Computer Science 513

Emulation of Computation — and a 'Universal" $\mathcal{S}$-Program

Instructor: Wayne Eberly

Department of Computer Science
University of Calgary

Lecture #9

Goals for Today

- Establish that a variety of functions and predicates concerning ***computation by $\mathcal{S}$-programs*** are primitive recursive
- Develop a "universal $\mathcal{S}$-program" — which receives encodings of another $\mathcal{S}$-program $\mathcal{P}$ and of inputs for $\mathcal{P}$, and simulates the execution of $\mathcal{P}$ on these inputs
- Use this to prove that every Turing-computable partial or total function is μ-recursive.

Expectations

- I *will not* be asking students to repeat, recall, or explain the proofs that have been presented today. They are too long and complicated!
- However, I ***will*** expect students to understand and, possibly, explain the ***results that have been established*** — and their implications.
- Some of the functions introduced today will also be needed for later proofs, so it will be helpful if students understand their definitions.

Review: Encodings of Sequences of Integers

Encodings of a variety of integer tuples or sequences were introduced in Lecture #5. Various functions, related to these, were shown to be primitive recursive:

- A ***pairing function*** $\langle \cdot, \cdot \rangle : \mathbb{N}^2 \to \mathbb{N}$ and its inverses $\ell, r : \mathbb{N} \to \mathbb{N}$.
- For each positive integer k, and encoding

  $$\langle \cdot, \cdot, \ldots, \cdot \rangle : \mathbb{N}^k \to \mathbb{N}$$

  of k-tuples of integers, and its inverses $\rho_i^k : \mathbb{N} \to \mathbb{N}$ for $1 \leq i \leq k$.
- ***Gödel Numbering*** was used to define another encoding

  $$[\cdot, \cdot, \ldots, \cdot] : \mathbb{N}^k \to \mathbb{N}$$

  of k-tuples of integers along with associated functions that we will be used in these notes.

Encodings of Programs

Our next objective is encode every program $\mathcal{P}$ in the programming language $\mathcal{S}$ using some natural number, $\#(\mathcal{P})$, in such a way that the encoding can be used to compute useful information about the program.

Encodings of Variables

- To begin, consider the following ordering of all ***variables*** that *might* appear in $\mathcal{P}$:

$$Y, X_1, Z_1, X_2, Z_2, X_3, Z_3, \ldots$$

For each variable V let $\#(V)$ the position of V in this ordering:

- $\#(Y) = 1$.
- $\#(X_i) = 2i$ for every integer $i \geq 1$.
- $\#(Z_i) = 2i + 1$ for every integer $i \geq 1$.

Note that every ***positive*** integer i is the encoding $\#(V)$ for exactly one variable V.

Encodings of Labels

- Similarly, consider the following ordering of all ***labels*** that *might* appear in $\mathcal{P}$:

$$A_1, B_1, C_1, D_1, E_1, A_2, B_2, C_2, D_2, E_2, \ldots$$

For each label L let $\#(L)$ be the position of L in the above ordering: For each integer i such that $i \geq 1$,

- $\#(A_i) = 5i - 4$,
- $\#(B_i) = 5i - 3$,
- $\#(C_i) = 5i - 2$,
- $\#(D_i) = 5i - 1$, and
- $\#(E_i) = 5i$.

Note that every ***positive*** integer i is the encoding $\#(L)$ for exactly one label L.

Encodings of Statements in $\mathcal{S}$

- Recall that the only ***statements*** — or ***instructions*** — in the programming language $\mathcal{S}$ have one of the following forms:
  (a) $V \leftarrow V + 1$, where V is one of the above variables,
  (b) $V \leftarrow V - 1$, where V is one of the above variables,
  (c) $V \leftarrow V$, where V is one of the above variables, and
  (d) IF $V \neq 0$ GOTO L where V is one of the above variables and L is one of the above labels.

  Each statement *might* have one of the above labels.
  Labels are not necessarily unique.

Encodings of Statements in $\mathcal{S}$

- Each instruction I will be encoded by a natural number $\#(I)$. In particular,

$$\#(I) = \langle a, \langle b, c \rangle \rangle$$

where a, b and c are as described below.

  - If I does not have a label then $a = 0$. Otherwise, if I has label L then $a = \#(L)$.
  - b indicates the *instruction type* as follows:

$$b = \begin{cases} 0 & \text{if } I \text{ is } V \leftarrow V \text{ for some variable } V, \\ 1 & \text{if } I \text{ is } V \leftarrow V + 1 \text{ for some variable } V, \\ 2 & \text{if } I \text{ is } V \leftarrow V - 1 \text{ for some variable } V, \\ 2 + \#(L) & \text{If } I \text{ is ``IF } V \neq 0 \text{ GOTO } L\text{'' for some variable } V \text{ and label } L. \end{cases}$$

  - If I mentions the variable V then $c = \#(V) - 1$.

