Lecture 1
Tuesday, August 27
Functional programming

LISP • APL • FP • Scheme • KRC • Hope Miranda™ • Erlang • Curry • Gofer • Mercury Charity • Cayenne • Mondrian • Epigram • Clean • Caml • Haskell

Everything else is just *dys*functional 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.

• 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
Equivalence

• For every type $\mathsf{t}$ there is a notion of equivalence for expressions of that type

• We usually just use $\equiv$

• When necessary we use $\equiv_\mathsf{t}$

Our examples so far illustrate:

$\equiv_{\text{int}}$

$\equiv_{\text{int list}}$

$\equiv_{\text{int} \rightarrow \text{int}}$
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)

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

[1,2,3] = 1 :: [2,3]
count

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 [L_1, ..., L_n] = sum L_1 + ... + 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 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

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

fun Sum L = sum’ (L, 0)

• Sum : int list -> int

• Sum and sum are extensionally equivalent
  For all L:int list, Sum L = sum L.
Hence...

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

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

• **Count** and **count** are *extensionally equivalent*

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

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

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

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

\[
\Rightarrow^* 1 + (2 + \text{sum } [3])
\]

\[
\Rightarrow^* 1 + (2 + (3 + \text{sum } [ ]))
\]

\[
\Rightarrow^* 1 + (2 + (3 + 0))
\]

\[
\Rightarrow^* 1 + (2 + 3)
\]

\[
\Rightarrow^* 1 + 5
\]

\[
\Rightarrow^* 6
\]

\("evaluates to, in finitely many steps\)

pattern of recursive calls, order of arithmetic operations
Evaluation

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

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

\[
\begin{align*}
\text{parcount } & \left( [[1,2,3], [4,5], [6,7,8]] \right) \\
= & \left( \text{sum} \left( \text{map sum } [[1,2,3], [4,5], [6,7,8]] \right) \right) \\
= & \left( \text{sum} \left[ \text{sum } [1,2,3], \text{sum } [4,5], \text{sum } [6,7,8] \right] \right) \\
= & \left( \text{sum } [6, 9, 21] \right) \\
= & 36
\end{align*}
\]
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 \( k \cdot m \)

With \( m = 20 \) and \( k = 12 \),
\[ k + m = 32, \quad \text{almost an 8-fold speedup over } k \cdot m = 240. \]
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