15-150 Fall 2013
Lecture 1

Stephen Brookes

1 Themes

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

2 Objectives

- Learn to write functional programs
- Learn to write specifications, and to use rigorous techniques to prove program correctness
- Learn tools and techniques for analyzing the sequential and parallel running time of your programs
- Learn how to choose data structures, design functions, and exploit parallelism to improve efficiency
- Learn to structure your code using abstract types and modules, with clear and well designed interfaces

3 Functional Programming and ML

- ML is a *functional* programming language
  - computation = expression evaluation, to produce a value
- ML is a *typed* language
  - only well-typed expressions can be evaluated
- ML is a *polymorphically* typed language
  - an expression can be used at any instance of its *most general* type
- ML is a *statically* typed language
  - types are determined by a syntax-directed algorithm
- ML is a *call-by-value* language
  - function calls evaluate their arguments
- Functional programs are *referentially transparent*
  - expressions are "equivalent" when they produce equal results
  - compositional reasoning: substitution of "equals for equals"
- Functional programs are *mathematical* objects
  - can exploit mathematical techniques to prove correctness
- ML allows *recursive* definition of functions and datatypes
  - can use mathematical *induction* to analyze recursive code
- Evaluation order is largely irrelevant
  - can use parallel evaluation for independent code

4 Principles

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

- Every function needs a specification.
  Well-specified programs are easy to understand.

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

- Large programs should be designed as modules.
  Well-interfaced programs are easier to maintain and develop.

- Data structures algorithms.
  Sensible choice of data structure leads to better code.

- Think parallel, when feasible.
  Parallel programs may go faster. Use trees instead of lists if you can.

- Strive for simplicity.
  Programs should be as simple as possible, but no simpler.
  Simple code is usually easier to debug and easier to prove correct!

5 Today's lecture

A brief intro to functional programming, with forward references to the most important concepts, techniques and themes that will be developed later in the semester. Don't worry about the details (of syntax, or of semantics!). For now, just try to appreciate the elegance and simplicity!

Types

Expressions in ML are typed; only well-typed expressions can be evaluated. *This prevents many common programming errors!*

For today, we will only refer to the types `int` (integers), `int list` (lists of integers), `int list list` (lists of lists of integers) and function types like `int list -> int` (functions from integer lists to integers).

Functions

First, a (recursive) function for adding the integers in a list. Defined using cases: empty list, and non-empty list.

```
(* sum : int list -> int                                    *)
 fun sum [ ] = 0
  |  sum (x::L) = x + (sum L);

(* Specification: *)
(* For all integer lists L,                                 *)
(*    sum(L) = the sum of the integers in L.                *)
```

Note how similar this function definition is to a mathematical definition of a function. List notation: `[ ]` is the empty list, and `[2,3]` is the integer list with `2` as head and `[3]` as tail. Note that `1::[2,3]=[1,2,3]`. We can use mathematical notation and math-style equational reasoning to explain how this function works:

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

Next, a function for adding the integers in a list of integer lists. Again recursive, again by cases. Uses `sum` from above.

```
(* count : (int list) list -> int                              *)
fun count [ ] = 0
 |  count (r::R) = (sum r) + (count R);

(* Specification: *)
(* For all lists of integer lists R,                           *)
(*    count = the sum of the integers in the lists of R.       *)
```

Use equational reasoning to show that `count [[1], [2,3,4], [5]] = 15`.

Evaluation

Computation is evaluation. Expression evaluation stops when we reach a "value", such as an integer numeral or a list of integer numerals. Addition expressions evaluate from left to right.

We write `=>*` for "evaluates in a finite number of steps to".

```
(* 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                                           *)
(*  Numeral additions get done after recursive call returns.  *)
```

Similarly,

```
(* 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                   *)
```

Tail recursion

A recursive function definition is called *tail recursive* if in each clause of the function's definition any recursive call is the last thing that gets done. The definitions of `sum` and `count` are not tail recursive, because for `sum` or `count` on a non-empty list there is an addition operation to do after the recursive call. You can see this in the layout of our evaluation display above.