Encodings of Programs in $\mathcal{S}$

- ***Note:*** It can be checked that each natural number $n \in \mathbb{N}$ is the encoding $\#(I)$ for *exactly one* instruction in $\mathcal{S}$.
- ***A Somewhat Important Example:*** If I is the "dummy statement"

$$Y \leftarrow Y$$

then $\#(I) = \langle 0, \langle 0, 0\rangle\rangle = \langle 0, 0\rangle = 0$. Note that copies of this instruction at the ***end*** of a program do not change the (partial) function computed by the program at all.

- Finally, let $\mathcal{P}$ be a program in $\mathcal{S}$. In particular, suppose that $\mathcal{P}$ is the sequence of instructions

$$I_1,\ I_2,\ I_3,\ \ldots,\ I_k.$$

then $\#(\mathcal{P}) = [\#(I_1), \#(I_2), \#(I_3), \ldots, \#(I_k)] - 1$.

Encodings of Programs in $\mathcal{S}$

Properties:

- If $\mathcal{P}$ is the ***empty program*** — that is, the program without any statements at all — then $\#(\mathcal{P}) = 0$.
- Suppose $\mathcal{P}$ is an $\mathcal{S}$-program including the sequence of instructions

  $$I_1,\ I_2,\ \ldots,\ I_k$$

  and $\widehat{\mathcal{P}}$ is the $\mathcal{S}$-program including the sequence of instructions

  $$I_1,\ I_2, \ldots,\ I_k, Y \leftarrow Y$$

  — that is, the same program as $\mathcal{P}$ except that a copy of the "dummy statement" $Y \leftarrow Y$ has been added at the end. Then $\#(\widehat{\mathcal{P}}) = \#(\mathcal{P})$.

Encodings of Programs in $\mathcal{S}$

- On the other hand, if $\mathcal{P}$ is an $\mathcal{S}$-program including a sequence of instructions

$$I_1, \ I_2, \ \ldots, I_k$$

and $\mathcal{Q}$ is an $\mathcal{S}$-program including a sequence of instructions

$$J_1, \ J_2, \ \ldots, J_\ell$$

such that *neither* I_k *nor* J_ℓ is the (unlabelled) instruction $Y \leftarrow Y$, then $\#(\mathcal{P}) = \#(\mathcal{Q})$ if and only if $k = \ell$ and I_i and J_i are *exactly* the same instruction for all i such that $1 \leq i \leq k$.

Encodings of Programs in $\mathcal{S}$

It follows that if $\mathcal{P}$ is an $\mathcal{S}$-program, and $x = \#(\mathcal{P})$, then

- $Lt(x+1)$ is the number of instructions in the program $\widehat{\mathcal{P}}$ that would be obtained, from $\mathcal{P}$, by removing all *trailing* copies of the (unlabelled) dummy statement $Y \leftarrow Y$. This is — essentially — the "length" of the program $\mathcal{P}$.
- If $k \in \mathbb{N}$ then $(x+1)_k$ is the encoding of the k^{th} instruction in $\mathcal{P}$ if $1 \leq i \leq \text{Lt}(x+1)$, and $(x+1)_k$ is 0, otherwise.

Consequently these functions — of x in the first case, and of x and k in the second case — are ***primitive recursive*** functions too.

States and Their Encodings

As defined in the initial lecture on $\mathcal{S}$-programs, a ***state*** of a program $\mathcal{P}$ is a sequence of equations $V = m$ such that

- Ideally, this includes ***exactly one*** equation $V = m$ for each variable V appearing in $\mathcal{P}$ — and no equations for variables that *do not* appear in $\mathcal{P}$ except, possibly, a finite number of *input* variables.[1]
- If the state includes an equation $V = m$ then $m \in \mathbb{N}$ — and one should think of m as being "the current value" of the variable V.

[1] Note that if an input variable does not appear in the program then its value cannot effect the program's computation.

States and Their Encodings

In any state that can be reached after a finite number of steps of a computation, only a finite number of variables can have nonzero values.

