Recursive functions and Turing machines

1 Primitive recursive and partial recursive functions

Definition 1.1 (Initial functions). *The following are the initial functions:*

Zero $Z(n) = 0$;

Successor $S(n) = n + 1$; *and*

Projection $\Pi^k_i(n_1, \dots, n_k) = n_i$ *(one projection for every pair k, i with $i \leqslant k$).*

Definition 1.2 (Composition). *A function $f : \mathbb{N}^k \to \mathbb{N}$ is obtained by* composition *from $g : \mathbb{N}^l \to \mathbb{N}$ and $h_1, \dots, h_l : \mathbb{N}^k \to \mathbb{N}$ if*

$$f(\vec{n}) = g(h_1(\vec{n}), \dots, h_l(\vec{n})).$$

We use the notation $f = g \circ (h_1, h_2, \dots, h_l)$.

Definition 1.3 (Primitive recursion). *A function $f : \mathbb{N}^{k+1} \to \mathbb{N}$ is obtained by* primitive recursion *from $g : \mathbb{N}^k \to \mathbb{N}$ and $h : \mathbb{N}^{k+2} \to \mathbb{N}$ if*

$$\begin{aligned} f(0, \vec{n}) &= g(\vec{n}) \\ f(i+1, \vec{n}) &= h(i, f(i, \vec{n}), \vec{n}) \end{aligned}$$

If g and h are total functions, f is also total.

Definition 1.4 (μ-recursion). *A function $f : \mathbb{N}^k \to \mathbb{N}$ is obtained by* μ-recursion *or* minimization *from $g : \mathbb{N}^{k+1} \to \mathbb{N}$ if*

$$f(\vec{n}) = \begin{cases} i & \text{if } g(i, \vec{n}) = 0 \text{ and } \forall j < i : g(j, \vec{n}) > 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

We use the notation $f(\vec{n}) = \mu i(g(i, \vec{n}) = 0)$. Note that f need not be total even when g is, and that if $f(\vec{n}) = i$, then $g(j, \vec{n})$ is defined for all $j \leqslant i$.

Definition 1.5 (Primitive recursive, recursive functions). *The class of* primitive recursive functions *is the smallest class of functions containing the initial functions, and closed under composition and primitive recursion.*

The class of (partial) recursive functions *is the smallest class of functions containing the initial functions, and closed under composition, primitive recursion and μ-recursion.*

2 Recursive functions are Turing computable

Since we know that Turing machines can simulate simple **while** programs, we show how recursive functions can be translated to programs.

- The initial functions have trivial programs.
- If $f : \mathbb{N}^k \to \mathbb{N}$ is defined by $f = g \circ (h_1, \ldots, h_l)$, the following program computes f, assuming programs already exist for g and the h_i's.

```
int f(int x1, int x2, ..., int xk) {
    y1 = h1(x1, x2, ..., xk);
    y2 = h2(x1, x2, ..., xk);
    ...
    yl = hl(x1, x2, ..., xk);
    return g(y1, y2, ..., yl);
}
```

- If $f : \mathbb{N}^{k+1} \to \mathbb{N}$ is defined from $g : \mathbb{N}^k \to \mathbb{N}$ and $h : \mathbb{N}^{k+2} \to \mathbb{N}$ by primitive recursion, the following program computes f, assuming programs already exist for g and h.

```
int f(int y, int x1, ..., int xk) {
    result = g(x1, ..., xk);   // f(0, x1, ..., xk)
    for (i = 0; i < y; i++) {  // computing f(i+1, x1, ..., xk)
        result = h(i, result, x1, ..., xk);
    }
    return result;
}
```

- If $f : \mathbb{N}^k \to \mathbb{N}$ is defined from $g : \mathbb{N}^{k+1} \to \mathbb{N}$ by μ-recursion, then here is the program for f.

```
int f(int x1, ..., int xk) {
    i = 0;
    while (g(i, x1, ..., xk) > 0) {
        i = i + 1;
    }
    return i;
}
```

3 Primitive recursive functions and relations – examples

The first few examples are written strictly in the official template, specifying the exact g and h used to obtain the function. Then we slip to an informal notation and just write recursive equations. The reader can convert them to the official template.

