Computability and the Church-Turing Thesis

Hayden Jananthan

Vanderbilt University

March 21, 2017

Computing

Computers.

Computing

Computers. We love ‘em.

Computing

They do stuff for us.

Computing

They do *a lot of stuff* for us.

Computing

Seemingly, they can do *anything* we give them.

But...

What is a computer?

But...

What is a computer? Maybe... A computer is a mechanical tool for running algorithms.

But...

What is a computer? Maybe... A computer is a mechanical tool for running algorithms.
What is an algorithm?

But...

What is a computer? Maybe... A computer is a mechanical tool for running algorithms.
What is an algorithm? Not so obvious...

Variations on Defining 'Algorithm'

Many different and sometimes strange models of computation have been proposed or created:

Variations on Defining 'Algorithm'

Many different and sometimes strange models of computation have been proposed or created:

(1) Partial Recursive Functions, defined by Kurt Gödel in 1933.

Variations on Defining 'Algorithm'

Many different and sometimes strange models of computation have been proposed or created:

(1) Partial Recursive Functions, defined by Kurt Gödel in 1933.

(2) λ-Calculus, defined by Alonzo Church in 1936.

Variations on Defining 'Algorithm'

Many different and sometimes strange models of computation have been proposed or created:

(1) Partial Recursive Functions, defined by Kurt Gödel in 1933.

(2) λ-Calculus, defined by Alonzo Church in 1936.

(3) Turing Machines, defined by Alan Turing in 1936.

Variations on Defining 'Algorithm'

Many different and sometimes strange models of computation have been proposed or created:

(1) Partial Recursive Functions, defined by Kurt Gödel in 1933.

(2) λ-Calculus, defined by Alonzo Church in 1936.

(3) Turing Machines, defined by Alan Turing in 1936.

(4) Register Machine Programs.

Variations on Defining 'Algorithm'

Many different and sometimes strange models of computation have been proposed or created:

(1) Partial Recursive Functions, defined by Kurt Gödel in 1933.

(2) λ-Calculus, defined by Alonzo Church in 1936.

(3) Turing Machines, defined by Alan Turing in 1936.

(4) Register Machine Programs.

(5) Minecraft.

Variations on Defining 'Algorithm'

Many different and sometimes strange models of computation have been proposed or created:

(1) Partial Recursive Functions, defined by Kurt Gödel in 1933.

(2) λ-Calculus, defined by Alonzo Church in 1936.

(3) Turing Machines, defined by Alan Turing in 1936.

(4) Register Machine Programs.

(5) Minecraft.

(6) Conways Game of Life.

Variations on Defining 'Algorithm'

Many different and sometimes strange models of computation have been proposed or created:

(1) Partial Recursive Functions, defined by Kurt Gödel in 1933.
(2) λ-Calculus, defined by Alonzo Church in 1936.
(3) Turing Machines, defined by Alan Turing in 1936.
(4) Register Machine Programs.
(5) Minecraft.
(6) Conways Game of Life.
(7) Minesweeper.

Variations on Defining 'Algorithm'

Many different and sometimes strange models of computation have been proposed or created:

(1) Partial Recursive Functions, defined by Kurt Gödel in 1933.
(2) λ-Calculus, defined by Alonzo Church in 1936.
(3) Turing Machines, defined by Alan Turing in 1936.
(4) Register Machine Programs.
(5) Minecraft.
(6) Conways Game of Life.
(7) Minesweeper.

All can be used to describe what it means for a function $f : \mathbb{N}^k \to \mathbb{N}$ to be *computable*.

Church-Turing Thesis

In 1936, after Church and Turing came up with their respective models of computation, they proved that the two were equivalent, in that the functions computed by them coincide.

Church-Turing Thesis

In 1936, after Church and Turing came up with their respective models of computation, they proved that the two were equivalent, in that the functions computed by them coincide.

The same was done with Gödel 's partial recursive functions as well, and a general trend was noticed:

Church-Turing Thesis

In 1936, after Church and Turing came up with their respective models of computation, they proved that the two were equivalent, in that the functions computed by them coincide.
The same was done with Gödel's partial recursive functions as well, and a general trend was noticed:

Every effectively calculable function (effectively decidable predicate) is [partial recursive].

Our Aim...

(a) Define what it means to be "partial recursive".

Our Aim...

(a) Define what it means to be “partial recursive”.

(b) Define what it means to be “register machine computable”.

Our Aim...

(a) Define what it means to be "partial recursive".
(b) Define what it means to be "register machine computable".
(c) Prove that they are equivalent.

Partial Recursive Functions

We shall attempt to define the "minimal" non-trivial class of functions which are computable.

Partial Recursive Functions

We shall attempt to define the "minimal" non-trivial class of functions which are computable.
To achieve this, we want to have some starting "simple" functions, and then some operations that represent the ideas of computability.