- Consider a state σ where m is the largest encoding of a variable whose value is nonzero.
- Suppose that the variable with encoding i has value $v_i \in \mathbb{N}$ for $1 \leq i \leq m$.
- Then this state can be encoded using Gödel number

$$\#(\sigma) = z = [v_1, v_2, \ldots, v_m].$$

States and Their Encodings

- In particular,
  - Y always receives value $(z)_1$.
  - For $i \geq 1$, X_i receives the value $(z)_{2i}$.
  - For $i \geq 1$, Z_i receives the value $(z)_{2i+1}$.
- If every variable receives the value 0 then $\text{Lt}(z) = 0$. Otherwise $\text{Lt}(z) \geq 1$ and $\text{Lt}(z)$ is the largest integer m, such that the variable V such that $\#(V) = m$ receives a value that is different from zero.

Snapshots and Their Encodings

- Suppose now that $\mathcal{P}$ is a program that does not include any copies of $Y \leftarrow Y$ at the end — so that if $\#(\mathcal{P}) = x$ then $n = \mathrm{Lt}(x + 1)$ is the number of instructions in n.
- Recall that a ***snapshot*** or ***instantaneous description***, for $\mathcal{P}$, is an ordered pair $S = (i, \sigma)$ such that
  - $1 \leq i \leq n + 1$. One should think of i as being the number of the statement in $\mathcal{P}$ that is about to be executed *next* — and where $i = n + 1$ if and only if the computation has already ended.
  - σ is a ***state*** of $\mathcal{P}$, as defined on the previous slides.
- The snapshot $S = (i, \sigma)$ can now be encoded as

$$\#(S) = \langle i, \#(\sigma) \rangle.$$

A Useful Predicate

- Consider the predicate halted : $\mathbb{N}^2 \to \{0, 1\}$ such that, for $x, y \in \mathbb{N}$,

$$\text{halted}(x, y) = \begin{cases} 1 & \text{if } \ell(y) > \text{Lt}(x + 1), \\ 0 & \text{otherwise.} \end{cases}$$

  Note that "halted" is a primitive recursive predicate.

- Suppose, as well, that $x = \#(\mathcal{P})$ for some program $\mathcal{P}$ that does not end with any trailing copies of the dummy statement $Y \leftarrow Y$, and suppose that $y = \#(S)$ for some snapshot S.

A Useful Predicate

- Notice that $\ell(y)$ is the number i of the next instruction to be executed — for the snapshot S encoded by y — and $\mathrm{Lt}(x+1)$ is the number n of instructions in $\mathcal{P}$.
- Thus

$$\mathrm{halted}(x,y) = \begin{cases} 1 & \text{if } i \geq n+1, \\ 0 & \text{otherwise,} \end{cases}$$

that is, $\mathrm{halted}(x,y) = 1$ if and only if $\mathcal{P}$ has already halted when snapshot S has been reached.

A Useful Function

- Consider the function output : $\mathbb{N}^2 \to \mathbb{N}$ such that, for $x, y \in \mathbb{N}$,

$$\text{output}(x, y) = \begin{cases} (r(y))_1 & \text{if halted}(x, y) = 1, \\ 0 & \text{otherwise.} \end{cases}$$

  Note that "output" is a primitive recursive function.

- Suppose, once again, that $x = \#(\mathcal{P})$ and $y = \#(S)$ for a program $\mathcal{P}$ and snapshot S as above.

A Useful Function

- Recall that $r(y) = \#(\sigma)$ is the encoding of the *state* σ included in S — so that $(r(y))_1$ is the value of the output variable Y.
- Thus output(x, y) is the value computed by the program $\mathcal{P}$ (if snapshot S has been reached) if this program has already halted, and this value is 0, otherwise.

Computing the Initial Snapshot

- Let $k \geq 1$ and suppose that a program $\mathcal{P}$ is executed when the i^{th} input has value x_i for $1 \leq i \leq k$ (with all other input variables having value 0).
- Since $\#(X_i) = 2i$ for $i \geq 1$, this corresponds to an initial *state* σ whose encoding is

$$[0, x_1, 0, x_2, \ldots, 0, x_k] = p_2^{x_1} \times p_4^{x_2} \times \cdots \times p_{2k}^{x_k}.$$

- We have already seen that the exponential function $f(x_1, x_2) = x_1^{x_2}$ is primitive recursive.

Computing the Initial Snapshot

- For fixed k we can now define

$$\text{start}_k(x_1, x_2, \ldots, x_k) = \langle 1, p_2^{x_1} \times p_4^{x_2} \times \cdots \times p_{2k}^{x_k} \rangle.$$

Then, since the function mapping a positive integer i to the prime p_{2i} is primitive recursive, the above function "start_k" — which computes the encoding of the initial snapshot for the given inputs — is primitive recursive.