- $f(n) = n + 2$ is $S \circ S$
- $plus(n, m) = n + m$ is got by primitive recursion from $g = \Pi^1_1$ and $h = S \circ \Pi^3_2$. It is easily verified that $plus(0, m) = g(m) = \Pi^1_1(m) = m$, and that $plus(n + 1, m) = h(n, plus(n, m), m) = (S \circ \Pi^3_2)(n, plus(n, m), m) = S(plus(n, m)) = (n + m) + 1 = (n + 1) + m$.
- $mult(n, m) = nm$ is got by primitive recursion from $g = Z$ and $h = plus \circ (\Pi^3_2, \Pi^3_3)$. Verifying the equations is left as an exercise.
- $exp(n, m) = m^n$ is got by primitive recursion from $g = S \circ Z$ and $h = mult \circ (\Pi^3_2, \Pi^3_3)$. Verifying the equations is again left as an exercise.
- $sum(n) = \sum_{i=0}^{n} i$ is defined as $sum' \circ (\Pi^1_1, \Pi^1_1)$, where $sum'(n, m) = \sum_{i=0}^{n} i$ is defined by primitive recursion from $g = Z$ and $h = plus \circ (S \circ \Pi^3_1, \Pi^3_2)$.

  Clearly $sum'(0, m) = Z(m) = 0$ as desired, while

  $$sum'(n+1, m) = h(n, sum'(n, m), m) = (plus \circ (S \circ \Pi^3_1, \Pi^3_2))\,(n, sum'(n, m), m) = (n+1) + \sum_{i=0}^{n} i = \sum_{i=0}^{n+1} i.$$

- The predecessor function on natural numbers is defined as follows:

  $$pred(n) = \begin{cases} 0 & \text{if } n = 0 \\ n - 1 & \text{otherwise} \end{cases}$$

  It is primitive recursive: $pred = pred' \circ (\Pi^1_1, \Pi^1_1)$ where $pred'$ is obtained by primitive recursion from $g = Z$ and $h = \Pi^3_1$.
- *Cutoff subtraction* $m - n$ on natural numbers is defined as usual, except that $m - n = 0$ if $m \leqslant n$. It can be defined using primitive recursion from $g = \Pi^1_1$ and $h = pred \circ \Pi^3_2$.
- *Factorial*

  $$0! = 1$$
  $$(n + 1)! = (n + 1) \cdot n!$$

- *Bounded sums* $g(z, \vec{x}) = \sum_{y \leqslant z} f(y, \vec{x})$ is defined as follows:

  $$g(0, \vec{x}) = f(0, \vec{x})$$
  $$g(y + 1, \vec{x}) = g(y, \vec{x}) + f(y + 1, \vec{x})$$

- *Bounded products* $g(z,\vec{x}) = \prod_{y \leqslant z} f(y,\vec{x})$ is defined as follows:

$$g(0,\vec{x}) = f(0,\vec{x})$$
$$g(y+1,\vec{x}) = g(y,\vec{x}) \cdot f(y+1,\vec{x})$$

Definition 3.1. *A relation* $R \subseteq \mathbb{N}^k$ *is primitive recursive if its characteristic function* c_R *is primitive recursive.*

- *iszero* is primitive recursive since c_{iszero} is a primitive recursive function.

$$\begin{aligned} iszero(0) &= true & \qquad c_{iszero}(0) &= succ(Z(0)) \\ iszero(n+1) &= false & \qquad c_{iszero}(n+1) &= Z(n) \end{aligned}$$

- $x \leqslant y$ iff $iszero(x - y)$, so $c_{\leqslant}(x,y) = c_{iszero}(x - y)$, and hence $\leqslant$ is a primitive recursive relation.

- $c_{\neg\varphi} = 1 - c_{\varphi}$, $c_{\varphi\wedge\psi} = c_{\varphi} \cdot c_{\psi}$, so primitive recursive relations are closed under boolean operations.

- For $\varphi(z,\vec{x}) = (\forall y \leqslant z)\psi(y,\vec{x})$, $c_{\varphi}(z,\vec{x}) = \prod_{y \leqslant z} c_{\psi}(y,\vec{x})$, hence primitive recursive relations are closed under bounded universal quantification.

- $x = y$, $x < y$, $\varphi \vee \psi$, $\varphi \to \psi$, $(\exists y \leqslant z)\varphi(y,\vec{x})$ &c. are obtained easily by combining the above logical operators.

