μ-Recursive Functions

Here we investigate the relationship between Turing machines and computable functions.

For convenience we will restrict ourselves to only look at numeric computations, this does not reflect any loss of generality since all computational problems can be encoded as numbers (think ASCII code). Kurt Gödel used this fact in his famous *incompleteness proof*.

We will show that,

> The functions computable by a Turing machine are exactly the μ-recursive functions.

μ-recursive functions were developed by Gödel and Stephen Kleene.

So, between Turing, Church, Gödel, and Kleene we obtain the following equivalence relation:

$$\text{Algorithms} \Leftrightarrow \text{Turing Machines} \Leftrightarrow \mu\text{-Recursive Functions} \Leftrightarrow \lambda\text{-Calculus}$$

In order to work towards a proof of this equivalence we start with *primitive recursive functions*.

Function Composition

A more general view of function composition in order to define primitive recursive functions,

> Let $g_1, g_2, \ldots, g_n$ be k-variable functions and let h be an n-variable function, then the k-variable function f defined by
>
> $$f(x_1, \ldots, x_k) = h(g_1(x_1, \ldots, x_k), \ldots, g_n(x_1, \ldots, x_k))$$
>
> is called the ***composition*** of h with $g_1, g_2, \ldots, g_n$ and is written as
>
> $$f = h \circ (g_1, \ldots, g_k).$$

NOTE: The function $f(x_1, \ldots, x_k)$ is undefined or $f(x_1, \ldots, x_k) \uparrow$ if either

1. $g_i(x_1, \ldots, x_k) \uparrow$ for some $1 \leq i \leq n$, or
2. $g_i(x_1, \ldots, x_k) = y_i$ for $1 \leq i \leq n$ and $h(y_1, \ldots, y_n) \uparrow$.

NOTE: Here $g(\cdot) \uparrow$ means that g is undefined.

NOTE: Composition is strict in the sense that if any of the arguments of a function are undefined then so is the whole function.

Function Composition

A function f is called a ***total function*** if it is completely defined over its domain, that is, $\forall x, f(x) \downarrow$.[a]

A function f is called a ***partial function*** if it is undefined for at least one element in its domain, that is, $\exists x, f(x) \uparrow$.

[a] You guessed it, the $\downarrow$ indicates that the function is defined.

Primitive Recursive Functions

Definition: The basic ***primitive recursive functions*** are defined as follows:

zero function: $z(x) = 0$ is primitive recursive

successor function: $s(x) = x + 1$ is primitive recursive

projection function: $p_i^{(n)}(x_1, \ldots, x_n) = x_i, 1 \leq i \leq n$ is primitive recursive

More complex primitive recursive function can be constructed by a finite number of applications of,

composition: let $g_1, g_2, \ldots, g_n$ be k-variable primitive recursive functions and let h be an n-variable primitive recursive function, then the k-variable function f defined by

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

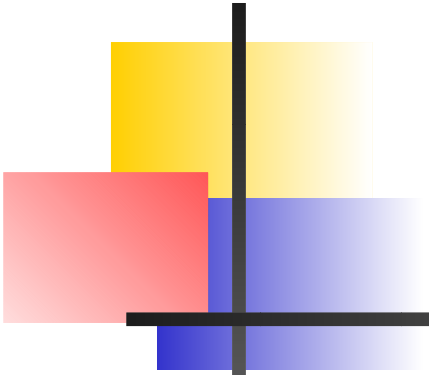
is also primitive recursive.

primitive recursion: let g and h be primitive recursive functions with n and $n + 2$ variables, respectively, then the $n + 1$-variable function f defined by

1. $f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$
2. $f(x_1, \ldots, x_n, s(y)) = h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)$

is also primitive recursive. Here, the variable y is called the ***recursive variable***.

Observation: The primitive recursive functions are total functions.



Primitive Recursive Functions

Why are primitive recursive functions interesting? Because they are so simple that they are considered intuitively computable without any additional proof.

Furthermore, we can use the primitive recursive functions to show that other functions are computable as well, by showing that these other functions can be constructed with primitive recursive functions.

This is similar to the proof approaches with Turing machines and reducibility.

