Parallelism & Sequences

15-150 M21

Lecture 0719
19 July 2021

- ✓ Basics of Functional Computation
- ✓ Induction and Recursion
- ✓ Polymorphism & Higher-Order Functions
- ✓ Functional Control Flow
- ✓ The SML Modules System
- Applications & Connections

0 n-ary Parallelism

Task : Applying a constant-time function to n things

The *work* of this is always going to be at least $O(n)$. But how good can we do on the *span*?

What if we could do them *all* at the same time?

We've defined a signature `SEQUENCE`, containing an abstract type `'a seq` and a variety of operations on `seq`s.

0719.0 (SEQUENCE.sig)

```
signature SEQUENCE =
sig

  type 'a t
  type 'a seq = 'a t
```

- We've implemented `Seq :> SEQUENCE` such that the functions meets the bounds specified in the documentation

0719.1 (Seq.sml)

```
structure Seq :> SEQUENCE =
struct
```

- How's it implemented? Who cares?!
- By analogy to lists, we'll write sequences as

$$\langle 1,\ 3,\ \sim 7,\ 2,\ 6,\ 4\rangle : \texttt{int Seq.seq}$$

This is a mathematical notation, *not* SML syntax.

Key Point:

Sequences are *parallel data structures*

In the sequence library, we have:

```
map : ('a -> 'b) -> 'a seq -> 'b seq
```

which takes a function `f` and applies it to every element in a sequence `S`.

But it performs all these applications **in parallel**: if `f` has $O(1)$ span, then so too does `map f`, i.e. the parallel runtime (assuming arbitrarily-many processors) of `map f S` *does not grow as the length of S grows*.

Documentation:
The Sequence Reference

Live Coding:
Basic Sequence Functions

0719.2 (sandbox.sml)

```
fun rev S =
  let
    val n = Seq.length S   (* O(1) *)
  in
    (* O(n) work, O(1) span *)
    Seq.tabulate (fn i => Seq.nth S (n-i-1)) n
  end
```

```
fun append(S1,S2) =
  let
    (* O(1) *)
    val (m,n) = (Seq.length S1,Seq.length S2)

    (* O(1) *)
    fun f i =  case i<m of
                  true => Seq.nth S1 i
                | false => Seq.nth S2 (i-m)
  in
    (* O(n+m) work, O(1) span *)
    Seq.tabulate f (m+n)
  end
```

0719.4 (sandbox.sml)

```
val sum = Seq.reduce op+ 0
```

0719.5 (sandbox.sml)

```
infix |>
fun x |> f = f x

fun mappartial f S =
      S |> (Seq.map f)
        |> (Seq.filter Option.isSome)
        |> (Seq.map Option.valOf)
```

0719.6 (sandbox.sml)

```
fun double S =
  let
    fun twoseq x = Seq.append(
                     Seq.singleton x,
                     Seq.singleton x)
  in
    Seq.mapreduce
      twoseq
      (Seq.empty()) Seq.append
      S
  end
```

0719.7 (sandbox.sml)

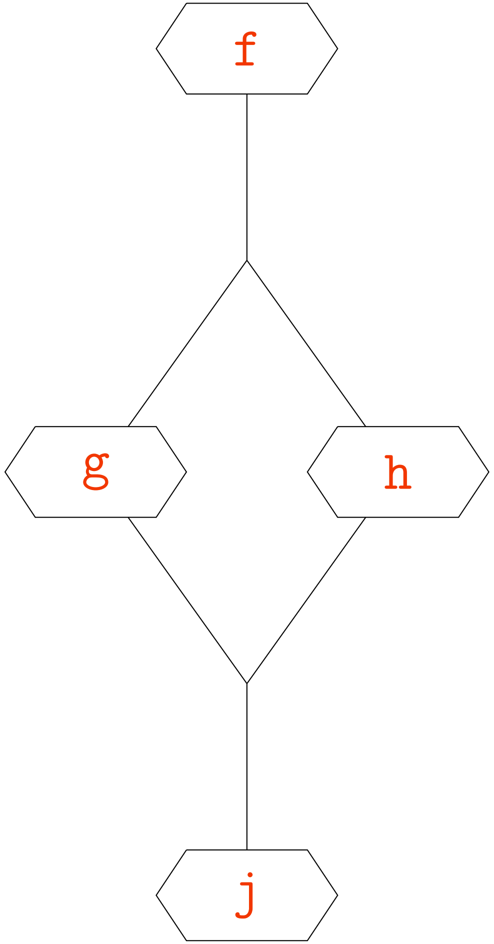
```
fun double' S =
  Seq.tabulate
    (fn i => Seq.nth S (i div 2))
    (2 * Seq.length(S))
```