- *Bounded* μ*-recursion*: If $\varphi(y,\vec{x})$ is a relation, then $\mu y : \varphi(y,\vec{x})$ is defined to be $\mu y.(1 - c_{\varphi}(y,\vec{x}) = 0)$, the smallest y for which $\varphi(y,\vec{x})$ holds. This is not necessarily primitive recursive, but when we apply a bound on the search, it is. Bounded μ-recursion is defined as follows:

$$\mu y_{\leqslant z}\, \varphi(y,\vec{x}) = \begin{cases} \mu y.\varphi(y,\vec{x}) & \text{if } (\exists y \leqslant z)\varphi(y,\vec{x}) \\ z+1 & \text{otherwise} \end{cases}$$

  It can be shown to be primitive recursive if φ is. Let $\psi'(y,\vec{x})$ be $(\forall w < y)\neg\varphi(w,\vec{x})$ and $\psi(y,\vec{x})$ be $\varphi(y,\vec{x}) \wedge \psi'(y,\vec{x})$. If φ is primitive recursive, so are ψ' and ψ, and

$$\mu y_{\leqslant z}\, \varphi(y,\vec{x}) = \left(\sum_{y \leqslant z} y \cdot c_{\psi}(y,\vec{x})\right) + (z+1) \cdot c_{\psi'}(z+1,\vec{x}).$$

- *x divides y*

$$x|y \ \text{ iff } \ (\exists z \leqslant y)\ (x \cdot z = y)$$

- *x is even*

$$even(x) \ \text{ iff } \ 2|x$$

- *x is odd*

$$odd(x) \text{ iff } \neg even(x)$$

- *x is a prime*

$$prime(x) \text{ iff } x \geqslant 2 \wedge (\forall y \leqslant x)(y|x \to y = 1 \vee y = x)$$

- *the n-th prime (this is a function)*

$$\begin{aligned} Pr(0) &= 2 \\ Pr(n+1) &= \text{the smallest prime greater than } Pr(n) \\ &= \mu y_{\leqslant Pr(n)!+1}\,(prime(y) \wedge y > Pr(n)) \end{aligned}$$

  The (very loose) bound is guaranteed by Euclid's proof. You can use Bertrand's postulate to get better bounds.

- *the exponent of (the prime) k in the decomposition of y*

$$exp(y,k) = \mu x_{\leqslant y}\left[k^x | y \wedge \neg(k^{x+1}|y)\right]$$

- $\frac{x}{2} = \mu y_{\leqslant x}(2y \geqslant x)$

- There is a primitive coding of the plane in natural numbers. The standard Cantor bijection between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$ is primitive recursive, defined by

$$pair(x,y) = \frac{(x+y)^2 + 3x + y}{2}$$

- The inverses are also primitive recursive:

$$\begin{aligned} fst(z) &= \mu x_{\leqslant z}\left[(\exists y \leqslant z)(z = pair(x,y))\right] \\ snd(z) &= \mu y_{\leqslant z}\left[(\exists x \leqslant z)(z = pair(x,y))\right] \end{aligned}$$

- Finally, finite sequences of natural numbers can be coded in a primitive recursive fashion.

  - The sequence $x_1, \ldots, x_n$ (of length n) is coded by

$$Pr(0)^n \cdot Pr(1)^{x_1} \cdot Pr(2)^{x_2} \cdots Pr(n)^{x_n}$$

  - *n-th element of the sequence coded by x*

$$(x)_n = exp(x, Pr(n))$$

  - *length of sequence coded by x*

$$ln(x) = (x)_0$$

  - *x is a sequence number, i.e. codes a sequence* (this is a predicate)

$$Seq(x) \text{ iff } (\forall n \leqslant x)\left[(n > 0 \wedge (x)_n \neq 0) \to n \leqslant ln(x)\right]$$

4 Turing computable functions are recursive

Definition 4.1 (Turing machines)**.** *A (two-way infinite, non-deterministic)* Turing machine M *is a triple* (Q, Σ, Δ) *where:*

