Reducibilities relations with applications to symbolic dynamics

Part I: Computability

E. Jeandel

LORIA (Nancy, France)

Introduction

- What is computability ?
- Why does computability interact with symbolic dynamics ?
- Computability results on symbolic dynamics.

What is computability

My point of view

Computability theory studies which sets, functions, reals... are computable

Computability theory tries to order sets, functions, reals by how far they are from being computable.

Warning

Main difficulty: There are a lot of computability notions

Computable function $f : \mathbb{N} \to \mathbb{N}$

Computable set $S \subseteq \mathbb{N}$

Computable real $r \in \mathbb{R}$

Computable function $\mathbb{R} \to \mathbb{R}$

Computable set $S \subseteq \{0, 1\}^{\mathbb{N}}$

The five definitions are different and cannot be unified.

Plan

Informal definition

Definition

A partial function from $\mathbb{N}^k$ to $\mathbb{N}$ is *recursive* if it can be given by a program in your favorite programming language.

Church-Turing thesis: the programming language (Java, Python, your brain, my brain) does not matter

Examples

- $f_1(n) = n^2$ is recursive
- $f_2(n, m) = n + \frac{(n+m)(n+m+1)}{2}$ is recursive
- $(f_3, f_4) = f_2^{-1}$ are recursive
- $f_5(n) =$ the n-th decimal of π is recursive

Examples

$$f(n) = \begin{cases} 1 & \text{if the sequence } \overbrace{66\cdots6}^{n} \text{ appears in the decimals of } \pi \\ 0 & \text{otherwise} \end{cases}$$

is recursive

- Either infinitely large sequences of 6 appear, in which case $f(n) = 1$ for all n and f is certainly recursive
- Or $f(n) = 1$ for $n < k$ and $f(n) = 0$ otherwise for some k, and any such f is certainly recursive

computability $\neq$ constructivity

Examples

$$f(n) = \begin{cases} 1 & \text{if the sequence } \overbrace{66\cdots6}^{n} \text{ appears in the decimals of } \pi \\ 0 & \text{otherwise} \end{cases}$$

is recursive

- Either infinitely large sequences of 6 appear, in which case $f(n) = 1$ for all n and f is certainly recursive
- Or $f(n) = 1$ for $n < k$ and $f(n) = 0$ otherwise for some k, and any such f is certainly recursive

computability $\neq$ constructivity

Partial functions

Functions computed by programs may be *partial*: $f(n)$ is not defined if the program does not halt on input n.

Sometimes we write $f(n) = \perp$ to say that f is not defined on n.

- $f(n) = 1$ if the Collatz sequence starting from n goes to the cycle $1/2/4$, $\perp$ otherwise is (partial) recursive.
- $f(n_1, n_2, n_3, n_4, n_5, n_6) = 1$ if the equation $n_1 x^2 + n_2 xy - n_3 y^2 - n_4 x - n_5 y + n_6 = 0$ has an integral solution, $\perp$ otherwise is (partial) recursive

Total variants of these functions (with 0 instead of $\perp$) are known not to be recursive for variants of the Collatz sequence (Conway 87) and for polynomials in 38 variables of degree 8 (J.P. Jones).

Variants

Once one agrees on codings of $\mathbb{Z}, \Sigma^\star$, finite subsets of $\mathbb{N}$, one can speak about computable functions from $\Sigma^\star$ to $\Sigma^\star$, etc.

- Factor maps (seen as maps from finite words to finite words) are recursive.
- The function that takes as input a SFT and an integer n, and outputs the number of periodic points of period n is recursive.
- The function that takes as input a SFT X and a word w, and outputs 1 if w is not allowed in X, 0 otherwise is recursive in 1D
- The function that takes as input a SFT X and a word w, and outputs 1 if w is not allowed in X, $\perp$ otherwise is (partial) recursive in 2D

Nonrecursiveness

For cardinality reasons, not all functions are recursive:

- Each function is given by a program, of which there are only countable many. We write $(\phi_i)_{i\in\mathbb{N}}$ for the collection of all programs
- Ideally we would like to say that $f(n) = \phi_n(n) + 1$ is not recursive, but this does not work (as $\bot + 1 = \bot$)
- Instead we do $f(n) = \phi_n(n) + 1$ if $\phi_n(n) \neq \bot$ and $f(n) = 0$ otherwise.
- Then $f(n)$ is not recursive obviously

Nonrecursiveness

$f(n) = \phi_n(n) + 1$ if $\phi_n(n) \neq \perp$ and $f(n) = 0$ otherwise is not (partial) recursive

$g(n) = \phi_n(n)$ *is* (partial) recursive: you can simulate the n-th program of input n.

This means there is no algorithm to test whether $\phi_n(n) = \perp$, which means there is no algorithm to test whether a program terminates.

Reducibilities

Computability theory is not interesting only in computable functions, as they are so few of them

Reducibility relations is the way to measure *how far from computable* a function is

$$f \leq_{XX} g$$

means intuitively that

- f is easier than g
- Assuming someone gives me g in some form, I will be able to compute f

Reducibilities

A lot of different reducibility relations depending on :

- How g is given
- What “computes f” means.

Most of them are similar:

- They are partial preorders. To each $\leq_{XX}$ an equivalence relation $\sim_{XX}$ is defined
- $f \sim_{XX} g$ means f is as easy as g.
- Each equivalence class is called a *degree*
- $\leq_{XX}$ has a minimal element, the *easiest* sets of functions, which usually is the set of “computable” functions.
- Due to diagonalization arguments, usually no maximal element.

Turing reducibility

$f \leq_T g$ if f is recursive with *oracle* g

This is only defined if g is total (f might be partial)
In the code of f, you can call the function g on any value x.

- g is a black box, f has no idea how g operates.

Examples: Given g

- $f(n) = g(n)^2 + 1664g(n) + 666$ is (partial) recursive ($f \leq_T g$)
- Suppose that $\lim g(m)/m > 1$. Then $f(n) = \min\{x | g(n+x) = x\}$ satisfies $f \leq_T g$
- $f(n) = \min\{g(n+x), x \in \mathbb{N}\}$. Then $f \not\leq_T g$ for almost all g.

Properties

The minimal elements are the (total) recursive functions:

- Suppose that $f \leq_T g$ for all g. Then in particuliar $f \leq_T 0$ where 0 is the constant function. This means f is (total) recursive.
- Conversely, if f is (total) recursive, $f \leq_T g$ for all g, as you don't even need g.

There is no maximal elements:

- There are only countably many f s.t. $f \leq_T g$ (each f is given by a program)
- $\leq_T$ is a *partial* relation. In fact $\{(f, g) | f \leq_T g\}$ is meager.

Properties

Let $f(n) = 0$ if the n-th program halts on input n, and 1 otherwise.

- Then $f \not\leq_T 0$ (as f is not recursive)
- The degree of f , the set of all functions g s.t. $g \sim_T f$, is called $0'$.

Functions in $0'$ are the easiest nonrecursive functions to define.
Examples of functions in $0'$:

- Let $g(n, m) = 0$ if the n-th program halts on input m and 1 otherwise
- Let $g(n) = 0$ if the n-th program halts on input 0 and 1 otherwise
- Let $g(n) = m + 1$ if the n-th program halts in m seconds, and 0 otherwise (Assuming the notion of time is well defined).

WARNING: There exists functions h s.t. $0 <_T h <_T 0'$.

Plan

Overview

Computability is defined for functions: $\mathbb{N} \to \mathbb{N}$ but can also be defined for *sets* $S \subseteq \mathbb{N}$.

What is the good definition of computability for sets ?

There are conflicting definitions.
For example

What does it mean for the set S of all written books to be computable ?

Three definitions

Claim 1 (Google)

S is computable because I know a website that can tell me, given an author x and a title y , whether x wrote y.

