

# PROGRAMMING LANGUAGES: FUNCTIONAL PROGRAMMING

## 4. SIMPLE PROGRAM CALCULATION

---

Shin-Cheng Mu

Spring 2022

National Taiwan University and Academia Sinica

## A QUICK REVIEW

- Functions are the basic building blocks. They may be passed as arguments, may return functions, and can be composed together.
- While one issues commands in an imperative language, in functional programming we specify values, and computers try to reduce the values to their normal forms.
- Formal reasoning: reasoning with the form (syntax) rather than the semantics. Let the symbols do the work!
- ‘Wholemeal’ programming: think of aggregate data as a whole, and process them as a whole.

## A QUICK REVIEW

- Once you describe the values as algebraic datatypes, most programs write themselves through structural recursion.
- Programs and their proofs are closely related. They share similar structure, by induction over input data.
- Properties of programs can be reasoned about in equations, just like high school algebra.

## SOME COMMENTS ON EFFICIENCY

---

## DATA REPRESENTATION

- So far we have (surprisingly) been talking about mathematics without much concern regarding efficiency. Time for a change.
- Take lists for example. Recall the definition:  
**data** *List a* = [] | *a* : *List a*.
- Our representation of lists is biased. The left most element can be fetched immediately.
  - Thus. (:), *head*, and *tail* are constant-time operations, while *init* and *last* takes linear-time.
- In most implementations, the list is represented as a linked-list.

## LIST CONCATENATION TAKES LINEAR TIME

- Recall ( $++$ ):

$$[] ++ ys =$$

$$(x : xs) ++ ys =$$

## LIST CONCATENATION TAKES LINEAR TIME

- Recall  $(++)$ :

$$[] ++ ys = ys$$

$$(x : xs) ++ ys = x : (xs ++ ys)$$

## LIST CONCATENATION TAKES LINEAR TIME

- Recall  $(++)$ :

$$\begin{aligned} [] ++ ys &= ys \\ (x : xs) ++ ys &= x : (xs ++ ys) \end{aligned}$$

- Consider  $[1, 2, 3] ++ [4, 5]$ :

$$\begin{aligned} &(1 : 2 : 3 : []) ++ (4 : 5 : []) \\ &= 1 : ((2 : 3 : []) ++ (4 : 5 : [])) \\ &= 1 : 2 : ((3 : []) ++ (4 : 5 : [])) \\ &= 1 : 2 : 3 : ([] ++ (4 : 5 : [])) \\ &= 1 : 2 : 3 : 4 : 5 : [] \end{aligned}$$

- $(++)$  runs in time proportional to the length of its left argument.

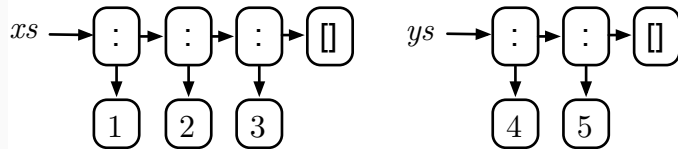


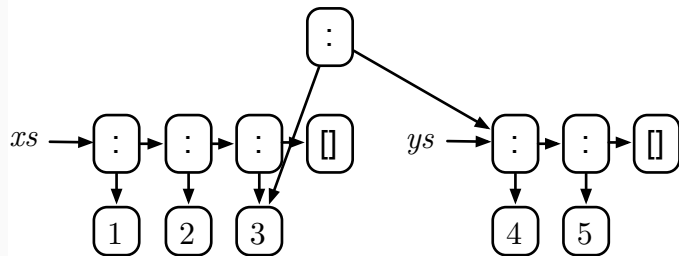
## FULL PERSISTENCY

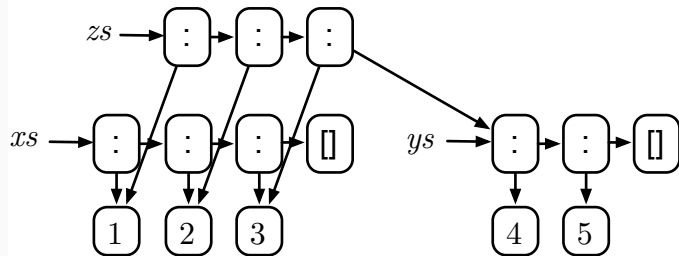
- Compound data structures, like simple values, are just values, and thus must be *fully persistent*.
- That is, in the following code:

```
let xs = [1, 2, 3]
    ys = [4, 5]
    zs = xs ++ ys
in ...body...
```

- The *body* may have access to all three values. Thus `++` cannot perform a destructive update.







