15-150 Fall 2018

Stephen Brookes

LECTURE I
Tuesday, August 28
Functional programming

LISP • APL • FP • Scheme • KRC • Hope Miranda™ • Erlang • Curry • Gofer • Mercury Charity • Cayenne • Mondrian • Epigram Clean • Caml • Haskell Everything else is just dysfunctional programming!
The SML language

- **functional**
  
  *computation = evaluation*

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

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

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

- referential transparency
  equivalent code is interchangeable

- mathematical foundations
  use math & logic to prove correctness

- functions are values
  functions can be used as data in lists, tuples, ...
  and as 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 \texttt{int} are equivalent if they evaluate to the same integer.

- Functions of type \texttt{int \rightarrow int} are equivalent if they map equivalent arguments to equivalent results.

(Also called extensional equivalence)

- Expressions of type \texttt{int list} are equivalent if they evaluate to the same list of integers.

Equivalence is a form of semantic equality.
We use = for equivalence
Don’t confuse with = in ML
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.
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

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

• sum : int list -> int

• sum [1,2,3] = 6

• For all values L : int list,
  sum L = the sum of the integers in L

  sum [v₁, ..., vₙ] = v₁ + ... + vₙ
**fun** sum [ ] = 0
| sum (x::L) = x + sum(L)

\[
\begin{align*}
\text{sum} \ [1,2,3] &= 1 + \text{sum} \ [2,3] \\
&= 1 + (2 + \text{sum} \ [3]) \\
&= 1 + (2 + (3 + \text{sum} \ [ ])) \\
&= 1 + (2 + (3 + 0)) \\
&= 6.
\end{align*}
\]

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

• count : (int list) list -> int

• count [[1,2,3], [1,2,3]] = 12

• For all values R : (int list) list,
  count R = the sum of the ints in the lists of R.

  count [L₁, …, Lₙ] = sum L₁ + … + sum Lₙ
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 accumulator.

sum : int list -> int
sum' : int list * int -> int

Q: This is a general technique. But why bother?
A: Sometimes tail recursive version is more efficient.
sum’

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

• sum’ : int list * int -> int

• sum’ ([1,2,3], 4) = 10

• For all L:int list and a:int,
   sum’ (L, a) = sum(L)+a
Sum

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

\textbf{fun} \texttt{Sum} L = \texttt{sum'} (L, 0)

- \texttt{Sum} : int list -> int
- \texttt{Sum} and \texttt{sum} are extensionally equivalent
  
  For all L:int list, \texttt{Sum} L = \texttt{sum} L.
Hence...

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

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

• \textbf{Count} and \textbf{count} are \textit{extensionally equivalent}

For all \( R: \) (int list) list, \( \text{Count} \ R = \text{count} \ R. \)
**Evaluation**

fun \text{sum} \ [ \ ] = 0

| \text{sum} (x::L) = x + \text{sum}(L) |

\[
\text{sum} (1::[2,3]) = \Rightarrow^* 1 + \text{sum} [2,3] \\
\Rightarrow^* 1 + (2 + \text{sum} [3]) \\
\Rightarrow^* 1 + (2 + (3 + \text{sum} [ ])) \\
\Rightarrow^* 1 + (2 + (3 + 0)) \\
\Rightarrow^* 1 + (2 + 3) \\
\Rightarrow^* 1 + 5 \\
\Rightarrow^* 6
\]

\(\Rightarrow^*\) means “evaluates to, in finitely many steps”

pattern of recursive calls, order of arithmetic operations
Evaluation

count [[1,2,3], [1,2,3]]

=>* \( \text{sum} \ [1,2,3] + \text{count} \ [[1,2,3]] \)

=>* \( 6 + \text{count} \ [[1,2,3]] \)

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

=>* \( 6 + (6 + \text{count} \ [ \ ]) \)

=>* \( 6 + (6 + 0) \)

=>* \( 6 + 6 \)

=>* 12
**Analysis**

(details later!)

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

(tail recursion **doesn’t** help here!)

These functions do sequential evaluation…
Potential for parallelism

+ is associative and commutative

The order in which we do additions doesn’t affect the result, so it’s safe to evaluate in parallel
parallel application

map f [x₁, ..., xₙ] =>* [f(x₁), ..., f(xₙ)]

fun parcount R = sum (map sum R)

parcount [[[1,2,3], [4,5], [6,7,8]]]

=>* sum (map sum [[[1,2,3], [4,5], [6,7,8]]])

=>* sum [sum [1,2,3], sum [4,5], sum [6,7,8]]

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

=>* sum [6, 9, 21]

=>* 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, parcount $R$ takes time proportional to $k + m*

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

Recall: 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 $k \times m = 240$. 
Exploiting parallelism with \texttt{map} and \texttt{reduce}

\begin{verbatim}
fun parcount R = reduce (op +) (map sum R)
\end{verbatim}

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

\begin{align*}
&=>* \texttt{reduce (op +) (map sum }[[[1,2,3], [4,5], [6,7,8]]]) \\
&=>* \texttt{reduce (op +) [sum [1,2,3], sum [4,5], sum [6,7,8]]} \\
&=>* \texttt{reduce (op +) [6, 9, 21]} \\
&=>* 36
\end{align*}

For k rows of length m, time is proportional to $\log k + m$

With $m=20$, $k=12$, $\log_2 k + m$ is 23, a 10-fold speedup over 240.
Can we do any better?

• Try other ways to compute sums of row sums, using **map** and **reduce**

• Better ways to exploit parallelism?

How can we tell?
work and span

We will introduce techniques for analysing

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

(that’s how we did the runtime calculations earlier)

*reduce* (op +) \([v_1, \ldots, v_k] \) has
work \(O(k)\) and span \(O(\log k)\)

*sum* \([v_1, \ldots, v_k] \) has
work \(O(k)\) and span \(O(k)\)
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 be able to use rigorous techniques to prove correctness

• Learn techniques for analyzing *sequential* and *parallel runtime*

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

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