Primitive Recursive Functions

Example: Computing the value for function $f(x_1, \ldots, x_n, s(y))$,

$$\begin{aligned}
f(x_1, \ldots, x_n, 0) &= g(x_1, \ldots, x_n) \\
f(x_1, \ldots, x_n, 1) &= h(x_1, \ldots, x_n, 0, f(x_1, \ldots, x_n, 0)) \\
f(x_1, \ldots, x_n, 2) &= h(x_1, \ldots, x_n, 1, f(x_1, \ldots, x_n, 1)) \\
&\vdots \\
f(x_1, \ldots, x_n, s(y)) &= h(x_1, \ldots, x_n, y, f(x_1, \ldots, x_n, y)).
\end{aligned}$$

Example: Prove that addition is primitive recursive. In order to show this we show that the addition function can be constructed from primitive recursive functions. We let $g = p_1^{(1)}$ and $h = s \circ p_3^{(3)}$,

$$\begin{aligned}
&add(x, 0) = g(x) = p_1^{(1)}(x) = x \\
&add(x, s(y)) = h(x, y, add(x, y)) = s(p^3_{(3)}(x, y, add(x, y))) = s(add(x, y))
\end{aligned}$$

Primitive Recursive Functions

Example: Prove that the constant functions $c_i^{(n)}(x_1, \ldots, x_n) = i$ are primitive recursive. We show that by constructing the constant functions from primitive recursive functions,

$$c_i^{(n)} = \underbrace{s \circ \ldots \circ s}_{i \text{ times}} \circ z \circ p_1^{(n)}.$$

Example: Multiplication. Assume the primitive recursive functions $g = z$ and $h = add \circ (p_3^{(3)}, p_1^{(3)})$, then

$$mult(x, 0) = g(x) = z(x) = 0$$
$$mult(x, s(y)) = h(x, y, mult(x, y)) = add(mult(x, y), x)$$

Example: Factorial. Let $g = c_1^{(1)}$ and $h = mult \circ (p_2^{(2)}, s \circ p_1^{(2)})$,

$$fact(0) = g(0) = c_1^{(1)}(0) = s(z(p_1^{(1)}(0))) = s(z(0)) = s(0) = 1$$
$$fact(s(y)) = h(y, fact(y)) = mult(fact(y), s(y))$$

Observation: We can build up a repertoire of primitive recursive functions that compute many "interesting" functions.

Primitive Recursive Functions

Description	Function	Definition
Addition	$add(x, y)$	$add(x, 0) = x$
	$x + y$	$add(x, y + 1) = add(x, y) + 1$
Multiplication	$mult(x, y)$	$mult(x, 0) = 0$
	$x \cdot y$	$mult(x, y + 1) = mult(x, y) + x$
Predecessor	$pred(y)$	$pred(0) = 0$
		$pred(y + 1) = y$
Proper subtraction	$sub(x, y)$	$sub(x, 0) = x$
	$x \dot{-} y$	$sub(x, y + 1) = pred(sub(x, y))$
Exponentation	$exp(x, y)$	$exp(x, 0) = 1$
	x^y	$exp(x, y + 1) = exp(x, y) \cdot x$

Primitive Recursive Predicates

We define primitive recursive predicates as primitive recursive functions with the co-domain $\{0, 1\}$.

Description	Predicate	Definition
Sign	$sg(x)$	$sg(0) = 0$ $sg(y+1) = 1$
Sign complement	$cosg(x)$	$cosg(0) = 1$ $cosg(y+1) = 0$
Less than	$lt(x, y)$	$sg(y \dot{-} x)$
Greater than	$gt(x, y)$	$sg(x \dot{-} y)$
Equal to	$eq(x, y)$	$cosg(lt(x, y) + gt(x, y))$
Not equal to	$ne(x, y)$	$cosg(eq(x, y))$

Primitive Recursive Functions

Theorem: Let $g(x)$ be a primitive recursive function, then the function

$$f(x) = \begin{cases} y_1 & \text{if } x = n_1 \\ y_2 & \text{if } x = n_2 \\ \vdots & \vdots \\ y_k & \text{if } x = n_k \\ g(x) & \text{otherwise} \end{cases}$$

is also primitive recursive.

Proof: We can express $f(x)$ as the following function using the primitive recursive predicates eq and ne, multiplication, and addition,

$$f(x) = eq(x, n1) \cdot y_1 + \ldots + eq(x, n_k) \cdot y_k + ne(x, n_1) \cdot \ldots \cdot ne(x, n_k) \cdot g(x)$$

Primitive Recursive Functions

Theorem: Let $g(x, y)$ and $h(x)$ be primitive recursive, then the functions,

1. $f(x, y, z_1, \ldots, z_n) = g(x, y)$
2. $f(x, y) = g(y, x)$
3. $f(x) = g(x, x)$
4. $f(x) = h(0)$

are also primitive recursive.

