Assignment 3: Solutions and Comments
CS 2MJ3 Fall 2009

Nick James

November 24, 2009

Note

There are two logic conventions that are commonly used in computability theory:

1. 0 = true, 1 = false
2. 1 = true, 0 = false

Obviously either one can be used; the symbols can be interpreted however you like. Unforunately, the convention chosen radically affects some of the solutions. Solutions that are correct using one convention are completely wrong using the other. The textbook sticks with Convention #2, so you may be most familiar with that. The assignment seems to exhibit no preference until Question 4, in which $Prime(x) = 0$ if x ***is*** prime, and 1 if isn't—suggesting Convention #1. It also appears that many students sometimes use one convention and then the other (which makes marking these assignments REALLY HARD, man!). Please... I beseech you... Stick with one convention! I don't care which one, but the text sticks with #2, so let's all use that.

That said, I'll present solutions using both conventions so you can see how it works (assuming I don't just make a huge mess).

1 Bounded Sum and Product

(a) The bounded sum is defined, using typical notation as,

$$sum(\vec{x}, y) = \sum_{z<y} q(\vec{x}, z)$$

We can define it using the primitive recursive scheme to show that it is a primitive recursive function if q is primitive recursive:

$$\begin{aligned} sum(\vec{x}, 0) &= 0 \\ sum(\vec{x}, y+1) &= sum(\vec{x}, y) + q(\vec{x}, y) \end{aligned}$$

That's a complete, but informal answer. There's nothing wrong with it, really, but if you prefer a more formal treatment, it looks like this:

$$\begin{aligned} sum(\vec{x}, 0) &= f(\vec{x}) \\ sum(\vec{x}, S(y)) &= g(\vec{x},\, sum(\vec{x}, y),\, y) \end{aligned}$$

where

$$\begin{aligned} f(\vec{x}) &= 0 \\ g(x_1, x_2, x_3) &= add(q(\pi_1(x_1, x_2, x_3),\, \pi_3(x_1, x_2, x_3)),\, \pi_2(x_1, x_2, x_3)) \end{aligned}$$

I think one of the trickier things about this is defining the empty sum, $\sum_{z<0} q(\vec{x}, z)$. There *are* no natural numbers, $z < 0$, so this expression, on its own, doesn't really give us any indication of how it should be defined. What does is the recursive part of our primitive recursive defintion. Let's look at $sum(\vec{x}, 2)$, just to see an easy example. The result we want is $sum(\vec{x}, 2) = q(\vec{x}, 0) + q(\vec{x}, 1)$. Using the recursive part of our definition, we get,

$$\begin{aligned} sum(\vec{x}, 2) &= sum(\vec{x}, 1) + q(\vec{x}, 1) \\ &= \Big(sum(\vec{x}, 0) + q(\vec{x}, 0)\Big) + q(\vec{x}, 1) \end{aligned}$$

At this point we need a value for $sum(\vec{x}, 0)$ that will make that line, $q(\vec{x}, 0) + q(\vec{x}, 1)$. Clearly, defining $sum(\vec{x}, 0) = 0$ is the only choice that will work.

(b) The bounded product is much the same, defined,

$$prod(\vec{x}, y) = \prod_{z<y} q(\vec{x}, z)$$

The short answer is simply,

$$\begin{aligned} prod(\vec{x}, 0) &= 1 \\ prod(\vec{x}, y+1) &= prod(\vec{x}, y) \cdot q(\vec{x}, y) \end{aligned}$$

Note that in this case we must define $prod(\vec{x}, 0) = 1$. If we make it 0, then $prod(\vec{x}, y)$ is *always* 0, no matter what values you use for $\vec{x}$ and y and what function you use for q. 1 is the multiplicative identity (multiplying it by any number has no effect), just like 0 is the additive identity (adding 0 to anything has no effect).

