(Optional) Recursion Theory

Recursion Theory

This **optional** reading gives you a brief introduction to some theory of recursion. This reading is much more mathematically heavy than the rest of this course. If you do not have the math background for this reading, you can skip over it and continue in the rest of the course with no problem. However, we recommend at least taking a look at it to see that there are important mathematical rules that govern when a recursive definition is well founded.

Recursion has a strong relationship to the mathematical proof technique of induction. If you need a quick refresher on *induction*, it is a technique that lets us prove $\forall x \in \mathbb{N}. p(x)$, where p(x) is some proposition about x. (*Translation*: we would like to prove that p(x) is true for all natural numbers, x.) Proof by induction starts by proving the base case, p(0). The proof then proceeds by showing the inductive case—either proving $\forall n \in \mathbb{N}. p(n) \rightarrow p(n+1)$ (weak induction) or $\forall n \in \mathbb{N}. (\forall x \in \mathbb{N} < n. p(x)) \rightarrow p(n+1)$ (strong induction).

The similarities between the two—having a base case, and a recursive/inductive case working on smaller values—are not a random coincidence. Our recursive function computes some answer (with certain properties) *for all* possible inputs. The recursive function works by assuming that the recursive case works correctly, and using that fact to make the current case work correctly—much like the inductive case assumes that the proposition holds for smaller numbers, and uses that fact to prove it for the "current" number. In fact, if we wanted to prove that a recursive function is correct, we would proceed by induction—and the structure of the inductive proof would mirror the structure of the recursive function.

Recursion is not just limited to the natural numbers (**unsigned ints**). We can recurse with arguments of different types, or on the natural numbers with a different ordering. In general, we can recurse on any set with a well-ordering (that is, a total ordering where every subset of the set has a least element) on it. We have seen this principle in action (though not explicitly discussed the theoretical implications) in our Fibonacci example. Here, our function operated on all integers (positive and negative). Proof by induction over all integers is a bit trickier, since they do not have a smallest value (thus base case) under their standard < ordering.

Our Fibonacci example was, however, theoretically sound, as we can well-order the integers by having a different ordering (which we will call $\sqsubseteq$). Specifically, for our Fibonacci function, we would want the ordering.

$$0 \sqsubseteq 1 \sqsubseteq -1 \sqsubseteq 2 \sqsubseteq -2 \sqsubseteq 3 \sqsubseteq -3 \sqsubseteq 4 \sqsubseteq -4 \sqsubseteq 5 \sqsubseteq -5 \sqsubseteq \ldots\ldots \sqsubseteq N \sqsubseteq -N \sqsubseteq (N+1) \sqsubseteq -(N+1) \sqsubseteq \ldots.$$

Now, whenever *fib*(*x*) recurses to *fib*(*y*), we can see that $y \sqsubseteq x$—it is "less" on this well-ordering. We can then prove the function correct by induction, using this same ordering.

We can use this principle to perform recursion (and correspondingly, induction) over types that are not directly numbers (of course, to a computer, everything is a number, as we learned earlier—which we will come back to in a second). As you continue to develop as a programmer, you will soon learn about data structures. Many of these data structures are recursively defined, such as lists and trees. This principle will be quite useful—you will want to recurse over the structure of the data. Fortunately, these structures have well-orderings, so we can recurse on them soundly.

The "everything is a number" principle actually appears in the mathematical theory related to well-ordered sets. We can take our well-ordered sets, and "number" the elements, then just consider the ordering of those numbers. Technically, we may need the *ordinal* numbers to perform this numbering, but if you are not familiar with them, you can just imagine the natural numbers as being sufficient.

All of this discussion of math is not just a theoretical curiosity. Instead, it gives us a formal way of understanding when our recursion is well-founded, versus when we may recurse infinitely. If we can construct a well-ordering of our parameters, and show that every time we recurse we are recursing with values that are "less than" (under this well-ordering) what was passed in, we can conclude that our recursion will always terminate—that is, we will never recurse infinitely. Observe that this property implies that we have a base case, as the well-ordering has a smallest element, so we are not allowed to recurse on that element (it is impossible to have anything smaller, so we cannot obey the rule of recursing on only smaller elements).

This property may sound both wondrous ("Great! I can guarantee my recursions will never be infinite.") and possibly daunting ("This sounds like a lot of math…My knowledge of well-ordered sets and the ordinals is kind of rusty."). Fear not—for many recursive tasks, this ordering is much simpler than it sounds. We can make a *measure function* which maps the parameter values to the naturals (or ordinals if needed), and convince ourselves that the measure decreases with every recursive call. For lists, we might measure them with their length, for trees we might measure them with their height, and so on. For most programming tasks, you will not actually need to formally prove any of this—you will just want to convince yourself that it is true to ensure you do not recurse infinitely.