Proof: We show primitive recursion by construction,

1. $f = g \circ (p_1^{(n+2)}, p_2^{(n+2)})$
2. $f = g \circ (p_1^{(2)}, p_2^{(2)})$
3. $f = g \circ (p_1^{(1)}, p_1^{(1)})$
4. $f = h \circ z$

Bounded Minimalization

Bounded minimalization is defined via the search operator μ^y,

$$f(x_1, \ldots, x_n, y) = \mu^y z[p(x_1, \ldots, x_n, z)],$$

and defines a function f which reads "return the least value of z satisfying $p(x_1, \ldots, x_n, z)$ or return the bound," more precisely

$$f(x_1, \ldots, x_n, y) = \begin{cases} \min z & \text{s. t. } p(x_1, \ldots, x_n, z) = 1 \text{ and } 0 \leq z \leq y \\ y + 1 & \text{otherwise} \end{cases}$$

The μ operator can be viewed as a search operator over the natural numbers $\leq y$ for the minimal value that satisfies the predicate p. Consider,

$$f(x, y) = \mu^y z[eq(x, mult(z, z)].$$

Here $f(4, 10) = 2$ and $f(3, 10) = 11$.

Theorem: Let $p(x_1, \ldots, x_n, z)$ be a primitive recursive predicate, then the function

$$f(x_1, \ldots, x_n, y) = \mu^y z[p(x_1, \ldots, x_n, z)],$$

is also primitive recursive.

Proof: On can construct the bounded minimalization of bounded sum. See proof on handout, Theorem 13.3.3.

Primitive Recursive Functions

Theorem: Every primitive recursive function is computable.

Proof Sketch: It is clear that the zero, successor, and projection functions are computable by their sheer simplicity. It remains to show that computable functions are closed under composition and recursion. This can be shown by constructing the appropriate TMs.

(Start by constructing TMs for the basis functions and then plug these simple machines together to obtain more complex ones. Idea: universal turing machines that executes encodings of machines.)

General Recursive Functions

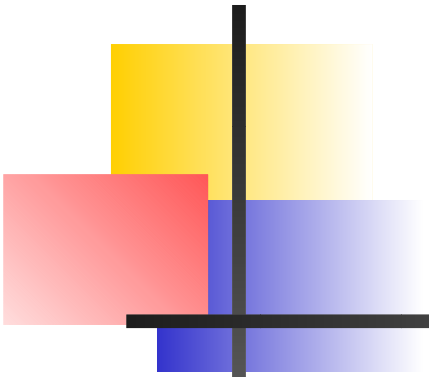
Theorem: There exist total computable functions not representable by primitive recursion.

Proof: By counter example. We can prove this by construction, e.g., Ackermann's function,

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m, n-1)) & \text{if } m > 0 \text{ and } n > 0, \end{cases}$$

where $m, n \geq 0$.

This is a recursive function, but it is not primitive recursive: you cannot define this function according to the primitive recursive function template.



Partial Functions

We relax our notion of computable functions by defining the notion of partial computable function as a Turing machine that does not halt for some of the inputs of the function it implements. In effect the function will then be undefined for these input values, as required by the mathematical definition of partial function.

Unbounded Minimalization

The consequence of the previous arguments is that in order for recursion to provide a computational framework equivalent to Turing machines we will need to *admit partial functions*.

We do so by introducing the ***unbounded minimalization operator*** μ,

$$f(x_1, \ldots, x_n) = \mu z[p(x_1, \ldots, x_n, z)],$$

defines a function f which reads "return the least value of z satisfying $p(x_1, \ldots, x_n, z)$," or

$$f(x_1, \ldots, x_n) = \min z, \qquad \text{s. t. } p(x_1, \ldots, x_n, z) = 1.$$

The μ operator can be viewed as a search operator over the natural numbers for the minimal value that satisfies the predicate p. That f represents a partial function comes from the fact that perhaps no such natural number exists and the search will go on forever. Consider,

$$f(x) = \mu z[eq(x, mult(z, z)].$$

Here, $f(3) \uparrow$.

