Lecture 6: Functional computation: Preamble to functional programming

K. V. S. Prasad

TDA384/DIT391 Principles of Concurrent Programming
Chalmers Univ. and Univ. of Gothenburg

16 September 2019

If you haven't seen functional programming (FP) before

For now, take *functional programming* as a *name*, what you call something.

> *What's in a name? That which we call a rose*
> *By any other name would smell as sweet;*
>
> — Juliet, in *Romeo and Juliet*, Act II, Scene 2

The *word* functional can mean (the Oxford Dictionary gives more):

1. Relating to the way in which something works or operates.
   - *there are important functional differences between left and right brain*
2. Designed to be practical and useful, rather than attractive.
   - *a small, functional bathroom*
3. In operation; working.
   - *the museum will be fully functional from the opening of the festival*
4. Mathematics: Relating to a variable quantity whose value depends upon one or more other variables.

Only the last applies. And even that tells you little.

Our first computations are *functional*! (in the FP sense)

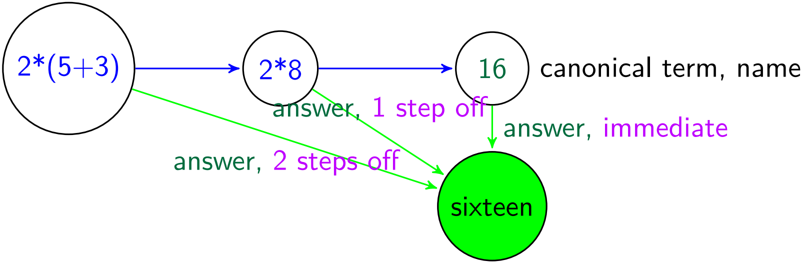
- $5 + 3 = 8$
  - where $5 + 3$ is the sum and 8 is the answer.
  - The answer is often a *name*, which we understand without further ado.
- But the symmetry in $5 + 3 = 8$ is partly misleading.
  - $8 = 5 + 3$ is true, but no child calls 8 the sum and $5 + 3$ the answer.
    - $5 + 3$ is not a name.
    - Also, 8 can also be $6+2$, etc.
- So $5 + 3 \rightarrow 8$ is better notation for this computation.
  - $5 + 3$ is the *expression* to be *evaluated* and 8 is a *canonical term*
    - A *canonical term* cannot be reduced further.
    - It is typically a *name*.
  - Evaluation may consist of several *reductions* $\rightarrow$, as in

    $$2*(5+3) \rightarrow 2*8 \rightarrow 16.$$

    Evaluation stops at a *canonical term*, 16.
    - When Europe learned the Indian decimal numerals, $16 \rightarrow \mathrm{XVI}$.
    - So what is canonical is a convention.

The answer doesn't change during evaluation

5+3 and 8 do have the same *value*, eight. So 5+3=8 is OK, even in FP.



Depending on study of arithmetic and language, we might

- see both 2*(5+3) and 16 as sixteen almost equally fast.
  - 2*(5+3), 2*8 and 16 have the same value even in a larger expression
- see (again, as adults) that
  - sixteen is ten+six, and that 8 is the name for the 7th. successor of 1.
- *Structured names* often become simple names.
  - A spårvagn is usually just seen as a tram, not as a track+carriage.
  - Mr. Johnson is not John's son (though someone was, at some point)

What *does* change in functional computation? *knowledge*

- Note that in the mini-computation $5 + 3 \rightarrow 8$
  - Neither 5 nor 3 "became" 8!
  - In fact, no data changed at all, not even the expression $5 + 3$.
- Then why bother compute?
  - What changed was our *knowledge*. We now know the answer, 8.
- Compare: "Do you see the girl in the blue blouse?"
  "Ah, you mean Alice."
  - Evaluation, the girl in the blue blouse → Alice
    a *description* → a *name* (a canonical value)

Function definitions and programs in FP

So far, we have only seen arithmetic evaluations, but we can illustrate FP by defining some arithmetic functions ourselves, though these are built-in to most practical FP systems.

Even in our toy system, we shall take *8* as the name for the 7th. successor of 1, or more conveniently the 8th. successor of 0. These integer names are taken to be defined as

```
    1 = succ(0)
    2 = succ(1), ... and so on
```

$+$ is the *infix* version of the function *add*, defined *recursively* by

```
add(0,y) = y
add(succ(x),y) = succ(add(x,y))
```

This definition and the built-in integer names constitute a *program* in FP.

Running FP programs

Running the program consists of giving it an expression to evaluate, using definitions in the program.

To evaluate an expression *pattern match* it against the given definitions.

So add(2,5) → add(succ(succ(0)),5) by the definition of 2
→ succ(add(succ(0),5)) by line 2 of add
→ succ(succ(add(0,5))) by line 2 of add
→ succ(succ(5)) by line 1 of add
→ succ(6) by the definition of 6
→ 7 by the definition of 7

So each reduction step replaces the left-hand-side (*lhs*) of some definition clause by the right-hand-side (*rhs*).

We can run the program with new input. Give it add(3,5), for example. Once we load a new program into an FP system, it will do a *read-eval-print* loop. (Read the new input, evaluate it, print the result).

Variables in FP

- The variables in the definition of add are *parameters*, as for functions in mathematics. In add(3,4), we have y=4 and y doesn't change for the duration of the evaluation, the *lifetime* of the variable.
- Names like 2 are defined in terms of previously known terms.
- Most FP languages allow "Let" as in algebra:
  - We are told "Mother gave me some apples. I gave 2 to Tim, and now have 3 left. How many did mother givve me?"
    - We go "Let x be the number of apples", so x-2=3", so x=5.
    - Notice that x here never changes. It was always 5, but we learn that only after solving the equation. The *scope* of the unknown x is only this problem. We can re-use x later.
- Variables that actually change while in the same scope and lifetime seem only to occur in imperative programming!

No commands in FP

- The only commands we've seen are *read, eval, print*. They are run-time system commands, not part of the program.
- No commands means no loops and no sequencing!
  - We use recursion instead of loops in FP.
  - We don't have sequencing either, instead we use "and" of timeless *statements* as in mathematics.
    - The term *statements* is a misnomer when used to describe imperative languages. Those are commands.
  - Sometimes we use if-then-else in FP instead of pattern matching.
    - The if-then used in imperative programming (else go on to the next command) makes no sense in FP.
- But Erlang *processes send* and *receive messages*, and *spawn* other processes. Those are not part of the FP subset of Erlang.

Copyright