Claim 2 (Alexandria Library)

S is computable because I have a way to list all existing books by just browsing the library of Alexandria

Claim 3 (Lucien's Library (from Sandman))

S is computable because I have a way to list all non-existing books by just browsing the library of Dream

Comparison

If I want to know if the book “To the Lighthouse” by Virginia Woolf exists:

- I can just use Google
- I can browse the whole Alexandria library until I find the book. If the book does not exist, I will never know
- I can browse the library of Dream until I find the book. If I do, then the book does not exists. If it exists, I will never know

Comparison

If I want to know if the book “The Sound and Nick Fury” by William Faulkner does not exist:

- I can just use Google
- I can browse the whole Alexandria library until I find the book. If the book does not exist, I will never know
- I can browse the library of Dream until I find the book. If I do, then the book does not exists. If it exists, I will never know

Comparison

However:

- If I have access to both libraries, I can answer both existence and nonexistence questions. Therefore both libraries together are equivalent to Google.
- If the library of Alexandria is *sorted*, then it is equivalent to Google.

Definitions

A set S is recursive if χ_S is recursive.

A set S is recursively enumerable if $S = range\, f$ for some partial recursive function f

$S = \{f(n), n \in \mathbb{N}\}$ is an enumeration of S.

A set S is co-recursively enumerable if the complement of S is recursively enumerable

Definitions

Theorem

Equivalence:

- *S is recursively enumerable, i.e. the range of a partial recursive function*
- *S is the domain of a partial recursive function.*
- *(for $S \neq \emptyset$) S is the range of a total recursive function*

Let $S = Dom\, g$ and fix some $a \in S$. On input (i, n):

- Launch an execution of g on input n during i seconds (you need a notion of time)
- If it stops, output n, otherwise output a

Definitions

Theorem

If S is the range of a partial recursive function, then S is the domain of a partial recursive function.

Soppose that $S = \mathit{range}\, f$. On input n:

- For all pairs (i, m) , compute $f(m)$ during exactly i seconds (you need a notion of time)
- If it halts and outputs n, then stop and returns 666
- Otherwise goes to the next pair.

This is called *dovetailing*: we are essentially executing f for all different inputs in parallel.

Properties

S is recursive iff it is both recursively enumerable and corecursively enumerable.

$S = \textit{range } f$ is recursive if f is increasing

Examples

For a function f, let $Diag(f) = \{(n, f(n)), n \in \mathbb{N}\}$.

- If f is total recursive, then $Diag(f)$ is recursive
- If f is partial recursive, then $Diag(f)$ is recursively enumerable
- If $Graph(f)$ is recursively enumerable, then f is partial recursive.

Let f be a total recursive function

- $f(S)$ is recursively enumerable if S is recursively enumerable.
- $f^{-1}(S)$ is recursive if S is recursive.
- $f^{-1}(S)$ is recursively enumerable if S is recursively enumerable.

Examples

- Let X be a SFT. The set of n s.t. there exists a point of period n is recursive.
- Let X be a 1D SFT. The set of patterns that cannot appear in X is recursive.
- Let X be a 2D SFT. The set of patterns that cannot appear in X is recursively enumerable.

Reducibilities

What is the good notion of reduction for sets ?

$A \leq_{XX} B$ iff A is "computable" given B.
Two notions of computable $\rightarrow$ Two notions of reductions.

Turing Reducibility

$S \leq_T S'$ iff $\chi_S \leq_T \chi_{S'}$

Assuming you have a way to decide if any $y \in S'$, you can devise an algorithm that decides whether some $x \in S$.

- For any set S, $S \leq_T \overline{S}$ (where $\overline{S}$ is the complement of S).
- For any set S, $S \times S \leq_T S$
- For any set S, $S^\star \leq_T S$
- For any set S, $f^{-1}(S) \leq_T S$ if f is total recursive.

