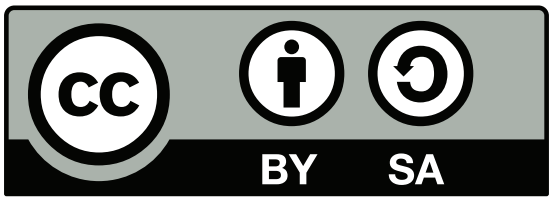
Turing Machines and Computability

Newcastle Junior Algebra Seminar

Graham Campbell

School of Mathematics, Statistics and Physics, Newcastle University, UK

February 2020



Strings

Definition 1 (Alphabet)

An alphabet Σ is a finite set. We call $\sigma \in \Sigma$ a symbol, letter, or character.

Strings

Definition 1 (Alphabet)

An alphabet Σ is a finite set. We call $\sigma \in \Sigma$ a symbol, letter, or character.

Definition 2 (String)

A string over Σ is a finite sequence of symbols from Σ. We usually write these symbols side-by-side (e.g. $w = abcabc$ is a string over $\{a, b, c\}$). We denote the set of all strings over Σ by Σ^*.

Strings

Definition 1 (Alphabet)

An alphabet Σ is a finite set. We call $\sigma \in \Sigma$ a symbol, letter, or character.

Definition 2 (String)

A string over Σ is a finite sequence of symbols from Σ. We usually write these symbols side-by-side (e.g. $w = abcabc$ is a string over $\{a, b, c\}$). We denote the set of all strings over Σ by Σ^*.

Definition 3 (Language)

A language over Σ is simply a subset $L \subseteq \Sigma^*$.

Strings

Definition 1 (Alphabet)

An alphabet Σ is a finite set. We call $\sigma \in \Sigma$ a symbol, letter, or character.

Definition 2 (String)

A string over Σ is a finite sequence of symbols from Σ. We usually write these symbols side-by-side (e.g. $w = abcabc$ is a string over $\{a, b, c\}$). We denote the set of all strings over Σ by Σ^*.

Definition 3 (Language)

A language over Σ is simply a subset $L \subseteq \Sigma^*$.

All languages are countable sets, but there are uncountably many languages over a fixed alphabet. Concatenation of strings is concatenation of sequences, and we denote the empty string by ϵ.

Rules

Definition 4 (Substring)

Given strings u, v over Σ, we say that u is a substring of v ($u \sqsubseteq v$) exactly when u is a subsequence of v.

Rules

Definition 4 (Substring)

Given strings u, v over Σ, we say that u is a substring of v ($u \sqsubseteq v$) exactly when u is a subsequence of v.

Definition 5 (Rewriting Rule)

A string rewriting rule over Σ is a pair of strings $p = (l, r)$.

Rules

Definition 4 (Substring)

Given strings u, v over Σ, we say that u is a substring of v ($u \sqsubseteq v$) exactly when u is a subsequence of v.

Definition 5 (Rewriting Rule)

A string rewriting rule over Σ is a pair of strings $p = (l, r)$.

Definition 6 (Rule Application)

We say that $p = (l, r)$ can be applied to w iff $l \sqsubseteq w$. We write $w \Rightarrow_p z$ to indicate that p has been applied to w, replacing l with r to yield z.

Formally, $\Rightarrow_{(l,r)} = \{(ulv, urv) \mid u, v \in \Sigma^*\} \subseteq \Sigma^* \times \Sigma^*$.

Rules

Definition 4 (Substring)

Given strings u, v over Σ, we say that u is a substring of v ($u \sqsubseteq v$) exactly when u is a subsequence of v.

Definition 5 (Rewriting Rule)

A string rewriting rule over Σ is a pair of strings $p = (l, r)$.

Definition 6 (Rule Application)

We say that $p = (l, r)$ can be applied to w iff $l \sqsubseteq w$. We write $w \Rightarrow_p z$ to indicate that p has been applied to w, replacing l with r to yield z.

Formally, $\Rightarrow_{(l,r)} = \{(ulv, urv) \mid u, v \in \Sigma^*\} \subseteq \Sigma^* \times \Sigma^*$.

Given a set of rules $\mathcal{R}$, we define $\Rightarrow_{\mathcal{R}} = \bigcup_{p \in \mathcal{R}} \Rightarrow_p$. That is, $w \Rightarrow_{\mathcal{R}} z$ means that w can be rewritten to z using one of the rules from $\mathcal{R}$.

SRSs and Grammars

Definition 7 (SRS)

Given an alphabet Σ and a finite set of rules $\mathcal{R}$ over Σ, we call the pair $(\Sigma, \mathcal{R})$ a string rewriting system (SRS).

SRSs and Grammars

Definition 7 (SRS)

Given an alphabet Σ and a finite set of rules $\mathcal{R}$ over Σ, we call the pair $(\Sigma, \mathcal{R})$ a string rewriting system (SRS).

Definition 8 (Grammar)

A grammar is a 4-tuple $\mathcal{G} = (T, N, \mathcal{R}, S)$ where T and N are finite disjoint sets called terminals and non-terminals respectively, $\mathcal{R}$ is a finite set of rules over $T \cup N$ such that each rule's LHS contains at least one non-terminal, and $S \in N$ is the start symbol.

SRSs and Grammars

Definition 7 (SRS)

Given an alphabet Σ and a finite set of rules $\mathcal{R}$ over Σ, we call the pair $(\Sigma, \mathcal{R})$ a string rewriting system (SRS).

Definition 8 (Grammar)

A grammar is a 4-tuple $\mathcal{G} = (T, N, \mathcal{R}, S)$ where T and N are finite disjoint sets called terminals and non-terminals respectively, $\mathcal{R}$ is a finite set of rules over $T \cup N$ such that each rule's LHS contains at least one non-terminal, and $S \in N$ is the start symbol.

Define the accepted language to be the set of all terminal strings derivable from s in 0 or more steps: $\mathrm{L}(\mathcal{G}) := \{w \in T^* \mid s \Rightarrow_{\mathcal{R}}^* w\}$.

Non-terminals really do add generational power. For example, we cannot describe palindromes without them.

Language Classes I

Theorem 9 (Regular Language Characterisations)

Given $L \subseteq \Sigma^*$, the following are equivalent:

1. *L is a recognisable subset of Σ^*;*
2. *L is a rational subset of Σ^*;*
3. *L is generated by a regular grammar;*
4. *L is accepted by a finite state automaton (FSA);*
5. *L is accepted by a deterministic FSA (DFSA);*
6. *L is accepted by a read-only Turing machine.*

Language Classes I

Theorem 9 (Regular Language Characterisations)

Given $L \subseteq \Sigma^*$, the following are equivalent:

1. *L is a recognisable subset of Σ^*;*
2. *L is a rational subset of Σ^*;*
3. *L is generated by a regular grammar;*
4. *L is accepted by a finite state automaton (FSA);*
5. *L is accepted by a deterministic FSA (DFSA);*
6. *L is accepted by a read-only Turing machine.*

Theorem 10 (Context-Free Language Characterisations)

Given $L \subseteq \Sigma^*$, the following are equivalent:

1. *L is generated by a context-free grammar;*
2. *L is accepted by a pushdown automaton.*

Language Classes II

Theorem 11 (Recursively Enumerable Language Characterisations)

Given $L \subseteq \Sigma^*$, the following are equivalent:

1. *L is generated by an unrestricted grammar;*
2. *L is accepted by a Turing Machine.*

Language Classes II

Theorem 11 (Recursively Enumerable Language Characterisations)

Given $L \subseteq \Sigma^*$, the following are equivalent:

1. *L is generated by an unrestricted grammar;*
2. *L is accepted by a Turing Machine.*

Recall that the deterministic context-free languages were those accepted by a deterministic pushdown automaton, and that the recursive languages were those accepted by a Turing Machine that halts on all inputs.