What are some of the simplest functions we can compute?

What are some of the simplest functions we can compute?

The zero function $Z(x) = 0$.

What are some of the simplest functions we can compute?

The zero function $Z(x) = 0$.

The successor function $S(x) = x + 1$.

What are some of the simplest functions we can compute?

The zero function $Z(x) = 0$.

The successor function $S(x) = x + 1$.

The projection functions $\pi_i^k(x_1, \ldots, x_k) = x_i$.

Our operations will capture the idea of doing computations in sequence:

Our operations will capture the idea of doing computations in sequence:

(1) Generalized Composition: can pre-compute some values (separately) to use in our computation.

Our operations will capture the idea of doing computations in sequence:

(1) Generalized Composition: can pre-compute some values (separately) to use in our computation.

(2) Primitive Recursion: can iterate a function using previously computed values repeatedly.

Our operations will capture the idea of doing computations in sequence:

(1) Generalized Composition: can pre-compute some values (separately) to use in our computation.
(2) Primitive Recursion: can iterate a function using previously computed values repeatedly.
(3) Minimization: Unbounded search, i.e. continue testing one by one until a condition is met.

Definition

If $g_1, \dots, g_n$ are k-ary functions and h is an n-ary function, then the **generalized composition** $f = h \circ (g_1, \dots, g_n)$ is defined by

$$f(x_1, \dots, x_k) = h(g_1(x_1, \dots, x_k), \dots, g_n(x_1, \dots, x_k))$$

Definition

If $g_1, \ldots, g_n$ are k-ary functions and h is an n-ary function, then the **generalized composition** $f = h \circ (g_1, \ldots, g_n)$ is defined by

$$f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$$

Example

The function $f(x, y) = x + 1$ is given by $S(\pi_1^2(x, y))$.

Definition

If h is a $k+2$-ary function and g a k-ary function, **primitive recursion** applied to g, h returns the $k+1$-ary function f defined by

$$f(\mathbf{0}, x_1, \ldots, x_k) = g(x_1, \ldots, x_k) \qquad \text{(Base Case)}$$

$$f(\mathbf{y+1}, x_1, \ldots, x_k) = h(y, f(\mathbf{y}, x_1, \ldots, x_k), x_1, \ldots, x_k) \qquad \text{(Iterative Step)}$$

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

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

$$f(0, y) = y \qquad \text{(Base Case)}$$

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

$$f(0,y) = y \qquad \text{(Base Case)}$$
$$f(x+1,y) = f(x,y) + 1 \qquad \text{(Iterative Step)}$$

Definition

If g is a $k+1$-ary function such that for all $x_1, \ldots, x_k$ there is y such that $g(y, x_1, \ldots, x_k) = 0$, then the **minimization** of g is the k-ary function defined by

$$f(x_1, \ldots, x_k) = \text{least } y \text{ such that } g(y, x_1, \ldots, x_k) = 0$$

Partial Functions

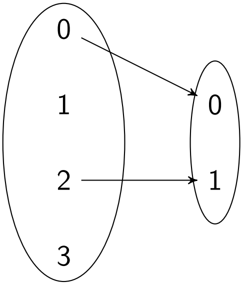
Definition

A **partial function** f is defined only on a subset of $\mathbb{N}^k$.
A function defined everywhere is a **total function**.

Partial Functions

Definition

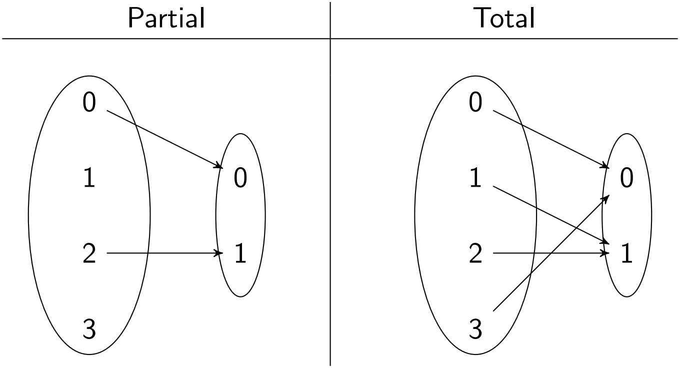
A **partial function** f is defined only on a subset of $\mathbb{N}^k$.
A function defined everywhere is a **total function**.

Partial	
	

Partial Functions

Definition

A **partial function** f is defined only on a subset of $\mathbb{N}^k$.
A function defined everywhere is a **total function**.