Computing the "Next" Snapshot

Consider the function instruction : $\mathbb{N}^2 \to \mathbb{N}$ such that, for all $x, y \in \mathbb{N}$,

$$\text{instruction}(x, y) = (x + 1)_{\ell(y)}.$$

Suppose, now, that $x = \#(\mathcal{P})$ for some program $\mathcal{P}$, and $y = \#(S)$ for some snapshot S.

- Recall that $\ell(y)$ is the number i of the next instruction in $\mathcal{P}$ that is to be executed — or is greater than the length of $\mathcal{P}$ if this computation has already halted.
- It follows that instruction(x, y) is the ***encoding*** $\#(I)$ of the instruction I in $\mathcal{P}$ to be executed next — or 0 if the computation has already halted.

Computing the "Next" Snapshot

Next consider the functions type, variable, label : $\mathbb{N} \to \mathbb{N}$ such that, for all $x, y \in \mathbb{N}$,

- $\text{type}(x, y) = \ell(r(\text{instruction}(x, y)))$,
- $\text{variable}(x, y) = r(r(\text{instruction}(x, y))) + 1$, and
- $\text{label}(x, y) = \ell(\text{instruction}(x, y))$.

Once again, it should be easy to see that all of these functions are primitive recursive.

Suppose, again, that $x = \#(\mathcal{P})$ for a program $\mathcal{P}$ and $y = \#(S)$ for a snapshot S — that is not "terminal" (the computation has not halted already).

Finally, let I be the instruction that is to be executed next — so that

$$\text{instruction}(x, y) = \#(I).$$

Computing the "Next" Snapshot

Referring back to encodings of instructions as needed, one can confirm the following.

- label$(x, y) = 0$ if the instruction I does not have a label. Otherwise label$(x, y) = \#(L)$ where L is the label for I.
- variable$(x, y) = \#(V)$ where V is the variable mentioned in the instruction I.
- Finally,

$$\text{type}(x, y) = \begin{cases} 0 & \text{if } I \text{ has the form } V \leftarrow V \\ 1 & \text{if } I \text{ has the form } V \leftarrow V + 1 \\ 2 & \text{if } I \text{ has the form } V \leftarrow V - 1 \\ 2 + \#(L) & \text{if } I \text{ if "IF } V \neq 0 \text{ GOTO } L\text{" for some variable } V \text{ and label } L \end{cases}$$

Computing the "Next" Snapshot

Next consider the predicate decrement : $\mathbb{N}^2 \to \{0, 1\}$ such that, for all $x, y \in \mathbb{N}$,

$$\text{decrement}(x, y) = \begin{cases} 1 & \text{if } (\text{type}(x, y) = 2) \wedge (r(y)_{\text{variable}(x,y)} > 0), \\ 0 & \text{otherwise.} \end{cases}$$

- Once again, this is a primitive recursive predicate.
- This predicate has value 1 if and only if the snapshot encoded by y is not terminal, the next instruction to be executed has the form $V \leftarrow V - 1$, and the current value of the variable V is positive.

Computing the "Next" Snapshot

Consider the function nextState : $\mathbb{N}^2 \to \mathbb{N}$ such that, for all $x, y \in \mathbb{N}$,

$$\text{nextState}(x, y) = \begin{cases} r(y) \times p_{\text{variable}(x,y)} & \text{if type}(x, y) = 1, \\ r(y) \text{ quo } p_{\text{variable}(x,y)} & \text{if decrement}(x, y) = 1, \\ r(y) & \text{otherwise.} \end{cases}$$

- This is also a primitive recursive function.
- Suppose, once again, that $x = \#(\mathcal{P})$ for some program $\mathcal{P}$ and $y = \#(S)$ for some snapshot S. Note that $\text{nextState}(x, y) = r(y)$ if S is a terminal snapshot.
- Otherwise, think of $r(y)$ is a *state* that might be included in a later snapshot.

Computing the "Next" Snapshot

Consider the instruction I to be executed next.

- If I is $V \leftarrow V + 1$ then $\text{type}(x, y) = 1$ — and $r(y)$ encodes the state obtained from the state σ, encoded by $r(y)$, by adding one to value of V.
- If I is $V \leftarrow V - 1$ and the value of V is positive then $\text{decrement}(x, y) = 1$, and $r(y)$ encodes the state obtained from the state σ, encoded by $r(y)$, by *subtracting* one from the value of V.
- In all other cases the execution of I should not change the *state* at all — so that $r(y)$ encodes the state obtained by executing the instruction I in this case too.