Theorem 12 (Chomsky Hierarchy)

We have the following strict inclusions of language classes:

$$\mathit{Finite} \subset \mathit{Regular} \subset \mathit{DCF} \subset \mathit{CF} \subset \mathit{CS} \subset \mathit{Recursive} \subset \mathit{r.e.}$$

where (D)CF (Deterministic) Context-Free is, CS is Context-Sensitive, and r.e. is Recursively Enumerable.

Introduction

I never actually showed you a formal definition of a Turing Machine last time. There are lots of genuinely useful equivalent definitions. We will use one that lends itself to computing functions.

Introduction

I never actually showed you a formal definition of a Turing Machine last time. There are lots of genuinely useful equivalent definitions. We will use one that lends itself to computing functions.

Our machine will have n read-only one-way input tapes and an unrestricted output tape. Note that it's actually possible (up to encoding) for a single tape machine to simulate our $n + 1$ tape machine.

Introduction

I never actually showed you a formal definition of a Turing Machine last time. There are lots of genuinely useful equivalent definitions. We will use one that lends itself to computing functions.

Our machine will have n read-only one-way input tapes and an unrestricted output tape. Note that it's actually possible (up to encoding) for a single tape machine to simulate our $n+1$ tape machine.

Our machine will be deterministic. This does not result in any reduction in computational power, compared to allowing non-determinism. A deterministic machine can simulate a non-deterministic machine and accept exactly when it accepts.

Introduction

I never actually showed you a formal definition of a Turing Machine last time. There are lots of genuinely useful equivalent definitions. We will use one that lends itself to computing functions.

Our machine will have n read-only one-way input tapes and an unrestricted output tape. Note that it's actually possible (up to encoding) for a single tape machine to simulate our $n+1$ tape machine.

Our machine will be deterministic. This does not result in any reduction in computational power, compared to allowing non-determinism. A deterministic machine can simulate a non-deterministic machine and accept exactly when it accepts.

But what about non-deterministic output? Well, we're only interested in computing functions, so being able to have non-deterministic output simply isn't useful. Moreover, it could be simulated by concatenation of output with a special separator, anyway.

A Formal Definition

We will fix the notation that h_a denotes the accept state, h_r denotes the reject state, and Δ denotes the blank symbol.

A Formal Definition

We will fix the notation that h_a denotes the accept state, h_r denotes the reject state, and Δ denotes the blank symbol.

Definition 13 (Turing Machine)

An n-input Turing Machine is the 5-tuple $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta)$ where

A Formal Definition

We will fix the notation that h_a denotes the accept state, h_r denotes the reject state, and Δ denotes the blank symbol.

Definition 13 (Turing Machine)

An n-input Turing Machine is the 5-tuple $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta)$ where

1. Q is a finite set of states, s.t. $\{h_a, h_r\} \subseteq Q$;

A Formal Definition

We will fix the notation that h_a denotes the accept state, h_r denotes the reject state, and Δ denotes the blank symbol.

Definition 13 (Turing Machine)

An n-input Turing Machine is the 5-tuple $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta)$ where

1. Q is a finite set of states, s.t. $\{h_a, h_r\} \subseteq Q$;
2. Σ is the input alphabet, s.t. $(Q \cup \{\Delta\}) \cap \Sigma = \emptyset$;

A Formal Definition

We will fix the notation that h_a denotes the accept state, h_r denotes the reject state, and Δ denotes the blank symbol.

Definition 13 (Turing Machine)

An n-input Turing Machine is the 5-tuple $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta)$ where

1. Q is a finite set of states, s.t. $\{h_a, h_r\} \subseteq Q$;
2. Σ is the input alphabet, s.t. $(Q \cup \{\Delta\}) \cap \Sigma = \emptyset$;
3. Γ is the tape alphabet, s.t. $(\Sigma \cup \{\Delta\}) \subseteq \Gamma$ and $Q \cap \Gamma = \emptyset$;

A Formal Definition

We will fix the notation that h_a denotes the accept state, h_r denotes the reject state, and Δ denotes the blank symbol.

Definition 13 (Turing Machine)

An n-input Turing Machine is the 5-tuple $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta)$ where

1. Q is a finite set of states, s.t. $\{h_a, h_r\} \subseteq Q$;
2. Σ is the input alphabet, s.t. $(Q \cup \{\Delta\}) \cap \Sigma = \emptyset$;
3. Γ is the tape alphabet, s.t. $(\Sigma \cup \{\Delta\}) \subseteq \Gamma$ and $Q \cap \Gamma = \emptyset$;
4. $q_0 \in Q$ is the initial state;

A Formal Definition

We will fix the notation that h_a denotes the accept state, h_r denotes the reject state, and Δ denotes the blank symbol.

Definition 13 (Turing Machine)

An n-input Turing Machine is the 5-tuple $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta)$ where

1. Q is a finite set of states, s.t. $\{h_a, h_r\} \subseteq Q$;
2. Σ is the input alphabet, s.t. $(Q \cup \{\Delta\}) \cap \Sigma = \emptyset$;
3. Γ is the tape alphabet, s.t. $(\Sigma \cup \{\Delta\}) \subseteq \Gamma$ and $Q \cap \Gamma = \emptyset$;
4. $q_0 \in Q$ is the initial state;
5. $\delta : ((Q \setminus \{h_a, h_r\}) \times \Gamma^{n+1}) \rightharpoonup (Q \times \{S, R\}^n \times \Gamma \times \{L, S, R\})$.

A Formal Definition

We will fix the notation that h_a denotes the accept state, h_r denotes the reject state, and Δ denotes the blank symbol.

Definition 13 (Turing Machine)

An n-input Turing Machine is the 5-tuple $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta)$ where

1. Q is a finite set of states, s.t. $\{h_a, h_r\} \subseteq Q$;
2. Σ is the input alphabet, s.t. $(Q \cup \{\Delta\}) \cap \Sigma = \emptyset$;
3. Γ is the tape alphabet, s.t. $(\Sigma \cup \{\Delta\}) \subseteq \Gamma$ and $Q \cap \Gamma = \emptyset$;
4. $q_0 \in Q$ is the initial state;
5. $\delta : ((Q \setminus \{h_a, h_r\}) \times \Gamma^{n+1}) \rightharpoonup (Q \times \{S, R\}^n \times \Gamma \times \{L, S, R\})$.

So δ is a partial function that says that when we are in a given state, with our $n+1$ tape heads looking at the given $n+1$ symbols, respectively, then we must move the n input tape heads in the way prescribed, replace the symbol at the current position in the output tape, and then move the output tape head.

Machine Configurations

We will use n-TM to abbreviate n-input Turing Machine. Note that in the definition, everything is finite, including the transition (partial) function δ.

Machine Configurations

We will use n-TM to abbreviate n-input Turing Machine. Note that in the definition, everything is finite, including the transition (partial) function δ.

The "current state" of a TM at any given time is called a configuration. Before we start, we need to define an auxiliary function $\Gamma^* \to \Gamma^*$:

Machine Configurations

We will use n-TM to abbreviate n-input Turing Machine. Note that in the definition, everything is finite, including the transition (partial) function δ.

The "current state" of a TM at any given time is called a configuration. Before we start, we need to define an auxiliary function $\Gamma^* \to \Gamma^*$:

Definition 14 (Blank Trimming)

Given a string $w \in \Gamma^*$, by $\texttt{trim}_\Delta(w)$ we mean the largest substring of w such that the last symbol is not Δ.

