Programming Languages: Functional Programming
4. Simple Program Calculation

Shin-Cheng Mu

Autumn 2023

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 $(++)$:

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

List Concatenation Takes Linear Time

- Recall $(+\!\!+)$:

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

List Concatenation Takes Linear Time

- Recall $(+\!\!+)$:

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

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

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

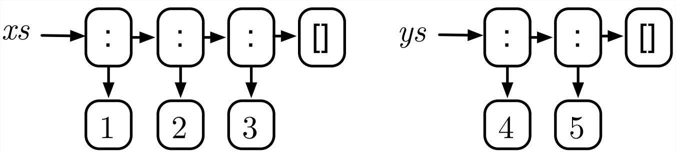
- $(+\!\!+)$ runs in time proportional to the length of its left argument.

Full Persistency

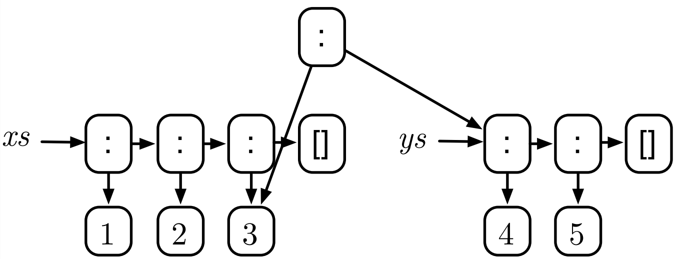
- Compound data structures, like simple values, are just values, and thus must be *fully persistent*.
- That is, in the following code:

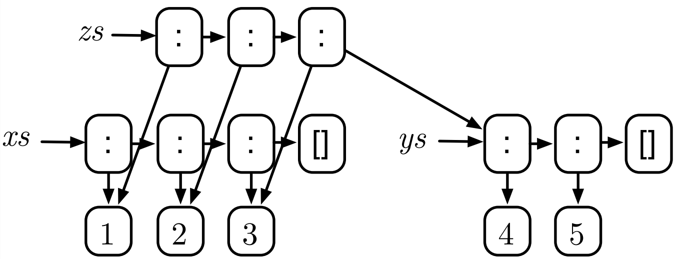
$$\begin{aligned}\textbf{let}\ & xs = [1, 2, 3] \\ & ys = [4, 5] \\ & zs = xs \mathbin{+\!\!+} ys \\ \textbf{in}\ & \ldots body \ldots\end{aligned}$$

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



xs
:
:
:
[]
1
2
3
ys
:
:
[]
4
5


:
xs
:
:
:
[]
ys
:
:
[]
1
2
3
4
5


zs
xs
ys
:
[]
1
2
3
4
5

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:

$$\begin{aligned}\textbf{let}\ & xs = [1..100] \\ & ys = \mathit{update}\ xs\ 10\ 20 \\ \textbf{in}\ & \ldots \mathit{body} \ldots\end{aligned}$$

- 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:

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

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

Another Linear-Time Operation

- Taking all but the last element of a list:

$$\begin{aligned} \mathit{init}\ [x] &= [\,] \\ \mathit{init}\ (x : xs) &= x : \mathit{init}\ xs \end{aligned}$$

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

Another Linear-Time Operation

- Taking all but the last element of a list:

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

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

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

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

$$\mathit{sumsq}\ [\,]$$

Sum of Squares

- Given a sequence $a_1,a_2,\ldots,a_n$, compute $a_1^2 + a_2^2 + \ldots + a_n^2$.
  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:

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

Sum of Squares

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

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

Sum of Squares

- Given a sequence a_1,a_2,...,a_n, compute $a_1^2 + a_2^2 + \ldots + a_n^2$. 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:

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

Sum of Squares

- Given a sequence $a_1,a_2,...,a_n$, compute $a_1^2 + a_2^2 + \ldots + a_n^2$. 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:

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

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:

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

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

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

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

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

Sum of Squares, the Inductive Case

- Consider the case when the input is not empty:

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

Alternative Definition for *sumsq*

- From $sumsq = sum \cdot map\ square$, we have proved that

$$
\begin{aligned}
sumsq\ [\,] &= 0 \\
sumsq\ (x : xs) &= square\ x + sumsq\ xs
\end{aligned}
$$

- Equivalently, we have shown that $sum \cdot map\ square$ is a solution of

$$
\begin{aligned}
f\ [\,] &= 0 \\
f\ (x : xs) &= square\ x + f\ xs
\end{aligned}
$$

- 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:

$$\begin{aligned} mss &:: List\ Int \rightarrow Int \\ mss &= maximum \cdot map\ sum \cdot segments \\ segments &:: List\ a \rightarrow List\ (List\ a) \\ segments &= concat \cdot map\ inits \cdot tails \end{aligned}$$

- 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.[1] 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.

[1] Unless introduced in a disciplined way.