Enumeration reducibility

Definition

$A \leq_e B$ if A is there is some program h, taking any enumeration of B as input (e.g. as a range of some function) s.t. $A = \textit{range}\, f$.

f uses g as a submodule. g will enumerate B in some random order. f should enumerate A, in an order depending on g.
(To simplify exposition we suppose $B \neq \emptyset$)

- $A \leq_e A$: $h(n) = g(n)$
- $A \times A \leq_e A$: $h(n, m) = (g(n), g(m))$
- If f is total recursive, $f(S) \leq_e S$: $h(n) = f(g(n))$.

Examples

Let F be a 1D set of patterns, and X_F the subshift that forbids all of F.
Let $D(X_F)$ the set of patterns that cannot appear in X_F.
Then $D(X_F) \leq_e F$.

On input (n, w), h works as follows:

- Ask for $F_n = \{g(1), g(2), \dots g(n)\}$, the "first" n forbidden patterns of F
- If w cannot appear in X_F, output w, otherwise output nothing.

Enumeration reducibility

For a subshift X, $\gamma(X) \leq_e D(X)$ means that the object $\gamma(X)$ can be enumerated if we have approximations of X from above by SFTs.

Indeed, if we know only n patterns of $D(X)$ we obtain a surapproximation of X by some SFT.

Examples

Let X be subshift and f a block map. Then $D(f(X)) \leq_e D(X)$.

On input (n, w), h works as follows:

- Ask for $F_n = \{g(1), g(2), \dots g(n)\}$, the "first" n forbidden patterns of $D(X)$
- Compute all words u s.t. $f(u) = w$. If none of them can appear in X_{F_n}, output w, otherwise output nothing.

$$f\left(\bigcap X_i\right) = \bigcap f(X_i)$$

Examples

Let X be a (nonempty) minimal subshift. Then $D(X)^c \leq_e D(X)$.

On input (n, w), h works as follows:

- Ask for $F_n = \{g(1), g(2), \ldots g(n)\}$, the "first" n forbidden patterns of $D(X)$
- If $X_{F_n \cup \{w\}}$ is empty, output w. Otherwise output nothing.

2D

All preceding examples work in 2D, with a slight modification: You cannot know if X_F is empty.

With the same definitions as before, in 2D,
$D(X_F) \leq_e F$.

On input (i, n, w), h works as follows:

- Ask for $F_n = \{g(1), g(2), \dots g(n)\}$, the "first" n forbidden patterns of F
- If no pattern of size i that is valid for F_n contains w at its center, output w, otherwise output nothing.

If X is sofic, then $D(X)$ is recursively enumerable

Application

> Let X be a minimal 2D subshift of finite type.
> Then $D(X)$ is recursive.

Let $X = X_F$ for some finite set F.

- $D(X) \leq_e F$. But F is finite therefore $D(X)$ is enumerable given a finite set, and therefore recursively enumerable.
- $D(X)^C \leq_e D(X)$. $D(X)^c$ is enumerable given a enumerable set, and therefore enumerable (plug into g the recursive function s.t. $range\, g = D(X)$.
- $D(X)$ is recursively enumerable and corecursively enumerable and therefore recursive.

Plan

Models

Computability theory has been defined in this talk informally.

To define it formally, we need a definition of a computer.

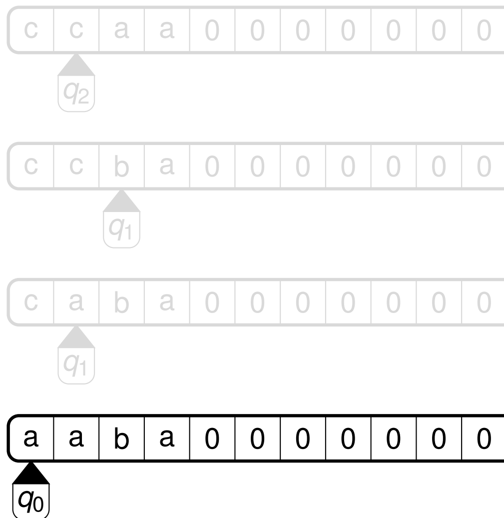
Turing Machines

The theoretical model of computation is *Turing machines*.
In its simplest form, a Turing Machine contains:

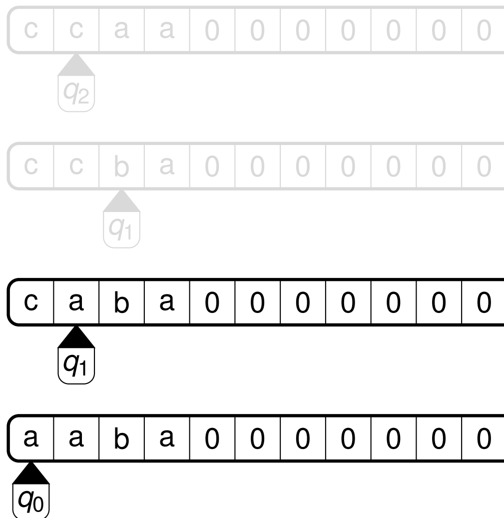
- An infinite tape, that can contain symbols in Σ
- A distinguished position on the tape (the head)
- A state in Q
- An update function $Q \times \Sigma \to Q \times \Sigma \times \{-1, 0, 1\}$

The input is initially written on the tape, and the machine evolves from a specific (initial) state until reaching a specific (halting) state.

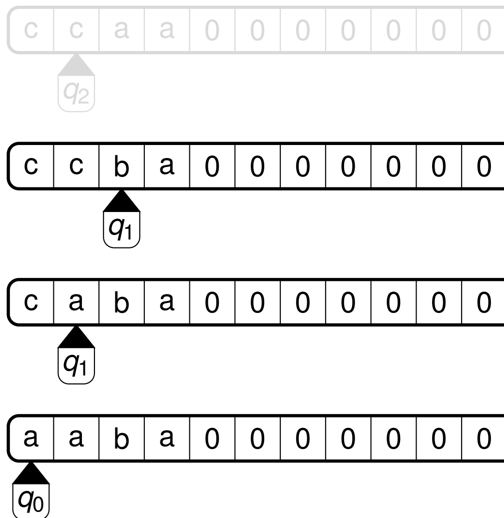
Turing Machines



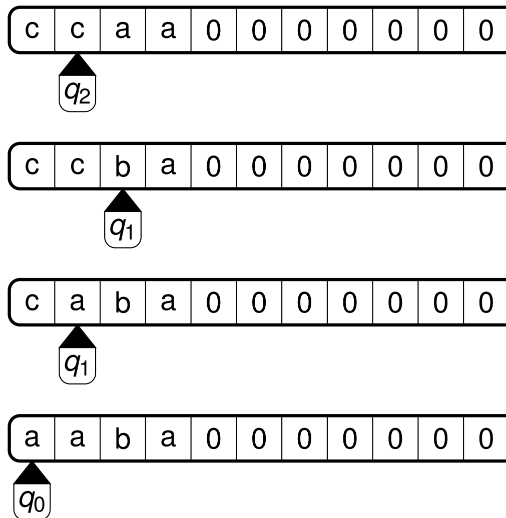
Turing Machines



Turing Machines



Turing Machines



Turing vs p.r.f.

Partial recursive functions are equivalent to Turing machines

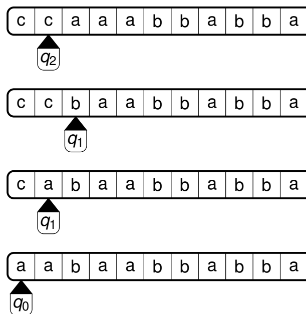
What about programs with black boxes ? We add additional socalled "*oracle tapes*"

It is easier to code programs, but easier to encode machines.