Machine Configurations

We will use n-TM to abbreviate n-input Turing Machine. Note that in the definition, everything is finite, including the transition (partial) function δ.

The "current state" of a TM at any given time is called a configuration. Before we start, we need to define an auxiliary function $\Gamma^* \to \Gamma^*$:

Definition 14 (Blank Trimming)

Given a string $w \in \Gamma^*$, by $\texttt{trim}_\Delta(w)$ we mean the largest substring of w such that the last symbol is not Δ.

Definition 15 (Configuration)

The set of configurations of an n-TM $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta)$ is:

$$\mathcal{C}_\mathcal{M} = \{(u_1 q v_1, \ldots, u_{n+1} q v_{n+1}) \mid u_i \in \Gamma^*, q \in Q, v_i \in \texttt{trim}_\Delta(\Gamma^*)\}.$$

Machine Configurations

We will use n-TM to abbreviate n-input Turing Machine. Note that in the definition, everything is finite, including the transition (partial) function δ.

The "current state" of a TM at any given time is called a configuration. Before we start, we need to define an auxiliary function $\Gamma^* \to \Gamma^*$:

Definition 14 (Blank Trimming)

Given a string $w \in \Gamma^*$, by $\texttt{trim}_\Delta(w)$ we mean the largest substring of w such that the last symbol is not Δ.

Definition 15 (Configuration)

The set of configurations of an n-TM $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta)$ is:

$$\mathcal{C}_\mathcal{M} = \{(u_1 q v_1, \ldots, u_{n+1} q v_{n+1}) \mid u_i \in \Gamma^*, q \in Q, v_i \in \texttt{trim}_\Delta(\Gamma^*)\}.$$

Note that not all the configurations may be reachable from a given starting configuration (it is in fact undecidable in general to ask!).

The Computation Relation

I'm actually not going to provide a formal definition in here, since it is quite fiddly, and won't help you understand it! It will be sufficient to know that it is the lifting of δ to operate on the configurations, giving us a binary relation $\vdash_{\mathcal{M}}$ on $\mathcal{C}_{\mathcal{M}}$.

The Computation Relation

I'm actually not going to provide a formal definition in here, since it is quite fiddly, and won't help you understand it! It will be sufficient to know that it is the lifting of δ to operate on the configurations, giving us a binary relation $\vdash_{\mathcal{M}}$ on $\mathcal{C}_{\mathcal{M}}$.

There are some subtleties that need addressing, however:

The Computation Relation

I'm actually not going to provide a formal definition in here, since it is quite fiddly, and won't help you understand it! It will be sufficient to know that it is the lifting of δ to operate on the configurations, giving us a binary relation $\vdash_{\mathcal{M}}$ on $\mathcal{C}_{\mathcal{M}}$.

There are some subtleties that need addressing, however:

1. If by moving left, we move off the end of a tape, then we don't allow this. Such cases are replaced with a transition to the reject state (h_r), leaving the all the tape heads in place.

The Computation Relation

I'm actually not going to provide a formal definition in here, since it is quite fiddly, and won't help you understand it! It will be sufficient to know that it is the lifting of δ to operate on the configurations, giving us a binary relation $\vdash_{\mathcal{M}}$ on $\mathcal{C}_{\mathcal{M}}$.

There are some subtleties that need addressing, however:

1. If by moving left, we move off the end of a tape, then we don't allow this. Such cases are replaced with a transition to the reject state (h_r), leaving the all the tape heads in place.
2. The relation is total as a function (it is actually a function since we are defining a deterministic machine). Any undefined transitions are replaced, one again, with a transition to the reject state.

The Computation Relation

I'm actually not going to provide a formal definition in here, since it is quite fiddly, and won't help you understand it! It will be sufficient to know that it is the lifting of δ to operate on the configurations, giving us a binary relation $\vdash_{\mathcal{M}}$ on $\mathcal{C}_{\mathcal{M}}$.

There are some subtleties that need addressing, however:

1. If by moving left, we move off the end of a tape, then we don't allow this. Such cases are replaced with a transition to the reject state (h_r), leaving the all the tape heads in place.
2. The relation is total as a function (it is actually a function since we are defining a deterministic machine). Any undefined transitions are replaced, one again, with a transition to the reject state.

Note that since δ has no transitions from a halting state (h_r or h_a), then a configuration is halted (in state h_r or h_a) iff there is no successor according to $\vdash_{\mathcal{M}}$. That is, either a machine is in a halted state and no progress can be made, or it is not and progress can be made.

TMs and Languages

Definition 16 (Recognised Language)

We define the language recognised by 1-TM $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta)$:

$$\mathrm{L}(\mathcal{M}) = \{w \in \Sigma^* \mid \exists \gamma_1, \gamma_1', \gamma_2, \gamma_2' \in \Gamma^*, (q_0 w, q_0) \vdash_{\mathcal{M}}^* (\gamma_1 h_a \gamma_1', \gamma_2 h_a \gamma_2')\}.$$

TMs and Languages

Definition 16 (Recognised Language)

We define the language recognised by 1-TM $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta)$:

$$\mathrm{L}(\mathcal{M}) = \{w \in \Sigma^* \mid \exists \gamma_1, \gamma_1', \gamma_2, \gamma_2' \in \Gamma^*, (q_0 w, q_0) \vdash_{\mathcal{M}}^* (\gamma_1 h_a \gamma_1', \gamma_2 h_a \gamma_2')\}.$$

Definition 17 (Recursive Languages)

1. A language $L \subseteq \Sigma^*$ is recursively enumerable (r.e.) iff there exists a 1-TM $\mathcal{M}$ over Σ such that $\mathrm{L}(\mathcal{M}) = L$.

TMs and Languages

Definition 16 (Recognised Language)

We define the language recognised by 1-TM $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta)$:

$$\mathrm{L}(\mathcal{M}) = \{w \in \Sigma^* \mid \exists \gamma_1, \gamma_1', \gamma_2, \gamma_2' \in \Gamma^*, (q_0 w, q_0) \vdash_{\mathcal{M}}^* (\gamma_1 h_a \gamma_1', \gamma_2 h_a \gamma_2')\}.$$

Definition 17 (Recursive Languages)

1. A language $L \subseteq \Sigma^*$ is recursively enumerable (r.e.) iff there exists a 1-TM $\mathcal{M}$ over Σ such that $\mathrm{L}(\mathcal{M}) = L$.
2. A language $L \subseteq \Sigma^*$ is recursively iff there exists a 1-TM $\mathcal{M}$ over Σ such that $\mathrm{L}(\mathcal{M}) = L$ that halts on all inputs.

TMs and Languages

Definition 16 (Recognised Language)

We define the language recognised by 1-TM $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta)$:

$$\mathrm{L}(\mathcal{M}) = \{w \in \Sigma^* \mid \exists \gamma_1, \gamma_1', \gamma_2, \gamma_2' \in \Gamma^*, (q_0 w, q_0) \vdash_{\mathcal{M}}^* (\gamma_1 h_a \gamma_1', \gamma_2 h_a \gamma_2')\}.$$

Definition 17 (Recursive Languages)

1. A language $L \subseteq \Sigma^*$ is recursively enumerable (r.e.) iff there exists a 1-TM $\mathcal{M}$ over Σ such that $\mathrm{L}(\mathcal{M}) = L$.
2. A language $L \subseteq \Sigma^*$ is recursively iff there exists a 1-TM $\mathcal{M}$ over Σ such that $\mathrm{L}(\mathcal{M}) = L$ that halts on all inputs.

