $x \notin A$ holds. Then, there is no y with $R(x, y)$, hence $g(x, e)\uparrow$ for every $e \in \mathbb{N}$. In particular, $g(x, f(x))\uparrow$, i.e., $\varphi_{f(x)}(f(x))\uparrow$ and $f(x) \notin K$. □

The s-m-n theorem is an invaluable tool in the construction of many m-reductions.

K is not the only r.e. set which is m-complete, as the following exercise establishes.

Exercise 7.8. (a) Let $A, B, C \subseteq \mathbb{N}$. Prove that if $A \leq_m B$ and $B \leq_m C$, then $A \leq_m C$.

(b) Prove that if any r.e. set C with $K \leq_m C$ is an m-complete.

(c) Show that $\{e \in \mathbb{N} : W_e \neq \emptyset\}$ is an m-complete r.e. set.

We will discuss m-complete r.e. sets further in Section **??**, but for now we explore an application of computable reductions. m-reductions play an important role is proving that certain sets are not computable, as well as proving certain sets are not r.e. The key properties that allow this are in the following proposition.

Proposition 7.9. Let $A, B \subseteq \mathbb{N}$ and assume $A \leq_m B$.

(1) If B is computable, then A is computable.

(2) If B is r.e., then A is r.e.

Proof. Both statements follow immediately from the fact that computable relations and semirecursive relations are both closed under recursive total substitution. □

We will often use this proposition in the form of the following corollary. It follows immediately from the contrapositives of the above statements and the fact that K is not computable and $\neg K$ is not r.e.

Corollary 7.10. Let $B \subseteq \mathbb{N}$

(1) If $K \leq_m B$, then B is not computable.

(2) $\neg K \leq_m B$, then B is not r.e.

We now give a few examples of how to use this corollary.

Example 7.11. We will show that

$$B := \{\langle e, n\rangle : \varphi_e(n)\downarrow\}$$

is not computable. By the corollary, all we have to do is show $K \leq_m B$. Define $f : \mathbb{N} \to \mathbb{N}$ by

$$f(e) := \langle e, e \rangle$$

It is easy to see that f is an m-reduction of K to B.

Note that B is not an index set, so we could not have applied Rice's Theorem to B.

Recall that

$$\text{Tot} =_{\text{df}} \{e \in \mathbb{N} : \varphi_e \text{ is total}\}$$

is not recursive by Rice's Theorem. Now, we can use Corollary **??** to show that Tot is not r.e.

Proposition 7.12. The set $\text{Tot} = \{e \in \mathbb{N} : \varphi_e \text{ is total}\}$ is not r.e.

Proof. By Corollary 7.10, it is enough to show that $\neg K \leq_m \text{Tot}$. Since K is r.e., there is a recursive $R(e, y)$ such that

$$e \in K \iff (\exists y)\ R(e, y).$$

Taking negations, we have

$$e \notin K \iff (\forall y)\ \neg R(e, y).$$

Recall that $\neg R$ is also recursive. Define $g : \mathbb{N}^2 \rightharpoonup \mathbb{N}$ by

$$g(e, y) = \begin{cases} 1 & \text{if } \neg R(e, y) \\ \uparrow & \text{otherwise.} \end{cases}$$

g is recursive by Exercise 6.14. For fixed $e \in \neg K$, we have $g(e, y)\downarrow = 1$ for all y; for fixed $e \in K$, there is y with $R(e, y)$, so that we have $g(e, y) \uparrow$.

By the s-m-n theorem, there is a recursive $f : \mathbb{N} \to \mathbb{N}$ such that

$$\varphi_{f(e)}(y) = g(e, y)$$

for all $e, y \in \mathbb{N}$. It is then easy to use our above observations to see that

$$e \in \neg K \iff f(e) \in \text{Tot},$$

so that f is an m-reduction from $\neg K$ to Tot. □

If we examine our proof, we didn't use anything about K other than it is an r.e. set. Thus, we can see that we actually proved something stronger: any co-r.e. set m-reduces to Tot.

Exercise 7.13. Prove that Tot is not co-r.e.

Exercise 7.14. Prove that $A = \{e \in \mathbb{N} : \varphi_e(x)\downarrow = 0 \text{ for all } x \in \mathbb{N}\}$ is not r.e. *Hint.* Since Tot is not r.e., it is enough to show that Tot $\leq_m A$.

7.2 Creative and simple sets

We have seen that K is an m-complete r.e. set, i.e., for any r.e. set $A \subseteq \mathbb{N}$, $A \leq_m K$. Moreover, we have seen many examples of r.e., undecidable sets B which have $K \leq_m B$. As we have remarked in the previous section, such a B is also m-complete. This leads us to ask a natural question:

Question. Does there exist a set $B \subseteq \mathbb{N}$ such that B is r.e., undecidable, but <u>not</u> m-complete?

In this section, we will establish that the answer is that such a set B does, indeed, exist. The definitions and proofs used below are due to Emil Post. First, we establish that all m-complete r.e. sets are *creative*. To define this new notion, we first make the following definition.

Definition 7.15. Let $A \subseteq \mathbb{N}$. A *productive function* for A is a recursive total $p : \mathbb{N} \to \mathbb{N}$ such that for any $e \in \mathbb{N}$,

$$W_e \subseteq A \implies f(e) \in A \setminus W_e.$$

We say that a set is *productive* if it has a productive function.

Let A be productive with productive function p. A cannot be an r.e. set. Indeed, if A is r.e., then $A = W_e$ for some e; in particular, $W_e \subseteq A$, hence $p(e) \in A \setminus W_e$. But this is a contradiction since $A \setminus W_e = A \setminus A = \emptyset$. So, productive sets are not r.e., but they are not r.e. in a strong sense; they have a productive function that computably produces counterexamples to the claim that $A = W_e$ for any r.e. set W_e contained in A.

Definition 7.16. A set $C \subseteq \mathbb{N}$ is *creative* if it is r.e. and its complement is productive.

Note that creative sets cannot be recursive; otherwise, their complements would be r.e., which is impossible since the complement is productive.

We will show that all m-complete sets are creative. Towards this end, we first show that our favorite m-complete set is creative.

Theorem 7.17. K is creative.

Proof. We already know K is r.e., so we only have to show that $\neg K$ is productive. Its productive function is very simple. Define $p : \mathbb{N} \to \mathbb{N}$ to be the identity function.

We claim this is a productive function for $\neg K$. Suppose $W_e \subseteq \neg K$. We show that $p(e) = e \in \neg K \setminus W_e$. Suppose towards a contradiction that $e \in W_e$; this means that $\varphi_e(e)\downarrow$, i.e., $e \in K$. But then $e \in W_e \cap K$, which contradicts our assumption that $W_e \subseteq \neg K$. Thus, we have proved by contradiction that $e \notin W_e$. So, $\varphi_e(e)\uparrow$, which means $e \in \neg K$. We conclude that $e \in \neg K \setminus W_e$, as desired. □

Theorem 7.18. If A is r.e. set and $B \leq_m A$ for some creative set B, then A is creative.

Proof. We only have to show that $\neg A$ has a productive function. Let p be a productive function for $\neg B$, and let $f : \mathbb{N} \to \mathbb{N}$ by an m-reduction from B to A. We will define a productive function for $\neg A$ as follows: given $W_e \subseteq \neg A$, we will pull back W_e using f, so that $f^{-1}[W_e] \subseteq \neg B$ since f is a reduction. Then, we will apply p to an index for the r.e. set $f^{-1}[W_e]$ to get an element $x \in \neg B \setminus f^{-1}[W_e]$. Then, we apply f to this x, so that $f(x) \in \neg A \setminus W_e$, again using the properties of the reduction.

To make the above procedure effective, we must be able to compute an index for $f^{-1}[W_e]$. But this is exactly what is established in Exercise 6.41: there is a computable total $u : \mathbb{N} \to \mathbb{N}$ such that $W_{u(e)} = f^{-1}[W_e]$ for every $e \in \mathbb{N}$. Now, define $q : \mathbb{N} \to \mathbb{N}$ by

$$q(e) =_{\text{df}} f(p(u(e)))$$

It is then easy to check that q is a productive function for $\neg A$. □

The previous two theorems has the following immediate consequence.

Corollary 7.19. Any r.e set which is m-complete is creative.

Thus, we have shown that every m-complete r.e. set is indeed creative. Next, we will show that all creative sets have complements which contain infinite r.e. sets. Later, this will give us a tool to show certain sets are not creative, hence not m-complete.

Theorem 7.20. If $A \subseteq \mathbb{N}$ is productive, then A contains an infinite r.e. set.

Proof. We will define a recursive injective total $f : \mathbb{N} \to \mathbb{N}$ whose image is contained in A. Then, the image is an infinite r.e. subset of A. We define this using the following recursion. First, let $f(0) = p(e_0)$, where $W_{e_0} = \emptyset$.

Then, since $W_{e_0} \subseteq A$, $f(0) \in A \setminus W_{e_0}$. Next, to define $f(1)$ we apply this same procedure to the r.e. set $W_{e_1} = \{f(0)\}$. Since $W_{e_1} \subseteq A$, we have $f(1) = p(e_1) \in A \setminus W_{e_1}$; thus, $f(1)$ is in A, but is different from $f(0)$. We continue this way, by obtaining $f(n+1)$ by applying p to the set $\{f(0), \ldots, f(n)\}$.

To make this into a formal proof, we need to show that we can compute indices for the sets $\{f(0), \ldots, f(n)\}$. So, we establish the following:

Claim. There exists a total recursive $S : \mathbb{N} \to \mathbb{N}$ such that for any positive length sequence code u,

$$W_{S(u)} = \{(u)_0, \ldots, (u)_{\mathrm{lh}(u)-1}\}.$$

The proof of the claim is left as an exercise. Once the claim is established, we are done since we can define our recursive $f : \mathbb{N} \to \mathbb{N}$ by the course-of-values recursion

$$f(0) = e_0, \qquad f(n+1) = p(S(\langle f(0), \ldots, f(n)\rangle))$$

where e_0 is some fixed index for $\emptyset$. □

Exercise 7.21. Prove the claim in the above proof.

Recall that our goal is to find an r.e. set which is not m-complete. We have so far shown that

$$A \text{ } m\text{-complete} \implies A \text{ creative} \implies \neg A \text{ contains an infinite r.e. set.}$$

Thus, if we can find an r.e., non-recursive set whose complement does not contain any infinite r.e. subsets, then that r.e., non-recursive set cannot be m-complete. We give such sets a name.

Definition 7.22. A set A is *simple* if (1) A is r.e.; (2) $\neg A$ is infinite; and (3) $\neg A$ does not contain any infinite r.e. subsets.

It is now a simple exercise to show that a simple set, if it exists, is a solution to our problem.

Exercise 7.23. Show that if $A \subseteq \mathbb{N}$ is simple, then it is an r.e., non-recursive set which is not m-complete.

Remark. We have shown that m-completeness of an r.e. set implies that it is creative. As described above, this is enough to solve our current problem. However, it turns out that for r.e. sets, m-completeness and creativity are actually *equivalent*. This will be prove in the next section.

Now, all that remains is to prove that simple sets exist.

Theorem 7.24. Simple sets exist. Thus, there are r.e., non-recursive sets which are not m-complete.

Proof. We want to build an r.e. set A so that whenever W_e is infinite, there is an element in $A \cap W_e$, so that W_e is not contained in $\neg A$. But, we also need to do this in a way that $\neg A$ is infinite. To this end, we will pick $x \in A \cap W_e$ so that $x > 2e$.

Define a relation R by

$$R(e,x) \iff_{\text{df}} x \in W_e \,\&\, x > 2e.$$

Easily, R is semirecursive. By the semirecursive-selection principal, there is a recursive partial $f : \mathbb{N} \rightharpoonup \mathbb{N}$ such that

$$\begin{aligned} f(e)\!\downarrow &\iff (\exists x) R(e,x) \\ f(e)\!\downarrow &\implies R(e, f(e)). \end{aligned}$$

Set $A = f[\mathbb{N}]$, which is clearly r.e. So, to show that A is simple, we only need to show (1) $\neg A$ does not contain any infinite r.e. sets; and (2) $\neg A$ is infinite.

(1) Suppose W_e is infinite. Then, it must contain elements larger than $2e$. Thus, $(\exists x) R(e,x)$ holds, and so $f(e)\!\downarrow$ and $R(e, f(e))$. This means that $f(e) \in W_e$ and, of course, $f(e) \in A$. Thus, $W_e \not\subseteq \neg A$.

(2) Next, we show that $\neg A$ is infinite. For $k > 0$, consider how many naturals $< 2k$ are in A. If $x < 2k$ is in A, then there is $e \in \mathbb{N}$ with $f(e)\!\downarrow = x$. Since $R(e, f(e))$ holds, $2e < f(e) = x < 2k$, hence $e < k$. Thus, the only elements of $\{0, 1, \ldots, 2k-1\}$ that are in A are of the form $f(e)$ with $e < k$. So, $\{0, 1, \ldots, 2k-1\}$ contains at most $k-1$ elements in A, meaning more than half are elements of $\neg A$. Since $\neg A$ contains at least half of the elements of every set of the form $\{0, 1, \ldots, 2k-1\}$, it follows that $\neg A$ is infinite. □

7.3 1-reductions

A 1-reduction between sets is just an m-reduction which is also injective (i.e., one-to-one).

Definition 7.25. Let $A, B \subseteq \mathbb{N}$. A 1-*reduction* from A to B is a computable injective function $f : \mathbb{N} \to \mathbb{N}$ such that, for all $n \in \mathbb{N}$,

$$n \in A \iff f(n) \in B$$

We say that A is 1-*reducible* to B, and write $A \leq_1 B$, to mean that there is a 1-reduction $f : \mathbb{N} \to \mathbb{N}$ from A to B.