Definition

The class of **partial recursive functions** is the smallest collection of partial functions for which

Definition

The class of **partial recursive functions** is the smallest collection of partial functions for which

(i) contains the *initial functions* Z, S, π_i^k

Definition

The class of **partial recursive functions** is the smallest collection of partial functions for which

(i) contains the *initial functions* Z, S, π_i^k

(ii) closed under generalized composition

Definition

The class of **partial recursive functions** is the smallest collection of partial functions for which

(i) contains the *initial functions* Z, S, π_i^k

(ii) closed under generalized composition

(iii) closed under primitive recursion (of its total functions)

Definition

The class of **partial recursive functions** is the smallest collection of partial functions for which

(i) contains the *initial functions* Z, S, π_i^k

(ii) closed under generalized composition

(iii) closed under primitive recursion (of its total functions)

(iv) closed under minimization (of its total functions)

Some Partial Recursive Functions

Addition: $(x, y) \mapsto x + y$

Some Partial Recursive Functions

Addition: $(x, y) \mapsto x + y$

Multiplication: $(x, y) \mapsto x \cdot y$

Some Partial Recursive Functions

Addition: $(x, y) \mapsto x + y$

Multiplication: $(x, y) \mapsto x \cdot y$

Exponentiation: $(x, y) \mapsto x^y$

Some Partial Recursive Functions

Addition: $(x, y) \mapsto x + y$

Multiplication: $(x, y) \mapsto x \cdot y$

Exponentiation: $(x, y) \mapsto x^y$

Remainder: $\mathrm{Remainder}(x, y)$ = remainder when dividing y by x

Some Partial Recursive Functions

Addition: $(x, y) \mapsto x + y$

Multiplication: $(x, y) \mapsto x \cdot y$

Exponentiation: $(x, y) \mapsto x^y$

Remainder: $\mathrm{Remainder}(x, y) =$ remainder when dividing y by x

Prime Enumeration: $n \mapsto p_n = n$-th prime number

Addition: $(x, y) \mapsto x + y$

Multiplication: $(x, y) \mapsto x \cdot y$

Exponentiation: $(x, y) \mapsto x^y$

Remainder: $\mathrm{Remainder}(x, y)$ = remainder when dividing y by x

Prime Enumeration: $n \mapsto p_n$ = n-th prime number

Prime-Power Encoding:

$(z)_n$ = least w such that $\mathrm{Remainder}(z, p_n^{w+1}) \neq 0$

Some Partial Recursive Functions

Addition: $(x, y) \mapsto x + y$

Multiplication: $(x, y) \mapsto x \cdot y$

Exponentiation: $(x, y) \mapsto x^y$

Remainder: $\mathrm{Remainder}(x, y)$ = remainder when dividing y by x

Prime Enumeration: $n \mapsto p_n$ = n-th prime number

Prime-Power Encoding:

$(z)_n$ = least w such that $\mathrm{Remainder}(z, p_n^{w+1}) \neq 0$

Kronecker Delta: $\alpha(x) = 0^x = \begin{cases} 0 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \end{cases}$

Register Machine Programs

We have infinitely-many **registers** $R_1, R_2, \ldots$ which each contain a natural number; at any give time, all but finitely-many are **empty**, i.e. contain 0.

Register Machine Programs

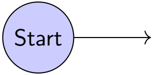
We have infinitely-many **registers** $R_1, R_2, \ldots$ which each contain a natural number; at any give time, all but finitely-many are **empty**, i.e. contain 0.

Four basic instructions:

Register Machine Programs

We have infinitely-many **registers** $R_1, R_2, \ldots$ which each contain a natural number; at any give time, all but finitely-many are **empty**, i.e. contain 0.

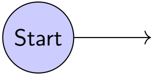
Four basic instructions:

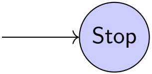
Start: 

Register Machine Programs

We have infinitely-many **registers** $R_1, R_2, \ldots$ which each contain a natural number; at any give time, all but finitely-many are **empty**, i.e. contain 0.

Four basic instructions:

Start: 

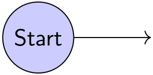
Stop: 

Register Machine Programs

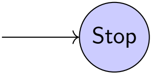
We have infinitely-many **registers** $R_1, R_2, \ldots$ which each contain a natural number; at any give time, all but finitely-many are **empty**, i.e. contain 0.

