Many Functions and Constructions Are Primitive Recursive

Previous and subtraction. We have shown that addition and multiplication are primitive recursive. The natural next-to-addition operation is subtraction $a \dot{-} b$, and, in particular, the operation of $a \dot{-} \dot{-}$ of subtracting 1, that produces a previous natural number (or 0 if the input was 0). For convenience, let us denote this subtracting-1 operation by $prev(a)$.

Theorem 1. *The function $prev(a)$ is primitive recursive.*

Proof. The function $prev(a)$ can be described by the following two equations:

$$prev(0) = 0;$$

$$prev(m+1) = m.$$

In general, a function of one variable described by primitive recursion has the form

$$h(0) = f;$$

$$h(m+1) = g(m, h(m)).$$

After renaming, the equations describing the function $prev(a)$ takes the following form:

$$h(0) = 0;$$

$$h(m+1) = m.$$

So here, $f = 0$, $g = \pi_1^2$, hence $prev = PR(0, \pi_1^2)$. Thus, the function $prev(a)$ is indeed primitive recursive. The theorem is proven.

Theorem 2. *Subtraction $sub(a, b) = a \dot{-} b$ is primitive recursive.*

Proof. Subtracting b means that we subtract one b times:

$$a \dot{-} b = a \dot{-} 1 \dot{-} 1 \dot{-} \ldots \dot{-} 1 \text{ (b times)}.$$

Thus, we can describe subtraction as a for-loop:

```
int c = a;
for(int i = 1, i <= b; i++)
  {c = prev(c);}
```

Here,

$$c(a, 0) = a;$$

$$c(a, m + 1) = prev(c(a, m)).$$

In general, a function of two variables described by primitive recursion has the form

$$h(n_1, 0) = f(n_1);$$

$$h(n_1, m + 1) = g(n_1, m, h(n_1, m)).$$

After renaming, the equations describing the function $prev(a)$ takes the following form:

$$h(n_1, 0) = n_1;$$

$$h(n_1, m + 1) = prev(h(n_1, m)).$$

So here, $f = \pi_1^1$ and $g = prev(\pi_3^3)$, hence $sub = PR(\pi_1^1, prev \circ \pi_3^3)$. Since $prev$ is primitive recursive, the function $sub(a, b)$ is indeed primitive recursive. The theorem is proven.

What next. To show that division and remainder are primitive recursive, we first need to show that conditional statements are primitive recursive. A general conditional statement has the form

```
if(<condition>)<statement> else <statement>
```

Here, a statement can be equality or inequality between numerical values, or a boolean combination of such statements – obtained by using "and", "or", and "not". Following the usual computer representation of truth values, we will assume that "false" is 0 and "true" is 1.

To prove that everything computed this way is primitive recursive, we will therefore follow the following steps:

- first, we will prove that boolean operations are primitive recursive;
- then, we will prove that equalities and inequalities are primitive recursive;
- finally, we will prove that every conditional statement is primitive recursive.

Theorem 3. *Negation $not(a)$ is primitive recursive.*

Proof. Since "false" is 0 and "true" is 1, negation is described by the following two formulas: $neg(0) = 1$ and $neg(1) = 0$. One can easily check that in both cases, we have $neg(a) = 1 \dot{-} a$. Since 1 is primitive recursive and $\dot{-}$ is primitive recursive, negation is primitive recursive too.

Theorem 4. *The "and"-operation $and(a, b)$ is primitive recursive.*

Proof. There are 4 possible combinations of truth values of a and b, and the corresponding truth values are as follows:

$$and(0, 0) = and(0, 1) = and(1, 0) = 0; \quad and(1, 1) = 1.$$

One can easily check that in all 4 cases, $and(a, b) = a \cdot b$. Since multiplication is primitive recursive, the function $and(a, b)$ is therefore also primitive recursive.

Comment. It is not accidental that in digital design, "and" is described as AB – the same way as multiplication: "and"-operation *is* multiplication. The only reason why in Java they are different is that Java is a typed language, so multiplication can be only applies to numbers, but not to truth values.

Theorem 5. *The "or"-operation $or(a, b)$ is primitive recursive.*

Proof. By De Morgan laws, $or(a, b) = not(and(not(a), not(b)))$. Since *and* and *not* are primitive recursive, we can conclude that the function *or* is also primitive recursive – as a composition of primitive recursive functions.

Theorem 6. *The function $eq0(a)$ – that checks whether a is equal to 0 – is primitive recursive.*

Proof. Indeed, we have

$$eq0(0) = 1;$$

$$eq0(m + 1) = 0.$$

In general, a function of one variable described by primitive recursion has the form

$$h(0) = f;$$

$$h(m + 1) = g(m, h(m)).$$

After renaming, the equations describing the function $eq0(a)$ takes the following form:

$$h(0) = 1;$$

$$h(m + 1) = 0.$$

So here, $f = 1 = \sigma \circ 0$, $g = 0$, hence $eq0 = PR(\sigma \circ 0, 0)$. Thus, the function $eq0(a)$ is indeed primitive recursive. The theorem is proven.

Theorem 7. *The function $leq(a, b)$ – that checks whether $a \le b$ – is primitive recursive.*

Proof. From the definition of $a \dot{-} b$, it follows that $a \dot{-} b$ is equal to 0 if and only if $a \le b$ is true. Thus, $leq(a, b) = eq0(a \dot{-} b)$. Since $eq0$ and $\dot{-}$ are primitive recursive, we can conclude that the function *leq* is also primitive recursive – as a composition of primitive recursive functions.