Computing the "Next" Snapshot

Consider the predicate jump : $\mathbb{N}^2 \to \{0, 1\}$ such that, for all $x, y \in \mathbb{N}$,

$$\text{jump}(x, y) = \begin{cases} 1 & \text{if } (\text{type}(x, y) > 2) \wedge ((r(y)_{\text{variable}(x,y)} > 0) \\ 0 & \text{otherwise.} \end{cases}$$

- Once again, this is a primitive recursive predicate.
- Let $x = \#(\mathcal{P})$, $y = \#(S)$. If S is a terminal snapshot then $\text{jump}(x, y) = 0$.

Computing the "Next" Snapshot

- Otherwise let I be the instruction to be executed next, so that $\#(I) = \text{instruction}(x, y)$.
- Then $\text{jump}(x, y) = 1$ if and only if I is a statement

$$\text{"IF } V \neq 0 \text{ GOTO } L\text{"}$$

and the current value of V is positive — so that instruction to be executed after this one should be the first with label L (and the program should halt if no such instruction exists).

Computing the "Next" Snapshot

Consider the predicate hasLabel : $\mathbb{N}^3 \to \{0, 1\}$ such that, for all $x, y, i \in \mathbb{N}$,

$$\text{hasLabel}(x, y, i) = \begin{cases} 1 & \text{if } \ell((x+1)_i) + 2 = \text{type}(x, y) \\ 0 & \text{otherwise.} \end{cases}$$

- Once again, this is a primitive recursive predicate.
- Consider the value of this function when $\text{type}(x, y) \geq 3$, that is, the snapshot encoded by y is not terminal and the next instruction to be executed is a conditional jump.
- If $i = 0$ or $i > \text{Lt}(x+1)$ then $\ell((x+1)_i) + 2 = \ell(0) + 2 = 2$, so $\text{hasLabel}(x, y, i) = 0$.

Computing the "Next" Snapshot

- On the other hand, if $1 \leq i \leq \mathrm{Lt}(x+1)$ then $(x+1)_i$ is the encoding of the i^{th} instruction in the program $\mathcal{P}$ such that $x = \#(\mathcal{P})$ — and $\ell((x+1)_i)$ encodes the *label* of this instruction, if it has one. Once again, this has value 0 — and $\mathrm{hasLabel}(x, y, i) = 0$ — otherwise.
- Checking the definition of the function "type" once again, one can see that — when $\mathrm{type}(x, y) \geq 3$ — $\mathrm{hasLabel}(x, y, i) = 1$ if and only if the next instruction I to be executed has the form

$$\text{if } V \neq 0 \text{ GOTO } L,$$

$1 \leq i \leq \mathrm{Lt}(x+1)$, and the i^{th} instruction in the program *does* have the same label L.

Computing the "Next" Snapshot

Consider the predicate $p : \mathbb{N}^3 \in \{0, 1\}$ obtained from hasLabel using a bounded existential quantifier: For all $x, y, i \in \mathbb{N}$,

$$p(x, y, i) = (\exists t)_{\leq i} \, \text{hasLabel}(x, y, t).$$

- This predicate is also primitive recursive.
- Once again, consider the value of this predicate when $\text{type}(x, y) \geq 3$.
- If $i = 0$ then $p(x, y, i) = 0$.
- If $1 \leq i < \text{Lt}(x + 1)$ then $p(x, y, i) = 1$ if and only if one of the first i instructions in the program has label L.
- If $i \geq \text{Lt}(x + 1)$ then $p(x, y, 1) = 1$ if and only if *there exists* a statement in the program with label L.

Computing the "Next" Snapshot

Consider the predicate foundLabel : $\mathbb{N}^2 \to \{0, 1\}$ such that, for all $x, y \in \mathbb{N}$,

$$\text{foundLabel}(x, y) = p(x, y, \text{Lt}(x + 1)).$$

- Once again, this is primitive recursive.
- It follows from the above that — if $\text{type}(x, y) \geq 3$ — $\text{foundLabel}(x, y) = 1$ if and only if there exists a statement in the program $\mathcal{P}$ whose label is L.
- We need the *number* of the first such statement, if one exists.