Four basic instructions:

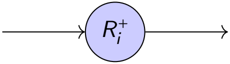
Start:



Stop:



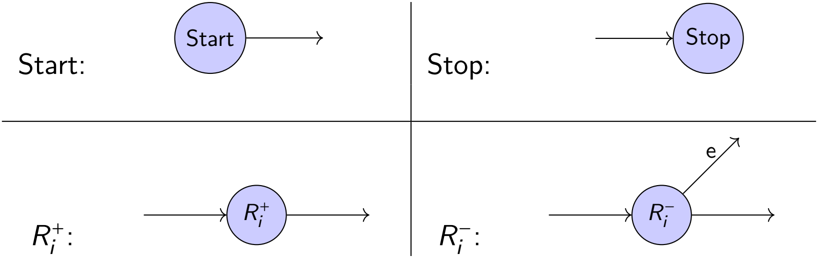
R_i^+:



Register Machine Programs

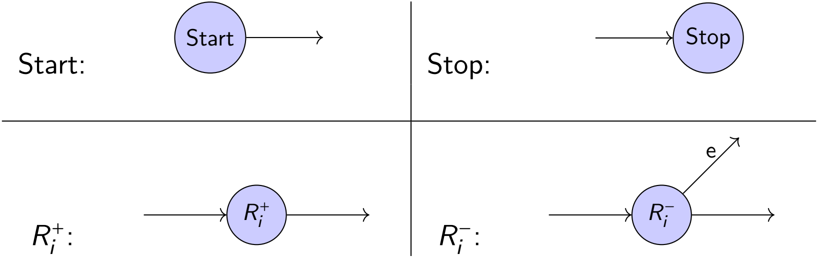
We have infinitely-many **registers** $R_1, R_2, \ldots$ which each contain a natural number; at any give time, all but finitely-many are **empty**, i.e. contain 0.

Four basic instructions:



Register Machine Programs

We have infinitely-many **registers** $R_1, R_2, \ldots$ which each contain a natural number; at any give time, all but finitely-many are **empty**, i.e. contain 0. Four basic instructions:



Definition

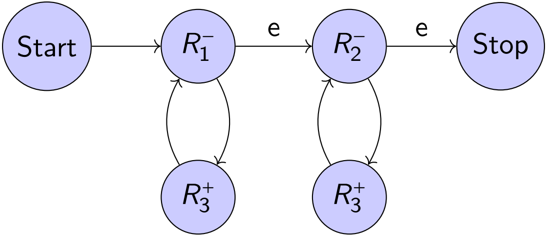
A **register machine program** is a finite diagram consisting of the aforementioned instructions, with exactly one start and at least one stop instruction.

As an example, consider the following register machine program:

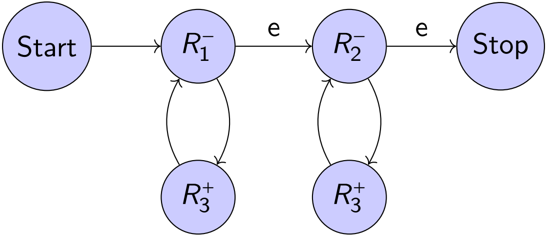
Register Machine Programs

A Simple Example

As an example, consider the following register machine program:



As an example, consider the following register machine program:



If we imagine R_1, R_2 as our input values x, y and R_3 as our output value, then this register machine program *computes* the addition function $(x, y) \mapsto x + y$.

Register Machine Computability

Register Machine Computability

Definition

$f : \mathbb{N}^k \xrightarrow{p} \mathbb{N}$ is **register machine computable** if there is a register machine program $\mathcal{P}$ such that

if $R_1, \ldots, R_k$ contain the values $x_1, \ldots, x_k$,

then $\mathcal{P}$ halts on this input with $f(x_1, \ldots, x_k)$ in the register R_{k+1}

(exactly when f is defined on the input $(x_1, \ldots, x_k)$).

Register Machine Computability

Definition

$f : \mathbb{N}^k \xrightarrow{p} \mathbb{N}$ is **register machine computable** if there is a register machine program $\mathcal{P}$ such that

if $R_1, \ldots, R_k$ contain the values $x_1, \ldots, x_k$,

then $\mathcal{P}$ halts on this input with $f(x_1, \ldots, x_k)$ in the register R_{k+1}

(exactly when f is defined on the input $(x_1, \ldots, x_k)$).

Church-Turing Thesis suggests this is the same as being partial recursive.

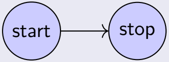
Initial Functions are Register Machine Computable

Lemma

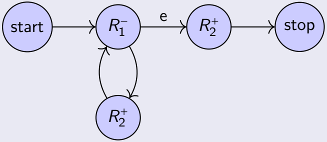
The functions Z, S, π_i^k (for $1 \leq i \leq k$) are register machine computable.

Proof.