Immediately, 1-reducibility always implies m-reducibility:

$$A \leq_1 B \implies A \leq_m B$$

However, the converse is false in general. In Example 7.3, we showed that

$$\{0, 2, 4, \dots\} \leq_m \{0\}.$$

But, certainly it is not the case that

$$\{0, 2, 4, \dots\} \leq_1 \{0\}$$

since (computability aside) we cannot map an infinite set injectively into a singleton set.

1-reducibility should be thought of as a computability theoretic version of *cardinality*. For sets X and Y, we say the *cardinality of X is less than or equal to the cardinality of Y*, and write $|X| \leq |Y|$, if there is an injective function $f : X \to Y$. For sets $A, B \subseteq \mathbb{N}$, we can think of $A \leq_1 B$ as meaning that the *computable cardinality* of A is less than or equal to the *computable cardinality* of B. This is of course a refinement of the usual notion of cardinality, as the below exercise shows.

Exercise 7.26. Let $A, B \subseteq \mathbb{N}$. Prove that if $A \leq_1 B$, then $|A| \leq |B|$ and $|\neg A| \leq |\neg B|$.

A fundamental fact about cardinality is the following:

Theorem 7.27 (Schroder-Bernstein)**.** Let X and Y be sets. Then, $|X| \leq |Y|$ and $|Y| \leq |X|$ if and only if there is a bijective (one-to-one and onto) function between X and Y.

Of course, the existence of a bijection $h : X \to Y$ immediately implies $|X| \leq |Y|$ (via h) and $|Y| \leq |X|$ (via h^{-1}). The more interesting fact is that, given a injective functions $f : X \to Y$ and $g : Y \to X$, we can always build a bijection $h : X \to Y$. We will prove that the computable version of Schroder-Bernstein is also true. The following is the computable version of a bijection existing between subsets of $\mathbb{N}$:

Definition 7.28. Let $A, B \subseteq \mathbb{N}$. We say A and B are *computably isomorphic* if there exists a 1-reduction $f : \mathbb{N} \to \mathbb{N}$ from A to B which is also onto.

Theorem 7.29 (Myhill)**.** Let $A, B \subseteq \mathbb{N}$. Then $A \leq_1 B$ and $B \leq_1 A$ if and only if A and B are computably isomorphic.

Proof. Let $f : \mathbb{N} \to \mathbb{N}$ be a 1-reduction from A to B and let $g : \mathbb{N} \to \mathbb{N}$ be a 1-reduction from B to A. we want to construct a recursive isomorphism $h : \mathbb{N} \to \mathbb{N}$ from A to B. We will begin by describing out the key ideas.

Let's think of the requirements on h as follows. We need to form a set of pairs of numbers (a, b), which will be the graph of h. For a pair (a, b) in the graph of h, let's say that a is a left-partner of b and b is a right-partner of a. We must meet the following requirements: (1) every $a \in \mathbb{N}$ has exactly one right-partner, (2) every $b \in \mathbb{N}$ has exactly one left-partner, (3) for every paired up (a, b), either $(a \in A \ \& \ b \in B)$ or $(a \notin A \ \& \ b \notin B)$.

Imagine the natural numbers in a line (in order) waiting to be assigned a right-partner and a left-partner. At any point, due to previous assignments made, our next customer might already have a left and/or right-partner, which is fine. Our first customer is 0 asking us for a right-partner, and we give him $f(0)$. Since f is a reduction from A to B, this pair obeys requirement (3) above. For concreteness, let's say $f(0) = 7$, so $(0, 7)$ is our first pair. We also have to give 0 a left-partner. A natural candidate is $g(0)$. If $g(0) \neq 0$, this is fine and we add the pair $(g(0), 0)$. But, if $g(0) = 0$, then putting in the pair $(0, 0)$ would mean 0 has two right-partners, violating requirement (1). So what do we do in this case? Apply f and then g again to $g(0)$, so we're looking $a := g(f(g(0))$. Since we're in the case where $g(0) = 0$, we have $a = g(f(0)) = g(7)$. Since g is injective and $g(0) = 0$, $a = g(7) \neq 0$. Thus, adding the pair $(a, 0)$ does not violate (1) or (2), and ensures that 0 has a left-partner. Next, 1 comes up asking for a right-partner and a left-partner, and we play the same game.

Now, we give the more formal details. We will define by recursion a computable total $H : \mathbb{N} \to \mathbb{N}$ so that for every $n \in \mathbb{N}$, $H(n)$ is a sequence code for a pair and so that $\{((H(n))_0, (H(n))_1 : n \in \mathbb{N}\}$ is the graph of a computable isomorphism from A to B. Then, we can computably define $h : \mathbb{N} \to \mathbb{N}$ by setting $h(x)$ to be the unique y such that $\langle x, y\rangle$ is in the range of H.

Now we describe the recursion to define H. Set $H(0) := \langle 0, f(0)\rangle$. For the recursive step, we distinguish between even and odd cases. For $H(2n+1)$, computably search for the first y such that y is not any of $(H(0))_1, \ldots, (H(2n))_1$, i.e., y does not yet have a left-partner. Then, start computing

$$g(y), gfg(y), gfgfg(y), \ldots, (gf)^i(g(y))$$

until the first time one of these is different from $(H(0))_0, \ldots, (H(2n))_0$ (the numbers which already have right-partners). You will find such a number $x := (gf)^i(g(y))$, for some $i \in \mathbb{N}$, because gf is injective. Then, set $H(2n + 1) := \langle x, y\rangle$.

The case of $H(2n+2)$ is similar. Begin by computably searching for the first x such that x is not any of $(H(0))_0, \ldots, (H(2n+1))_0$, i.e., x does not yet have a right-partner. Then, start computing

$$f(x), fgf(x), fgfgf(x), \ldots, (fg)^i(g(x))$$

until the first time one of these is different from $(H(0))_1, \ldots, (H(2n+1))_1$ (the numbers which already have left-partners). You will find such a number $y := (fg)^i(f(y))$, for some $i \in \mathbb{N}$, because fg is injective. Then, set $H(2n+2) := \langle x, y \rangle$.

It is clear that this a computable process. Using the construction, it is fairly straight-forward to check that requirements (1) and (2) above are met. We have to check requirement (3). This is done by establishing that for any $x, y \in \mathbb{N}$ and any $i \in \mathbb{N}$,

$$y \in B \iff (gf)^i(g(y)) \in A,$$

and

$$x \in A \iff (fg)^i(f(x)) \in B.$$

Both of these are proved by a simple induction, using that f and g are reductions. □

§ 1**-completeness**. We end this section by discussing 1-complete r.e. sets, the 1-reduction analog of m-complete r.e. sets. It turns out that the 1-complete r.e. sets are exactly the same as the m-complete sets and, indeed, they are the same as the creative sets. This is a remarkable theorem of Myhill and we will show how to prove it, although it does require some technicalities.

Definition 7.30. An r.e. set $B \subseteq \mathbb{N}$ is called 1-*complete* if $A \leq_1 B$ for every r.e. set A.

In the definition of productive function, we required the productive function to be total, but not necessarily injective; however, it turns out that productive sets always have injective productive functions.

Lemma 7.31. Every productive set has an injective productive function.

Proof. Let $B \subseteq \mathbb{N}$ be productive. and let $q : \mathbb{N} \to \mathbb{N}$ be a (not necessarily injective) productive function. We will use q to define a new productive function which is injective. First, we can use the s-m-n theorem to get a computable total function $S : \mathbb{N} \to \mathbb{N}$ with

$$W_{S(e)} = W_e \cup \{q(e)\}$$

for all $e \in \mathbb{N}$. Define $p : \mathbb{N} \to \mathbb{N}$ by recursion as follows: begin by setting $p(0) := q(0)$. To define $p(e)$ for $e > 0$, start computing

$$q(e), \quad S(q(e)), \quad S(q(q(e))), \quad \ldots \quad , S(q^i(e))$$

until either:

(1) you find a $S(q^i(e))$, some $i \in \mathbb{N}$, which is different than $p(0), \ldots, p(e-1)$. In this case, set $p(e) := S(q^i(e))$; or,

(2) you compute a $S(q^i(e))$ which is equal to a previous $S(q^j(e))$, $j < i$. In this case, set $p(e)$ to be any number different than $p(0), \ldots, p(e-1)$.

It is clear that p is total and injective. We must show that it is a productive function. Suppose $W_e \subseteq B$. The key is to prove by induction that all the values $S(q^i(e))$, $i \in \mathbb{N}$, are distinct from each other and are in $B \setminus W_e$. Then, we will be in case (1) and we will return a value in $B \setminus W_e$. □

Now, we are ready to prove Myhill's theorem. In the proof, it is important to remember that the functions provided by the second recursion theorem with parameters may be assumed to be injective.

Theorem 7.32 (Myhill)**.** Let $C \subseteq \mathbb{N}$. The following are equivalent.

(1) C is a 1-complete r.e. set.

(2) C is an m-complete r.e. set.

(3) C is creative.

Proof. (1) implies (2) is immediate, and we proved (2) implies (3) in the previous section. So, we only have to prove that (3) implies (1).

Assume C is creative. By the previous lemma, we have a total and injective productive function $p : \mathbb{N} \to \mathbb{N}$ for $\neg C$. Fix an r.e. set $A \subseteq \mathbb{N}$, and we must show $A \leq_1 C$. Our aim is to use the second recursion theorem with parameters to get an injective total function $g : \mathbb{N} \to \mathbb{N}$ which satisfies

$$W_{g(e)} = \begin{cases} \{p(g(e))\} & \text{if } e \in A \\ \emptyset & \text{otherwise.} \end{cases}$$

To do this, first define the computable partial function

$$f(z, e, x) = \begin{cases} 0 & \text{if } e \in A \text{ and } x = p(z) \\ \uparrow & \text{otherwise.} \end{cases}$$

By the second recursion theorem with parameters, there is a recursive total $g : \mathbb{N} \to \mathbb{N}$ which is injective such that

$$\varphi_{g(e)}(x) = f(g(e), e, x)$$

for all $e, x \in \mathbb{N}$. It is easy to verify that this is our desired g.

Now, we show that $p \circ g$ is a 1-reduction from A to C. It is obviously computable, total, and injective, since it is the composition of two computable, total, and injective functions. If $e \notin A$, then $W_{g(e)} = \emptyset \subseteq \neg C$, hence $p(g(e)) \in \neg C$ (using that p is a productive function for $\neg C$). Now let $e \in A$ and suppose towards a contradiction that $p(g(e)) \notin C$. Then, since $e \in A$, $W_{g(e)} = \{p(g(e))\}$, which is a subset of $\neg C$. Thus,

$$p(g(e)) \in \neg C \setminus W_{g(e)} = \neg C \setminus \{p(g(e))\},$$

which is absurd. □

7.4 Additional exercises

Exercise 7.33. (a) Show that $A \not\leq_m \emptyset$ for any $A \neq \emptyset$ and that $\emptyset \leq_m B$ for any $B \neq \mathbb{N}$. Then, show that $A \not\leq_m \mathbb{N}$ for any $A \neq \mathbb{N}$ and that $\mathbb{N} \leq_m B$ for any $B \neq \emptyset$. (In particular, $\emptyset$ and $\mathbb{N}$ are both $\leq_m$-minimal and they are $\leq_m$-incomparable to each other.)

(b) Call a subset of A *non-trivial* if $A \neq \emptyset$ and $A \neq \mathbb{N}$. Show that if A, B are non-trivial subsets with A recursive, then $A \leq_m B$.

Exercise 7.34. Prove that the set

$$A =_{\text{df}} \{e \in \mathbb{N} : \varphi_e(x)\!\downarrow = 0 \text{ for all } x \in \mathbb{N}\}$$

is not r.e. *Hint.* It is enough to show $\text{Tot} \leq_m A$.

Exercise 7.35. Prove the the binary relation

$$R(e, z) \quad \iff_{\text{df}} \quad W_e \subseteq W_z$$

is not r.e. *Hint.* It is enough to computably reduce Tot to R.

Exercise 7.36. Let A and C be subsets of $\mathbb{N}$. Prove that if C is creative, A is r.e., and $A \cap C = \emptyset$, then $A \cup C$ is creative.

Exercise 7.37. Prove that if A and B are simple sets, then

$$A \oplus B =_{\text{df}} \{2x : x \in A\} \cup \{2x+1 : x \in B\}$$

is also simple.

Exercise 7.38. Let $f : \mathbb{N} \to \mathbb{N}$ be a recursive injective total function whose range is not recursive. Prove that the set

$$A =_{\text{df}} \{x \in \mathbb{N} : (\exists y)[y > x \ \&\ f(y) < f(x)]\}$$

is simple.

Hint. To prove that $\neg A$ is infinite, suppose towards a contradiction that it is finite and use this assumption to build an infinite sequence $y_0 < y_1 < \cdots$ such that $f(y_0) > f(y_1) > \cdots$. To prove that $\neg A$ does not contain any infinite r.e. subsets B, work by contradiction and assume we have such a $B \subseteq \neg A$; use this assumption to show that range(f) is decidable. To decide if $z \in \text{range}(f)$, find an $n \in B$ such that $f(n) > z$, then argue that since $n \notin A$, we have a computable way to check if $z \in \text{range}(f)$.

Exercise 7.39. Let A and B be disjoint subsets of $\mathbb{N}$. Prove that if A is productive and B is recursive, then $A \cup B$ is productive.

Exercise 7.40. Let $\text{Sur} = \{e \in \mathbb{N} : \varphi_e \text{ is surjective}\}$ and let $\text{Tot} = \{e \in \mathbb{N} : \varphi_e \text{ is total}\}$.