Proposition 18

A language $L \subseteq \Sigma^*$ is recursive iff both L and $\Sigma^* \setminus L$ are r.e..

Encoded TMs

It is possible to uniquely (up to renaming of states) encode every Turing Machine as a string over $\{0, 1\}$.

Encoded TMs

It is possible to uniquely (up to renaming of states) encode every Turing Machine as a string over $\{0, 1\}$. The detail is not interesting (other than noting that we must restrict the universe of states and alphabets to some countable set), but it is useful to be able to speak of the "code" of a TM. We will call this encoding function e.

Encoded TMs

It is possible to uniquely (up to renaming of states) encode every Turing Machine as a string over $\{0, 1\}$. The detail is not interesting (other than noting that we must restrict the universe of states and alphabets to some countable set), but it is useful to be able to speak of the "code" of a TM. We will call this encoding function e.

Proposition 19

Let $SA = \{e(\mathcal{M}) \mid \mathcal{M}\ is\ a\ 1\text{-}TM\ and\ e(\mathcal{M}) \in L(\mathcal{M})\}$ *be the codes of all* $1-$*TMs that accept themselves, and let* $NSA = \{0, 1\}^* \setminus SA$ *be the complement (all machines that don't accept themselves, and everything that isn't an encoding of a* 1*-TM).*

Encoded TMs

It is possible to uniquely (up to renaming of states) encode every Turing Machine as a string over $\{0, 1\}$. The detail is not interesting (other than noting that we must restrict the universe of states and alphabets to some countable set), but it is useful to be able to speak of the "code" of a TM. We will call this encoding function e.

Proposition 19

Let $SA = \{e(\mathcal{M}) \mid \mathcal{M}\ is\ a\ 1\text{-}TM\ and\ e(\mathcal{M}) \in L(\mathcal{M})\}$ be the codes of all $1-$TMs that accept themselves, and let $NSA = \{0,1\}^* \setminus SA$ be the complement (all machines that don't accept themselves, and everything that isn't an encoding of a 1-TM).

Then SA is r.e. but not recursive, and NSA is not even r.e..

Encoded TMs

It is possible to uniquely (up to renaming of states) encode every Turing Machine as a string over $\{0, 1\}$. The detail is not interesting (other than noting that we must restrict the universe of states and alphabets to some countable set), but it is useful to be able to speak of the "code" of a TM. We will call this encoding function e.

Proposition 19

Let $SA = \{e(\mathcal{M}) \mid \mathcal{M}\ is\ a\ 1\text{-}TM\ and\ e(\mathcal{M}) \in L(\mathcal{M})\}$ be the codes of all $1-$TMs that accept themselves, and let $NSA = \{0,1\}^* \setminus SA$ be the complement (all machines that don't accept themselves, and everything that isn't an encoding of a 1-TM).

Then SA is r.e. but not recursive, and NSA is not even r.e..

Proposition 20

Most languages are not even r.e.. There are $|\mathbb{R}|$ languages over $\{0, 1\}$ but only $|\mathbb{N}|$ r.e. languages, since there are only countably many TMs!

Decision Problems

Definition 21 (Decision Problem)

A decision problem is a set of questions, each of which has a yes or no answer.

Decision Problems

Definition 21 (Decision Problem)

A decision problem is a set of questions, each of which has a yes or no answer.

We can view the language SA as a decision problem.

Example 22 (Self Accepting Problem)

Input: A 1-TM $\mathcal{M}$.
Question: Does it accept its own code?

Decision Problems

Definition 21 (Decision Problem)

A decision problem is a set of questions, each of which has a yes or no answer.

We can view the language SA as a decision problem.

Example 22 (Self Accepting Problem)

Input: A 1-TM $\mathcal{M}$.
Question: Does it accept its own code?

Encoding is usually left implicit. In particular, we can see that language of yes-instances is exactly SA, and the no-instances is $\mathrm{NSA} \cap \mathrm{EM}$ where EM is the language of all encoded 1-TMs.

Decision Problems

Definition 21 (Decision Problem)

A decision problem is a set of questions, each of which has a yes or no answer.

We can view the language SA as a decision problem.

Example 22 (Self Accepting Problem)

Input: A 1-TM $\mathcal{M}$.
Question: Does it accept its own code?

Encoding is usually left implicit. In particular, we can see that language of yes-instances is exactly SA, and the no-instances is $\mathrm{NSA} \cap \mathrm{EM}$ where EM is the language of all encoded 1-TMs.

Definition 23 (Decidable Problem)

We call a problem decidable whenever its yes-instances are a recursive language, and undecidable otherwise (up to a "reasonable" encoding).

The Membership Problem

There are two important versions of the membership problem.

The Membership Problem

There are two important versions of the membership problem.

Example 24 (Membership Problem)

Input: A string $w \in \Sigma^*$.
Question: Is $w \in \mathrm{L}(\mathcal{M})$, for some fixed 1-TM $\mathcal{M}$?

The Membership Problem

There are two important versions of the membership problem.

Example 24 (Membership Problem)

Input: A string $w \in \Sigma^*$.
Question: Is $w \in \mathrm{L}(\mathcal{M})$, for some fixed 1-TM $\mathcal{M}$?

Example 25 (Universal Membership Problem)

Input: A string $w \in \Sigma^*$ and a 1-TM $\mathcal{M}$.
Question: Is $w \in \mathrm{L}(\mathcal{M})$?

The Membership Problem

There are two important versions of the membership problem.

Example 24 (Membership Problem)

Input: A string $w \in \Sigma^*$.
Question: Is $w \in \mathrm{L}(\mathcal{M})$, for some fixed 1-TM $\mathcal{M}$?

Example 25 (Universal Membership Problem)

Input: A string $w \in \Sigma^*$ and a 1-TM $\mathcal{M}$.
Question: Is $w \in \mathrm{L}(\mathcal{M})$?

In the first case, the machine is fixed, and so often it will be decidable (even if the machine is not terminating - there may exist a different machine that recognises the same language and does terminate on all inputs!). The universal version is obviously undecidable (and implicitly has as input both the encoded input string and the encoded machine).

More Undecidable Problems

Example 26 (Uniform Halting Problem)

Input: A TM $\mathcal{M}$.
Question: Does $\mathcal{M}$ halt on all inputs?

More Undecidable Problems

Example 26 (Uniform Halting Problem)

Input: A TM $\mathcal{M}$.
Question: Does $\mathcal{M}$ halt on all inputs?

Example 27 (CF Equivalence Problem)

Input: Two context-free grammars $\mathcal{G}_1, \mathcal{G}_2$.
Question: Does $\mathrm{L}(\mathcal{G}_1) = \mathrm{L}(\mathcal{G}_2)$?

More Undecidable Problems

Example 26 (Uniform Halting Problem)

Input: A TM $\mathcal{M}$.
Question: Does $\mathcal{M}$ halt on all inputs?

Example 27 (CF Equivalence Problem)

Input: Two context-free grammars $\mathcal{G}_1, \mathcal{G}_2$.
Question: Does $\mathrm{L}(\mathcal{G}_1) = \mathrm{L}(\mathcal{G}_2)$?

Example 28 (Finiteness Test)

Input: A finite group presentation (S, R).
Question: Is the group finite?

More Undecidable Problems

Example 26 (Uniform Halting Problem)

Input: A TM $\mathcal{M}$.
Question: Does $\mathcal{M}$ halt on all inputs?

Example 27 (CF Equivalence Problem)

Input: Two context-free grammars $\mathcal{G}_1, \mathcal{G}_2$.
Question: Does $\mathrm{L}(\mathcal{G}_1) = \mathrm{L}(\mathcal{G}_2)$?

