15-150 Fall 2020
Stephen Brookes

LECTURE 1

Introduction to Functional Programming
Plan

This is a REMOTE class

- Lectures using Zoom **at class time** (then saved)
  - Please show up online, **on time**
    - *If in a different time zone, watch promptly.*
  - Study, work through examples, later.
- Homeworks and exams **online**
  - Do your own work!
Logistics

• Get to know course staff
  • email, request a zoom chat, …

• TAs will announce online office hours, …

• Email me about any concerns
  • Class size is large, so please be patient
  • Can also cc my assistant, Christina Contreras (cc8k@andrew)
Diversity

This class aims to give full and fair consideration to students from diverse backgrounds.

Diversity will be appreciated as a resource, a strength and a benefit.

Course staff aim to be respectful, and responsive to needs.

If any class meetings or deadlines conflict with religious events, let me know in advance so we can make arrangements.

Your suggestions are encouraged and appreciated.

Please let me know ways to improve the course.
Functional programming

Everything else is just dysfunctional programming!
The SML language

- **functional**
  - computation = expression evaluation

- **typed**
  - only well-typed expressions are evaluated

- **polymorphic**
  - well-typed expressions have a most general type

- **call-by-value**
  - function calls evaluate their argument
Advantages

• *functional*
  
easy to design and analyze

• *typed*
  
common errors caught early

• *polymorphic*
  
easy to re-use code

• *call-by-value*
  
predictable control flow
example

Standard ML of New Jersey [...]  

```ml
fun length [] = 0
| length (x::L) = 1 + length L;

- val length = fn - : 'a list -> int

length [1, 2, 4, 8];
- val it = 4 : int

length [true, false];
- val it = 2 : int

length 42;
- type error!
```
Features

• **referential transparency**
  - equivalent code is interchangeable

• **mathematical foundations**
  - use math to define equivalence
  - use logic to prove correctness, termination, …

• **functions are values**
  - can be used as data in lists, tuples, ...
  - can be an argument or result of other functions
Referential transparency

- The *type* of an expression depends only on the *types* of its sub-expressions
- The *value* of an expression depends only on the *values* of its sub-expressions

safe substitution, compositional reasoning
Equivalence

- Expressions of type `int` are equivalent if they evaluate to the same integer.
- Functions of type `int -> int` are equivalent if they map equivalent arguments to equivalent results.
- Expressions of type `int list` are equivalent if they evaluate to the same list of integers.

Equivalence is a form of semantic equality.
Equivalence

• $21 + 21$ is equivalent to $42$

• $[2, 4, 6]$ is equivalent to $[1+1, 2+2, 3+3]$

• $\text{fn } x \Rightarrow x+x$ is equivalent to $\text{fn } y \Rightarrow 2*y$

\[
21 + 21 = 42 \\
\text{fn } x \Rightarrow x+x = \text{fn } y \Rightarrow 2*y \\
(\text{fn } x \Rightarrow x+x) (21 + 21) = (\text{fn } y \Rightarrow 2*y) 42 = 84
\]

We use $=$ for equivalence
Don’t confuse with $=$ in ML
equality in ML

• ML has a built-in \( = \) operator

• Can use with expressions of simple types like \texttt{int}, \texttt{bool}, \texttt{int list}, … (called \textit{equality types})

• Will check if expressions evaluate to same value

\[(2 + 2) = 4 \quad \text{evaluates to} \quad \text{true}\]
Equivalence

- For every type \( t \) there is a notion of equivalence for expressions of that type
  - We usually just use \( = \)
  - When necessary we use \( =_t \)

Our examples so far illustrate:

\[ =_{\text{int}} \]
\[ =_{\text{int list}} \]
\[ =_{\text{int} \rightarrow \text{int}} \]
Compositionality

- Replacing a sub-expression of a program with an equivalent expression always gives an equivalent program.

The key to compositional reasoning about programs.
Parallelism

• Expression evaluation has no side-effects
  • can evaluate independent code in parallel
  • evaluation order has no effect on value

• Parallel evaluation may be faster than sequential
  Learn to exploit parallelism!
Principles

• **Expressions must be well-typed.**
  
  Well-typed expressions don't go wrong.