- $Q = \{q_0, q_1, \ldots, q_l\}$ *is a finite set of* states *(we adopt the convention that the first one listed,* q_0*, is the* initial state*, and that the next one,* q_1*, is the* final state*);*
- $\Sigma = \{0, 1\}$ *is the* tape alphabet*; and*
- Δ *is a finite set of* transitions*, each of the form*

$$(q_i, a) \longrightarrow (q_j, b, d),$$

where $i, j \leqslant l$, $a, b \in \Sigma$, $d \in \{L, R\}$*. A transition of the above form means that the machine, in state* q_i *and reading symbol* a *on the tape, switches to state* q_j*, overwriting the tape cell with the symbol* b*, and moves in direction specified by* d *(either left or right).*

Definition 4.2 (Turing machine configurations)**.** *Suppose* $M = (Q, \Sigma, \Delta)$ *is Turing machine. A* configuration *is a triple* (q, t, i) *where:*

- $q \in Q$ *is the* current state*;*
- $t : \mathbb{Z} \to \Sigma$ *is the* tape contents *(we are assuming that the tape cells are indexed by the integers), such that* $t(i) = 0$ *for all but finitely many* $i \in \mathbb{Z}$*; and*
- $i \in \mathbb{Z}$ *is the position of the tape head.*

An initial configuration *is a triple* (q, t, i) *where* $q = q_0$ *(the initial state) and* $t(j) = 0$ *for all* $j > i$*.*

A final configuration *is a triple* (q, t, i) *where* $q = q_1$ *(the final state) and* $t(j) = 0$ *for all* $j > i$*.*

For any binary string $w = b_{n-1}b_{n-2}\ldots b_0$ *(where* $n \geqslant 0$*), the number it represents, denoted* $val(w)$*, is defined to be* $\sum_{0 \leqslant i < n} b_i \cdot 2^i$*. When* $n = 0$*, then* w *is the empty word, and* $val(w) = 0$*.*

For any configuration $C = (q, t, i)$*, if* $t(j) = 0$ *for all* $j \in \mathbb{Z}$*, define* $L(C) = R(C) = 0$*. Otherwise let* j *and* k *being the smallest and largest indices such that* $t(j) = 1$ *and* $t(k) = 1$*. We define* $L(C)$ *and* $R(C)$ *as below:*

- $L(C) = val\big(t(j)t(j+1)\cdots t(i)\big)$ *(if* $i < j$*, the string is empty and its value is* 0*); and*
- $R(C) = val\big(t(k)t(k-1)\cdots t(i+1)\big)$ *(if* $k \leqslant i$*, the string is empty and represents* 0*). We read the tape to the right of the head in reverse, to make it easy to define* $L(C')$ *and* $R(C')$ *from* $L(C)$ *and* $R(C)$*, when there is a transition from* C *to* C'*.*

We can always define our machines in such a way that there is always some transition out of every non-final configuration, but there is no transition out of any final configuration. Then a machine halts on an input if and only if it reaches a final configuration, starting from the initial configuration representing the input.

Definition 4.3 (Turing computability). *A (partial) function $f : \mathbb{N} \to \mathbb{N}$ is Turing computable if there is a Turing machine M such that for all $n \in \mathbb{N}$, $f(n) = m$ iff M started with initial configuration C_i such that $L(C_i) = n$ eventually halts in a final configuration C_f such that $L(C_f) = m$.*

We emphasize that M does not halt on inputs where f is not defined. It suffices to consider unary functions, since we can code up multiple inputs into one number.

Coding configurations Fix a Turing machine $M = (\{q_0, q_1, \dots, q_l\}, \{0, 1\}, \Delta)$. The following encodings are primitive recursive.

- A configuration $C = (q_j, t, i)$ of M is coded by the number $pair(j, pair(L(C), R(C)))$.
- The state of a configuration coded by n is given by $state(n) = fst(n)$.
- The tape contents to the left of the head in a configuration coded by n is given by $left(n) = fst(snd(n))$.
- The tape contents to the right of the head in a configuration coded by n is given by $right(n) = snd(snd(n))$.
- The predicate $config(n)$, that says that n codes up a configuration of M, is defined by $0 \leqslant state(n) \leqslant l$.
- $initial(n) \iff state(n) = 0 \wedge right(n) = 0$ says that n codes up an initial configuration.
- $final(n) \iff state(n) = 1 \wedge right(n) = 0$ says that n codes up a final configuration.