Example 28 (Finiteness Test)

Input: A finite group presentation (S, R).
Question: Is the group finite?

Example 29 (Abalian Test)

Input: A finite group presentation (S, R).
Question: Is the group Abelian?

TMs and Functions

Definition 30 (Computed Function)

Let $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta)$ be an n-TM. We can define the (partial) function $f_{\mathcal{M}} : (\Sigma^*)^n \rightharpoonup \Sigma^*$ by its graph:

$$\{(w_1, \ldots, w_n, w) \in (\Sigma^*)^{n+1} \mid \exists \gamma_1, \ldots, \gamma_n, \gamma'_1, \ldots, \gamma'_n \in \Gamma^*, w \in \Sigma^*,$$
$$(q_0 w_1, \ldots, q_0 w_n, q_n) \vdash^*_{\mathcal{M}} (\gamma_1 h_a \gamma'_1, \ldots, \gamma_n h_a \gamma'_n, w h_a)\}.$$

If the machine does something other than finish in one of the prescribed configurations, then we say the function is undefined at that value.

TMs and Functions

Definition 30 (Computed Function)

Let $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta)$ be an n-TM. We can define the (partial) function $f_{\mathcal{M}} : (\Sigma^*)^n \rightharpoonup \Sigma^*$ by its graph:

$$\{(w_1, \ldots, w_n, w) \in (\Sigma^*)^{n+1} \mid \exists \gamma_1, \ldots, \gamma_n, \gamma'_1, \ldots, \gamma'_n \in \Gamma^*, w \in \Sigma^*,$$
$$(q_0 w_1, \ldots, q_0 w_n, q_n) \vdash^*_{\mathcal{M}} (\gamma_1 h_a \gamma'_1, \ldots, \gamma_n h_a \gamma'_n, w h_a)\}.$$

If the machine does something other than finish in one of the prescribed configurations, then we say the function is undefined at that value.

Definition 31 (Characteristic Function)

The characteristic function of a language $L \subseteq \Sigma^*$ is the total function $\chi_L : \Sigma^* \to \{w_a, w_r\}$ where $w_a, w_r \in \Sigma^*$ are two distinct strings and $\forall w \in \Sigma^*, \chi_L(w) = w_a$ iff $w \in L$.

TMs and Functions

Definition 30 (Computed Function)

Let $\mathcal{M} = (Q, \Sigma, \Gamma, q_0, \delta)$ be an n-TM. We can define the (partial) function $f_{\mathcal{M}} : (\Sigma^*)^n \rightharpoonup \Sigma^*$ by its graph:

$$\{(w_1, \ldots, w_n, w) \in (\Sigma^*)^{n+1} \mid \exists \gamma_1, \ldots, \gamma_n, \gamma'_1, \ldots, \gamma'_n \in \Gamma^*, w \in \Sigma^*, (q_0 w_1, \ldots, q_0 w_n, q_n) \vdash^*_{\mathcal{M}} (\gamma_1 h_a \gamma'_1, \ldots, \gamma_n h_a \gamma'_n, w h_a)\}.$$

If the machine does something other than finish in one of the prescribed configurations, then we say the function is undefined at that value.

Definition 31 (Characteristic Function)

The characteristic function of a language $L \subseteq \Sigma^*$ is the total function $\chi_L : \Sigma^* \to \{w_a, w_r\}$ where $w_a, w_r \in \Sigma^*$ are two distinct strings and $\forall w \in \Sigma^*, \chi_L(w) = w_a$ iff $w \in L$.

Proposition 32

$L \subseteq \Sigma^*$ is recursive iff its characteristic function is computable.

Encoding Numbers

Anyone who studied a logic course will be aware of Gödel numberings: a way to identify logical sentences with numbers. Similarly, we can do this with strings over an arbitrary an alphabet, and vice versa.

Encoding Numbers

Anyone who studied a logic course will be aware of Gödel numberings: a way to identify logical sentences with numbers. Similarly, we can do this with strings over an arbitrary an alphabet, and vice versa.

Definition 33 (Bijective Standard Numbering)

For a given alphabet $\Sigma = \{a_1, \ldots, a_n\}$, define the numbering $\upsilon_\Sigma : \mathbb{N} \to \Sigma^*$ by $\upsilon_\Sigma^{-1}(\epsilon) = 0$ and $\upsilon_\Sigma^{-1}(a_{i_k} \cdots a_{i_0}) = i_k \cdot n^k + \cdots + i_0 \cdot n^0$.

Encoding Numbers

Anyone who studied a logic course will be aware of Gödel numberings: a way to identify logical sentences with numbers. Similarly, we can do this with strings over an arbitrary an alphabet, and vice versa.

Definition 33 (Bijective Standard Numbering)

For a given alphabet $\Sigma = \{a_1, \ldots, a_n\}$, define the numbering $\upsilon_\Sigma : \mathbb{N} \to \Sigma^*$ by $\upsilon_\Sigma^{-1}(\epsilon) = 0$ and $\upsilon_\Sigma^{-1}(a_{i_k} \cdots a_{i_0}) = i_k \cdot n^k + \cdots + i_0 \cdot n^0$.

Definition 34 (Computable Function)

Call a (partial) function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ computable iff $\upsilon_\Sigma \circ f \circ \upsilon_\Sigma^{-1}$ is computable (and correspondingly for $f : \mathbb{N}^m \rightharpoonup \mathbb{N}$).

Encoding Numbers

Anyone who studied a logic course will be aware of Gödel numberings: a way to identify logical sentences with numbers. Similarly, we can do this with strings over an arbitrary an alphabet, and vice versa.

Definition 33 (Bijective Standard Numbering)

For a given alphabet $\Sigma = \{a_1, \ldots, a_n\}$, define the numbering $\upsilon_\Sigma : \mathbb{N} \to \Sigma^*$ by $\upsilon_\Sigma^{-1}(\epsilon) = 0$ and $\upsilon_\Sigma^{-1}(a_{i_k} \cdots a_{i_0}) = i_k \cdot n^k + \cdots + i_0 \cdot n^0$.

Definition 34 (Computable Function)

Call a (partial) function $f : \mathbb{N} \rightharpoonup \mathbb{N}$ computable iff $\upsilon_\Sigma \circ f \circ \upsilon_\Sigma^{-1}$ is computable (and correspondingly for $f : \mathbb{N}^m \rightharpoonup \mathbb{N}$).

Example 35

1. The projection functions are computable.
2. The Cantor pairing function is computable.

Church-Turing Thesis

Definition 36 (Church-Turing Thesis)

The Church-Turing Thesis says that a function on the natural numbers can be calculated by an effective method iff it is computable by a Turing machine. That is, Turing Machine can compute anything that can be computed in finite time, given finite input.

Church-Turing Thesis

Definition 36 (Church-Turing Thesis)

The Church-Turing Thesis says that a function on the natural numbers can be calculated by an effective method iff it is computable by a Turing machine. That is, Turing Machine can compute anything that can be computed in finite time, given finite input.

The Church-Turing thesis cannot be "proved": it is a philosophical standpoint (much like most of us believe the continuum hypothesis). However, there is compelling evidence for the Thesis (enough to convince the originally skeptical Gödel)...

Church-Turing Thesis

Definition 36 (Church-Turing Thesis)

The Church-Turing Thesis says that a function on the natural numbers can be calculated by an effective method iff it is computable by a Turing machine. That is, Turing Machine can compute anything that can be computed in finite time, given finite input.