Theorem 8. *The function $eq(a, b)$ – that checks whether $a = b$ – is primitive recursive.*

Proof. One can easily check that $a = b$ if and only if $a \le b$ and $b \le a$. Thus, $eq(a, b) = and(leq(a, b), leq(b, a))$. Since *and* and *leq* are primitive recursive, we can conclude that the function *eq* is also primitive recursive – as a composition of primitive recursive functions.

Theorem 9. *The function $lt(a, b)$ – that checks whether $a < b$ – is primitive recursive.*

Proof. One can easily check that $a < b$ if and only if $a \leq b$ and $b \not\leq a$. Thus, $lt(a, b) = and(leq(a, b), not(leq(b, a)))$. Since *and*, *leq*, and *not* are primitive recursive, we can conclude that the function lt is also primitive recursive – as a composition of primitive recursive functions.

Theorem 10. *The function $neq(a, b)$ – that checks whether $a \neq b$ – is primitive recursive.*

Proof. One can easily check that $a \neq b$ if and only if it is not true that $a = b$. Thus, $neq(a, b) = not(eq(a, b))$. Since *not* and *eq* are primitive recursive, we can conclude that the function *neq* is also primitive recursive – as a composition of primitive recursive functions.

Definition. *Let $P(\overline{n})$ be a function whose values are 0 or 1, and let $f(\overline{n})$ and $g(\overline{n})$ are functions. By an* if-then-else function

$$h(\overline{n}) = if(P(\overline{n}))\ f(\overline{n})\ else\ g(\overline{n}),$$

we mean the following function:

- *$h(\overline{n}) = f(\overline{n})$ if $P(\overline{n}) = 1$, and*
- *$h(\overline{n}) = g(\overline{n})$ if $P(\overline{n}) = 0$.*

Theorem 11. *If $P(\overline{n})$, $f(\overline{n})$, and $g(\overline{n})$ are primitive recursive, then the function*

$$h(\overline{n}) = if(P(\overline{n}))\ f(\overline{n})\ else\ g(\overline{n})$$

is also primitive recursive.

Proof. Let us take $h(\overline{n}) = P(\overline{n}) \cdot f(\overline{n}) + (1 \dot{-} P(\overline{n})) \cdot g(\overline{n})$. Then:

- if $P(\overline{n}) = 1$, then $1 \dot{-} P(\overline{n}) = 0$, so $h(\overline{n}) = 1 \cdot f(\overline{n}) + 0 \cdot g(\overline{n}) = f(\overline{n})$;
- if $P(\overline{n}) = 0$, then $1 \dot{-} P(\overline{n}) = 1$, so $h(\overline{n}) = 0 \cdot f(\overline{n}) + 1 \cdot g(\overline{n}) = g(\overline{n})$.

So, we indeed have the desired function. Since $P(\overline{n})$, $f(\overline{n})$, $g(\overline{n})$, $\cdot$, $+$, and $\dot{-}$ are primitive recursive, the function $h(\overline{n})$ is indeed primitive recursive as a composition of primitive recursive functions.

Comment. The above formula makes sense:

- we get $f(n)$ when $P(n)$ is true, i.e., when $P(n) = 1$, and
- we get $g(n)$ when $P(n)$ is false, i.e., when $notP(n) = 1 \dot{-} P(n)$ is true, i.e., when $1 \dot{-} P(n) = 1$.

Theorem 12. *The remainder function $rem(a, b) = b \ \% \ a$ is primitive recursive.*

Proof. Let us recall how remainder with respect to a fixed number a – e.g., with respect to $a = 3$ – change when a changes. We have:

$$0 \ \% \ 3 = 0;$$

$$1 \ \% \ 3 = 1;$$

$$2 \ \% \ 3 = 2;$$

$$3 \ \% \ 3 = 0;$$

$$4 \ \% \ 3 = 1;$$

$$5 \ \% \ 3 = 2;$$

$$6 \ \% \ 3 = 0;$$

$$7 \ \% \ 3 = 1, etc.$$

In general, the remainder starts with 0, and then grows by 1 unless adding 1 will lead to a – in this case the remainder is set back at 0.

This shows that the remainder can be described as follows:

$$rem(a, 0) = 0;$$

$$rem(a, m + 1) = if(rem(a, m) + 1 < a) \ rem(a, m) + 1 \ else \ 0.$$

Since $<$, $+1$ (σ), and if-then-else construction are all primitive recursive, the right-hand side of the last formula is also primitive recursive. Thus, the remainder function *rem* is primitive recursive.

Theorem 13. *The integer division function $div(a, b) = b \ / \ a$ is primitive recursive.*

Proof. Let us recall how the result $b \ / \ a$ of dividing by a fixed number a – e.g., by $a = 3$ – change when a changes. We have:

$$0 \ / \ 3 = 0;$$

$$1 \ / \ 3 = 0;$$

$$2 \ / \ 3 = 0;$$

$$3 \ / \ 3 = 1;$$

$$4 \ / \ 3 = 1;$$

$$5 \ / \ 3 = 1;$$

$$6 \ / \ 3 = 2;$$

$$7 \ / \ 3 = 2, etc.$$

In general, the value $b \ / \ a$ starts with 0, and then remains the same unless we face a number which is divisible by a – i.e., for which $rem(a, b) = 0$.

This shows that the integer division function can be described as follows:

$$div(a, 0) = 0;$$

$$div(a, m + 1) = if(0 < rem(a, m + 1)) \ div(a, m) \ else \ div(a, m) + 1.$$

Since $<$, $+1$ (σ), and if-then-else construction are all primitive recursive, the right-hand side of the last formula if also primitive recursive. Thus, the division function *div* is primitive recursive.