(a) Find a computable total function $f : \mathbb{N} \to \mathbb{N}$ which computably reduces Sur to Tot.

(b) Find a computable total function $g : \mathbb{N} \to \mathbb{N}$ which computably reduces Tot to Sur.

8 Relative computability

In Ancient Greece, an oracle was a person who was believed to be able to divine the answers to seemingly unanswerable questions. In this section, we will consider what we would be able to compute if we had access to an oracle that knows uncomputable information. In our modern context, we could also think of the oracle as some mysterious machine that somehow is able to compute information that we normally cannot.

Formally speaking, our oracles will not be people or mysterious machines, but rather they will be total functions $\alpha : \mathbb{N} \to \mathbb{N}$. Note that we can also think of α as an infinite sequence of numbers,

$$(\alpha(0), \alpha(1), \alpha(2), \dots).$$

Intuitively speaking, in a computation with oracle α, we are able to do all of our usual operations, but in addition we can also ask the oracle for a particular value $\alpha(n)$, and the oracle will always provide the correct value when asked. This type of computation is called a *computation relative to* α.

If the oracle α is not a computable function, then we have increased our computing power since we have access to uncomputable information. Consider, for example, using oracle $\alpha =_{\text{df}} \text{Char}_K$. With this oracle, we can now compute Char_K itself; to compute $\text{Char}_K(n)$ relative to the oracle, just ask the oracle for its nth value.

Relative computation does not have practical applications to, say, computer science; we do not have any realistic expectation that we will ever have access to an oracle that can provide uncomputable information. Rather, relative computation has applications to pure mathematics. Within computability theory, we will see that oracle computation can help us compare how complicated functions, sets, and relations are relative to each other. In particular, we can ask questions like does function α compute function β?

Oracle computability also has applications outside of logic; for instance, with the correct definitions for functions on $\mathbb{R}$, we can establish

$$f : \mathbb{R} \to \mathbb{R} \text{ is continuous } \iff f \text{ is computable relative to some oracle.}$$

This is outside the scope of these notes, but this connection between topology and computability theory has many important applications.

8.1 α-recursive partial functions

Like the computable functions, the α-computable functions have many equivalent definitions. These definitions are all "relativized" versions of the definitions of computable functions. We will start with the relativized version of recursive functions.

Recall that the class of recursive partial functions $\mathcal{R}$ is the smallest class which is recursively closed, i.e., contains the successor function, the projection functions, the constant functions, and is closed under definition by substitution, definition by primitive recursion, and definition by minimization.

For an oracle $\alpha : \mathbb{N} \to \mathbb{N}$, the α-recursive functions form the smallest class which, in addition to all the properties of a recursively closed class, also contains α itself.

Definition 8.1. Let $\alpha : \mathbb{N} \to \mathbb{N}$ be a total function. A class $\mathcal{C}$ of partial functions is *α-recursively closed* if it has the following properties:

(i) $\mathcal{C}$ contains the successor function, all the projection functions, and all the constant functions.

(ii) $\mathcal{C}$ contains α.

(iii) $\mathcal{C}$ is closed under definition by substitution, definition by primitive recursion, and definition by minimization.

A partial function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ is *α-recursive* if it belongs to every α-recursively closed class. In other words, the class $\mathcal{R}(\alpha)$ of α-recursive partial functions is the smallest α-recursively closed class of partial functions.

Note that every α-recursively closed class is also recursively closed. Thus, we easily have the following fact.

Proposition 8.2. Let α be an oracle. Every recursive partial function is also α-recursive.

This provides us with many basic examples of α-recursive partial functions. But, in general, there will be α-recursive partial functions which are not recursive:

Example 8.3. α is α-recursive. Thus, if $\alpha : \mathbb{N} \to \mathbb{N}$ is not recursive, then there are α-recursive partial functions which are not recursive.

On the other hand, if our oracle α is computable, then the α-recursive partial functions are exactly the same as the recursive partial functions.

Exercise 8.4. Show that if α is recursive, then $\mathcal{R}(\alpha) = \mathcal{R}$.

This is a special case of a more general fact:

Proposition 8.5. Suppose that $\alpha, \beta : \mathbb{N} \to \mathbb{N}$ are total functions. If α is β-recursive, then $\mathcal{R}(\alpha) \subseteq \mathcal{R}(\beta)$.

Proof. Since α is β-recursive, we have $\alpha \in \mathcal{R}(\beta)$. It follows that $\mathcal{R}(\beta)$ is α-recursively closed. Since $\mathcal{R}(\alpha)$ is the smallest α-recursively closed class of partial functions, we conclude that $\mathcal{R}(\alpha) \subseteq \mathcal{R}(\beta)$. □

In addition to α-recursive partial functions, we have α-recursive relations:

Definition 8.6. An n-ary relation R on $\mathbb{N}$ is *α-recursive* or *α-decidable* if Char_R is α-recursive.

Exercise 8.7. Prove that if α is recursive, then any α-recursive relation is also recursive.

Exercise 8.8. For any set $A \subseteq \mathbb{N}$, show that A is Char_A-recursive.

As we will see, although in general the classes of α-recursive partial functions and relations are strictly larger than the classes of recursive partial functions and relations, their theory is quite similar. In fact, even the proofs are similar. This general phenomenon is known as "relativization", and so in this language we say that the theory and the proofs relativize to any oracle.

For example, Theorem 2.11 relativizes to any oracle:

Exercise 8.9. Let $f : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ be an α-recursive partial function. Define $g : \mathbb{N}^n \rightharpoonup \mathbb{N}$ by setting $g(\vec{x}, 0) = 0$ and

$$g(\vec{x}, y+1)\downarrow \iff f(\vec{x}, 0)\downarrow \,\&\, f(\vec{x}, 1)\downarrow \,\&\, \cdots \,\&\, f(\vec{x}, y)\downarrow$$
$$g(\vec{x}, y+1)\downarrow \implies g(\vec{x}, y+1) = \sum_{i=0}^{y} f(\vec{x}, i).$$

Prove that g is also α-recursive.

Then, formulate and prove a relativized version of part (b) of Theorem 2.11.

The proofs for this exercise are practically identical to their non-relativized counterparts; this is because the class of α-recursive partial functions is also closed under primitive recursion.

Most of the theory of decidable relations also relativizes to any oracle:

Theorem 8.10. The class of α-recursive relations is closed under $\neg$, &, $\vee$, $\rightarrow$, and bounded existential quantification of both kinds.

We will give one example of how to relativize proofs by showing that α-recursive relations are closed under &.

Let R and Q be α-recursive n-ary relations. Recall that

$$\mathrm{Char}_{R\&Q}(\vec{x}) = \mathrm{Char}_R(\vec{x}) \cdot \mathrm{Char}_Q(\vec{x}).$$

Thus, $\mathrm{Char}_{R\&Q}$ is defined via a substitution from multiplication and Char_R and Char_Q. Since these functions are α-recursive, $\mathrm{Char}_{R\&Q}$ is also α-recursive by closure under definition by substitution.

Exercise 8.11. Prove the closure under bounded quantification of both types.

Exercise 8.12. Prove that a total function $f : \mathbb{N}^n \to \mathbb{N}$ is α-recursive if and only if its graph relation $G_f(\vec{x}, y)$ is α-recursive.

Exercise 8.13. Formulate and prove a version of Theorem 2.28 regarding closure under definition by recursive cases for α-recursion.

8.2 α-URM-computable functions

Here, we study the relativized version of URM-computable functions. The URM relative to an oracle α is exactly the same as the regular URM, except it has one additional type of instruction.

The instruction $O(n)$ takes the content r_n of register R_n and replaces it with $\alpha(r_n)$. For example,

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	
9	1	0	2	4	1	0	11	2	$\cdots$

$$\Downarrow O(3)$$

R_0	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	
9	1	0	$\alpha(2)$	4	1	0	11	2	$\cdots$

An oracle URM program is a finite, ordered list of zero, successor, transfer, jump, and oracle instructions. Note that programs do depend on a particular

choice of oracle; any program can be used with any oracle. For a program P, we write P^α when we are thinking of the program P running with oracle α.

We define α-URM-computable functions in the same way as before. Let $P^\alpha(a_1, \ldots, a_n)$ denote the computation of the URM with oracle α, using oracle program P, with initial register state $(a_1, a_2, \ldots, a_n, 0, 0, \ldots)$. The notation

$$P^\alpha(a_1, \ldots, a_n)\downarrow, \qquad P^\alpha(a_1, \ldots, a_n)\downarrow = y$$

is defined analogously to regular URM's.

Definition 8.14. A partial function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is *α-URM-computable* if there is a oracle URM program P such that

$$f(\vec{x})\downarrow = y \iff P^\alpha(a_1, \ldots, a_n)\downarrow = y.$$

We say that the program P *α-computes* f.

Exercise 8.15. Write a program P which α-computes the oracle α.

As we foreshadowed above, the α-URM-computable partial functions are exactly the α-recursive partial functions.

Theorem 8.16. A partial function is α-recursive if and only if it is α-URM-computable.

As before, we also use the term *α-computable* to refer to α-recursive partial functions and relations.

The proof of the Theorem is very similar to the proof the non-relativized version. We can do some oracle URM programming to show that the class of α-URM-computable partial functions is α-recursively closed. This shows that the class of α-recursive partial functions is included in the class of α-URM-computable partial functions.

For the other inclusion, we numerically code URM computations much in the same way as before. We must add a coding for oracle instructions,

$$\#O = 4, \qquad \#O(j) = \langle \#O, j \rangle.$$

It is then obvious how to code oracle URM programs. The codes for machines states must then be updated to account for the presence for oracle instructions. Then we must also update the relation CmpSt, which arithmatizes the operation of the machine. As we defined it previously, CmpSt includes clauses that coded how every instruction works; to these clauses, we must add the

disjunct

$$\begin{aligned}\mathrm{Ins}(u) = \#O(j) \,\&\, \mathrm{Reg}(v,j) = \alpha(\mathrm{Reg}(u,j)) \\ \&\, (\forall i < \ell)(i \neq j \to \mathrm{Reg}(u,i) = \mathrm{Reg}(v,i)) \\ \&\, \mathrm{InsNum}(v) = \mathrm{InsNum}(u) + 1\end{aligned}$$

which codes how oracle instructions work. This results in a new relation CmpSt^α, and its definition depends on the oracle α. It is α-recursive since it is defined in terms of recursive relations and the graph of α, which is α-recursive.

This proof provides us with a relativized version of the Kleene T-predicate:

Theorem 8.17. Let α be an oracle and let $n > 0$. There is an α-recursive relation $T_n^\alpha(e, \vec{x}, y)$ and a recursive $U : \mathbb{N} \to \mathbb{N}$ such that for any α-recursive partial function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$, there exists e such that

$$f(\vec{x}) U(\mu y\ T_n^\alpha(e, \vec{x}, y)).$$

This theorem also gives us an enumeration of the α-recursive partial functions. For n, e, define

$$\varphi_e^{(n),\alpha}(\vec{x}) =_{\mathrm{df}} U(\mu y\ T_n^\alpha(e, \vec{x}, y)).$$

We denote the enumeration of the unary α-recursive partial functions by

$$\varphi_0^\alpha, \varphi_1^\alpha, \varphi_2^\alpha, \ldots.$$

These enumerations are also effective in the same sense as for the non-relativized case: for any $n > 0$, there is an α-recursive $\psi_U^{(n),\alpha} : \mathbb{N}^{n+1} \rightharpoonup \mathbb{N}$ such that for any $e \in \mathbb{N}$ and $\vec{x} \in \mathbb{N}^n$,

$$\psi_U^{(n),\alpha}(e, \vec{x}) =_{\mathrm{df}} \varphi_e^{(n),\alpha}(\vec{x}).$$

Also, the enumerations satisfies a version of the s-m-n theorem:

Theorem 8.18. Let $f : \mathbb{N}^{n+m} \rightharpoonup \mathbb{N}$ be α-recursive. Then, there is a recursive total function $u : \mathbb{N}^n \to \mathbb{N}$ such that

$$\varphi_{u(\vec{x})}^{(m),\alpha}(\vec{y}) = f(\vec{x}, \vec{y})$$

for any $\vec{x} \in \mathbb{N}^n$ and $\vec{y} \in \mathbb{N}^m$.

Note that the code function u is, in fact, recursive. This is because u just manipulates the codes of programs and does not need access to information

about the oracle α. This version of the s-m-n theorem can be proved using a similar method as before.

8.3 α-semirecursive relations

We can also relativize the notion of α-semirecursive relations. An n-ary relation R is *α-semirecursive* if there is an α-recursive $(n+1)$-ary relation R_0 such that

$$R(\vec{x}) \iff (\exists y)\ R_0(\vec{x}, y)$$

for every $\vec{x} \in \mathbb{N}^n$. We also call a set $A \subseteq \mathbb{N}$ *α-recursively enumerable* (abbreviated α-r.e.) if A is α-semirecursive (as a unary relation on $\mathbb{N}$).

All of the basic properties of α-semirecursive sets relativize with proofs nearly identical the the non-relativized versions:

Theorem 8.19. Let $\alpha : \mathbb{N} \to \mathbb{N}$ be an oracle.

(i) Every α-recursive relation is α-semirecursive.

(ii) The α-semirecursive relations are closed under &, $\vee$, bounded quantification of both types over $\mathbb{N}$, unbounded existential quantification over $\mathbb{N}$, and α-recursive substitution.

(iii) A partial function $f : \mathbb{N}^n \rightharpoonup \mathbb{N}$ is α-recursive if and only if its graph relation G_f is α-semirecursive.

(iv) A relation R is α-recursive if and only if R and $\neg R$ are both α-semirecursive.