The Church-Turing thesis cannot be "proved": it is a philosophical standpoint (much like most of us believe the continuum hypothesis). However, there is compelling evidence for the Thesis (enough to convince the originally skeptical Gödel)...

Theorem 37 (Church (1936), Turing (1937))

A function on the natural numbers is computable (in the sense we have defined) iff it is general recursive (a notion of Gödel, 1933) iff it is λ-computable (a notion of Church 1936).

Beyond Countable

Everything we have seen so far only details with functions on countable domains. Kleene, Church, Turing were also particularly aware of this issue in the 1930s when notion of recursive functions on the naturals were being developed. Turing first developed a notation of computable real numbers in 1936.

Beyond Countable

Everything we have seen so far only details with functions on countable domains. Kleene, Church, Turing were also particularly aware of this issue in the 1930s when notion of recursive functions on the naturals were being developed. Turing first developed a notation of computable real numbers in 1936.

Even now, there is no generally accepted definition of computability on the real numbers (and other). I will be following Wihrauch (2000)'s view on the subject (but not necessarily all their notation), which is based on their own work, rooted in the definition of computable real functions based on the work by Grzegoizyk (1955) and Hauck (1973, 1978, 1980, 1981, 1982).

Beyond Countable

Everything we have seen so far only details with functions on countable domains. Kleene, Church, Turing were also particularly aware of this issue in the 1930s when notion of recursive functions on the naturals were being developed. Turing first developed a notation of computable real numbers in 1936.

Even now, there is no generally accepted definition of computability on the real numbers (and other). I will be following Wihrauch (2000)'s view on the subject (but not necessarily all their notation), which is based on their own work, rooted in the definition of computable real functions based on the work by Grzegoizyk (1955) and Hauck (1973, 1978, 1980, 1981, 1982).

We (Wihrauch) call this theory Type-2 Theory of Effectivity (TTE), and the corresponding machines Type-2 Machines. The study of such machine models is called Computable Analysis.

Computable Analysis

Some basic questions computable analysis wants to formalise and ask are:

Computable Analysis

Some basic questions computable analysis wants to formalise and ask are:

1. Is the exponential function computable?

Computable Analysis

Some basic questions computable analysis wants to formalise and ask are:

1. Is the exponential function computable?
2. Are union and intersection of closed subsets of the real plane computable?

Computable Analysis

Some basic questions computable analysis wants to formalise and ask are:

1. Is the exponential function computable?
2. Are union and intersection of closed subsets of the real plane computable?
3. Are differentiation and integration computable operators?

Computable Analysis

Some basic questions computable analysis wants to formalise and ask are:

1. Is the exponential function computable?
2. Are union and intersection of closed subsets of the real plane computable?
3. Are differentiation and integration computable operators?
4. Is zero-finding for complex polynomials computable?

Computable Analysis

Some basic questions computable analysis wants to formalise and ask are:

1. Is the exponential function computable?
2. Are union and intersection of closed subsets of the real plane computable?
3. Are differentiation and integration computable operators?
4. Is zero-finding for complex polynomials computable?

In their 1996 paper "Complexity and Real Computation: A Manifesto", Blum, Cucker, Shub and Smale say:

Computable Analysis

Some basic questions computable analysis wants to formalise and ask are:

1. Is the exponential function computable?
2. Are union and intersection of closed subsets of the real plane computable?
3. Are differentiation and integration computable operators?
4. Is zero-finding for complex polynomials computable?

In their 1996 paper "Complexity and Real Computation: A Manifesto", Blum, Cucker, Shub and Smale say:

Our perspective is to formulate the laws of computation. Thus, we write not from the point of view of an engineer who looks for a good algorithm [...]. The perspective is more that of a physicist, truing to understand the laws of scientific computation [...]

Computable Analysis

Some basic questions computable analysis wants to formalise and ask are:

1. Is the exponential function computable?
2. Are union and intersection of closed subsets of the real plane computable?
3. Are differentiation and integration computable operators?
4. Is zero-finding for complex polynomials computable?

In their 1996 paper "Complexity and Real Computation: A Manifesto", Blum, Cucker, Shub and Smale say:

Our perspective is to formulate the laws of computation. Thus, we write not from the point of view of an engineer who looks for a good algorithm [...]. The perspective is more that of a physicist, truing to understand the laws of scientific computation [...]

There is a substantial conflict between theoretical computer science and numerical analysis. These two subjects with common goals have grown apart. [...]

Computable Analysis

Some basic questions computable analysis wants to formalise and ask are:

1. Is the exponential function computable?
2. Are union and intersection of closed subsets of the real plane computable?
3. Are differentiation and integration computable operators?
4. Is zero-finding for complex polynomials computable?

In their 1996 paper "Complexity and Real Computation: A Manifesto", Blum, Cucker, Shub and Smale say:

Our perspective is to formulate the laws of computation. Thus, we write not from the point of view of an engineer who looks for a good algorithm [...]. The perspective is more that of a physicist, truing to understand the laws of scientific computation [...]

There is a substantial conflict between theoretical computer science and numerical analysis. These two subjects with common goals have grown apart. [...]

The conflict has its roots in another age-old conflict, that between the continuous and the discrete [...] Algorithms are primarily a means to solve practical problems. There is not even a formal definition of algorithm in the subject [of numerical analysis]. [...] Thus, we view numerical analysis as an eclectic subject with weak foundations; this certainly in no way denies the great achievements through the centuries.

Infinite Words

A real number cannot be described by finite data, and so our current model of finite sequences of symbols as machine input will not suffice. We need to name real numbers using infinite sequences of symbols.

Infinite Words

A real number cannot be described by finite data, and so our current model of finite sequences of symbols as machine input will not suffice. We need to name real numbers using infinite sequences of symbols.

Definition 38 (Infinite String)

Define $\Sigma^\omega = \{p \mid p : \mathbb{N} \to \Sigma\}$ to be all infinite sequences over Σ.

Infinite Words

A real number cannot be described by finite data, and so our current model of finite sequences of symbols as machine input will not suffice. We need to name real numbers using infinite sequences of symbols.

Definition 38 (Infinite String)

Define $\Sigma^\omega = \{p \mid p : \mathbb{N} \to \Sigma\}$ to be all infinite sequences over Σ.

Definition 39 (Naming System)

A naming system of a set M is a surjective function $\upsilon : Y \rightharpoonup M$ where $Y \in \Sigma^*, \Sigma^\omega$. We say that $p \in Y$ is a υ-name of $x \in M$ iff $\upsilon : Y \rightharpoonup M$ is a naming system and $\upsilon(p) = x$.

Infinite Words

A real number cannot be described by finite data, and so our current model of finite sequences of symbols as machine input will not suffice. We need to name real numbers using infinite sequences of symbols.

Definition 38 (Infinite String)

Define $\Sigma^\omega = \{p \mid p : \mathbb{N} \to \Sigma\}$ to be all infinite sequences over Σ.

Definition 39 (Naming System)

A naming system of a set M is a surjective function $\upsilon : Y \rightharpoonup M$ where $Y \in \Sigma^*, \Sigma^\omega$. We say that $p \in Y$ is a υ-name of $x \in M$ iff $\upsilon : Y \rightharpoonup M$ is a naming system and $\upsilon(p) = x$.

We have already seen a naming system for the natural numbers in Σ^*. Leaving the countable setting. It will turn out that different naming systems will induce different computability theories (no time for all the detail), but I will give a concrete example at the end...

Towards Type-2 Machines

Recall our definition of a Turing Machine had n one-way read-only input tapes, and a single output tape.

Towards Type-2 Machines

Recall our definition of a Turing Machine had n one-way read-only input tapes, and a single output tape. Without increasing power, we could add k additional unrestricted tapes, called "work tapes" (it is possible to simulate them on a single tape).