2 Bounded Quantification

Here's where we run into that problem of true/false conventions because quantified expressions are statements (with true/false) results. In each case here we'll be using a solution that involves the ordinary composition scheme (a primitive recursive composition of primitve recursive functions) rather than the primitive recursive scheme itself.

(a) If you use Convention 1 where 0 is "true" and 1 is "false," then you want,

$$S(\vec{x}, y) = \begin{cases} 0 & \text{if there is a } z < y \text{ such that } R(\vec{x}, z) = 0 \\ 1 & \text{if there isn't} \end{cases}$$

So we've got a bunch of 0s and/or 1s and we want the result, 0, if any one of those numbers is 0, and 1 otherwise? Well, that's easy! Just multiply them all together:

$$S(\vec{x}, y) = \prod_{z<y} R(\vec{x}, z)$$

If you use Convention 2, however, things get a bit trickier. The easy way out is to put this one aside and do T first. Then, with T shown to be primitive recursive we can get S like this:

$$S(\vec{x}, y) = \neg \forall z < y \ (\neg R(\vec{x}, z))$$

where the *"not"* predicate can be defined in several primitive recursive ways. One, for example, being like this:

$$\begin{aligned} \neg(0) &= 1 \\ \neg(n+1) &= 0 \end{aligned}$$

The other way to define bounded existential quantification (i.e. S) is to use our bounded summation from Question 1. We have a bunch of 0s and/or 1s and we want the result to be 0 if there are only 0s, and 1 if there is at least one 1 among the 0s. As a first attempt, we might try this:

$$S(\vec{x}, y) = \sum_{z<y} R(\vec{x}, z)$$

This is correct if all the $R(\vec{x}, z)$'s are zero, or if *exactly* one of them is 1. If more than one of them is 1, then this expression would evaluate to numbers other than 0 or 1 (which is not the behaviour we expect from a predicate). This is easy to rectify, though, using a primitive recursive function, $isPositive$, which can be defined as $isPositive(x) = \neg(\neg(x))$. Yeah, I know... It looks weird—especially if you're one of those crazy "Convention 1" cultists. If you don't like it, you can define it easily in any number of other ways. However you choose to get it, you can now define,

$$S(\vec{x}, y) = isPositive\left(\sum_{z<y} R(\vec{x}, z)\right)$$

(b) This part is essentially identical to (a), but backward. If you use Convention 1 where 0 is "true" and 1 is "false," then you want,

$$T(\vec{x}, y) = \begin{cases} 0 & \text{if R}(\vec{x}\text{,z})\text{=0 for every } z < y \\ 1 & \text{if there's even one obnoxious z who ruins it for everyone} \end{cases}$$

This is like the Convention 2 solution for part (a). We can either define S directly (as I showed initially) and then make,

$$T(\vec{x}, y) = \neg\exists z < y \; \neg R(\vec{x}, z)$$

...or do it like this:

$$T(\vec{x}, y) = isPositive\left(\sum_{z<y} R(\vec{x}, z)\right)$$

In a Convention 2 solution, we have a pile of 0s and/or 1s, and we want the result 0 if there is even one 0 among the 1s, and 1 otherwise. So,

$$T(\vec{x}, y) = \prod_{z<y} R(\vec{x}, z)$$

I know this is all kind of confusing (moreso than it should be since we have to be aware of the two conventions), but please email me—as always—if you're having trouble.

3 The *divides* Predicate: $x|y$

Here we have another predicate:

$$x|y = \begin{cases} true & \text{if } x \text{ is a factor of } y \\ false & \text{otherwise} \end{cases}$$

Note that x divides y (or x is a factor of y) iff y is a multiple of x. That is, if there is another number, z, such that $xz = y$. So, as a first attempt we might be tempted to try this:

$$x|y = ((\exists z)\ xz = y)$$

or, more formaly,

$$x|y = ((\exists z)\ eq(mul(x, z), y)$$

The only problem is that this is an *unbounded* quantification; it ranges over all natural numbers, z. And that's not a primitive recursive operation. The quantification itself is a good idea. We just need a bound for it. If we have that, then it's a bounded quantification, which we just showed *is* primitive recursive. So we need some number, w, that will allow us to express $x|y$ as $((\exists z < w)\ xz = y)$.