5-minute break

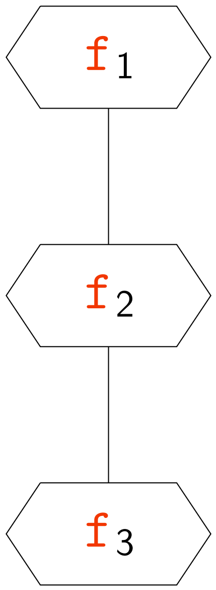
1 Cost Graphs

Idea

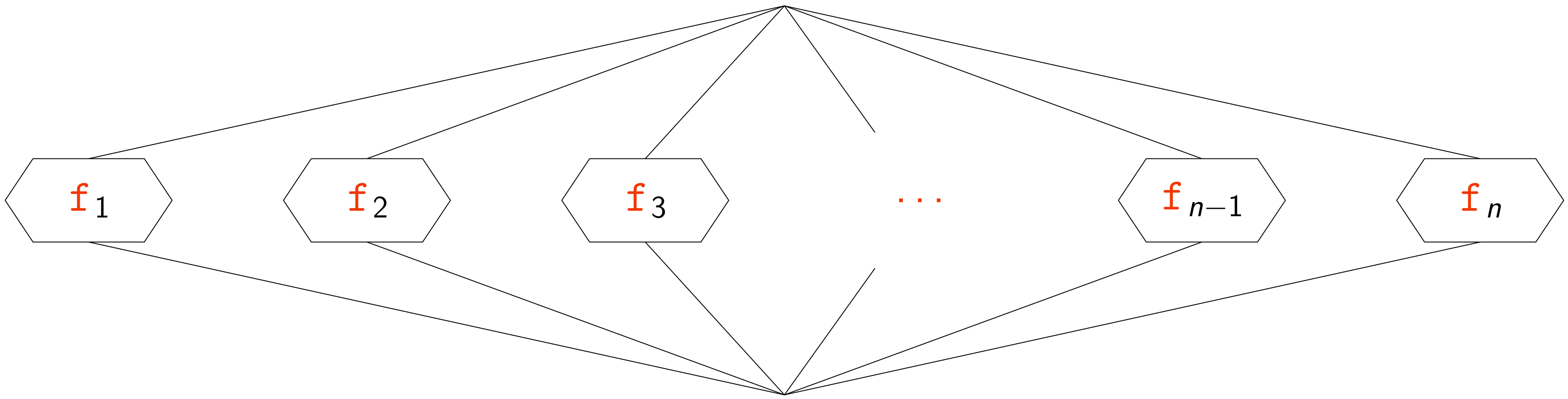
- Goal: establish a way to reason visually about the asymptotic runtime of algorithms
- Will represent algorithms as *directed acyclic graphs*
- Will reason about the runtime properties of the algorithm as properties of the graph.



Depicting Sequential Evaluation



$f_1; f_2; f_3$



$$\texttt{f}_1 \parallel \texttt{f}_2 \parallel \texttt{f}_3 \parallel \cdots \parallel \texttt{f}_{n-1} \parallel \texttt{f}_n$$

Key Idea: Can use cost graphs to determine the work & span of a function

Work: Sum of total cost in graph
Span: Height of longest path through graph

Demonstration:

Work & Span Analysis using cost graphs

Summary

- Prove equivalence of two structures ascribing to the same signature by relating values representing the same structure
- Carefully invariants and dutifully maintain them, using the opacity of the modules system to prevent the user from breaking them

Next Time

- Start *Applications* portion of course
- Parallel data structures and algorithms

Thank you!