(v) A relation is α-semirecursive if and only if it is the domain of an α-recursive partial function.

(vi) A subset $A \subseteq \mathbb{N}$ is α-r.e. if and only if $A = \emptyset$ or it is enumerated by an α-recursive total function.

(vii) A subset $A \subseteq \mathbb{N}$ is α-r.e. if and only if it is the range of an α-recursive partial function.

(viii) A subset $A \subseteq \mathbb{N}$ is α-r.e. if and only if A is finite or it is enumerated without repetitions by an α-recursive total function.

We will not prove every statement here, mostly because the proofs are so easily relativized from the proofs we have already seen. We will give one example of relativizing of proofs by proving that α-semircursive relations are closed under &:

Let R and Q be α-recursive n-ary relations. Pick α-recursive $(n+1)$-ary relations R_0 and Q_0 satisfying

$$R(\vec{x}) \iff (\exists y) R_0(\vec{x}, y), \qquad Q(\vec{x}) \iff (\exists y) Q_0(\vec{x}, y).$$

The equivalences

$$\begin{aligned} R(\vec{x}) \,\&\, Q(\vec{x}) &\iff \big((\exists y) R_0(\vec{x}, y)\big) \,\&\, \big((\exists y) Q_0(\vec{x}, y)\big) \\ &\iff (\exists u)[R_0(\vec{x}, (u)_0) \,\&\, Q_0(\vec{x}, (u)_1)] \end{aligned}$$

are valid for the same reasons as in the proof that semirecursive relations are closed under &. Now, we can define

$$P(\vec{x}, u) \iff_{\text{df}} R_0(\vec{x}, (u)_0) \,\&\, Q_0(\vec{x}, (u)_1),$$

which is α-recursive since α-recursive relations are closed under total α-recursive substitutions and &. Thus,

$$(R \,\&\, Q)(\vec{x}) \iff (\exists u) P(\vec{x}, u),$$

which shows that $R \,\&\, Q$ is α-semirecursive.

Compare this proof with the proof of Theorem 6.4. You will see it is exactly the same, just substituting the relativized notions in for the non-relativized notions.

Exercise 8.20. Prove (iii) of Theorem 8.19. You may assume that (i) and (ii) are true.

For $n > 0$ and $e \in \mathbb{N}$, we define

$$W_e^{(n),\alpha} =_{\text{df}} \text{domain}(\varphi_e^{(n),\alpha}), \qquad E_e^{(n),\alpha} =_{\text{df}} \text{range}(\varphi_e^{(n),\alpha})$$

As in the non-relative case, this give us an enumeration of the α-semirecursive n-ary relations,

$$W_0^{(n),\alpha}, W_1^{(n),\alpha}, W_2^{(n),\alpha}, \ldots$$

This enumeration is effective in the sense that the relation

$$V_U^{(n),\alpha}(e, \vec{x}) \iff_{\text{df}} W_e^{(n),\alpha}(\vec{x})$$

is α-semirecursive. Similar to before, when $n = 1$, we use the notation

$$W_e^{\alpha}, \qquad V_U^{\alpha}.$$

Exercise 8.21. Formulate and prove the statement that the union operation on α-r.e. sets is effective in the codes. Can the "code function" be recursive, or can we only say it is α-recursive in general?

§ **The Turing jump of α.** For any α, the diagonal halting problem relative to α is the set

$$K^\alpha =_{\text{df}} \{e \in \mathbb{N} : \varphi_e^\alpha(e)\downarrow\}.$$

K^α is also called the *Turing jump of* α.

Most of the properties of K relativize to K^α. For instance, K^α is α-r.e. since

$$e \in K^\alpha \iff (\exists y)\ T_1^\alpha(e, e, y),$$

where T_1^α is α-recursive. Moreover, K^α is not α-recursive. This can be proved by relativing the diagonalization argument we used for K.

Exercise 8.22. Prove that K^α is not α-recursive.

Another property that easily relativizes to K^α is m-completeness.

Theorem 8.23. If R is an α-semirecursive relation, then $R \leq_m K^\alpha$.

Exercise 8.24. Prove Theorem 8.23.

We will discuss Turing jumps more in the next section.

8.4 Turing reducibility

When discussing relative computability, we often use the name of a set in place of its characteristic function. For example, for a set $A \subseteq \mathbb{N}$, we say that a partial function is A-recursive if it Char_A-recursive. Similarly, we say that a relation is A-recursive (respectively, A-semirecursive) if it is Char_A-recursive (respectively, Char_A-semirecursive).

Relative computability gives us another way to compare the complexity of sets:

Definition 8.25. Let $A, B \subseteq \mathbb{N}$. We say that A is *Turing reducible* to B and write $A \leq_T B$ if A is B-recursive, i.e.,

$$A \leq_T B \iff A \text{ is computable relative to } B.$$

We also use the notation

$$A <_T B \iff_{\text{df}} A \leq_T B \,\&\, B \not\leq_T A.$$

We say that A and B are *Turing equivalent* and write $A \equiv_T B$ if $A \leq_T B$ and $B \leq_T A$.

Theorem 8.26. For every $A, B, C \subseteq \mathbb{N}$,

(i) $A \leq_T A$;

(ii) if $A \leq_T B$ and $B \leq_T C$, then $A \leq_T C$.

Proof. (i) is trivial. For (ii), let $\alpha = \text{Char}_A$, $\beta = \text{Char}_B$, and $\gamma = \text{Char}_C$. By our hypothesis, α is β-recursive and β is γ-recursive. By two applications of Proposition 8.5, we have $\mathcal{R}(\alpha) \subseteq \mathcal{R}(\beta)$ and $\mathcal{R}(\beta) \subseteq \mathcal{R}(\gamma)$. Thus, $\text{Char}_A = \alpha \in \mathcal{R}(\alpha) \subseteq \mathcal{R}(\gamma)$, hence $A \leq_T C$. □

Exercise 8.27. (a) If $A \subseteq \mathbb{N}$ is recursive and $B \subseteq \mathbb{N}$ is any set, then show $A \leq_T B$.

(b) Show that if $A \leq_T B$ and B is recursive, then A is recursive.

Example 8.28. For every $A \subseteq \mathbb{N}$, $\neg A \leq_T A$. Indeed, since Char_A is clearly A-recursive, it follows that

$$\text{Char}_{\neg A} = 1 \dot{-} \text{Char}_A$$

is also A-recursive by closure under α-recursive substitution.

Note that applying the statement to $\neg A$ in place of A, we have $\neg(\neg A) \leq_T \neg A$. Thus, for any $A \subseteq \mathbb{N}$, we have $\neg A \equiv_T A$

The relation $\leq_T$ on subsets of $\mathbb{N}$ helps us compare the complexity of sets. However, we already studied another way to compare the complexity of sets,

$$A \leq_m B \iff \text{there is an } m\text{-reduction from } A \text{ to } B.$$

$\leq_T$ and $\leq_m$ are related in some ways, but they are different from each other.

Theorem 8.29. Let $A, B \subseteq \mathbb{N}$. Then,

$$A \leq_m B \implies A \leq_T B.$$

Proof. Let $f : \mathbb{N} \to \mathbb{N}$ be a recursive function such that

$$x \in A \iff f(x) \in B.$$

Since B is B-recursive and the class of B-recursive relations is closed under B-recursive substitution, it follows that A is B-recursive. □

We make a few remarks on Theorem 8.29.

(1) Essentially the same proof can be used to prove a more general fact: if R is an n-ary relation which is computably reducible to B, then R is a B-recursive relation.

(2) The converse of Theorem 8.29 is <u>not</u> true:

Example 8.30. By Example 8.28, $\neg K \leq_T K$. However, $\neg K \not\leq_m K$; this is because K semirecursive and $\neg K \leq_m K$ would imply that $\neg K$ is semirecursive, which it is not.

Note that this example shows that, in general, $A \leq_T B$ and B semirecursive does <u>not</u> imply that A is semirecursive.

Exercise 8.31. Show that if A is r.e., then $A \leq_T K$. For this reason, we say that K is *T-complete for r.e. sets*.

§ **The Turing Jump of a set**. For $\alpha : \mathbb{N} \to \mathbb{N}$, we defined the diagonal halting problem relative to α,

$$K^\alpha =_{\text{df}} \{e \in \mathbb{N} : \varphi_e^\alpha(e)\downarrow\}.$$

Recall in section 8.3 we showed that K^α is α-r.e. and in an exercise it was established that K^α is not α-recursive.

By our usual notational convention, for a set $A \subseteq \mathbb{N}$ we write

$$K^A = K^{\text{Char}_A} = \{e \in \mathbb{N} : \varphi_e^A(e)\downarrow\}.$$

K^A is called the *Turing jump of A*. By the previously mentioned facts about the relativized halting problem, we have $K^A \not\leq_m A$. But since A is clearly A-r.e., it follows by Theorem 8.23 that $A \leq_m K^A$. Thus, we have established the following fact.

Theorem 8.32. For any $A \subseteq \mathbb{N}$, $A <_T K^A$.

Next, we show that the Turing jump respects the ordering $\leq_T$.

Theorem 8.33. Let $A, B \subseteq \mathbb{N}$.

(i) If $A \leq_T B$, then $K^A \leq_T K^B$.

(ii) If $A \equiv_T B$, then $K^A \equiv_T K^B$.

Proof. (ii) clearly follows from (i), so we just prove (i). Suppose $A \leq_T B$, so that every A-recursive partial function is B-recursive. In particular, the universal function $\psi_U^A : \mathbb{N}^2 \rightharpoonup \mathbb{N}$,

$$\psi_U^A(e, x) = \varphi_e^A(x)$$

is B-recursive. By the s-m-n theorem, there is a recursive $f : \mathbb{N} \to \mathbb{N}$ such that

$$\varphi_{f(e)}^B(x) \cong \psi_U^A(e, x) = \varphi_e^A(x)$$

for all $e, x \in \mathbb{N}$. Then,

$$e \in K^A \iff \varphi_e^A(e)\downarrow \iff \varphi_{f(e)}^B(e)\downarrow,$$

which shows that K^A is B-semirecursive. By Theorem 8.23, $K^A \leq_m K^B$, hence $K^A \leq_T K^B$. □

It is not true in general that $A <_T B$ implies $K^A <_T K^B$. Indeed, there are non-recursive sets B with $K^B \equiv_T K$. Thus, $\emptyset <_T B$ but $K^\emptyset \equiv_T K \equiv_T K^B$. The proof that such sets exists is beyond the methods we have at our disposal.

§ **Turing degrees**. The *Turing degree* of a set A is the collection of all sets which are Turing equivalent to A,

$$d_T(A) =_{\text{df}} \{B \subseteq \mathbb{N} : A \equiv_T B\}.$$

$\equiv_T$ is an *equivalence* relation on the collection of subsets of $\mathbb{N}$, and $d_T(A)$ is the *equivalence class* of A.

Exercise 8.34. Let $A, B \subseteq \mathbb{N}$. Prove that either $d_T(A) = d_T(B)$ or $d_T(A) \cap d_T(B) = \emptyset$.

We use bold letters like $\boldsymbol{a}, \boldsymbol{b}$, etc., to denote arbitrary Turing degrees.

Proposition 8.35. Let $\boldsymbol{a}, \boldsymbol{b}$ be Turing degrees. The following are equivalent:

(i) there exists $A \in \boldsymbol{a}$ and $B \in \boldsymbol{b}$ with $A \leq_T B$.

(ii) for any $A \in \boldsymbol{a}$ and any $B \in \boldsymbol{b}$, we have $A \leq_T B$.

Proof. (ii) obviously implies (i). So assume (i), that there exists $A \in \boldsymbol{a}$ and $B \in \boldsymbol{b}$ with $A \leq_T B$. Let $A_0 \in \boldsymbol{a}$ and $B_0 \in \boldsymbol{b}$, so that $A_0 \equiv_T A$ and $B_0 \equiv_T B$. By several applications of Proposition 8.5, it follows that

$$\mathcal{R}(A_0) = \mathcal{R}(A) \subseteq \mathcal{R}(B) = \mathcal{R}(B_0).$$

Since $\text{Char}_{A_0} \in \mathcal{R}(A_0)$, we have $\text{Char}_{A_0} \in \mathcal{R}(B_0)$, hence $A_0 \leq_T B_0$. □

We can define $\leq_T$ on Turing degrees by

$$\begin{aligned} \boldsymbol{a} \leq_T \boldsymbol{b} &\iff_{\text{df}} (\exists A \in \boldsymbol{a})(\exists B \in \boldsymbol{b})\ A \leq_T B \\ &\iff (\forall A \in \boldsymbol{a})(\forall B \in \boldsymbol{b})\ A \leq_T B. \end{aligned}$$

Exercise 8.36. Show that $\leq_T$ is a partial ordering of the Turing degrees, i.e., show that $\leq_T$ is reflexive ($\boldsymbol{a} \leq_T \boldsymbol{a}$ for all $\boldsymbol{a}$), transitive ($\boldsymbol{a} \leq_T \boldsymbol{b}$ and $\boldsymbol{b} \leq_T \boldsymbol{c}$ imply $\boldsymbol{a} \leq_T \boldsymbol{c}$), and antisymmetric ($\boldsymbol{a} \leq_T \boldsymbol{b}$ and $\boldsymbol{b} \leq_T \boldsymbol{a}$ imply $\boldsymbol{a} = \boldsymbol{b}$).

In the ordering $\leq_T$, the smallest degree is

$$\boldsymbol{0} = d_T(\emptyset).$$

Exercise 8.37. Show that $\boldsymbol{0} = \{A \subseteq \mathbb{N} : A \text{ is recursive}\}$ and that $\boldsymbol{0} \leq_T \boldsymbol{a}$ for all Turing degrees $\boldsymbol{a}$.