Computing the "Next" Snapshot

With that noted, consider the function $f : \mathbb{N}^3 \to \mathbb{N}$ obtained from the predicate hasLabel using ***bounded minimization:*** For all $x, y, i \in \mathbb{N}$

$$f(x, y, i) = \min_{t \leq i} \text{hasLabel}(x, y, t).$$

- This is a primitive recursive function.
- If $\text{type}(x, y) \geq 3$, and $1 \leq i \leq \text{Lt}(x + 1)$, then the value of this function is the number of the *first* instruction with label L, if this is one of the first i instructions in the program. Otherwise (when $\text{type}(x, y) \geq 3$) the function has value 0.
- If $\text{type}(x, y) \geq 3$, and $i \geq \text{Lt}(x + 1)$ then the value of this function is the number of the *first* instruction with label L, if such an instruction exists. Otherwise (when $\text{type}(x, y) \geq 3$) the value of this function is 0.

Computing the "Next" Snapshot

Let firstLabel : $\mathbb{N}^2 \to \mathbb{N}$ be the function such that, for all $x, y \in \mathbb{N}$,

$$\text{firstLabel}(x, y) = \begin{cases} f(x, y, \text{Lt}(x+1)) & \text{if foundLabel}(x, y), \\ \text{Lt}(x+1) + 1 & \text{otherwise.} \end{cases}$$

- This is yet another primitive recursive function.
- If type$(x, y) \geq 3$, so that the next instruction I to be executed has the form

$$\text{IF } V \neq 0 \text{ GOTO } L$$

then the value of this function is the number of the *first* instruction with label L — or $n + 1$ (where n is the number of instructions in $\mathcal{P}$) if no such instruction exists.

Computing the "Next" Snapshot

Consider the function nextInstruction : $\mathbb{N}^2 \to \mathbb{N}$ such that, for all $x, y, \in \mathbb{N}$,

$$\text{nextInstruction}(x, y) = \begin{cases} \text{firstLabel}(x, y) & \text{if jump}(x, y), \\ \ell(y) & \text{if halted}(x, y), \\ \ell(y) + 1 & \text{otherwise.} \end{cases}$$

- This is yet another primitive recursive function.
- Recall that the function jump confirms that type$(x, y) \geq 3$ – so that the snapshot encoded by y is not terminal — and, furthermore, that the next instruction I to be executed is an instruction

  $$\text{IF } V \neq 0 \text{ GOTO } L$$

  where the current value of V is positive.
- It follows that this really *is* equal to the number of the instruction that should be executed after I is.

Computing the "Next" Snapshot

Finally, consider the function next : $\mathbb{N}^2 \to \mathbb{N}$ such that, for all $x, y \in \mathbb{N}$,

$$\text{next}(x, y) = \langle \text{nextInstruction}(x, y), \text{nextState}(x, y) \rangle.$$

- This is another primitive recursive function.
- If $x = \#(\mathcal{P})$ for a program $\mathcal{P}$ and $y = \#(S)$ for a snapshot S, then
  - If the program has already halted (when S was reached) then next(x, y) also encodes the snapshot S.
  - Otherwise this encodes the *next* snapshot that would be obtained by executing one more instruction.

Computing the n^{th} Snapshot

- Consider the function $g_k : \mathbb{N}^{k+3} \to \mathbb{N}$ such that, for all $t, y, x, z \in \mathbb{N}$,

$$g_k(t, y, x_1, x_2, \ldots, x_k, z) = \text{next}(z, y).$$

  This function is also primitive recursive.

- Let $\text{snapshot}_k : \mathbb{N}^{k+2} \to \mathbb{N}$ be the function obtained from start_k and g_k by ***primitive recursion:*** For all $x, z \in \mathbb{N}$,

$$\text{snapshot}_k(x_1, x_2, \ldots, x_k, z, 0) = \text{start}_k(x_1, x_2, \ldots, x_k)$$

  and, for all $t \geq 0$,

$$\begin{aligned}
&\text{snapshot}_k(x_1, x_2, \ldots, x_k, z, t+1) \\
&\quad = g_k(t, \text{snapshot}_k(x_1, x_2, \ldots, x_k, z, t), x_1, x_2, \ldots, x_k, z) \\
&\quad = \text{next}(z, \text{snapshot}_k(x_1, x_2, \ldots, x_k, z, t)).
\end{aligned}$$

Computing the n^{th} Snapshot

- It is easy to show (by induction on n) that if z encodes a program $\mathcal{P}$, $x_1, x_2, \ldots, x_k \in \mathbb{N}$, and $n \geq 0$, then

$$\text{snapshot}_k(x_1, x_2, \ldots, x_k, z, n)$$

  encodes the snapshot obtained by taking the first n steps in the execution of program $\mathcal{P}$ on the inputs $x_1, x_2, \ldots, x_k$ — or the final snapshot, if the program halted after taking fewer than n steps.