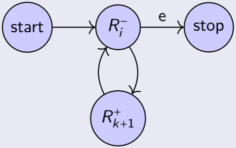
Zero:



Successor:



Projection:



□

Closure under Generalized Composition

Lemma

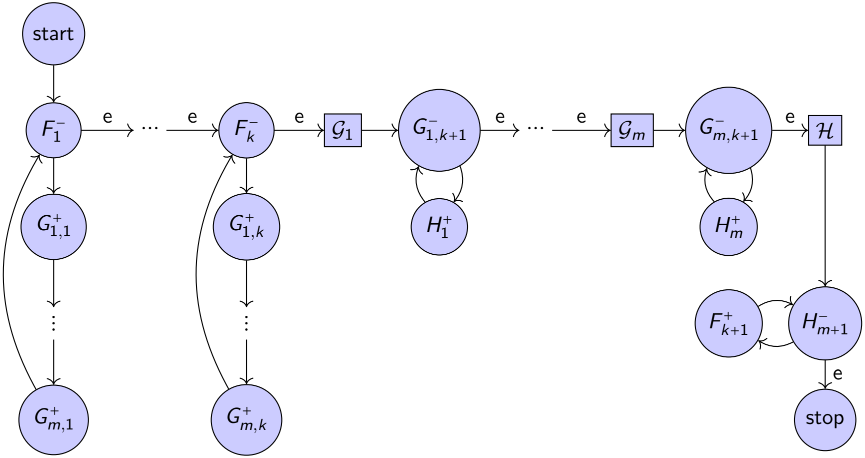
If h is an n-ary register machine computable function, and $g_1, \ldots, g_n$ are k-ary register machine computable functions, then the k-ary function

$$f(x_1, \ldots, x_k) \simeq h(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$$

is register machine computable.

Proof.
Let $\mathcal{H}$ be a register machine program computing h with registers $H_1, \ldots$ and $\mathcal{G}_i$ be register machine programs computing g_i with registers $G_{i,1}, \ldots$.
Then consider the register machine program...

Closure under Generalized Composition



Closure under Primitive Recursion

Lemma

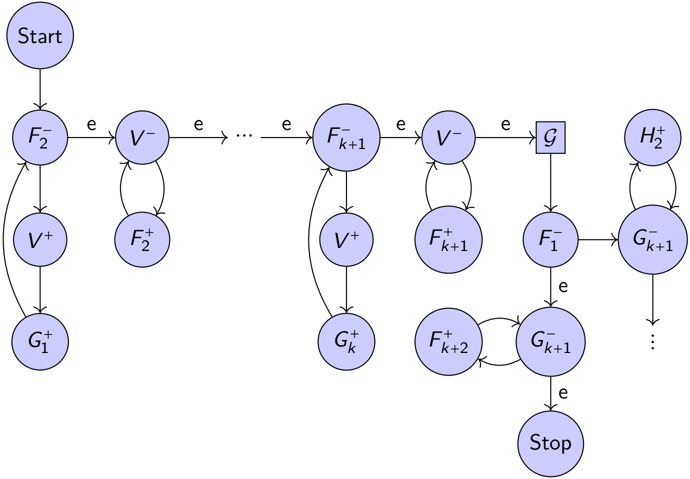
If g is a k-ary total register machine computable function and h is a $k+2$-ary total register machine computable function, then the $k+1$-ary function

$$f(0, x_1, \dots, x_k) = g(x_1, \dots, x_k)$$
$$f(n+1, x_1, \dots, x_k) = h(n, f(n, x_1, \dots, x_k), x_1, \dots, x_k)$$

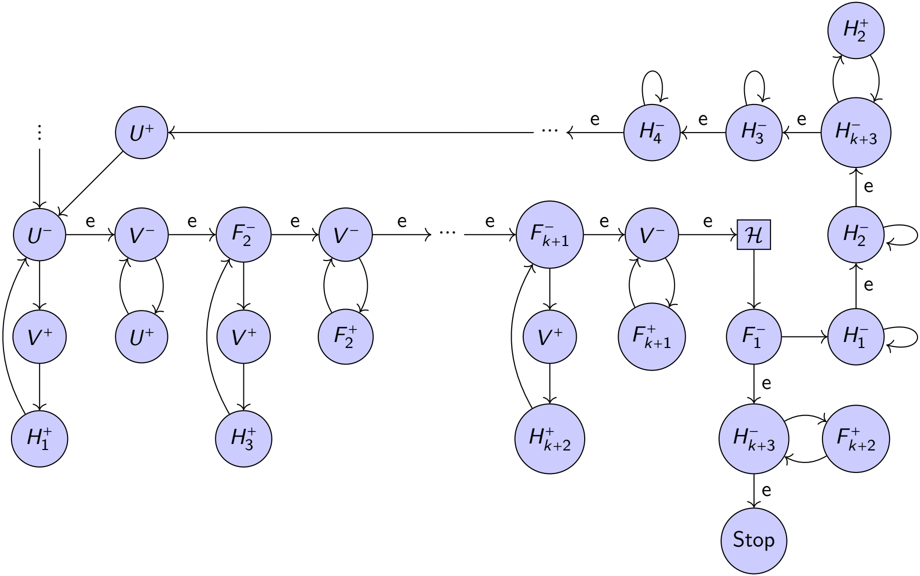
is register machine computable.