On the other hand, there is no largest Turing degree. To see this, first consider a Turing degree $\boldsymbol{a} = d_T(A)$. Define

$$\boldsymbol{a}' = d_T(K^A).$$

Since $A <_T K^A$, it follows that $\boldsymbol{a} <_T \boldsymbol{a}'$. Recall that for any $A \equiv_T B$, we have $K^A \equiv_T K^B$. This means that for any $A, B \in \mathbf{a}$, we have $K^A \equiv K^B$, so our definition of $\boldsymbol{a}' = d_T(K^A)$ does not depend on the choice of $A \in \boldsymbol{a}$. $\boldsymbol{a}'$ is called the *Turing jump* of the degree $\boldsymbol{a}$. We can keep iterating the Turing jump to get an infinite $<_T$-increasing sequence of degrees:

$$\boldsymbol{a} <_T \boldsymbol{a}' <_T \boldsymbol{a}'' <_T \cdots <_T \boldsymbol{a}^{(n)} <_T \cdots .$$

8.5 Computation relative to finite sequences

In relative computability, we have used oracles $\alpha : \mathbb{N} \to \mathbb{N}$ which are infinite sequences of natural numbers. In this section, we would like to make sense of the idea of computation relative to finite sequences $\sigma : \{0, 1, \ldots, k-1\} \to \mathbb{N}$.

Computation relative to finite sequences helps us approximate oracles in complicated constructions.

Towards this goal, we first introduce some notation for finite sequences. If $\sigma : \{0, 1, \ldots, k-1\} \to \mathbb{N}$ is a finite sequence, then its length is

$$|\sigma| =_{\text{df}} k.$$

If σ and τ are both finite sequences, then we write

$$\sigma \sqsubseteq \tau \iff_{\text{df}} |\sigma| \leq |\tau| \,\&\, (\forall i < |\sigma|)\ \sigma(i) = \tau(i).$$

When $\sigma \sqsubseteq \tau$, we say that τ *extends* σ. If τ *strictly* extends σ, we write

$$\sigma \sqsubset \tau \iff_{\text{df}} \sigma \sqsubseteq \tau \,\&\, \sigma \neq \tau.$$

We also use this notation to when τ is an infinite sequence. If $\alpha : \mathbb{N} \to \mathbb{N}$, then

$$\sigma \sqsubset \alpha \iff (\forall i < |\sigma|)\ \sigma(i) = \alpha(i)$$

and say that α is an *infinite extension* of σ.

Next, we define the *use* of a convergent computation. Consider a convergent computation $\varphi_e^\alpha(x)\!\downarrow$. The *use* of this computation is the least $s \in \mathbb{N}$ so that every oracle call $\alpha(n)$ made in the computation has $n < s$. We denote the use by $\text{use}(\alpha; e, x)$. Note that if no oracle calls are made, then $\text{use}(\alpha; e, x) = 0$. Also, $\text{use}(\alpha; e, x)$ is undefined when $\varphi_e^\alpha(x)\!\uparrow$.

Definition 8.38. Let σ be a finite sequence of natural numbers. We define

$$\varphi_e^\sigma(x)\!\downarrow = y \iff_{\text{df}} \text{there is an infinite } \alpha \sqsupset \sigma \text{ with } \varphi_e^\alpha(x)\!\downarrow = y \text{ and } \text{use}(\alpha; e, x) \leq |\sigma|$$

Theorem 8.39 (Finite use principle I)**.** Let σ be a finite sequence with $\varphi_e^\sigma(x)\!\downarrow = y$. Then, $\varphi_e^\beta(x)\!\downarrow = y$ for all infinite $\beta \sqsupset \sigma$.

Proof. Since $\varphi_e^\sigma(x)\!\downarrow = y$, there is an infinite $\alpha \sqsupset \sigma$ with $\varphi_e^\alpha(x)\!\downarrow = y$ and $\text{use}(\alpha; e, x) \leq |\sigma|$. Let P be the (relative) URM program with code e. Let $P^\alpha(x)$ denote the computation of P at x with oracle α. By our assumption, this computation converges with output y.

Consider an infinite sequence $\beta \sqsupset \sigma$. We must show that $\varphi_e^\beta(x)\!\downarrow = y$. Consider the computation $P^\beta(x)$ of program P at x with oracle β. An induction argument can show that each step of the computation $P^\beta(x)$ is exactly the same as the computation of $P^\alpha(x)$. Steps whose instructions are not of

the form $O(n)$ do not reference the oracle, so they are trivial to handle. So, suppose $P^\alpha(x)$ and $P^\beta(x)$ are the same up to step t and the next instruction is $O(n)$. Since the computations are the same up till now, the register R_n has the same content r_n in both machines. Thus, $r_n < \text{use}(\alpha; e, x) \leq |\sigma|$. But since $\sigma \sqsubset \alpha$ and $\sigma \sqsubset \beta$, we have

$$\beta(r_n) = \sigma(r_n) = \alpha(r_n).$$

So, this next step in both computations replaces r_n in register R_n with $\sigma(r_n)$. Thus, by induction, the computations $P^\alpha(x)$ and $P^\beta(x)$ are exactly the same, and so $\varphi_e^\alpha(x) = y = \varphi_e^\beta(x)$, as desired. □

Corollary 8.40. If σ and τ are finite sequences, then

$$\sigma \sqsubseteq \tau \,\&\, \varphi_e^\sigma(x)\!\downarrow = y \implies \varphi_e^\tau(x)\!\downarrow = y.$$

Exercise 8.41. Prove Corollary 8.40.

Theorem 8.42 (Finite use principle II)**.** Let α be an infinite sequence. If $\varphi_e^\alpha(x)\!\downarrow = y$, then there is a finite $\sigma \sqsubset \alpha$ such that $\varphi_e^\sigma(x)\!\downarrow = y$.

Proof. Assume $\varphi_e^\alpha(x)\!\downarrow = y$. Let $s = \text{use}(\alpha; e, x)$. Then, define

$$\sigma : \{0, 1, \ldots, s-1\} \to \mathbb{N}, \qquad \sigma(i) = \alpha(i) \quad (i < s),$$

i.e., σ is the *restriction* of α to $\{0, 1, \ldots, s-1\}$. Then, it easily follows by definition that $\varphi_e^\sigma(x)\!\downarrow = y$. □

We also have a Kleene T-predicate for computation relative to finite sequences.

Theorem 8.43. There is a recursive relation $T^*(s, e, x, y)$ on $\mathbb{N}$ and a recursive total function $U^* : \mathbb{N} \to \mathbb{N}$ such that for every finite sequence σ, we have

$$\varphi_e^\sigma(x) = U^*(\mu y T^*(\#\sigma, e, x, y)),$$

where $\#\sigma = \langle \sigma(0), \ldots, \sigma(|\sigma| \dot{-} 1) \rangle$.

The proof is small elaboration on the proof of the existence of the relativized Kleene T-predicate, T_1^α. The coding of programs and machine states does not change. We must again update the predicate CmpSt, but the change this time is different. Of course, CmpSt now needs an argument s for the code of σ, so

that it is of the form $\mathrm{CmpSt}(s, n, e, u, v)$. We also add to it the disjunct

$$\begin{aligned}&\mathrm{Ins}(u) = \#O(j) \mathbin{\&} \mathrm{Reg}(u, j) < \mathrm{lh}(s) \mathbin{\&} \mathrm{Reg}(v, j) = (s)_{\mathrm{Reg}(u,j)} \\ &\mathbin{\&} (\forall i < \ell)(i \neq j \to \mathrm{Reg}(u, i) = \mathrm{Reg}(v, i)) \mathbin{\&} \mathrm{InsNum}(v) = \mathrm{InsNum}(u) + 1\end{aligned}$$

to code the implementation of oracle instructions. Note that we need the clause "$\mathrm{Reg}(u, j) < \mathrm{lh}(s)$" to ensure that the computation does not ask for a value of σ which is not defined.

We end this section with an example of an application of computation relative to finite sequences. The following theorem shows that that there are incomparable Turing degrees.

Theorem 8.44 (Kleene and Post)**.** There exist incomparable Turing degrees, i.e., there are sets A and B such that $A \not\leq_T B$ and $B \not\leq_T A$.

Proof. We will define A and B by defining finite sequence approximations to their characteristic functions; more formally, we will define σ_n and τ_n such that

(i) Each σ_n and τ_n is a finite binary sequence and

$$\sigma_0 \sqsubset \sigma_1 \sqsubset \sigma_2 \sqsubset \cdots, \qquad \tau_0 \sqsubset \tau_1 \sqsubset \tau_2 \sqsubset \cdots$$

Note that this means $\lim_n \sigma_n$ and $\lim_n \tau_n$ are infinite binary sequences. They are defined by

$$(\lim_n \sigma_n)(x) = i \iff (\exists m)\ \sigma_m(x)\!\downarrow = i,$$

and similarly for $\lim_n \tau_n$.

(ii) If we let A and B be the sets with $\mathrm{Char}_A = \lim_n \sigma_n$ and $\mathrm{Char}_B = \lim_n \tau_n$, then $A \not\leq_T B$ and $B \not\leq_T A$.

We will construct the approximations in infinitely many stages. At stage $s+1$, we will pick σ_{s+1} and τ_{s+1} to satisfy the requirements

$$\begin{aligned} R_{2e}: &\quad \mathrm{Char}_A \neq \varphi_e^B \qquad \text{if } s = 2e \\ R_{2e+1}: &\quad \mathrm{Char}_B \neq \varphi_e^A \qquad \text{if } s = 2e+1. \end{aligned}$$

Clearly, if all the requirements are satisfied, then A and B are Turing incomparable.

At stage 0, just take σ_0, τ_0 to both be the empty sequence.

Next, we describe stage $s+1$, where $s = 2e$ for some e. We assume we already have constructed σ_s and τ_s and we want to choose σ_{s+1} and τ_{s+1} so that we satisfy the requirement

$$R_{2e}: \qquad \mathrm{Char}_A \neq \varphi_e^B.$$

Let x_0 be the smallest number so that $\sigma_s(x_0)\uparrow$. We want to extend σ_s to a longer finite sequence σ_{s+1} so that

$$\sigma_{s+1}(x_0) \neq \varphi_e^B(x_0).$$

The trick is that we don't know exactly what B is yet; we only have the approximation τ_s to its characteristic function. So, look for a finite binary $\tau \sqsupset \tau_s$ such that $\varphi_e^\tau(x_0)\downarrow$.

Case 1. If such a τ does exist, then set $\tau_{s+1} = \tau$ and define σ_{s+1} by

$$\sigma_{s+1}(x) = \begin{cases} \sigma_s(x) & \text{if } x < x_0 \\ 1 \dot{-} \varphi_e^{\tau_{s+1}}(x_0) & \text{if } x = x_0 \end{cases}$$

Then, we have $\sigma_s \sqsubset \sigma_{s+1}$ and $\tau_s \sqsubset \tau_{s+1}$; moreover, we have satisfied requirement R_{2e} since in the end we will have $\tau_{s+1} \sqsubset \mathrm{Char}_B$, hence $\varphi_e^{\tau_{s+1}}(x_0) = \varphi_e^B(x_0)$. This implies that

$$\mathrm{Char}_A(x_0) = \sigma_{s+1}(x_0) = 1 \dot{-} \varphi_e^{\tau_{s+1}}(x_0) = 1 \dot{-} \varphi_e^B(x_0) \neq \varphi_e^B(x_0).$$

Case 2. Suppose such a τ does not exist. This means that for all finite, binary $\tau \sqsupset \tau_s$, we have $\varphi_e^\tau(x_0)\uparrow$. Thus, by the finite use principle, no matter how we proceed we will have $\varphi_e^B(x_0)\uparrow$ since $\mathrm{Char}_B \sqsupset \tau_s$. Thus, R_{2e} is satisfied no matter how we proceed. Just define σ_{s+1}, τ_{s+1} by extending them any way you like to longer finite sequences.

The construction for stage $s+1$ where $s = 2e+1$ for some e is done exactly as above, just swapping the roles of A and B and, of course, σ_n and τ_n. $\square$

Further analyzing the above proof can show that A and B both have degrees $\leq_T K$. Thus, we have found Turing degrees less than K which are note comparable. Note this implies that neither A nor B are recursive. However, it is not clear from our proof that A and B are r.e. sets. The question of whether there are Turing incomparable r.e. sets was a big open question in computability theory, known as *Post's problem*. It was solved independently by Friedberg and Muchnik. Both used a method known as a *finite injury priority argument*. It bears some resemblance to our proof above, except at

some stages of the construction it is necessary to violate (or "injure") a lower priority argument.

8.6 Additional exercises

Exercise 8.45. Prove that every infinite α-r.e. set has an infinite α-recursive subset.

Exercise 8.46. Formulate and prove the relativized version of the semirecursive-selection principle.

Exercise 8.47. Let A, B, C be subsets of $\mathbb{N}$.

(a) Prove that if A is B-r.e. and B is C-recursive, then A is C-r.e.

(b) Show that the following statement is false: if A is B-recursive and B is C-r.e., then A is C-r.e.

Exercise 8.48. Formulate and prove the statement that the union operation is effective in the codes for α-r.e. sets.

Exercise 8.49. Formulate and prove a relativized version of Kleene's theorem.

Exercise 8.50. Prove that for any oracle $\alpha : \mathbb{N} \to \mathbb{N}$, there exists a set $A \subseteq \mathbb{N}$ which is Turing equivalent to α, in the sense that A is α-recursive and α is A-recursive. (Thus, we don't "lose anything" by just considering Turing degrees of sets.)