Computing the n^{th} Snapshot: Application

Next consider the predicate Halt : $\mathbb{N}^{k+2} \to \{0, 1\}$ such that, for all $z, x_1, x_2, \ldots, x_k, n \in \mathbb{N}$,

$$\text{Halt}_k(x_1, x_2, \ldots, x_k, z, n) = \text{halted}(z, \text{snapshot}_k(x_1, x_2, \ldots, x_k, z, n)).$$

- This is a primitive recursive predicate.
- If $z = \#(\mathcal{P})$ for a program $\mathcal{P}$, and $x_1, x_2, \ldots, x_k, n \in \mathbb{N}$, then $\text{Halt}_k(x_1, x_2, \ldots, x_k, z, n) = 1$ if and only if $\mathcal{P}$ halts after executing at most n instructions, when executed on the inputs $x_1, x_2, \ldots, x_k$.

Computing the n^{th} Snapshot: Application

Finally, consider the function Output : $\mathbb{N}^{k+2} \to \mathbb{N}$ such that, for all $z, x_1, x_2, \ldots, x_k, n \in \mathbb{N}$,

$$\begin{aligned}\text{Output}_k(x_1, x_2, \ldots, x_k, z, n) \\ = \text{output}(z, \text{snapshot}_k(x_1, x_2, \ldots, x_k, z, n)).\end{aligned}$$

- This is a primitive recursive function.
- If $z = \#(\mathcal{P})$ for a program $\mathcal{P}$, and $x_1, x_2, \ldots, x_k, n \in \mathbb{N}$,
  - If $\mathcal{P}$ halts, when executed on the inputs $x_1, x_2, \ldots, x_k$, after at most n steps, then $\text{Output}_k(x_1, x_2, \ldots, x_k, z, n)$ is the value that $\mathcal{P}$ returns as output.
  - On the other hand, if this execution of $\mathcal{P}$ requires the execution of $n + 1$ or more steps, then $\text{Output}_k(x_1, x_2, \ldots, x_k, z, n) = 0$.

A Universal $\mathcal{S}$-Program

Once again, consider the following primitive recursive predicates and functions that have now been defined.

- $\text{start}_k : \mathbb{N}^k \to \mathbb{N}$. For all $x_1, x_2, \ldots, x_k \in \mathbb{N}$, $\text{start}_k(x_1, x_2, \ldots, x_k)$ is the encoding of the initial snapshot corresponding to the execution of any program $\mathcal{P}$ using the inputs $x_1, x_2, \ldots, x_k$.
- $\text{next} : \mathbb{N}^2 \to \mathbb{N}$: For all $z, y \in \mathbb{N}$, if $z = \#(\mathcal{P})$ for some program $\mathcal{P}$ and y encodes some snapshot S for $\mathcal{P}$, then
  - if $\mathcal{P}$ has halted before (or when) S is reached, then $\text{next}(z, y) = y$.
  - Otherwise $\text{next}(z, y)$ is an encoding of the snapshot $\widehat{S}$ that would be reached (by $\mathcal{P}$ from S) by executing one more instruction of $\mathcal{P}$.

A Universal $\mathcal{S}$-Program

- halted : $\mathbb{N}^2 \to \{0, 1\}$. If $z = \#(\mathcal{P})$ for some program $\mathcal{P}$ and y encodes a snapshot S then halted$(z, y) = 1$ if and only if $\mathcal{P}$ has halted before (or when) the snapshot S has been reached.
- output : $\mathbb{N}^2 \to \mathbb{N}$. If $z = \#(\mathcal{P})$ for some program $\mathcal{P}$ and y encodes a snapshot S then
  - If $\mathcal{P}$ has halted before (or when) reaching S then output(z, y) is the output $\mathcal{P}$ returned on termination.
  - Otherwise output$(z, y) = 0$.

Since these are primitive recursive, they are all ***computable***. Thus, if they are used in a program $\mathcal{U}$ as macros then (after macro expansion) an $\mathcal{S}$-program computing the same (partial) function could also be obtained.