Proof.
Let $\mathcal{G}$ be a register machine program computing g with registers $G_1, \dots$ and $\mathcal{H}$ be a register machine program computing h with registers $H_1, \dots$, and let U, V be two other registers. Then consider the register machine program... $\square$

Closure under Primitive Recursion



Closure under Primitive Recursion



Closure under Minimization

Lemma

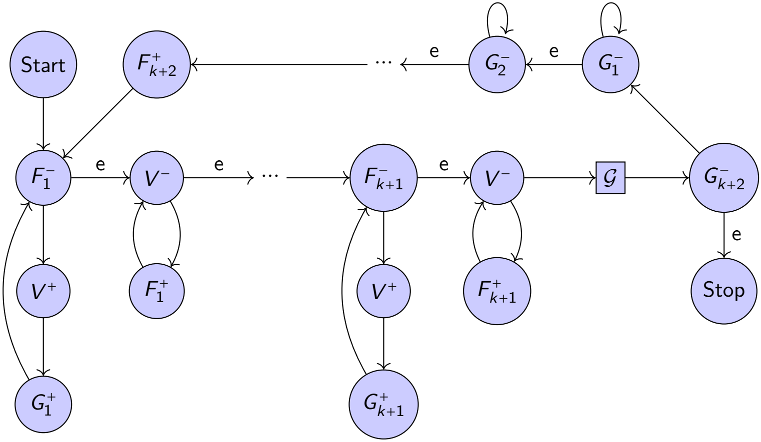
Suppose g is a $k+1$-ary total register machine computable function. Then the partial function

$$f(x_1, \ldots, x_k) = \textit{least } y \textit{ such that } g(y, x_1, \ldots, x_k) = 0$$

is partial recursive.

Proof.
Let $\mathcal{G}$ be a register machine program computing g with registers $G_1, \ldots$.
Then consider the register machine program... $\square$

Closure under Minimization



Gödel Numbering of Instructions

Want to encode the instructions of a register machine program.

Gödel Numbering of Instructions

Want to encode the instructions of a register machine program.

Gödel Numbering of Instructions

Want to encode the instructions of a register machine program.
We number each of the instructions in our register machine program $\mathcal{E}$ by $I_1, \ldots, I_\ell$, with I_1 the first instruction, and halting when we reach the (non-existent) instruction I_0.

Gödel Numbering of Instructions

Want to encode the instructions of a register machine program.

We number each of the instructions in our register machine program $\mathcal{E}$ by $I_1, \ldots, I_\ell$, with I_1 the first instruction, and halting when we reach the (non-existent) instruction I_0.

Instructions take two forms:

Gödel Numbering of Instructions

Want to encode the instructions of a register machine program.
We number each of the instructions in our register machine program $\mathcal{E}$ by $I_1, \ldots, I_\ell$, with I_1 the first instruction, and halting when we reach the (non-existent) instruction I_0.
Instructions take two forms:

(i) increment R_i and go to instruction I_{n_0}

Gödel Numbering of Instructions

Want to encode the instructions of a register machine program.
We number each of the instructions in our register machine program $\mathcal{E}$ by $I_1, \ldots, I_\ell$, with I_1 the first instruction, and halting when we reach the (non-existent) instruction I_0.
Instructions take two forms:

(i) increment R_i and go to instruction I_{n_0}

(ii) if R_i is empty (0) go to I_{n_0}, otherwise decrement R_i and go to I_{n_1}.

Gödel Numbering of Instructions

Want to encode the instructions of a register machine program.
We number each of the instructions in our register machine program $\mathcal{E}$ by $I_1, \ldots, I_\ell$, with I_1 the first instruction, and halting when we reach the (non-existent) instruction I_0.
Instructions take two forms:

(i) increment R_i and go to instruction I_{n_0}

(ii) if R_i is empty (0) go to I_{n_0}, otherwise decrement R_i and go to I_{n_1}.

Definition

$$\#(I_m) = \begin{cases} 3^i \cdot 5^{n_0} & \text{if } I_m \text{ is of the form given in (i)} \\ 2 \cdot 3^i \cdot 5^{n_0} \cdot 7^{n_1} & \text{if } I_m \text{ is of the form given in (ii)} \end{cases}$$

Gödel Numbering of Register Machine Programs

Prime-power encoding of instructions gives Gödel numbering of register machine programs:

Gödel Numbering of Register Machine Programs

Prime-power encoding of instructions gives Gödel numbering of register machine programs:

Definition

Then we define the **Gödel Numbering** of $\mathcal{E}$ by

$$\#(\mathcal{E}) = \prod_{m=1}^{\ell} p_m^{\#(I_m)}$$

where $p_0, p_1, p_2, \ldots$ are the prime numbers $2, 3, 5, \ldots$ in increasing order.