## LINKED V.S. BLOCK DATA STRUCTURES

- Trees are usually represented in a similar manner, through links.
- Fully persistency is easier to achieve for such linked data structures.
- Accessing arbitrary elements, however, usually takes linear time.
- In imperative languages, constant-time random access is usually achieved by allocating lists (usually called arrays in this case) in a consecutive block of memory.

## LINKED V.S. BLOCK DATA STRUCTURES

- Consider the following code, where *xs* is an array (implemented as a block), and *ys* is like *xs*, apart from its 10th element:

```
let xs = [1..100]
    ys = update xs 10 20
in ...body...
```

- To allow access to both *xs* and *ys* in *body*, the *update* operation has to duplicate the entire array.
- Thus people have invented some smart data structure to do so, in around  $O(\log n)$  time.
- On the other hand, *update* may simply overwrite *xs* if we can somehow make sure that *nobody* other than *ys* uses *xs*.
- Both are advanced topics, however.

## ANOTHER LINEAR-TIME OPERATION

- Taking all but the last element of a list:

$init\ [x] =$

$init\ (x : xs) =$

- Consider  $init\ [1, 2, 3, 4]$ :

## ANOTHER LINEAR-TIME OPERATION

- Taking all but the last element of a list:

$$\textit{init} [x] = []$$

$$\textit{init} (x : xs) = x : \textit{init} xs$$

- Consider  $\textit{init} [1, 2, 3, 4]$ :



## ANOTHER LINEAR-TIME OPERATION

- Taking all but the last element of a list:

$$\text{init } [x] = []$$

$$\text{init } (x : xs) = x : \text{init } xs$$

- Consider  $\text{init } [1, 2, 3, 4]$ :

$$\text{init } (1 : 2 : 3 : 4 : [])$$

$$= 1 : \text{init } (2 : 3 : 4 : [])$$

$$= 1 : 2 : \text{init } (3 : 4 : [])$$

$$= 1 : 2 : 3 : \text{init } (4 : [])$$

$$= 1 : 2 : 3 : []$$

## SUM, MAP, ETC

- Functions like *sum*, *maximum*, etc. needs to traverse through the list once to produce a result. So their running time is definitely  $O(n)$ .
- If *f* takes time  $O(t)$ , *map f* takes time  $O(n \times t)$  to complete. Similarly with *filter p*.
  - In a lazy setting, *map f* produces its first result in  $O(t)$  time. We won't need lazy features for now, however.

# A FIRST TASTE OF PROGRAM CALCULATION

---

## SUM OF SQUARES

- Given a sequence  $a_1, a_2, \dots, a_n$ , compute  $a_1^2 + a_2^2 + \dots + a_n^2$ .  
Specification: *sumsq = sum · map square*.
- The spec. builds an intermediate list. Can we eliminate it?
- The input is either empty or not. When it is empty:

*sumsq []*

## SUM OF SQUARES

- Given a sequence  $a_1, a_2, \dots, a_n$ , compute  $a_1^2 + a_2^2 + \dots + a_n^2$ .  
Specification:  $sumsq = sum \cdot map\ square$ .
- The spec. builds an intermediate list. Can we eliminate it?
- The input is either empty or not. When it is empty:

$$\begin{aligned} &sumsq [] \\ = &\{ \text{definition of } sumsq \} \\ &(sum \cdot map\ square) [] \end{aligned}$$

## SUM OF SQUARES

- Given a sequence  $a_1, a_2, \dots, a_n$ , compute  $a_1^2 + a_2^2 + \dots + a_n^2$ .  
Specification:  $\text{sumsq} = \text{sum} \cdot \text{map square}$ .
- The spec. builds an intermediate list. Can we eliminate it?
- The input is either empty or not. When it is empty:

$$\begin{aligned} & \text{sumsq []} \\ = & \quad \{ \text{definition of } \text{sumsq} \} \\ & (\text{sum} \cdot \text{map square}) [] \\ = & \quad \{ \text{function composition} \} \\ & \text{sum} (\text{map square []}) \end{aligned}$$

## SUM OF SQUARES

- Given a sequence  $a_1, a_2, \dots, a_n$ , compute  $a_1^2 + a_2^2 + \dots + a_n^2$ .  
Specification:  $\text{sumsq} = \text{sum} \cdot \text{map square}$ .
- The spec. builds an intermediate list. Can we eliminate it?
- The input is either empty or not. When it is empty:

$$\begin{aligned} & \text{sumsq } [] \\ = & \quad \{ \text{definition of } \text{sumsq} \} \\ & (\text{sum} \cdot \text{map square}) [] \\ = & \quad \{ \text{function composition} \} \\ & \text{sum } (\text{map square } []) \\ = & \quad \{ \text{definition of } \text{map} \} \\ & \text{sum } [] \end{aligned}$$