Exercise 8.51. For sets $A, B \subseteq \mathbb{N}$, define

$$A \oplus B =_{\text{df}} \{2x : x \in A\} \cup \{2x + 1 : x \in B\}.$$

(a) Prove that $A \leq_T A \oplus B$ and $B \leq_T A \oplus B$.

(b) Prove that if $C \subseteq \mathbb{N}$ satisfies $A \leq_T C$ and $B \leq_T C$, then $A \oplus B \leq_T C$.

(c) Prove that any pair of Turing degrees $\mathbf{a}, \mathbf{b}$ has a unique least upper bound in the ordering $\leq_T$.

Exercise 8.52. Let $\alpha : \mathbb{N} \to \mathbb{N}$ be an oracle.

(a) Formulate and prove the relativized version of the semirecursive-selection principle for α-r.e. relations.

(b) A set $A \subseteq \mathbb{N}$ is *α-simple* if it is α-r.e., $\neg A$ is infinite, and $\neg A$ does not contain any infinite α-r.e. subsets. Prove there exists an α-simple set.

Exercise 8.53. A Turing degree $\mathbf{a}$ is called an *r.e. degree* if there is an r.e. set A in $\mathbf{a}$.

(a) True or false: if $\mathbf{a}$ is an r.e. degree, then every set in $\mathbf{a}$ is r.e. Justify your answer.

(b) Prove that any r.e. degree which is not the degree of recursive sets contains a simple set.

Hint for (b). Let $B \in \mathbf{a}$ be r.e. Pick a total injective computable function $f : \mathbb{N} \to \mathbb{N}$ which enumerates B without repetitions. Let

$$A =_{\text{df}} \{x : (\exists y)[y > x \ \& \ f(y) < f(x)]\}.$$

We already showed in previous homework that A is simple. Now, show that $A \equiv_T B$.

Exercise 8.54. Let A be an r.e. set which is not recursive. Define the function $f : \mathbb{N} \to \mathbb{N}$ by $f(0) = 0$ and

$$f(x) = \text{ the number of elements in the set } A \cap \{0, \ldots, x-1\}.$$

for $x > 0$.

(a) Prove that f is not computable.

(b) Prove that f is K-computable.

Exercise 8.55. Let $f : \mathbb{N} \to \mathbb{N}$ be a strictly increasing total function and let $A = \text{range}(f)$.

Prove that f and A are Turing equivalent in the following sense: f is an A-recursive function and A is an f-recursive set.

9 The arithmetical hierarchy

The first class of special relations we studied were the *recursive relations*. Later, we studied the *semirecursive relations*, which are obtained by placing an existential quantifier in front of each recursive relation, i.e., $R(\vec{x})$ is semirecursive if and only if there is a recursive $R_0(\vec{x}, y)$ such that

$$R(\vec{x}) \iff (\exists y)\ R_0(\vec{x}, y).$$

The *co-semirecursive* relations are the negations of semirecursive relations; so, each co-semirecursive relation $Q(\vec{x})$ is of the form

$$Q(\vec{x}) \iff \neg[(\exists y)\ R_0(\vec{x}, y)] \iff (\forall y)\ \neg R_0(\vec{x}, y),$$

where R_0 is recursive. If we define $Q_0 = \neg R_0$, then we can see that every co-semirecursive relation Q is of the form

$$Q(\vec{x}) \iff (\forall y)\ Q_0(\vec{x}, y),$$

for some recursive relation Q_0.

In this section, we will study the relations we obtain by allowing any number of alternating quantifiers in front of a recursive relation. This will result in a hierarchy of definable relations called the *arithmetical hierarchy*.

We begin by giving alternative names to the relations mentioned above. Recursive relations are also called Δ_1^0 relations, semirecursive relations are also called Σ_1^0 relations, and co-semirecursive relations are also called Π_1^0 relations.

Before defining higher classes in the arithmetical hierarchy, we pause for a moment to mention some closure properties for the Π_1^0 relations.

Theorem 9.1. The class of Π_1^0 relations is closed under &, $\vee$, bounded quantification of both kinds, unbounded universal quantification, and substitution by recursive total functions.

Proof. All of these follow easily from the closure properties for Σ_1^0. For instance, consider closure under &. Let R and Q be Π_1^0 n-ary relations. Then, using De Morgan's law we have

$$(R \,\&\, Q)(\vec{x}) \iff R(\vec{x}) \,\&\, Q(\vec{x}) \iff \neg\Big(\neg R(\vec{x}) \vee \neg Q(\vec{x})\Big).$$

$\neg R$ and $\neg Q$ are both Σ_1^0, hence $\neg R \vee \neg Q$ is also Σ_1^0 since that class is closed under $\vee$. Thus, $R\&Q$ is Π_1^0 since it is the negation of the Σ_1^0 relation $\neg R \vee \neg Q$. The closure of Π_1^0 under $\vee$ is similarly proved.

Next, we show closure under substitution by recursive total functions. Let $R(\vec{y})$ be a Π^0_1 k-ary relation, and let $f_0, \ldots, f_{k-1} : \mathbb{N}^n \to \mathbb{N}$ be recursive total functions. We show that the n-ary relation

$$Q(\vec{x}) \iff_{\text{df}} R(f_0(\vec{x}), \ldots, f_{k-1}(\vec{x}))$$

is Π^0_1. It is enough to show that $\neg Q$ is Σ^0_1. We have

$$\neg Q(\vec{x}) \iff \neg R(f_0(\vec{x}), \ldots, f_{k-1}(\vec{x})),$$

so that $\neg Q$ is obtain via a recursive total substitution into the Σ^0_1 relation $\neg R$. Since Σ^0_1 is closed under recursive total substitutions, we conclude that $\neg Q$ is Σ^0_1, as desired.

The closures for the quantifications are proved similarly, using negation and the corresponding closure properties for Σ^0_1. □

Exercise 9.2. Prove that Π^0_1 is closed under bounded quantification of both kinds, and that it is closed under unbounded universal quantification.

We obtained the Σ^0_1 relations by placing an existential quantifier (over $\mathbb{N}$) in front of recursive relations. We proved that Σ^0_1 relations are closed under existential quantification, but what if we put an existential quantifier in front of a Π^0_1 relation? We call such relations Σ^0_2. In other words, a relation $R(\vec{x})$ is Σ^0_2 if and only if there is a Π^0_1 relation $Q(\vec{x}, y)$ such that

$$R(\vec{x}) \iff (\exists y)\ Q(\vec{x}, y).$$

Unwinding this definition a little further, we could pick a recursive $Q_0(\vec{x}, y, z)$ such that

$$Q(\vec{x}, y) \iff (\forall z)\ Q_0(\vec{x}, y, z).$$

Then, R also satisfies the equivalence

$$R(\vec{x}) \iff (\exists y)(\forall z)\ Q_0(\vec{x}, y, z),$$

where Q_0 is recursive.

Thus, a Σ^0_2 relation is a relation obtained by placing two alternating quantifiers, $\exists\forall$, in front of a recursive relation. In the same way that we obtained Π^0_1 relations by negating Σ^0_1 relations, the Π^0_2 relations are the negations of Σ^0_2 relations. Thus, any Π^0_2 relation $P(\vec{x})$ is of the form

$$P(\vec{x}) \iff \neg[(\exists y)(\forall z)\ Q_0(\vec{x}, y, z)] \iff (\forall y)(\exists z)\ \neg Q_0(\vec{x}, y, z),$$

where $\neg Q_0$ is recursive.

Recall that Kleene's theorem states that a relation is recursive if and only if it is both semirecursive and co-semirecursive. In other words,

$$\Delta_1^0 = \Sigma_1^0 \cap \Pi_1^0.$$

This theorem inspires us to make the following definition:

$$\Delta_2^0 = \Sigma_2^0 \cap \Pi_2^0,$$

so that a relation is Δ_2^0 if and only if it is both Σ_2^0 and Π_2^0.

We continue this definition scheme to define the classes Σ_n^0, Π_n^0, and Δ_n^0 for $n > 0$.

Definition 9.3. A relation $R(\vec{x})$ is Σ_{n+1}^0 if it is of the form

$$R(\vec{x}) \iff (\exists y)\ R_0(\vec{x}, y)$$

for some Π_n^0 relation $R(\vec{x}, y)$.

A relation is Π_{n+1}^0 if it is the negation of a Σ_{n+1}^0 relation.

A relation is Δ_{n+1}^0 if it is both Σ_{n+1}^0 and Π_{n+1}^0.

We use the following equations to symbolically express the above definition:

$$\Sigma_{n+1}^0 = \exists^{\mathbb{N}} \Pi_n^0, \qquad \Pi_{n+1}^0 = \neg \Sigma_{n+1}^0, \qquad \Delta_{n+1}^0 = \Sigma_{n+1}^0 \cap \Pi_{n+1}^0.$$

Note that it is easy to show that the negation of a Π_n^0 relation is Σ_n^0,

$$\Sigma_n^0 = \neg \Pi_n^0.$$

If one continues the above discussion, we can easily establish the following normal forms for Σ_n^0 relations:

$$\begin{array}{ll} \Sigma_1^0 & (\exists y_0)\ R(\vec{x}, y_0) \\ \Sigma_2^0 & (\exists y_0)(\forall y_1)\ R(\vec{x}, y_0, y_1) \\ \Sigma_3^0 & (\exists y_0)(\forall y_1)(\exists y_2)\ R(\vec{x}, y_0, y_1, y_2) \\ & \vdots \end{array}$$

where R is a recursive relation in each case. For Π^0_n, the normal forms are

$$\begin{array}{ll} \Pi^0_1 & (\forall y_0)\ R(\vec{x}, y_0) \\ \Pi^0_2 & (\forall y_0)(\exists y_1)\ R(\vec{x}, y_0, y_1) \\ \Pi^0_3 & (\forall y_0)(\exists y_1)(\forall y_2)\ R(\vec{x}, y_0, y_1, y_2) \\ & \vdots \end{array}$$

We can write this fact about normal forms more formally in the following proposition.

Proposition 9.4. Let $n > 0$. A relation $P(\vec{x})$ is Σ^0_n if and only if

$$P(\vec{x}) \iff (\exists y_0)(\forall y_1) \cdots (\mathsf{Q} y_{n-1})\ R(\vec{x}, y_0, y_1 \ldots, y_{n-1}),$$

where R is recursive and $\mathsf{Q} = \exists$ if n is odd and $\mathsf{Q} = \forall$ if n is even.

A relation $P(\vec{x})$ is Π^0_n if and only if

$$P(\vec{x}) \iff (\forall y_0)(\exists y_1) \cdots (\mathsf{Q} y_{n-1})\ R(\vec{x}, y_0, y_1 \ldots, y_{n-1}),$$

where R is recursive and $\mathsf{Q} = \forall$ if n is odd and $\mathsf{Q} = \exists$ if n is even.

We end this section by looking at a few examples.

Example 9.5. As we have noted before, K is Σ^0_1 and $\neg K$ is, of course, Π^0_1.

Example 9.6. Recall the set

$$\mathrm{Tot} =_{\mathrm{df}} \{e \in \mathbb{N} : \varphi_e \text{ is total}\}.$$

By noting the equivalence

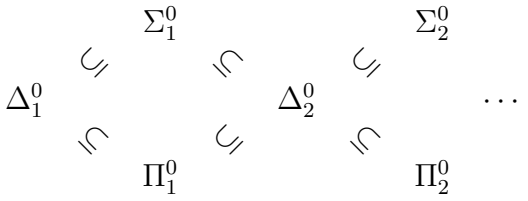
$$\mathrm{Tot}(e) \iff (\forall x)(\exists y)\ T_n(e, x, y),$$

we can see that Tot is Π^0_2. Thus,

$$\neg\mathrm{Tot} =_{\mathrm{df}} \{e \in \mathbb{N} : \varphi_e \text{ is not total}\}$$

is Σ^0_2.

We will see more examples once we establish closure properties in the next section, but first we observe that we have the following diagram of inclusions.



Thus, each class at a level in the hierarchy contains all the classes in earlier levels. To establish these inclusions, first note that $\Delta_n^0 \subseteq \Sigma_n^0$ and $\Delta_n^0 \subseteq \Pi_n^0$ are immediate by definition of $\Delta_n^0 = \Sigma_n^0 \cap \Pi_n^0$. The inclusions $\Sigma_n^0 \subseteq \Delta_{n+1}^0$ and $\Pi_n^0 \subseteq \Sigma_{n+1}^0$ are proved using "dummy quantifiers". For instance, consider a Σ_2^0 relation $R(\vec{x})$. To see that R is also Π_3^0, pick a recursive relation R_0 with

$$R(\vec{x}) \iff (\exists y)(\forall z)\ R_0(\vec{x}, y, z),$$

and consider the equivalence

$$R(\vec{x}) \iff (\forall w)(\exists y)(\forall z)\ R_0(\vec{x}, y, z)$$

Note that the relation $Q(\vec{x}, y, z, w) \iff_{\text{df}} R_0(\vec{x}, y, z)$ is recursive using closure under recursive total substitution. Thus, R is also Π_3^0. To see that R is also Σ_3^0, just note the equivalence

$$R(\vec{x}) \iff (\exists y)(\forall z)(\exists w)\ R_0(\vec{x}, y, z),$$

and argue like before.

9.1 Closure properties

Proposition 9.7. For any $n > 0$, the classes Σ_n^0 and Π_n^0 are closed under $\&, \vee$, bounded quantification of both kinds, and substitution by recursive total functions. Moreover, Σ_n^0 is closed under unbounded existential quantification, and Π_n^0 is closed under unbounded universal quantification.

Proof. We proceed by induction on n. We have proved earlier the base case for Σ_1^0 and Π_1^0.