Gödel Numbering of Register Machine Programs

Prime-power encoding of instructions gives Gödel numbering of register machine programs:

Definition

Then we define the **Gödel Numbering** of $\mathcal{E}$ by

$$\#(\mathcal{E}) = \prod_{m=1}^{\ell} p_m^{\#(I_m)}$$

where $p_0, p_1, p_2, \ldots$ are the prime numbers $2, 3, 5, \ldots$ in increasing order.

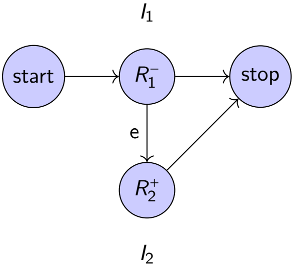
(Technically depends on how we order the instructions, so will have multiple Gödel numbers that correspond to the same register machine program.)

Gödel Numbering Example

$\alpha(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x > 0 \end{cases}$ is computed by the register machine program $\mathcal{E}$

Gödel Numbering Example

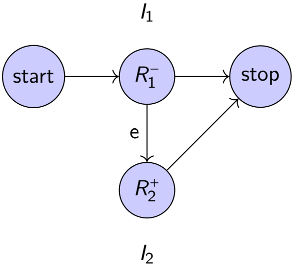
$$\alpha(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x > 0 \end{cases}$$ is computed by the register machine program $\mathcal{E}$



Gödel Numbering Example

$$\alpha(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{if } x > 0 \end{cases}$$ is computed by the register machine program $\mathcal{E}$

Then



$$\#(I_1) = 2 \cdot 3^1 \cdot 5^2 \cdot 7^0 = 150$$
$$\#(I_2) = 3^2 \cdot 5^0 = 9$$

so that

$$\#(\mathcal{E}) = 2^{150} \cdot 3^9$$

Definition

If e is the Gödel number of a register machine program $\mathcal{E}$ and $k \geq 1$, we define

$$\varphi_e^k(x_1, \ldots, x_k)$$

to be the value in register R_{k+1} when $R_1, \ldots, R_k$ are given $x_1, \ldots, x_k$ and $\mathcal{E}$ is run and halts (assuming it halts).

Definition

If e is the Gödel number of a register machine program $\mathcal{E}$ and $k \geq 1$, we define

$$\varphi_e^k(x_1, \ldots, x_k)$$

to be the value in register R_{k+1} when $R_1, \ldots, R_k$ are given $x_1, \ldots, x_k$ and $\mathcal{E}$ is run and halts (assuming it halts).

φ_e^k is a register machine computable function by definition.

Some Lemmas

Say that the **predicate** $P \subset \mathbb{N}^k$ is **recursive** if its characteristic function χ_P is recursive.

Lemma

The 1-ary predicate
$\mathrm{Program}(e) \equiv e$ *a Gödel number of a register machine program is recursive.*

Lemma

If $P_1, P_2 \subset \mathbb{N}^k$ are recursive predicates with $P_1 \cap P_2 = \varnothing$ and $P_1 \cup P_2 = \mathbb{N}^k$. Suppose f_1, f_2 are k-ary recursive functions. Then

$$f(x_1, \ldots, x_k) = \begin{cases} f_1(x_1, \ldots, x_k) & \text{if } P_1(x_1, \ldots, x_k) \text{ holds} \\ f_2(x_1, \ldots, x_k) & \text{if } P_2(x_1, \ldots, x_k) \text{ holds} \end{cases}$$

is recursive.

Enumeration Theorem (Statement)

Theorem (Enumeration Theorem)

The $k+1$-ary function

$$\Phi(e, x_1, \ldots, x_k) \simeq \begin{cases} \varphi_e^k(x_1, \ldots, x_k) & \textit{if } \mathrm{Program}(e) \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

is partial recursive.

Enumeration Theorem (Proof)

e a Gödel number of a register machine program $\mathcal{E}$.

Enumeration Theorem (Proof)

e a Gödel number of a register machine program $\mathcal{E}$.
Want to show that the $k+2$-ary function

$$z = \mathrm{State}(e, x_1, \ldots, x_k, n) = p_0^m \cdot \prod_{i=1}^{\infty} p_i^{z_i}$$

is (total) recursive, where z_i is the number in register R_i and I_m is the next instruction to be executed.

Enumeration Theorem (Proof)

e a Gödel number of a register machine program $\mathcal{E}$.
Want to show that the $k+2$-ary function

$$z = \mathrm{State}(e, x_1, \ldots, x_k, n) = p_0^m \cdot \prod_{i=1}^{\infty} p_i^{z_i}$$

is (total) recursive, where z_i is the number in register R_i and I_m is the next instruction to be executed.
Then $(z)_0 = m$ and $(z)_i = z_i$ for $i \geq 1$.