• **Every function needs a specification.**
  
  Well-specified programs are easier to understand.

• **Every specification needs a proof.**
  
  Well-proven programs do the right thing.

Those are my principles, and if you don't like them... well, I have others.
Principles

• Large programs should be modular.
  Well-interfaced code is easier to maintain.

• Data structures algorithms.
  Good choice of representation can lead to better code.

• Exploit parallelism.
  Parallel code may run faster.

• Strive for simplicity.
  Programs should be as simple as possible, but no simpler.
**sum**

```plaintext
fun sum [] = 0
| sum (x::L) = x + sum(L)
```

A recursive function declaration
using list patterns and integer arithmetic

- **sum** has type `int list -> int`
- **sum [1,2,3]** evaluates to **6**
- For all **n ≥ 0** and integer values **v₁, …, vₙ**

  \[
  \text{sum [v₁, …, vₙ]} = v₁ + … + vₙ
  \]
sum

fun sum [ ] = 0
| sum (x::L) = x + sum(L)

sum [1,2,3]
= 1 + sum [2,3]
= 1 + (2 + sum [3])
= 1 + (2 + (3 + sum [ ]))
= 1 + (2 + (3 + 0))
= 6

equational reasoning

[1,2,3] = 1 :: [2,3]
fun count [ ] = 0
  | count (r::R) = (sum r) + (count R)

• count has type (int list) list -> int
count

fun count [ ] = 0
   | count (r::R) = (sum r) + (count R)

- count has type (int list) list -> int
- count [[1,2,3], [1,2,3]] evaluates to 12
fun count [ ] = 0
  |
  | count (r::R) = (sum r) + (count R)

• count has type (int list) list -> int
• count [[[1,2,3], [1,2,3]]] evaluates to 12
• For all \( n \geq 0 \) and integer lists \( L_1, \ldots, L_n \)
  
  \[
  \text{count} [L_1, \ldots, L_n] = \text{sum} L_1 + \ldots + \text{sum} L_n
  \]
Since
\[
\text{sum} [1,2,3] = 6
\]
and
\[
\text{count} [[[1,2,3], [1,2,3]]] = \text{sum}[1,2,3] + \text{sum} [1,2,3]
\]
it follows that
\[
\text{count} [[[1,2,3], [1,2,3]]] = 6 + 6 = 12
\]
tail recursion

fun sum [ ] = 0
  | sum (x::L) = x + sum(L)

• The definition of sum is not tail-recursive

• Can define a tail recursive helper function sum’ that uses an integer accumulator

  sum : int list -> int

  sum’ : int list * int -> int

Q: This is a general technique. But why bother?
A: Sometimes tail recursion is more efficient.
sum'

\[
\text{fun} \; \text{sum}' \; ([ \; ], \; a) = a
\]
\[
| \; \text{sum}' \; (x::L, \; a) = \text{sum}' \; (L, \; x+a)
\]

- \text{sum}' has type \text{int list} * \text{int} \rightarrow \text{int}
- \text{sum}' ([1,2,3], 4) evaluates to 10
- For all integer lists \(L\) and integers \(a\), \(\text{sum}'(L, a) = \text{sum}(L) + a\)
Sum

fun sum' ([ ] , a) = a
  | sum' (x::L, a) = sum' (L, x+a)

fun Sum L = sum' (L, 0)

• Sum has type int list -> int
• Sum and sum are equivalent

For all integer lists L,
  Sum L = sum L
Hence...

\[
\textbf{fun} \ \text{count} [\ ] = 0 \\
| \quad \text{count} (r::R) = (\text{sum} \ r) + (\text{count} \ R)
\]

\[
\textbf{fun} \ \text{Count} [\ ] = 0 \\
| \quad \text{Count} (r::R) = (\text{Sum} \ r) + (\text{Count} \ R)
\]