μ-Recursive Functions

Definition

The family of **μ-recursive functions** is defined as follows:

i) The successor, zero, and projection functions are μ-recursive.

ii) If h is an n-variable μ-recursive function and $g_1, \ldots, g_n$ are k-variable μ-recursive functions, then $f = h \circ (g_1, \ldots, g_n)$ is μ-recursive.

iii) If g and h are n and $n+2$-variable μ-recursive functions, then the function f defined from g and h by primitive recursion is μ-recursive.

iv) If $p(x_1, \ldots, x_n, y)$ is a total μ-recursive predicate, then $f = \mu z[p(x_1, \ldots, x_n, z)]$ is μ-recursive.

v) A function is μ-recursive only if it can be obtained from i) by a finite number of applications of the rules in ii), iii), and iv).

μ-Recursive Functions

Theorem: Turing machines and μ-recursive functions are equivalent.

Proof Sketch: By construction.

(a) We first show that Turing machines can simulate μ-recursive functions. We have already shown that TMs can implement the primitive recursive functions. Since composition is strict it suffices to show that a Turing machine implementation of the μ operator exists for the “if” direction. Since this is a search procedure it is clear that it is algorithmic and a machine can be built for that.

(b) For the converse we provide a procedure to encode any TM as a function based on an enumeration of all possible configurations. Since the enumerations are countable (think of them as binary encoded, finite strings), this is possible and therefore it is possible to construct a function representing a TM.

λ-Calc Implementation

Theorem: If a function is computable by a λ-expression then it is computable by a Turing machine.

Proof Sketch: By construction. Computing with λ-expressions is algorithmic. It is straightforward to construct an encoding for λ-expressions and then perform the algorithm for function applications. That this is possible is demonstrated that we have functional programming languages running on Von Newmann style computers.

λ-Calc Implementation

Theorem: μ-recursive functions can be implemented in the λ-calculus.

Proof Sketch: Since they are functions and the μ operator is algorithmic it is clear that the functions can be implemented in the λ-calculus. Consider,

- The zero, successor, and projection functions,

$$\begin{aligned} \text{ZERO} &= \lambda x.\, 0 \\ \text{SUCC} &= \lambda x.\, x+1 \\ \text{PROJ}_i^n &= \lambda(x_1, \ldots, x_n).\, x_i \end{aligned}$$

- The composition of the function h with the functions $g_1, g_2, \ldots, g_n$, applied to the tuple $(x_1, x_2, \ldots, x_k)$ is $h(g_1(x_1, x_2, \ldots, x_k), g_2(x_1, x_2, \ldots, x_k), \ldots, g_n(x_1, x_2, \ldots, x_k))$, or as a λ-expression:

$$\lambda f g_1 \ldots g_n(x_1, x_2, \ldots, x_k).\, f\,(g_1(x_1, x_2, \ldots, x_k)) \ldots (g_n(x_1, x_2, \ldots, x_k))$$

λ-Calc Implementation

- The primitive recursive function f with

$$\begin{aligned} f(x_1, \ldots, x_n, 0) &= h(x_1, \ldots, x_n) \\ f(x_1, \ldots, x_n, y) &= g(x_1, \ldots, x_n, y-1, f(x_1, \ldots, x_n, y-1)) \end{aligned}$$

can be written as the λ-expression

$$\lambda hg.\, Y\, (\lambda f(x_1, \ldots, x_n)y.\, y = 0?\, (h\, (x_1, \ldots, x_n)) : (g\, (x_1, \ldots, x_n)\, (y-1)\, (f\, (x_1, \ldots, x_n)\, (y-1))))$$

Where Y is called the fixed-point operator (or Y combinator) and is defined as

$$Y = \lambda f.\, (\lambda x.\, f\, (x\, x))(\lambda x.\, f\, (x\, x))$$

This operator expresses recursive computation in pure λ-calculus.

λ-Calc Implementation

- The expression $\mu z[p(x_1, \ldots, x_n, z)]$ returns the smallest z such that $p(x_1, \ldots, x_n, z) = 0$, as a λ-expression:

$$\lambda p\,(x_1, \ldots, x_n).\,(Y\,(\lambda h z.\,(p\,(x_1, \ldots, x_n)\,z) = 0?\,z : (h\,(z+1)))\,0)$$

This completes the proof. $\square$

The Equivalence

λ-Calculus $\prec$ Turing Machines $\prec$ μ-Recursive Functions $\prec$ λ-Calculus

where $a \prec b$ means b implements a.