Towards Type-2 Machines

Recall our definition of a Turing Machine had n one-way read-only input tapes, and a single output tape. Without increasing power, we could add k additional unrestricted tapes, called "work tapes" (it is possible to simulate them on a single tape).

If we insist now that our output tape cannot move backwards, and after writing a non-blank symbol, it must immediately move forwards, so long as we have at least one work tape, we have not decreased our power (if we don't have a work tape, we have just become a (multi-input) FSA).

Towards Type-2 Machines

Recall our definition of a Turing Machine had n one-way read-only input tapes, and a single output tape. Without increasing power, we could add k additional unrestricted tapes, called "work tapes" (it is possible to simulate them on a single tape).

If we insist now that our output tape cannot move backwards, and after writing a non-blank symbol, it must immediately move forwards, so long as we have at least one work tape, we have not decreased our power (if we don't have a work tape, we have just become a (multi-input) FSA).

So, we have an some inputs that are read in order, and an output that is written in order and once each symbol is written, it cannot be changed.

Towards Type-2 Machines

Recall our definition of a Turing Machine had n one-way read-only input tapes, and a single output tape. Without increasing power, we could add k additional unrestricted tapes, called "work tapes" (it is possible to simulate them on a single tape).

If we insist now that our output tape cannot move backwards, and after writing a non-blank symbol, it must immediately move forwards, so long as we have at least one work tape, we have not decreased our power (if we don't have a work tape, we have just become a (multi-input) FSA).

So, we have an some inputs that are read in order, and an output that is written in order and once each symbol is written, it cannot be changed. This is going to be perfect for modifying our machine definition to handle infinite length inputs and outputs.

Towards Type-2 Machines

Recall our definition of a Turing Machine had n one-way read-only input tapes, and a single output tape. Without increasing power, we could add k additional unrestricted tapes, called "work tapes" (it is possible to simulate them on a single tape).

If we insist now that our output tape cannot move backwards, and after writing a non-blank symbol, it must immediately move forwards, so long as we have at least one work tape, we have not decreased our power (if we don't have a work tape, we have just become a (multi-input) FSA).

So, we have an some inputs that are read in order, and an output that is written in order and once each symbol is written, it cannot be changed. This is going to be perfect for modifying our machine definition to handle infinite length inputs and outputs. The idea is that machines can still only run for finite time, but we'd like for them to converge on the answer. The more time we give them, the more precise an answer they provide us with, and they cannot change their mind about what they wrote before.

Type-2 Machines

Definition 40 (Type-2 Machines)

A Type-2 Machine is an n-TM $\mathcal{M}$ together with a type specification $(Y_1, \ldots, Y_n, Y_0)$ with $Y_i \in \{\Sigma^*, \Sigma^\omega\}$ $(0 \leq i \leq n)$, giving the type for each input tape and the output tape.

Type-2 Machines

Definition 40 (Type-2 Machines)

A Type-2 Machine is an n-TM $\mathcal{M}$ together with a type specification $(Y_1, \ldots, Y_n, Y_0)$ with $Y_i \in \{\Sigma^*, \Sigma^\omega\}$ $(0 \leq i \leq n)$, giving the type for each input tape and the output tape.

Definition 41 (Computed Function)

We can now define the string function $f_{\mathcal{M}} : Y_1 \times \cdots \times Y_n \rightharpoonup Y_0$ computed by a Type-2 Machine $\mathcal{M}$. The initial tape configuration for input $(y_1, \ldots, y_n)$ is given in the obvious way. We define:

Type-2 Machines

Definition 40 (Type-2 Machines)

A Type-2 Machine is an n-TM $\mathcal{M}$ together with a type specification $(Y_1, \ldots, Y_n, Y_0)$ with $Y_i \in \{\Sigma^*, \Sigma^\omega\}$ $(0 \leq i \leq n)$, giving the type for each input tape and the output tape.

Definition 41 (Computed Function)

We can now define the string function $f_{\mathcal{M}} : Y_1 \times \cdots \times Y_n \rightharpoonup Y_0$ computed by a Type-2 Machine $\mathcal{M}$. The initial tape configuration for input $(y_1, \ldots, y_n)$ is given in the obvious way. We define:

1. Case $Y_0 = \Sigma^* : f_{\mathcal{M}}(y_1, \ldots, y_n) := y_0 \in \Sigma^*$ iff $\mathcal{M}$ halts on input $(y_1, \ldots, y_n)$ and writes y_0 on the output tape.

Type-2 Machines

Definition 40 (Type-2 Machines)

A Type-2 Machine is an n-TM $\mathcal{M}$ together with a type specification $(Y_1, \ldots, Y_n, Y_0)$ with $Y_i \in \{\Sigma^*, \Sigma^\omega\}$ $(0 \leq i \leq n)$, giving the type for each input tape and the output tape.

Definition 41 (Computed Function)

We can now define the string function $f_{\mathcal{M}} : Y_1 \times \cdots \times Y_n \rightharpoonup Y_0$ computed by a Type-2 Machine $\mathcal{M}$. The initial tape configuration for input $(y_1, \ldots, y_n)$ is given in the obvious way. We define:

1. Case $Y_0 = \Sigma^* : f_{\mathcal{M}}(y_1, \ldots, y_n) := y_0 \in \Sigma^*$ iff $\mathcal{M}$ halts on input $(y_1, \ldots, y_n)$ and writes y_0 on the output tape.
2. Case $Y_0 = \Sigma^\omega : f_{\mathcal{M}}(y_1, \ldots, y_n) := y_0 \in \Sigma^\omega$ iff $\mathcal{M}$ computes forever on input $(y_1, \ldots, y_n)$ and writes y_0 on the output tape.

Computability Refines Continuity

Roughly speaking, the finiteness property says that, given a Type-2 Machine $\mathcal{M}$ such that $f_{\mathcal{M}} : \Sigma^\omega \rightharpoonup \Sigma^\omega$, every finite portion of the output $f_{\mathcal{M}}(p)$ is already determined by a finite portion of the input p.

Computability Refines Continuity

Roughly speaking, the finiteness property says that, given a Type-2 Machine $\mathcal{M}$ such that $f_{\mathcal{M}} : \Sigma^\omega \rightharpoonup \Sigma^\omega$, every finite portion of the output $f_{\mathcal{M}}(p)$ is already determined by a finite portion of the input p.

The finiteness property is equivalent to continuity of $f_{\mathcal{M}}$ if we consider the Cantor topology on Σ^ω.

Computability Refines Continuity

Roughly speaking, the finiteness property says that, given a Type-2 Machine $\mathcal{M}$ such that $f_{\mathcal{M}} : \Sigma^\omega \rightharpoonup \Sigma^\omega$, every finite portion of the output $f_{\mathcal{M}}(p)$ is already determined by a finite portion of the input p.

The finiteness property is equivalent to continuity of $f_{\mathcal{M}}$ if we consider the Cantor topology on Σ^ω.

Theorem 42

Every computable string function $f : Y \rightharpoonup Y_0$ is continuous in discrete topology when $Y = \Sigma^*$ and in the Cantor topology when $Y = \Sigma^*$.

Computability Refines Continuity

Roughly speaking, the finiteness property says that, given a Type-2 Machine $\mathcal{M}$ such that $f_{\mathcal{M}} : \Sigma^\omega \rightharpoonup \Sigma^\omega$, every finite portion of the output $f_{\mathcal{M}}(p)$ is already determined by a finite portion of the input p.

The finiteness property is equivalent to continuity of $f_{\mathcal{M}}$ if we consider the Cantor topology on Σ^ω.