A Universal $\mathcal{S}$-Program

It follows that the ***partial function*** from $\mathbb{N}^{k+1}$ to N, computed by the following program $\mathcal{U}$, is computable.

$$
\begin{array}{ll}
 & Z_1 \leftarrow \text{start}_k(X_1, X_2, \ldots, X_k) \\
A_1 & Z_2 \leftarrow \text{halted}(X_{k+1}, Z_1) \\
 & \text{IF } Z_2 \neq 0 \text{ GOTO } A_2 \\
 & Z_3 \leftarrow Z_1 \\
 & Z_1 \leftarrow \text{next}(X_{k+1}, Z_3) \\
 & \text{GOTO } A_1 \\
A_2 & Y \leftarrow \text{output}(X_{k+1}, Z_1)
\end{array}
$$

A Universal $\mathcal{S}$-Program

- Suppose $\mathcal{U}$ is executed with the following inputs:
  - $x_1, x_2, \ldots, x_n \in \mathbb{N}$, and
  - x_{k+1}: The encoding of some $\mathcal{L}$-program $\mathcal{P}$.

  Then $\mathcal{U}$ is computing (and storing in Z_1) the sequence of snapshots

  $$S_1, S_2, S_3, \ldots$$

  corresponding to the execution of $\mathcal{P}$ on the inputs $x_1, x_2, \ldots, x_k$.
- The partial function computed by $\mathcal{U}$ is defined at values $x_1, x_2, \ldots, x_k, x_{k+1}$ and only if $\mathcal{P}$ halts when executed on the inputs $x_1, x_2, \ldots, x_k$.

A Universal $\mathcal{S}$-Program

- Finally, if this partial function is defined at values $x_1, x_2, \ldots, x_k, x_{k+1}$ then the value of this function is the output returned when $\mathcal{P}$ is executed on the inputs encoded by $x_1, x_2, \ldots, x_k$, for $x_{k+1} = \#(\mathcal{P})$.
- Thus $\mathcal{U}$ is an "$\mathcal{S}$-Program Emulator" — or a ***Universal $\mathcal{S}$-Program***.

Normal Form Theorem

An extremely limited use of unbounded minimization — along with a primitive recursive function — is sufficient to obtain *any* Turing-computable (total or partial) functions:

Normal Form Theorem: For every integer $k \geq 1$ and every Turing-computable (partial or total) function $f : \mathbb{N}^k \to \mathbb{N}$ there exists a primitive recursive predicate $p : \mathbb{N}^{k+1} \to \{0, 1\}$ such that, for all $x_1, x_2, \ldots, x_k \in \mathbb{N}$,

$$f(x_1, x_2, \ldots, x_k) = \ell(\mu t(p(x_1, x_2, \ldots, x_k, t))),$$

where the function on the right hand side is understood to be undefined if $\mu t(p(x_1, x_2, \ldots, x_k, t))$ is.

Normal Form Theorem

Sketch of Proof: Let $f : \mathbb{N}^k \to \mathbb{N}$ be a Turing-computable partial or total function, for a positive integer k.

- Then f is also $\mathcal{S}$-computable, so there exists an $\mathcal{S}$-program, $\mathcal{P}$, which computes f.

Normal Form Theorem

- Consider the primitive recursive predicate

$$\text{Halt}_k : \mathbb{N}^{k+2} \to \{0, 1\}$$

and the primitive recursive function

$$\text{Output} : \mathbb{N}^{k+2} \to \mathbb{N}$$

that were introduced when developing a "universal $\mathcal{S}$-program".

- Let $p : \mathbb{N}^{k+1} \to \{0, 1\}$ be the predicate such that, for all $x_1, x_2, \ldots, x_k, t \in \mathbb{N}$, $p(x_1, x_2, \ldots, x_k, t) = 1$ if and only if

$$\text{Halt}_k(x_1, x_2, \ldots, x_k, \#(\mathcal{P}), r(t)) \wedge (\ell(t) = \text{Output}_k(x_1, x_2, \ldots, x_k, \#(\mathcal{P}), r(t))).$$

Normal Form Theorem

- This (rather complicated) predicate is primitive recursive.
- Now, for all $x_1, x_2, \ldots, x_k \in \mathbb{N}$,

$$f(x_1, x_2, \ldots, x_k) = \ell(\mu t(p(x_1, x_2, \ldots, x_k, t))),$$

as needed to establish the claim. □

Normal Form Theorem

Since we already know that every μ-recursive function is Turing-computable, this establishes the following.

Corollary: Let $f : \mathbb{N}^k \to \mathbb{N}$ be a partial or total function, for some positive integer k. Then f is μ-recursive *if and only if* f is Turing computable.

Universal Turing Machines

The fundamental concept being introduced here — ***universal computation*** — can also be introduced when a ***Turing machine*** is being used as the basic computational model, by describing and studying the properties of a ***universal Turing machine***.

- This is, very likely, how students in this course have already seen this.