For the inductive step, assume the theorem is true at level n. We establish the closure properties for Σ_{n+1}^0, then from these we deduce the closure properties for Π_{n+1}^0 in much the same way we did for Π_1^0 from Σ_1^0 in Theorem 9.1.

First, we show that Σ_{n+1}^0 is closed under $\exists$. Let $R(\vec{x}, y)$ be a Σ_{n+1}^0 relation. We want to show that

$$Q(\vec{x}) \iff_{\text{df}} (\exists y)\ R(\vec{x}, y)$$

is Σ^0_{n+1}. Pick a Π^0_n relation R_0 such that

$$R(\vec{x}, y) \iff (\exists z)\ R_0(\vec{x}, y, z).$$

Then,

$$Q(\vec{x}) \iff (\exists y)\ R(\vec{x}, y) \iff (\exists y)(\exists z)\ R_0(\vec{x}, y, z) \iff (\exists u)\ R_0(\vec{x}, (u)_0, (u)_1).$$

By our inductive hypothesis, the relation '$R_0(\vec{x}, (u)_0, (u)_1)$' is Π^0_n since that class is closed under recursive total substitution. Thus, Q is Σ^0_{n+1}.

We can then establish the closure of Σ^0_1 under $\&$, $\vee$, and bounded quantification of both kinds. The proofs here are very similar to the proofs of the corresponding closure properties for Σ^0_1, just using Π^0_n relations in place of recursive relations.

Finally, we will show that, under our inductive hypothesis, Σ^0_{n+1} is closed under total recursive substitutions. Let $R(\vec{y})$ be a Σ^0_{n+1} k-ary relation, and let $f_0, \ldots, f_{k-1} : \mathbb{N}^m \to \mathbb{N}$ be recursive total functions. We want to show

$$Q(\vec{x}) \quad \iff_{\text{df}} \quad R(f_0(\vec{x}), \ldots, f_{k-1}(\vec{x}))$$

is Σ^0_1. Pick a Π^0_n relation R_0 such that

$$R(\vec{y}) \iff (\exists z)\ R_0(\vec{y}, z).$$

Then,

$$Q(\vec{x}) \iff (\exists z)\ R_0(f_0(\vec{x}), \ldots, f_{k-1}(\vec{x}), z).$$

But the relation '$R_0(f_0(\vec{x}), \ldots, f_{k-1}(\vec{x}), z)$' is obtained via recursive total substitutions into a Π^0_n relation, hence it is Π^0_n by our inductive hypothesis. Thus, Q is indeed Σ^0_{n+1}. □

Note that if Σ^0_n and Π^0_n both have a particular closure property, then Δ^0_n has that same closure property. For instance, consider two Δ^0_n relations P and Q. Since P and Q are both Σ^0_n, it follows that $P \,\&\, Q$ is also Σ^0_n since that class is closed under $\&$. Similarly, since P and Q are both Π^0_n, it follows that $P \,\&\, Q$ is Π^0_n. Then $P \,\&\, Q$ is Δ^0_n.

The classes Δ^0_n are not closed under unbounded quantification of either type. However, they are closed under negation, which, we will see later, is a closure property neither Σ^0_n nor Π^0_n has.

As an immediate consequence of closure under recursive total substitutions, we get the following important fact.

Corollary 9.8. Let R and Q be relations such that $R \leq_M Q$, and let Let Γ be any of Σ^0_n, Π^0_n or Δ^0_n. If Q is Γ, then R is also Γ.

Next, we establish closure under recursive partial substitutions.

Proposition 9.9. For every $n > 1$, both Σ^0_n and Π^0_n are closed under substitution by recursive partial functions. (Σ^0_1 is also closed under these substitutions, but Π^0_1 is not.)

Proof. Fix $n > 1$. Let $R(\vec{y})$ be a Σ^0_n k-ary relation, and let $f_0, \ldots, f_{k-1} : \mathbb{N}^n \rightharpoonup \mathbb{N}$ be recursive partial functions. We want to show that

$$Q(\vec{x}) \iff f_0(\vec{x})\downarrow \,\&\, \cdots \,\&\, f_{k-1}(\vec{x})\downarrow \,\&\, R(f_0(\vec{x}), \ldots, f_{k-1}(\vec{x}))$$

is Σ^0_{n+1}. We have

$$Q(\vec{x}) \iff (\exists y_0) \ldots (\exists y_{k-1})[f_0(\vec{x})\downarrow = y_0 \,\&\, \cdots f_{k-1}(\vec{x})\downarrow = y_{k-1} \,\&\, R(\vec{y})].$$

All the graph relations are Σ^0_1, hence also Σ^0_n by the remark preceding this theorem. Since Σ^0_n relations are closed under & and existential quantification, it follows that Q is Σ^0_n.

Now, consider instead the case where R is instead a Π^0_n relation. This time, we want to show that Q is Π^0_n. We have

$$\begin{aligned} Q(\vec{x}) &\iff f_0(\vec{x})\downarrow \,\&\, \cdots \,\&\, f_{k-1}(\vec{x})\downarrow \\ &\quad \,\&\, (\forall y_0) \ldots (\forall y_{k-1}) \Big[(f_0(\vec{x})\downarrow = y_0 \,\&\, \cdots \,\&\, f_{k-1}(\vec{x})\downarrow = y_{k-1}) \to R(\vec{y})] \\ &\iff f_0(\vec{x})\downarrow \,\&\, \cdots \,\&\, f_{k-1}(\vec{x})\downarrow \\ &\quad \,\&\, (\forall y_0) \ldots (\forall y_{k-1}) \Big[\neg (f_0(\vec{x})\downarrow = y_0 \,\&\, \cdots \,\&\, f_{k-1}(\vec{x})\downarrow = y_{k-1}) \vee R(\vec{y})] \end{aligned}$$

Convergence of partial functions is Σ^0_1. Since $n > 1$, we have $\Sigma^0_1 \subseteq \Pi^0_n$. Then, by several applications of closure properties, we conclude that Q is Π^0_n. □

We write down for the record a single theorem summarizing all of the closure properties we have established.

Theorem 9.10. For any $n > 0$, Σ^0_n is closed under $\&, \vee$, bounded and unbounded existential quantification, bounded universal quantification, and substitution by recursive partial functions.

For any $n > 0$, Π^0_n is closed under $\&, \vee$, bounded existential quantification, and bounded and unbounded universal quantification. For $n > 1$, Π^0_n is also

closed under substitution by recursive partial functions. (Π^0_1 is only closed under substitution by recursive total functions.)

For any $n > 0$, Δ^0_n is closed under $\neg$, $\&$, $\vee$, and bounded quantification of both kinds. For $n > 1$, Δ^0_n is also closed under substitution by recursive partial functions. (Δ^0_1 is only closed under substitution by recursive total functions.)

Example 9.11. Consider the relation

$$R(e, z) \iff W_e \subseteq W_z.$$

Writing out this definition with quantifiers,

$$\begin{aligned} R(e, z) &\iff (\forall x)[\varphi_e(x)\!\downarrow \to \varphi_z(x)\!\downarrow] \\ &\iff (\forall x)[\varphi_e(x)\!\uparrow \vee \varphi_z(x)\!\downarrow]. \end{aligned}$$

The relation '$\varphi_e(x)\!\uparrow$' is Π^0_1, and the relation '$\varphi_e(x)\!\downarrow$' is Σ^0_1. Thus, they are both Δ^0_2 relations, so their disjunction is Δ^0_2. Thus, the relation R is of the form $(\forall x)[\text{–}\Delta^0_2\text{–}]$. It follows that R is Π^0_2 (why?).

9.2 Universal relations

Recall that we showed that, for any arity $k > 0$, the relation

$$V^{(k)}_U(e, \vec{x}) \quad \iff_{\text{df}} \quad W^{(k)}_e(\vec{x})$$

is Σ^0_1, and called it the *universal relation* for k-ary Σ^0_1 relations.

Definition 9.12. A *universal relation for k-ary Σ^0_n relations* is a relation $G^{(k)}_n \subseteq \mathbb{N}^{k+1}$ such that

(i) $G^{(k)}_n$ is a Σ^0_n relation; and

(ii) for every k-ary Σ^0_n relation $R(\vec{x})$, there exists $e \in \mathbb{N}$ such that

$$R(\vec{x}) \iff G^{(k)}_n(e, \vec{x}).$$

In other words, if we define $G^{(k)}_{n,e} = \{\vec{x} : G^{(k)}_n(e, \vec{x})\}$, we say that

$$G^{(k)}_{n,0}, \quad G^{(k)}_{n,1}, \quad G^{(k)}_{n,2}, \ldots$$

is an enumeration of all the k-ary Σ^0_n relations which is *uniformly* Σ^0_n.

We now prove that these universal relations exist.

Theorem 9.13. For every $n > 0$, Σ^0_n has universal relations $G^{(k)}_n$ of every arity k.

Proof. We prove the theorem by induction on n. As we discussed above, we have already prove the base case for Σ^0_1. So, we define $G^{(k)}_1 =_{\text{df}} V^{(k)}_U$.

Inductively, we assume that Σ^0_n has universal relations of every arity, which we denote by $G^{(k)}_n$. For $k > 0$, define $G^{(k)}_{n+1}$ by

$$G^{(k)}_{n+1}(e, \vec{x}) \iff (\exists y)\ \neg G^{(k+1)}_n(e, \vec{x}, y).$$

$G^{(k)}_{n+1}$ is clearly Σ^0_{n+1}; indeed, $\neg G^{(k+1)}_n$ is Π^0_n by induction hypothesis, and placing $\exists y$ in front makes the relation Σ^0_{n+1}. Now, we have to check property (ii) of the definition of universal relation.

Let $R(\vec{x})$ be a k-ary Σ^0_{n+1} relation. Pick a Σ^0_n $(k+1)$-ary relation R_0 such that

$$R(\vec{x}) \iff (\exists y)\ \neg R_0(\vec{x}, y).$$

Since $G^{(k+1)}_n$ is universal for $(k+1)$-ary Σ^0_n relations, there is an $e \in \mathbb{N}$ such that

$$R_0(\vec{x}, y) \iff G^{(k+1)}_n(e, \vec{x}, y).$$

Thus,

$$R(\vec{x}) \iff (\exists y)\ \neg R_0(\vec{x}, y) \iff (\exists y)\ \neg G^{(k+1)}_n(e, \vec{x}, y) \iff G^{(k)}_{n+1}(e, \vec{x}),$$

so condition (ii) is satisfied. □

A very important consequence of universal relations is the following:

Corollary 9.14. For every $n > 0$, there are Σ^0_n relations which are not Π^0_n, and there are Π^0_n relations which are not Σ^0_n.

Proof. This is a diagonalization argument. Fix $n > 0$. Define

$$R(e) \iff \neg G^{(1)}_n(e, e).$$

R is easily shown to be Π^0_n. Suppose towards a contradiction that R is Σ^0_n. Then, by the properties of universal sets, there is $e_0 \in \mathbb{N}$ such that

$$R(e) \iff G^{(1)}_n(e_0, e).$$

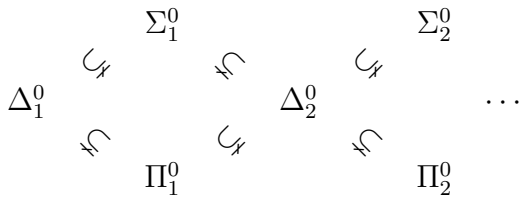
But now, combining the two equivalences with $e = e_0$ yields

$$G^{(1)}_n(e_0, e_0) \iff R(e_0) \iff \neg G^{(1)}_n(e_0, e_0),$$

which is absurd. Thus R is $\Pi^0_n \setminus \Sigma^0_n$.

It then follows easily that $\neg R$ is $\Sigma^0_n \setminus \Pi^0_n$. □

Corollary 9.15. All of the inclusions in the arithmetical hierarchy diagram are all strict inclusions:



Proof. We already now the non-strict $\subseteq$ inclusions hold everywhere.

Let $n > 0$. Since there is a relation which is Σ^0_n but not Π^0_n, it follows that $\Delta^0_n = \Sigma^0_n \cap \Pi^0_n \subsetneq \Sigma^0_n$. Similarly, since there is a relation in $\Pi^0_n \setminus \Sigma^0_n$, we have $\Delta^0_n \subsetneq \Pi^0_n$.

Now, suppose towards a contradiction that $\Sigma^0_n = \Delta^0_{n+1}$. Then, we have $\Pi^0_n \subseteq \Delta^0_{n+1} = \Sigma^0_n$, but we know that not every Π^0_n relation is Σ^0_n. Thus, $\Sigma^0_n \subsetneq \Delta^0_{n+1}$. A similar contradiction argument can establish that $\Pi^0_n \subsetneq \Delta^0_{n+1}$. □

9.3 Classifying relations in the arithmetical hierarchy

Recall the we showed that Tot, the set of indices of recursive total unary functions, is Π^0_2. We did this by establishing that

$$e \in \text{Tot} \iff (\forall x)(\exists y)\ T_1(e, x, y).$$

So, we showed that Tot has a Π^0_2 definition; but this fact alone does not rule out the possibility that there is simpler definition for Tot. Perhaps, for instance Tot is also Σ^0_2, which would place it into the simpler class Δ^0_2.

This is indeed not the case. But how can we prove it? An important tool in this sort of proof is the fact that every class Γ in the arithmetical hierarchy is "closed downward" under computably reduction: if R can be computably reduced to Q and Q is in Γ, then R is in Γ. Recall that this fact just follows from closure under recursive total substitution.

So, one way we can prove Tot is not Σ^0_2 is to show that every Π^0_2 set can be computably reduced to Tot. Then, if Tot was Σ^0_2, it would follow that every Π^0_2 set is also Σ^0_2. But, from the previous section we know that not every Π^0_2 set is Σ^0_2.

