Introduction to CS389r: *Recursion and Induction*

Concepts to Stimulate Discussion

Warren A. Hunt, Jr.
hunt@cs.utexas.edu

Computer Science Department
University of Texas
2317 Speedway, M/S D9500
Austin, TX 78712-0233

September, 2021

Computational Layers and Their Relationships

Computer users want simple mechanisms to indicate their intent.

Computers provide very, low-level operations; such operations are based on basic Boolean, logical operations, such as **AND**, **OR**, and bit-holding devices.

How do we assure that our intent is correctly translated to the language and hardware of computing?

Addition Definition

Can we define natural number (0, 1, 2, ...) addition through recursion? How would we know that our definition is correct?

```
(defun plus (x y)
  (if (zp x)
      y
    (plus (1- x) (1+ y))))
```

Addition Definition

Can we define natural number (0, 1, 2, ...) addition through recursion? How would we know that our definition is correct?

```
(defun plus (x y)
  (if (zp x)
      y
    (plus (1- x) (1+ y))))
```

Can we use proof (by induction) to establish properties about **plus**?

```
(defthm associativity-of-plus
  (implies (and (natp x) (natp y) (natp z))
           (equal (plus (plus x y) z)
                  (plus x (plus y z)))))

(defthm communitivity-of-plus
  (implies (and (natp x) (natp y))
           (equal (plus x y)
                  (plus y x))))
```

Even if we accept induction as a proof method, we still have to have *faith* that **plus** performs addition. Therefore, we must study our concepts carefully.

Warning: There are no known *correct* methods for defining original concepts.

Fibonacci Definition

Using our mechanical assistant, ACL2, we can define the Fibonacci function. ACL2 is a math checking program; but we don't know if it is itself correct.

```
(defun fib (x)
  (declare (xargs :guard (natp x)))
  (if (zp x)
      0
    (if (= x 1)
        1
      (+ (fib (- x 2))
         (fib (- x 1))))))
```

Our mechanical assistant can process this definition automatically, and it observes that **fib** always returns a natural number.

For this definition to be accepted, two termination conjectures must be checked – one for each inferior call to `fib`.

To carry out analysis of large systems, we use tools to help us. But, we must also have faith in the tools we develop on existing computers.

Can we arrange to certify the correctness of our computing systems?

Fibonacci Execution

The newly defined function, `fib`, can be executed immediately.

```
ACL2 !>(fib 10)
55
ACL2 !>(fib 20)
6765
```

However, when we evaluate the Fibonacci function with larger arguments, we must wait for the answer...

```
ACL2 !>(time$ (fib 50))
; (EV-REC *RETURN-LAST-ARG3* ...) took
; 191.77 seconds realtime, 188.34 seconds runtime
; (16 bytes allocated).
12586269025
```

Alternative Fibonacci Definition

ACL2 can accept an alternative execution (**:exec**) definition, when it can prove that the two definitions are equal.

```
(defun fib (x)
  (declare (xargs :guard (natp x)))
  (mbe
   :logic
   (if (zp x)
       0
     (if (= x 1)
         1
       (+ (fib (- x 2))
          (fib (- x 1)))))
   :exec
   (if (< x 10)
       (case x
         (0 0)
         (1 1)
         (2 1)
         (3 2)
         (4 3)
         (5 5)
         (6 8)
         (7 13)
         (8 21)
         (9 34))
     (+ (fib (- x 2))
        (fib (- x 1))))))
```

Execution

Defined functions may be executed.

```
ACL2!>(time$ (fib 50))
; (EV-REC *RETURN-LAST-ARG3* ...) took
; 3.65 seconds realtime, 3.59 seconds runtime
; (16 bytes allocated).
12586269025
```

Wow! Now, this function call takes less than 4 seconds!

By pre-computing the first ten values – the execution speeds up considerably.

Why? Because every execution can stop when `(< X 10)` – and just return the answer from the `CASE` statement.

The `:exec` version of `FIB` includes *memoized* results for when `(< X 10)`.

But, it will still be slow for yet larger inputs.

A New Fibonacci Function

By looking at the first ten values, we see that each entry is the sum of the preceding two entries. The first two values are given as 0 and 1.

Can we encode this relationship in a new function?

```
(defun f1 (fx-1 fx n-more)
  (declare (xargs :guard (and (natp fx-1)
                              (natp fx)
                              (natp n-more))))
  (if (zp n-more)
      fx
    (f1 fx              ;;  register: Current answer
        (+ fx-1 fx)     ;;  register: Next answer
        (1- n-more)     ;;  register: Steps remaining
        )))
```

Function F1 has three *registers*; the third argument indicates how many times to iterate this function.

The Completed New Fibonacci Function

We create a *wrapper* function with the first two values already computed.

```
(defun fib2 (x)
  (declare (xargs :guard (natp x)))
  (if (zp x)
      x
    (f1 0 1 (1- x))))
```

We run some tests to make sure that `FIB` and `FIB2` agree on some values.

```
ACL2 !>(equal (fib 10) (fib2 10))
T
ACL2 !>(equal (fib 30) (fib2 30))
T
```

Looks good! So, can we compute `(FIB2 100)` ? Yes, in an instant!

FIB2 is Equal to FIB

So, we would like to prove this conjecture:

```
(defthm fib2-is-fib
  (implies (natp x)
           (equal (fib2 x)
                  (fib x))))
```

This observation relates the logical definitions of `FIB` and `FIB2`.

- The theorem prover uses the logical definitions to compare these functions.
- For fast evaluation, we use the `FIB2` definition.

So, now how fast is `FIB`, or more to the point, how fast is `FIB2`?

Executing FIB2

Now, we can compute (FIB 1000) easily by computing (FIB2 1000).

```
ACL2 !>(time$ (integer-length (fib2 1000)))
; (EV-REC *RETURN-LAST-ARG3* ...) took
; 0.00 seconds realtime, 0.00 seconds runtime
; (61,200 bytes allocated).
694
```

If we want the complete answer, we can get it (which we split across 5 lines):

```
ACL2 >(FIB2 1000)
43466557686937456435688527675040625802564660517371...
78040248172908953655541794905189040387984007925516...
92959225930803226347752096896232398733224711616429...
96440906533187938298969649928516003704476137795166...
849228875
```