## SUM OF SQUARES

- Given a sequence  $a_1, a_2, \dots, a_n$ , compute  $a_1^2 + a_2^2 + \dots + a_n^2$ .  
Specification:  $\text{sumsq} = \text{sum} \cdot \text{map square}$ .
- The spec. builds an intermediate list. Can we eliminate it?
- The input is either empty or not. When it is empty:

$$\begin{aligned} & \text{sumsq } [] \\ = & \quad \{ \text{definition of } \text{sumsq} \} \\ & (\text{sum} \cdot \text{map square}) [] \\ = & \quad \{ \text{function composition} \} \\ & \text{sum } (\text{map square } []) \\ = & \quad \{ \text{definition of } \text{map} \} \\ & \text{sum } [] \\ = & \quad \{ \text{definition of } \text{sum} \} \\ & 0 \end{aligned}$$



## SUM OF SQUARES, THE INDUCTIVE CASE

- Consider the case when the input is not empty:

*sumsq* ( $x : xs$ )

## SUM OF SQUARES, THE INDUCTIVE CASE

- Consider the case when the input is not empty:

$$\begin{aligned} & \text{sumsq } (x : xs) \\ = & \{ \text{definition of } \text{sumsq} \} \\ & \text{sum } (\text{map square } (x : xs)) \end{aligned}$$

## SUM OF SQUARES, THE INDUCTIVE CASE

- Consider the case when the input is not empty:

$$\begin{aligned} & \text{sumsq } (x : xs) \\ = & \{ \text{definition of } \text{sumsq} \} \\ & \text{sum } (\text{map square } (x : xs)) \\ = & \{ \text{definition of } \text{map} \} \\ & \text{sum } (\text{square } x : \text{map square } xs) \end{aligned}$$

## SUM OF SQUARES, THE INDUCTIVE CASE

- Consider the case when the input is not empty:

$$\begin{aligned} & \text{sumsq } (x : xs) \\ = & \{ \text{definition of } \text{sumsq} \} \\ & \text{sum } (\text{map square } (x : xs)) \\ = & \{ \text{definition of } \text{map} \} \\ & \text{sum } (\text{square } x : \text{map square } xs) \\ = & \{ \text{definition of } \text{sum} \} \\ & \text{square } x + \text{sum } (\text{map square } xs) \end{aligned}$$

## SUM OF SQUARES, THE INDUCTIVE CASE

- Consider the case when the input is not empty:

$$\begin{aligned} & \text{sumsq } (x : xs) \\ = & \{ \text{definition of } \text{sumsq} \} \\ & \text{sum } (\text{map square } (x : xs)) \\ = & \{ \text{definition of } \text{map} \} \\ & \text{sum } (\text{square } x : \text{map square } xs) \\ = & \{ \text{definition of } \text{sum} \} \\ & \text{square } x + \text{sum } (\text{map square } xs) \\ = & \{ \text{definition of } \text{sumsq} \} \\ & \text{square } x + \text{sumsq } xs \end{aligned}$$

## ALTERNATIVE DEFINITION FOR *sumsq*

- From *sumsq* = *sum* · *map square*, we have proved that

$$\text{sumsq } [] = 0$$

$$\text{sumsq } (x : xs) = \text{square } x + \text{sumsq } xs$$

- Equivalently, we have shown that *sum* · *map square* is a solution of

$$f [] = 0$$

$$f (x : xs) = \text{square } x + f xs$$

- However, the solution of the equations above is unique.
- Thus we can take it as another definition of *sumsq*.  
Denotationally it is the same function; operationally, it is (slightly) quicker.
- Exercise: try calculating an inductive definition of *count*.

## HOW FAR CAN WE GET?

- Specification of maximum segment sum:

*mss*            :: *List Int* → *Int*

*mss*            = *maximum* · *map sum* · *segments*

*segments* :: *List a* → *List (List a)*

*segments* = *concat* · *map inits* · *tails*

- From the specification we can calculate a linear time algorithm.

## REMARK: WHY FUNCTIONAL PROGRAMMING?

- Time to muse on the merits of functional programming.  
Why functional programming?
  - Algebraic datatype? List comprehension? Lazy evaluation? Garbage collection? These are just language features that can be migrated.
  - No side effects.<sup>1</sup> But why taking away a language feature?
- By being pure, we have a simpler semantics in which we are allowed to construct and reason about programs.
  - In an imperative language we do not even have
$$f\ 4 + f\ 4 = 2 \times f\ 4.$$
- Ease of reasoning. That's the main benefit we get.

---

<sup>1</sup>Unless introduced in a disciplined way.