Enumeration Theorem (Proof)

$$z = \mathrm{State}(e, x_1, \ldots, x_k, n) = p_0^m \cdot \prod_{i=1}^{\infty} p_i^{z_i}$$

Enumeration Theorem (Proof)

$$z = \mathrm{State}(e, x_1, \ldots, x_k, n) = p_0^m \cdot \prod_{i=1}^{\infty} p_i^{z_i}$$

We define it using primitive recursion:

Enumeration Theorem (Proof)

$$z = \mathrm{State}(e, x_1, \dots, x_k, n) = p_0^m \cdot \prod_{i=1}^{\infty} p_i^{z_i}$$

We define it using primitive recursion:

$$\mathrm{State}(e, x_1, \dots, x_k, 0) = p_0^1 \cdot (p_1^{x_1} \cdots p_k^{x_k})$$

Enumeration Theorem (Proof)

$$z = \mathrm{State}(e, x_1, \ldots, x_k, n) = p_0^m \cdot \prod_{i=1}^{\infty} p_i^{z_i}$$

We define it using primitive recursion:

$$\mathrm{State}(e, x_1, \ldots, x_k, 0) = p_0^1 \cdot (p_1^{x_1} \cdots p_k^{x_k})$$
$$\mathrm{State}(e, x_1, \ldots, x_k, n+1) = \mathrm{NextState}(e, \mathrm{State}(e, x_1, \ldots, x_k, n))$$

Enumeration Theorem (Proof)

$$z = \mathrm{State}(e, x_1, \ldots, x_k, n) = p_0^m \cdot \prod_{i=1}^{\infty} p_i^{z_i}$$

We define it using primitive recursion:

$$\mathrm{State}(e, x_1, \ldots, x_k, 0) = p_0^1 \cdot (p_1^{x_1} \cdots p_k^{x_k})$$
$$\mathrm{State}(e, x_1, \ldots, x_k, n+1) = \mathrm{NextState}(e, \mathrm{State}(e, x_1, \ldots, x_k, n))$$

with $m = (z)_0$, $i = ((e)_m)_1$, $n_0 = ((e)_m)_2$, and $n_1 = ((e)_m)_3$,

Enumeration Theorem (Proof)

$$z = \mathrm{State}(e, x_1, \ldots, x_k, n) = p_0^m \cdot \prod_{i=1}^{\infty} p_i^{z_i}$$

We define it using primitive recursion:

$$\begin{aligned} \mathrm{State}(e, x_1, \ldots, x_k, 0) &= p_0^1 \cdot (p_1^{x_1} \cdots p_k^{x_k}) \\ \mathrm{State}(e, x_1, \ldots, x_k, n+1) &= \mathrm{NextState}(e, \mathrm{State}(e, x_1, \ldots, x_k, n)) \end{aligned}$$

with $m = (z)_0$, $i = ((e)_m)_1$, $n_0 = ((e)_m)_2$, and $n_1 = ((e)_m)_3$,
where

$$\mathrm{NextState}(e, z) = \begin{cases} z \cdot p_i \cdot p_0^{-m+n_0} & \text{if } ((e)_m)_0 = 0 \\ z \cdot p_0^{-m+n_0} & \text{if } ((e)_m)_0 = 1 \text{ and } (z)_i = 0 \\ z \cdot p_i^{-1} \cdot p_0^{-m+n_1} & \text{if } ((e)_m)_0 = 1 \text{ and } (z)_i > 0 \\ z & \text{otherwise} \end{cases}$$

(which is recursive.)

Enumeration Theorem (Proof)

Then define

$$\mathrm{Stop}(e, x_1, \ldots, x_k) \simeq \underset{(\mathrm{State}(e, x_1, \ldots, x_k, n))_0 + \alpha(\chi_{\mathrm{Program}}(e)) = 0}{\text{least } n \text{ such that}}$$

which is partial recursive.

Enumeration Theorem (Proof)

Then define

$$\mathrm{Stop}(e, x_1, \ldots, x_k) \simeq \underset{(\mathrm{State}(e, x_1, \ldots, x_k, n))_0 + \alpha(\chi_{\mathrm{Program}}(e)) = 0}{\text{least } n \text{ such that}}$$

which is partial recursive.
Then

$$\varphi_e^k(x_1, \ldots, x_k) \simeq (\mathrm{State}(e, x_1, \ldots, x_k, \mathrm{Stop}(e, x_1, \ldots, x_k)))_{k+1}$$

is partial recursive. $\square$

A Corollary to Enumeration Theorem

The proof of the Enumeration Theorem shows the following:

Theorem

A partial function $f : \mathbb{N}^k \xrightarrow{p} \mathbb{N}$ *is partial recursive if and only if it is register machine computable.*