Functional programming

SML

Everything else is just dysfunctional programming!
The SML language

- **functional**
  
  \[ \text{computation} = \text{evaluation} \]

- **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 `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.
  (also called extensional equivalence)
- 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 $[1+1, 2+2, 3+3]$
- $\text{fn } x \mapsto x+x$ is equivalent to $\text{fn } y \mapsto 2*y$

$$21 + 21 = 42$$

$$\text{fn } x \mapsto x+x = \text{fn } y \mapsto 2*y$$

$$(\text{fn } x \mapsto x+x) (21 + 21) = (\text{fn } y \mapsto 2*y) 42 = 84$$

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**

\[
\textbf{fun } \text{sum} \ [ ] = 0 \\
\quad \text{sum} \ (x::L) = x + \text{sum}(L)
\]

- \text{sum} : \text{int list} -> \text{int}  
- \text{sum} \ [1,2,3] = 6  
- For all values \( L : \text{int list} \), \( \text{sum} \ L = \text{the sum of the integers in} \ L \)  
- \text{sum} \ [v_1, \ldots, v_n] = v_1 + \ldots + v_n
\[\text{sum}\]

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

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

\textit{equational reasoning}
count

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

• count : (int list) list -> int  type
• count [[1,2,3], [1,2,3]] = 12  example
• For all values R : (int list) list,
  count R = the sum of the ints in the lists of R.  spec

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

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.
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} \text{sum'} ([ ], a) = a
\mid \text{sum'} (x::L, a) = \text{sum'} (L, x+a)

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

\begin{itemize}
  \item \textbf{Sum} : \text{int list} -> \text{int}
  \item \textbf{Sum} and \textbf{sum} are \textit{extensionally equivalent}
\end{itemize}

For all \textbf{L: int list}, \textbf{Sum} \textbf{L} = \textbf{sum} \textbf{L}.
Hence...

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

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

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

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

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))

=>* 1 + (2 + 3)

=>* 1 + 5

=>* 6

pattern of recursive calls, order of arithmetic operations

means "evaluates to, in finitely many steps"

=>* means "evaluates to, in finitely many steps"
count [[1,2,3], [1,2,3]]

=>* sum [1,2,3] + count [[1,2,3]]

=>* 6 + count [[1,2,3]]

=>* 6 + (sum [1,2,3] + count [ ])

=>* 6 + (6 + 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>sum(L), Sum(L)</td>
<td>length of L</td>
</tr>
<tr>
<td>count(R), Count(R)</td>
<td>sum of lengths of lists in R</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

\[
\text{map } f \left[ x_1, \ldots, x_n \right] \Rightarrow^* \left[ f(x_1), \ldots, f(x_n) \right]
\]

fun parcount \( R \) = sum \( \left( \text{map sum } R \right) \)

parcount \( \left[ \left[ 1,2,3 \right], \left[ 4,5 \right], \left[ 6,7,8 \right] \right] \)

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

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

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

\Rightarrow^* 36

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

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

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

$k + m$ is 32, almost an 8-fold speedup over $k \times m = 240$. 
Exploiting parallelism with map and reduce

fun parcount R = reduce (op +) (map sum R)

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

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

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

=>* reduce (op +) [6, 9, 21]

=>* 36

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
Notes

• Lab tomorrow

• Homework I out tomorrow

Course policy: no cheating!

Do your own work!

See course website, piazza, for details.

Always attend class! Then read the notes!