Proposition 9.16. Tot is $\Pi^0_2 \setminus \Sigma^0_2$.

Proof. All that is left to show is that Tot is not Σ^0_2. By our comments above, it is enough to show that $A \leq_m \text{Tot}$ for every Π^0_2 set.

Let A be a Π^0_2 set. We show that $A \leq_m \text{Tot}$. Pick a recursive relation R_0 such that

$$x \in A \iff (\forall y)(\exists z)\ R_0(x, y, z).$$

Define a partial function

$$f(x, y) = \mu z[R_0(x, y, z)].$$

By the s-m-n theorem, there is a recursive total $S : \mathbb{N} \to \mathbb{N}$ such that

$$\varphi_{S(x)}(y) = f(x, y) \qquad \text{for all } x, y \in \mathbb{N}.$$

We claim that this S is an m-reduction from A to Tot. We easily check that

$$\begin{aligned} x \in A \iff (\forall y)(\exists z)\ R_0(x, y, z) &\iff f(x, y)\!\downarrow \text{ for all } y \in \mathbb{N} \\ &\iff \varphi_{S(x)} \text{ is total} \iff S(x) \in \text{Tot}. \end{aligned}$$

Thus, Tot is not Σ^0_n, or else all Π^0_n sets would be Σ^0_n. □

In general, classifying a relation R in the arithmetical hierarchy means establishing exactly one of the following:

(1) R is Σ^0_n but not Π^0_n,
(2) R is Π^0_n but not Σ^0_n, or
(3) R is Δ^0_{n+1} but neither Π^0_n or Σ^0_n.

Example 9.17. In the previous section, we showed that for any $n > 0$,

$$R(e) \iff_{\text{df}} G^{(1)}_n(e, e)$$

is Σ^0_n but not Π^0_n. Of course, then $\neg R$ is Π^0_n but not Σ^0_n.

Example 9.18. Consider the set

$$\text{Inf} = \{e \in \mathbb{N} : W_e \text{ is infinite}\}.$$

We show that Inf is $\Pi^0_2 \setminus \Sigma^0_n$.

First, Inf is Π^0_2 since

$$e \in \text{Inf} \iff (\forall x)(\exists y)[y > x \,\&\, y \in W_e].$$

One can easily use closure properties to show that this is a Π^0_2 definition.

To show that Inf is not Σ_2^0, it is enough to show that $A \leq_m \text{Inf}$ for all Π_2^0 sets A.

Let A be Π_2^0 and pick a recursive R_0 such that

$$x \in A \iff (\forall y)(\exists z)\ R_0(x, y, z).$$

Now define

$$f(x, y) = \begin{cases} 0 & \text{if } (\forall i \leq y)(\exists z)\ R_0(x, i, z) \\ \uparrow & \text{otherwise.} \end{cases}$$

f is recursive (why?). Use the s-m-n theorem to obtain a recursive S such that $\varphi_{S(x)}(y) = f(x, y)$ for all $x, y \in \mathbb{N}$. Then, one can check that $S(x)$ is an m-reduction from A to Inf. The details are left as an exercise. Note that, when $x \notin A$, there is some y_0 such that $\neg R_0(x, y_0, z)$ for all $z \in \mathbb{N}$. It follows that $f(x, y) \uparrow$ for all $y \geq y_0$. Then, $\varphi_{S(x)}$ has a finite domain for this x.

Exercise 9.19. Fill in the details that S above is an m-reduction from A to Inf.

We give an example of a relation that is in a Δ class and now lower class.

Example 9.20. As usual, K is the diagonal halting problem. Define

$$A = K \oplus \neg K = \{2x : x \in K\} \cup \{2x + 1 : x \in \neg K\}$$

We will show that A is Δ_2^0 but not $\Sigma_1^0 \cup \Pi_1^0$.

To see that A is Δ_2^0, just note that

$$y \in A \iff (\exists x < y)[x = 2y \,\&\, y \in K] \vee (\exists x < y)[y = 2x + 1 \,\&\, x \in \neg K].$$

Then, one can use closure properties to show that this is a disjunction of a Σ_1^0 relation and a Π_1^0 relation. Thus, A is Δ_2^0 since it is a disjunction of two Δ_2^0 relations.

Next, we show that A is not Π_1^0. To see this, just note that we can m-reduce K to A:

$$x \in K \iff 2x \in A.$$

Since K is note Π_1^0, neither is A. Similarly, to show that A is not Σ_1^0, just note that we can m-reduce the non-Σ_1^0 set $\neg K$ to A:

$$x \in \neg K \iff 2x + 1 \in A.$$

Thus, A is Δ_2^0 but neither Π_1^0 nor Σ_1^0.

9.4 Turing jumps and the arithmetical hierarchy

In this section, we will explore the connection between the arithmetical hierarchy and relative computability. Specifically, we will focus on computability relative to K and its jumps. We will use the notation

$$0' = K, \quad 0'' = K^K = \text{ the jump of } K, \ldots, \quad 0^{(n)} = \text{ the } (n-1)\text{th jump of } K.$$

The important facts to remember for this section are that we have a Kleene T-predicate for computation relative to finite sequences, and that for any set A, the jump K^A is m-complete for A-semirecursive relations. We will also need the relativized version of Kleene's theorem.

Exercise 9.21. Formulate and prove a relativized version of Kleene's theorem.

Theorem 9.22. A relation is Σ^0_2 if and only if it is $0'$-semirecursive.

Proof. Suppose $R(\vec{x})$ is Σ^0_2. Pick a Σ^0_1 relation $R_0(\vec{x}, y)$ such that

$$R(\vec{x}) \iff (\exists y)\, \neg R_0(\vec{x}, y).$$

Since $K = 0'$ is m-complete for semirecursive relations, there is a a recursive total f such that

$$R_0(\vec{x}, y) \iff f(\vec{x}, y) \in 0'.$$

Thus,

$$R(\vec{x}) \iff (\exists y)\, \neg R_0(\vec{x}, y) \iff (\exists y)\, f(\vec{x}, y) \notin 0',$$

which shows that R is $0'$-semirecursive.

Conversely, suppose $R(\vec{x})$ be a $0'$-semirecursive k-ary relation. Then, there is a $0'$-recursive partial function $\varphi_e^{(k),0'} : \mathbb{N}^k \rightharpoonup \mathbb{N}$ such that

$$R(\vec{x}) \iff \varphi_e^{(k),0'}(\vec{x})\!\downarrow.$$

Then, using the basic facts about computability relative to finite sequences, we have

$$R(\vec{x}) \iff (\exists \sigma \text{ finite})[\sigma \sqsubset \mathrm{Char}_{0'} \,\&\, \varphi_e^{(k),\sigma}(\vec{x})\!\downarrow].$$

It is routine to change the right-hand-side into a statement about sequence codes, but we will continue to discuss in terms of actual finite sequences. From the Kleene T-predicate for computation relative to finite sequences, it follows that '$\varphi_e^{(k),\sigma}(\vec{x})\!\downarrow$' is Σ^0_1 as a relation of $e, \vec{x}$, and (a sequence code for) σ.

The relation '$\sigma \sqsubset \mathrm{Char}_{0'}$' can be shown to be Δ^0_2 by noting

$$\sigma \sqsubset \mathrm{Char}_{0'} \iff (\forall i < |\sigma|)[(\sigma(i) = 0 \vee \sigma(i) = 1) \\ \& (\sigma(i) = 0 \to i \notin 0') \,\&\, (\sigma(i) = 1 \to i \in 0')].$$

Thus, R is of the form $(\exists u)[-\Delta^0_2-]$, hence R is Σ^0_2. □

Corollary 9.23. A relation is Δ^0_2 if and only if it is $0'$-recursive.

Exercise 9.24. Prove the above corollary.

Theorem 9.25. For any $n > 0$, a relation is Σ^0_{n+1} if and only if it is $0^{(n)}$-semirecursive.

Proof. We proceed by induction on n. The base case $n = 1$ was handled by the previous theorem. So, assume the theorem is true for n. We want to show that a relation is Σ^0_{n+2} if and only if it is $0^{(n+1)}$-semirecursive. The proof is very similar to the proof of the base case.

Assume $R(\vec{x})$ is Σ^0_{n+2}. Pick a Σ^0_{n+1} relation R_0 such that

$$R(\vec{x}) \iff (\exists y)\, \neg R_0(\vec{x}, y).$$

Since R_0 is Σ^0_{n+1}, our induction hypothesis implies that it is $0^{(n)}$-semirecursive. Thus, R_0 can be computably reduced to $(0^{(n)})' = 0^{(n+1)}$ by some recursive total f,

$$R_0(\vec{x}, y) \iff f(\vec{x}, y) \in 0^{(n+1)}.$$

It follows that

$$R(\vec{x}) \iff (\exists y)\, \neg R_0(\vec{x}, y) \iff (\exists y)\, f(\vec{x}, y) \notin 0^{(n+1)},$$

which shows that R is indeed $0^{(n+1)}$-semirecursive.

Now suppose R_0 is a $0^{(n+1)}$-semirecursive k-ary relation. Then, there is a $0^{(n+1)}$-recursive partial function $\varphi_e^{(k),0^{(n+1)}} : \mathbb{N}^k \rightharpoonup \mathbb{N}$ such that

$$R(\vec{x}) \iff \varphi_e^{(k),0^{(n+1)}}(\vec{x})\!\downarrow.$$

Then, just as in the proof of the previous theorem, we have

$$R(\vec{x}) \iff (\exists \sigma \text{ finite})[\sigma \sqsubset \mathrm{Char}_{0^{(n+1)}} \,\&\, \varphi_e^{(k),\sigma}(\vec{x})\!\downarrow],$$

and this time we must show that '$\sigma \sqsubset \mathrm{Char}_{0^{(n+1)}}$' is Δ^0_{n+1}. This follows using the same reasoning as before. We will additionally need to note that $0^{(n+1)}$ is

Σ^0_{n+1}. Indeed, the equivalence

$$x \in 0^{(n+1)} \iff \varphi_x^{0^{(n)}}(x)\downarrow$$

shows that $0^{(n+1)}$ is $0^{(n)}$-semirecursive, hence Σ^0_{n+1} by our inductive hypothesis. □

Corollary 9.26. A relation is Δ^0_{n+1} if and only if it is $0^{(n)}$-recursive.

Corollary 9.27. For every $n > 0$, $0^{(n)}$ is both m-complete and Turing complete for Σ^0_n relations.

Proof. We already noted in the previous proof that $0^{(n)}$ is Σ^0_n. Let $R(\vec{x})$ be Σ^0_n. By the theorem, R is $0^{(n-1)}$-semirecursive. But this implies that R can be m-reduced to $(0^{(n-1)})' = 0^{(n)}$. □

9.5 Additional exercises

Exercise 9.28. We have shown that for every $n > 0$ there is a *unary* relation which is Π^0_n but not Σ^0_n. Show that for every $n, k > 0$, there is a k-ary relation which is Π^0_n but not Σ^0_n.

Exercise 9.29. For each $n, k > 0$, define $H^{(k)}_n \subseteq \mathbb{N}^{k+1}$ by

$$H^{(k)}_n(e, x_0, \ldots, x_{k-1}) \iff_{\text{df}} G^{(1)}_n(e, \langle x_0, \ldots, x_{k-1}\rangle)$$

Prove that $H^{(k)}_n$ is a universal relation for k-ary Σ^0_n relations.

Exercise 9.30 (s-m-n theorem for relations.)**.** Prove that for any $(m+n)$-ary Σ^0_k relation $P(\vec{x}, \vec{y})$, there exists a recursive total $S : \mathbb{N}^m \to \mathbb{N}$ such that

$$P(\vec{x}, \vec{y}) \iff G^{(n)}_k(S(\vec{x}), \vec{y})$$

for all $\vec{x} \in \mathbb{N}^m$ and $\vec{y} \in \mathbb{N}^n$. *Hint.* Prove it by induction on k.

Exercise 9.31. Prove that for any $n > 0$, there does not exist a universal relation for Δ^0_n sets, i.e., prove that there does not exist a Δ^0_n binary relation H_n such that for every Δ^0_n set A, there exists $e \in \mathbb{N}$ with

$$x \in A \iff H_n(e, x) \qquad \text{for all } x \in \mathbb{N}.$$

Exercise 9.32. Classify the following sets and relations into the arithmetical hierarchy, i.e., find n so that the relation is in $\Sigma^0_n \setminus \Pi^0_n$, or $\Pi^0_n \setminus \Sigma^0_n$, or $\Delta^0_n \setminus (\Sigma^0_n \cup \Pi^0_n)$. Prove your assertions.

(a) $\text{Fin} = \{e \in \mathbb{N} : W_e \text{ is finite}\}$

(b) $A = \{e \in \mathbb{N} : E_e \text{ is infinite}\}$

(c) $B = \{e \in \mathbb{N} : \varphi_e \text{ is not surjective}\}$

(d) $P(e, z) \iff_{\text{df}} W_e \cup W_z = \mathbb{N}$

(e) $Q(e, z) \iff_{\text{df}} W_e \cap W_z = \emptyset$

(f) $R(e, z) \iff_{\text{df}} \phi_e = \phi_z$

Exercise 9.33. Place the following sets into the arithmetical hierarchy. You do not have to provide a proof that they are not in any simpler class, but you should aim to place them in the simplest class that you can.

(a) $A = \{e \in \mathbb{N} : \neg W_e \text{ is infinite}\}$.

(b) $B = \{e \in \mathbb{N} : W_e^{(2)} \text{ is a symmetric relation}\}$. (Recall that a binary relation $R \subseteq \mathbb{N}^2$ is symmetric if $R(y, x)$ holds whenever $R(x, y)$ holds.)