Coding transitions Fix a Turing machine $M = (\{q_0, q_1, \dots, q_l\}, \{0, 1\}, \Delta)$ like before. We show how to code transitions by primitive recursive predicates, by way of two examples.

- Suppose $t \in \Delta$ is the transition $(q_4, 0) \longrightarrow (q_8, 1, L)$. We define the primitive recursive predicate $step_t(c, c')$ meaning that t can be fired in configuration coded by c, yielding a configuration coded by c'. Letting $c = pair(i, pair(l, r))$ and $c' = pair(i', pair(l', r'))$, we have the following constraints:
  - $i = 4$ and $i' = 8$;
  - rightmost bit of l is 0, i.e. $even(l)$ holds;
  - l' is got by dropping the last bit of l, .i.e. $l' = \frac{l}{2}$; and
  - r' acquires a new rightmost bit, which is 1, i.e. $r' = 2r + 1$.

  We can define $step_t(c, c')$ as follows:

$$config(c) \wedge config(c') \wedge state(c) = 4 \wedge state(c') = 8 \wedge even(left(c)) \wedge$$
$$2 \cdot left(c') = left(c) \wedge right(c') = 2 \cdot right(c) + 1$$

- Suppose $t \in \Delta$ is the transition $(q_7, 1) \longrightarrow (q_2, 0, R)$. Letting $c = pair(i, pair(l, r))$ and $c' = pair(i', pair(l', r'))$, we have the following constraints:
  - $i = 7$ and $i' = 2$;
  - rightmost bit of l is 1, i.e. $odd(l)$ holds;
  - if we let b be the rightmost bit of r, i.e. $b = c_{odd}(r)$, l' acquires b as its rightmost bit, and its second bit from right changes from 1 to 0, i.e..i.e. $l' = 2(l - 1) + b$; and
  - r' is got by dropping the rightmost bit of r i.e. $r' = \frac{r}{2}$.

  We can define $step_t(c, c')$ as follows:

$$config(c) \wedge config(c') \wedge state(c) = 7 \wedge state(c') = 2 \wedge odd(left(c)) \wedge$$
$$left(c') = 2(left(c) - 1) + c_{odd}(right(c)) \wedge 2 \cdot right(c') = right(c)$$

Coding transitions and runs Fix a Turing machine $M = (\{q_0, q_1, \ldots, q_l\}, \{0, 1\}, \Delta)$ like before. We present primitive recursive encodings of runs.

- $step_M(c, c') \Leftrightarrow \bigvee_{t \in \Delta} step_t(c, c')$.
- A (terminating) run of M on input n is a sequence of configurations $c_1, \ldots, c_k$ such that:
  - c_1 is an initial configuration with $left(c_1) = n$;
  - c_k is a final configuration, with the output recoverable as $left(c_k)$; and
  - for all $i < k$, $step_M(c_i, c_{i+1})$ holds.
- Here is the primitive recursive predicate $run_M(n, s)$, which says that s codes up a terminating run of M on input n (we always put the result m, which is recoverable from the last configuration of the run, in an easily accessible position of s):

$$\exists r, k, m \leqslant s \{ s = pair(m, r) \wedge Seq(r) \wedge k = ln(r) \wedge initial((r)_1) \wedge final((r)_k) \wedge$$
$$left((r)_1) = n \wedge left((r)_k) = m \wedge (\forall i < k)[step_M((r)_i, (r)_{i+1})] \}$$

- If s codes a run of M, $fst(s)$ returns the output of the run.

Turing computable functions are recursive

Theorem 4.4. *If $f : \mathbb{N} \to \mathbb{N}$ is a Turing computable (partial) function, it is also partial recursive.*

Proof. Suppose f is computed by a Turing machine M. We define f on input $n \in \mathbb{N}$ as follows:

$$f(n) = fst\left[\mu s.run_M(n, s)\right].$$

♣

A consequence is *Kleene's normal form theorem*, which states that recursive functions are precisely those that can be expressed as $fst(\mu s.T(n, s))$ for a primitive recursive predicate T. (Anything of the form $fst(\mu s.T(n, s))$ for primitive recursive T is clearly recursive. In the other direction, given a recursive function f, simply translate it to its Turing machine description, and translate back using the above theorem.)