What should that w be?

Well, it's pretty obvious that if x, y, z are all natural numbers and $xz = y$ then $x, z \leq y$. In other words, you can't multiply two natural numbers and get a result that is LESS than one of the numbers in the product. Therefore, y is a perfect choice for our w, right? Well... almost. Since we defined bounded quantification using the expression, "$z < y$" instead of "$z \leq y$," we actully have to choose $w = y + 1$. This accounts for the case in which $x = 1$ ($1 \cdot y = y$ and $y < y + 1$, but it's not the case that $y < y$).

So our final answer is this:

$$x|y = ((\exists z < y + 1)\ xz = y)$$

Note that in this case, we don't have to worry about the $true/false$ convention chosen: whatever convention is used for quantification simply carries through.

4 $Prime(x)$

A natural number, x, is prime if it's greater than 1 and the only factors it has are 1 and x. In other words, there are no numbers (among *all* naturnal numbers), $z \notin \{1, x\}$ such that $z|x$. As in Question 3, this is a good initial attempt, but we must make that quantification bounded. We must find a y such that x is prime when we consider only numbers, $z < y$ (rather than $z \in \mathbb{N} - \{1, x\}$). The obvious choice is x, because (as we observed before) if $z > x$ then it follows that$\neg(z|x)$ automatically, so there's no need to check for factors any larger than x itself.

There are several ways to express all this, one of them being as follows:

$$Prime(x) = (1 < x)\ \wedge\ ((\forall z < x)\ \neg(z|x)\ \vee\ (z = 1))$$

This reads, "x is prime if $x > 1$ and for all $z < x$, z doesn't divide x (or it does but it's just 1 anyway, so it doesn't matter)."

The one thing missing here is that we haven't shown that the predicates $\wedge$ and $\vee$ are primitive recursive (the textbook does $<$ and $=$ for us, though). As with $\neg$, it's ok if you just assumed it for the assignment, but I'll show you how to do it just in case you'd like to see.

Unfortunately, the two conventions come back to haunt us. If we use Convention 1, we could define them like this (among several other ways):

$$x \wedge y = \neg((\neg x) \vee (\neg y))$$
$$x \vee y = mul(x, y)$$

We simply interchange them for Convention 2:

$$x \wedge y = mul(x, y)$$
$$x \vee y = \neg((\neg x) \wedge (\neg y))$$

I could have avoided the mess of the two conventions by using bounded quantification instead, but that requires all kinds of other annoying details.

Unlike the other questions, $Prime(x)$ is required to follow Convention 2 according to the assignment (if you want to be strict about it). Even if you used Convention 2 everywhere else, this is easy to remedy by simply putting $\neg$ in front of your whole definition. I haven't had a chance to mark this question, but I think that's probably too pedantic. If you've been using Convention 2 all along and don't switch just for the *Prime* predicate, I think that should be fine.

5 $G(x, y)$

The assignment spells this out fairly clearly for us: "the number of distinct primes dividing x which are less than or equal to y." Following either convention, we can use the *Case Construction* of Definition 4.7, together with the primitive recursive scheme to get this:

$$G(x, 0) = 0$$
$$G(x, y+1) = \begin{cases} G(x, y) + 1 & \text{if } Prime(y+1) \wedge ((y+1)|x) \\ G(x, y) & \text{otherwise} \end{cases}$$

If we follow Convention 2, we could do it like this:

$$G(x, y) = \sum_{z<y+1} Prime(z) \wedge (z|x)$$

If we follow Convention 1, it's almost as easy:

$$G(x, y) = \sum_{z<y+1} \neg(Prime(z) \wedge (z|x))$$

6 $F(x)$: The Number of Distinct Primes Dividing x

I think I'll just let the answer speak for itself:

$$F(x) = G(x, x)$$