Tutorial 12: Working with unknown functions

CS 135 Fall 2007

November 28-30, 2007

Today's tutorial (the last one!) covers accumulative recursion (lecture module 12). We'll be using accumulators to achieve tail recursion, which allows us to write some functions more naturally and more efficiently. You might want to think about what the invariant is for each of the accumulative functions we ask for below, and how you could use it to justify the correctness of the function.

1 ... And you thought you were done with foldr

The `foldr` function is (as we have seen) a very useful one which does some of the work for us in implementing structural recursion on lists. But what about structual recursion on natural numbers? Your task is to write the function `foldr-nat` with the following behavior:

```
foldr-nat: (num num -> num) num nat -> num
(foldr-nat f base n) => (f n (f (- n 1) (f (- n 2) ... (f 1 base))))
```

So, for example, we could compute the factorial of a number n by calling `(foldr-nat * 1 n)`. A simple implementation using structural recursion is as follows:

```
(define (foldr-nat1 f base n)
  (cond [(= n 1) (f 1 base)]
        [else (f n (foldr-nat1 f base (sub1 n)))]))
```

Now write a version of `foldr-nat` which uses an accumulator to achieve tail recursion.

2 Monotone increasing divergent functions

These questions deal with functions which are strictly increasing for all $x > 0$ and which diverge (i.e. grow to infinity). Some examples of functions like these are certain polynomials, exponential and logarithmic functions, square root, cube root, etc., and any combination of these.

2.1 Inverse floor function

If $f(x)$ is monotone increasing and divergent on the positive real numbers, then for any real number v, there must exist a positive integer u such that $f(u) > v$. The inverse floor function computes the greatest integer u such that $f(u) \leq v$ (assuming that $f(0) \leq v$), given f and v. Intuitively, we could just accomplish this by testing each of $f(1)$, $f(2)$, ..., until we find $f(u) > v$, and then returning $u-1$. Write a function `inverse-floor1` which uses accumulative recursion to accomplish this.

2.2 Anything bigger

By simply adding 1 to the result of `inverse-floor1`, we could easily find the least integer u such that $f(u) > v$. But suppose we don't need the least such value, but *any* integer u with $f(u) > v$. This could be found much, much more quickly by repeatedly doubling the test value for u (rather than adding 1 each time). Write a function `find-bigger-evaluation` which uses tail recursion to accomplish this.

2.3 Inverse floor revisited

We saw a few weeks ago how to use the binary search algorithm to quickly search for a number in a binary search tree. This same idea can be used to search over a finite interval of the natural numbers. The idea is, given a lower and an upper bound on the result, to test some value in the middle (roughly), and use this to reduce the size of the search interval by one half (roughly).

Use this notion of a binary search on the natural numbers, along with your function `find-bigger-evaluation`, to write a second version of `inverse-floor` which works much, much faster (of course I want you to again use accumulation and tail recursion).