Theorem 42

Every computable string function $f : Y \rightharpoonup Y_0$ is continuous in discrete topology when $Y = \Sigma^*$ and in the Cantor topology when $Y = \Sigma^*$.

Proof: If $u \in \Sigma^*$ with $u \sqsubseteq f_{\mathcal{M}}(p)$, then on input p, the machine $\mathcal{M}$ writes the prefix u of the output $f_{\mathcal{M}}(p)$ in t steps for some $t \in \mathbb{N}$. Within t steps, $\mathcal{M}$ can read not more than the prefix $w := p_{<t}$ of the input $p \in \Sigma^\omega$. Therefore, the output of u depends only on the prefix $w \sqsubseteq p$, that is, $f_{\mathcal{M}}(w\Sigma^\omega) \subseteq u\Sigma^\omega$ (continuity at the point p). □

Computability Refines Continuity

Roughly speaking, the finiteness property says that, given a Type-2 Machine $\mathcal{M}$ such that $f_{\mathcal{M}} : \Sigma^\omega \rightharpoonup \Sigma^\omega$, every finite portion of the output $f_{\mathcal{M}}(p)$ is already determined by a finite portion of the input p.

The finiteness property is equivalent to continuity of $f_{\mathcal{M}}$ if we consider the Cantor topology on Σ^ω.

Theorem 42

Every computable string function $f : Y \rightharpoonup Y_0$ is continuous in discrete topology when $Y = \Sigma^*$ and in the Cantor topology when $Y = \Sigma^*$.

Proof: If $u \in \Sigma^*$ with $u \sqsubseteq f_{\mathcal{M}}(p)$, then on input p, the machine $\mathcal{M}$ writes the prefix u of the output $f_{\mathcal{M}}(p)$ in t steps for some $t \in \mathbb{N}$. Within t steps, $\mathcal{M}$ can read not more than the prefix $w := p_{<t}$ of the input $p \in \Sigma^\omega$. Therefore, the output of u depends only on the prefix $w \sqsubseteq p$, that is, $f_{\mathcal{M}}(w\Sigma^\omega) \subseteq u\Sigma^\omega$ (continuity at the point p). □

This generalises to arbitrarily typed machines using the product topology.

Decidable Sets

In TTE there is a distinction between recursive and decidable sets. We don't have time for their definition of recursive sets, but we can look at decidable sets, which are easy to define from what we have, and have a slick characterisation.

Decidable Sets

In TTE there is a distinction between recursive and decidable sets. We don't have time for their definition of recursive sets, but we can look at decidable sets, which are easy to define from what we have, and have a slick characterisation.

Definition 43

We call $A \subseteq \Sigma^\omega$ decidable iff its characteristic function is computable.

Decidable Sets

In TTE there is a distinction between recursive and decidable sets. We don't have time for their definition of recursive sets, but we can look at decidable sets, which are easy to define from what we have, and have a slick characterisation.

Definition 43

We call $A \subseteq \Sigma^\omega$ decidable iff its characteristic function is computable.

Theorem 44

Let $X \subseteq \Sigma^\omega$. Then, the following are equivalent statements:

1. *X is decidable;*
2. *X is clopen (in the Cantor topology);*
3. *$X = A\Sigma^\omega$ for some finite set $A \subseteq \Sigma^*$.*

Decidable Sets

In TTE there is a distinction between recursive and decidable sets. We don't have time for their definition of recursive sets, but we can look at decidable sets, which are easy to define from what we have, and have a slick characterisation.

Definition 43

We call $A \subseteq \Sigma^\omega$ decidable iff its characteristic function is computable.

Theorem 44

Let $X \subseteq \Sigma^\omega$. Then, the following are equivalent statements:

1. *X is decidable;*
2. *X is clopen (in the Cantor topology);*
3. *$X = A\Sigma^\omega$ for some finite set $A \subseteq \Sigma^*$.*

Proof: Compactness...

Decimal Fractions

You learned at school that every real number could be represented by an infinite decimal fraction. That is, an infinite string in

$$N := (\{1,\ldots,9\}\{0,\ldots,9\}^* \cup \{0\})\{.\}\{0,\ldots,9\}^\omega.$$

Decimal Fractions

You learned at school that every real number could be represented by an infinite decimal fraction. That is, an infinite string in

$$N := (\{1,\ldots,9\}\{0,\ldots,9\}^* \cup \{0\})\{.\}\{0,\ldots,9\}^\omega.$$

Definition 45 ($\mathbb{R}$ Naming System I)

The obvious surjection $N \to \mathbb{R}$ is a naming system for $\mathbb{R}$.

Decimal Fractions

You learned at school that every real number could be represented by an infinite decimal fraction. That is, an infinite string in

$$N := (\{1,\ldots,9\}\{0,\ldots,9\}^* \cup \{0\})\{.\}\{0,\ldots,9\}^\omega.$$

Definition 45 ($\mathbb{R}$ Naming System I)

The obvious surjection $N \to \mathbb{R}$ is a naming system for $\mathbb{R}$.

Proposition 46

No Type-2 machine computes the real function $f(x) = 3 \cdot x$ using the above naming system.

Decimal Fractions

You learned at school that every real number could be represented by an infinite decimal fraction. That is, an infinite string in

$$N := (\{1,\ldots,9\}\{0,\ldots,9\}^* \cup \{0\})\{.\}\{0,\ldots,9\}^\omega.$$

Definition 45 ($\mathbb{R}$ Naming System I)

The obvious surjection $N \to \mathbb{R}$ is a naming system for $\mathbb{R}$.

Proposition 46

No Type-2 machine computes the real function $f(x) = 3 \cdot x$ using the above naming system.

A computability concept under which multiplication by a constant is not computable, is not every useful!

A Better Naming System

Definition 47 ($\mathbb{R}$ Naming System II)

Define a name of a real number $x \in \mathbb{R}$ to be a sequence $(I_0, I_1, \dots)$ of closed rational intervals $[a, b]$ $(a < b, a, b \in \mathbb{Q})$ (formally, we can encode rational intervals using Cantor pairing followed by the standard numbering) such that $I_{n+1} \leq I_n$ for all $n \in \mathbb{N}$ and $\bigcap_{n \in \mathbb{N}} = \{x\}$.

A Better Naming System

Definition 47 ($\mathbb{R}$ Naming System II)

Define a name of a real number $x \in \mathbb{R}$ to be a sequence $(I_0, I_1, \dots)$ of closed rational intervals $[a, b]$ $(a < b, a, b \in \mathbb{Q})$ (formally, we can encode rational intervals using Cantor pairing followed by the standard numbering) such that $I_{n+1} \leq I_n$ for all $n \in \mathbb{N}$ and $\bigcap_{n \in \mathbb{N}} = \{x\}$.

Proposition 48

Real multiplication $(x, y) \mapsto x \cdot y$ is computable using the above naming system.

A Better Naming System

Definition 47 ($\mathbb{R}$ Naming System II)

Define a name of a real number $x \in \mathbb{R}$ to be a sequence $(I_0, I_1, \dots)$ of closed rational intervals $[a, b]$ $(a < b, a, b \in \mathbb{Q})$ (formally, we can encode rational intervals using Cantor pairing followed by the standard numbering) such that $I_{n+1} \leq I_n$ for all $n \in \mathbb{N}$ and $\bigcap_{n \in \mathbb{N}} = \{x\}$.

Proposition 48

Real multiplication $(x, y) \mapsto x \cdot y$ is computable using the above naming system.

Proposition 49

Every computable real function using the above naming system is continuous in the standard topology on $\mathbb{R}$.