- **Count** and **count** are equivalent because **Sum** and **sum** are equivalent.
**Evaluation**

```plaintext
fun sum [ ] = 0
  | sum (x::L) = x + sum(L)
```

\[
\begin{align*}
\text{sum (1::[2,3])} & \implies^* 1 + \text{sum [2,3]} \\
& \implies^* 1 + (2 + \text{sum [3]}) \\
& \implies^* 1 + (2 + (3 + \text{sum [ ]})) \\
& \implies^* 1 + (2 + (3 + 0)) \\
& \implies^* 1 + (2 + 3) \\
& \implies^* 1 + 5 \\
& \implies^* 6
\end{align*}
\]

“evaluates to, in finitely many steps”

pattern of recursive calls, order of arithmetic operations
Evaluation

count \([[[1,2,3], [1,2,3]]]\)

\[\implies^* \text{sum } [1,2,3] + \text{count } [[1,2,3]]\]

\[\implies^* 6 + \text{count } [[1,2,3]]\]

\[\implies^* 6 + (\text{sum } [1,2,3] + \text{count } [\ ] )\]

\[\implies^* 6 + (6 + \text{count } [\ ] )\]

\[\implies^* 6 + (6 + 0)\]

\[\implies^* 6 + 6\]

\[\implies^* 12\]
**Analysis**

(details later!)

<table>
<thead>
<tr>
<th>code fragment</th>
<th>evaluation time proportional to</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sum(L), Sum(L)</code></td>
<td>length of L</td>
</tr>
<tr>
<td><code>count(R), Count(R)</code></td>
<td>sum of lengths of lists in R</td>
</tr>
</tbody>
</table>

(tail recursion doesn’t help here!)

These functions do *sequential evaluation*...
parallelism

The combination order doesn’t affect result, so it’s safe to evaluate in parallel.

Suppose we have a function map such that

$$\text{map } f [x_1, \ldots, x_n] \mapsto \ast [f(x_1), \ldots, f(x_n)]$$

and we can evaluate the $f(x_i)$ in parallel…

+ is associative and commutative
parallel counting

\[
\textbf{fun} \ \text{parcount} \ R = \text{sum} \ (\text{map} \ \text{sum} \ R)
\]

\[
\text{parcount} \ [[1,2,3], \ [4,5], \ [6,7,8]]
\]

\[
\Rightarrow^* \ \text{sum} \ (\text{map} \ \text{sum} \ [[1,2,3], \ [4,5], \ [6,7,8]])
\]

\[
\Rightarrow^* \ \text{sum} \ [\text{sum} \ [1,2,3], \ \text{sum} \ [4,5], \ \text{sum} \ [6,7,8]]
\]

parallel evaluation of \text{sum}[1,2,3], \text{sum}[4,5] and \text{sum}[6,7,8]

\[
\Rightarrow^* \ \text{sum} \ [6,9,21]
\]

\[
\Rightarrow^* \ 36
\]
Analysis

• Let $R$ be a list of $k$ rows, and each row be a list of $m$ integers

• *If we have enough parallel processors, $\text{parcount } R$ takes time proportional to $k + m$*

  computes each row sum, in parallel
  then
  adds the row sums

Contrast: $\text{count } R$ takes time proportional to $k \times m$

With $m=20$ and $k=12$,

  $k + m$ is 32, almost an 8-fold speedup over $k \times m = 240$. 
work and span

We will introduce techniques for analysing

- work (sequential runtime)
- span (optimal parallel runtime)

(that’s how we did those runtime calculations)
Themes

• functional programming
• correctness, termination, and performance
• types, specifications and proofs
• evaluation, equivalence and referential transparency
• compositional reasoning
• exploiting parallelism
Objectives

• Write well-designed *functional programs*

• Write *specifications*, and prove correctness

• Techniques for analyzing runtime *(sequential and parallel)*

• Choose data structures wisely and exploit *parallelism* to achieve *efficiency*

• Design code using *modules* and *abstract types*, with clear interfaces
Summary

• Don’t worry if you don’t know SML syntax
• Don’t panic about so-far-undefined terminology
  • We will cover the details in lectures
• This introduction should help you appreciate the main ideas and see where we’re going…