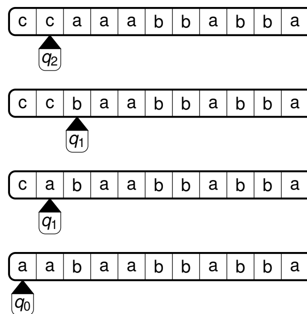
Encodings

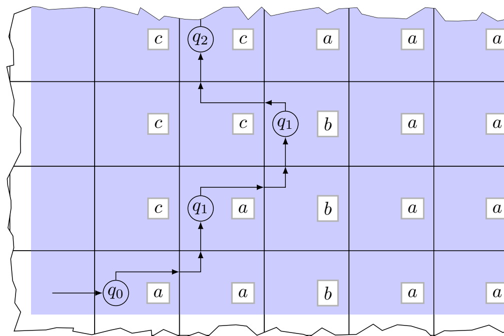
There is an encoding of Turing machines into SFTs.

The proof

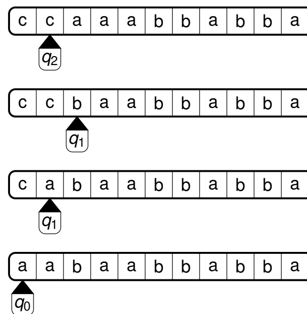


The proof

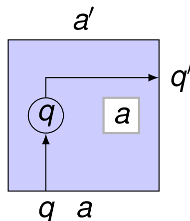




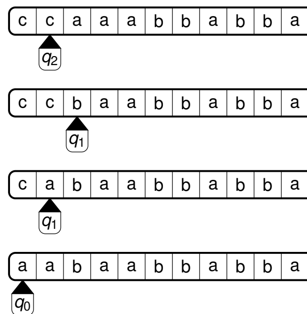
The proof



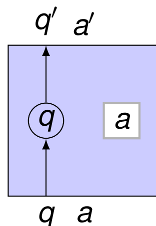
$$(q, a) \longrightarrow (q', a', \rightarrow)$$



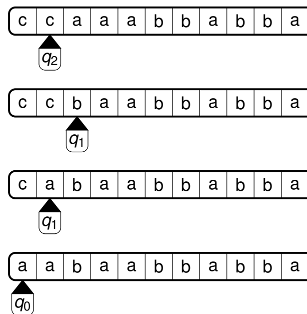
The proof



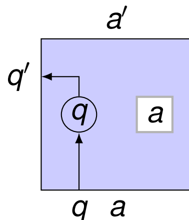
$$(q, a) \longrightarrow (q', a', \uparrow)$$



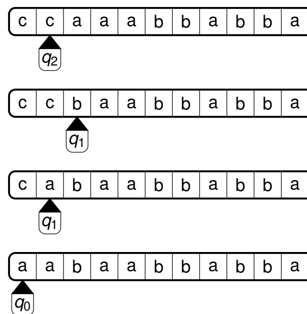
The proof

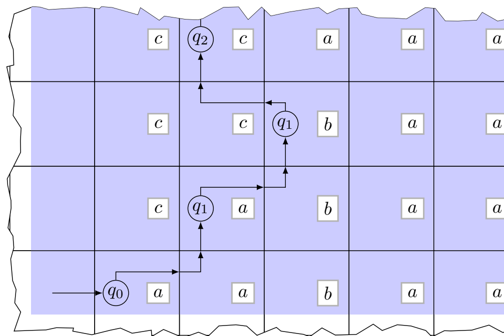


$$(q, a) \longrightarrow (q', a', \leftarrow)$$



The proof





First result

Suppose we start from a machine M which a nonrecursive domain.

- Before building X, we delete the final state from M so that either M runs indefinitely or M hangs.
- To "feed" the input n to the SFT, we only have to look at configurations that contains the $n \times 1$ pattern $u_n = q_0 a^n 0$ at the origin.
- Then u_n can appear in the SFT iff M runs forever on input n.

Therefore the language of X is not recursive.

Theorem (Robinson)

There exists a SFT X with a nonrecursive language.

Berger theorem

Theorem (Berger)

There is no algorithm that decides if a SFT is empty

A SFT can be given by finite patterns and therefore by an integer n. So it means “the set of all empty SFTs is not recursive”.

Idea of the proof

- For each machine M, build a SFT X s.t. X is nonempty iff M does not halt on input 0 (note: a machine is just an integer).
- Previous construction does not work (why ?)