Here is an addition function for integer lists that uses an extra argument as an *accumulator* to hold an integer representing the result of the additions so far, and does an addition onto the accumulator value before making the recursive call. The definition of `sum'` is tail recursive.

```
(* sum' : int list * int -> int *)
fun sum' ([ ], a) = a
 |  sum' (x::L, a) = sum' (L, x+a);

(* Specification: for all L:int list, a:int, sum'(L,a) = sum(L)+a. *)
```

Ue equational reasoning to show that `sum'([1,2,3], 4) = 10.`

Later we will discuss tail recursion in more detail, in connection with efficient implementation; typically tail recursive functions need less stack space. Compare the shape of the evaluation diagram for `sum'` with that of `sum`, when applied to `[1,2,3]`. For the tail recursive version the shape doesn't grow wide then shrink back.

```
(* sum' ([1,2,3], 0) =>* sum' ([2,3], 1)      *)
(*                   =>* sum' ([3], 3)        *)
(*                   =>* sum' ([ ], 6)        *)
(*                   =>* 6                    *)
```

Using `sum'` we can define a function `Sum` that is "equivalent" to `sum`:

```
(* Sum : int list -> int *)
fun Sum L = sum' (L, 0);
```

Equivalence and Referential Transparency

It is easy to show by *induction* on the length of `L` that for all integer lists `L`, `Sum L = sum L`. Hence we say the functions `Sum` and `sum` are "extensionally equivalent", or just "equivalent", and we write `Sum = sum`.

Induction is a key technique for proving correctness of recursive functions, and for proving termination. We will make extensive use of induction, in various and general forms, throughout the course.

A key property of functional programming is *referential transparency*, sometimes paraphrased as the assertion that in a functional programming language it is safe to replace "equals by equals", or that "the value of an expression depends only on the values of its sub-expressions". Here we state a (slightly simplified, but more precise) version of this property.

First, for each type we define a notion of "equivalence":

```
(* Expressions of type int are "equivalent"        *)
(* if they evaluate to the same integer.           *)

(* Expressions of type int list are "equivalent"   *)
(* if they evaluate to the same integer list.      *)

(* Function expressions are "equivalent"           *)
(* if, when applied to "equivalent" arguments      *)
(* they produce "equivalent" results.              *)
```

Referential transparency is the property that: replacing a sub-expression by an "equivalent" sub-expression produces an "equivalent" expression.

WARNING: we limit attention here to "pure" functional programs. For the full ML language we need a more sophisticated version of referential transparency in which we take account of "effects" such as runtime errors and side-effects.

Examples

```
(* 21+21 and 42 are equivalent expressions of type int.    *)
(* The expressions (21+21)*3 and 42*3 have the same value. *)

(* The functions Sum and sum, of type int list -> int, are equivalent. *)
```

Now consider the following function definition:

```
(* Count : int list list -> int *)
fun Count [ ] = 0
 |  Count (r::R) = Sum r + Count R;
```

By referential transparency, it follows that `Count` is equivalent to `count`.

The above discussion is an example of *compositional reasoning* based on referential transparency. For the "pure" functional subset of ML, it is safe to replace an expression by an equivalent expression (of the same type). The result will be equivalent to the original program. We do this kind of compositional or substitutive reasoning all the time in math. You will get used to doing this with functional programs, too!

6 Parallelism

When we evaluate a functional program fragment, the order of evaluation of independent sub-expressions does not affect the final result.For example, the value of `(2+4)*(3+4)` is `42`, and it doesn't matter if we calculate the value using left-right evaluation, or right-left evaluation, or parallel evaluation of the two sub-expressions. In mathematical notation we would all agree that:

```
(2+4)*(3+4) = 6*(3+4) = 6*7 = 42
(2+4)*(3+4) = (2+4)*7 = 6*7 = 42
(2+4)*(3+4) = 6*7 = 42.
```

Addition on the integers is an associative operation, i.e. $x + (y + z) = (x + y) + z$, for all integers x, y, z. Hence it should be fairly obvious that in summing the integers in a collection of integer lists, we should be able to add up the entries in each list *independently* of all the other lists in the collection. Later in the course we will explore the use of data structures (such as trees, and sequences) that support parallel evaluation of collections of expressions. We will talk about ways to estimate the asymptotic runtime and space usage of a functional program, and we will see that parallelsm can often yield more efficient code.

If we have n integers in a single list and we add them using sequentially (like in `sum`), it obviously takes $O(n)$ time, provided we make the natural assumption that a single addition takes constant time. There's no obvious way to exploit parallelism if we just have a single list of integers like this and we can only access its items from the front of the list. So sequential summation of a list of length n has to do $O(n)$ "work"; further, even the smartest parallel implementation is hampered by the list structure and must also do $O(n)$ additions in a row, so we say that `sum` has "span" $O(n)$.

Similarly, if we have n integers held in a list of rows, each row being an integer list, using the `count` function to add all these integers (sequentially) is going to take time proportional to n, regardless of the lengths of the rows.

If we have n integers in a list of k rows, each row of length n/k, and an unlimited supply of processors, we could in principle[1] add the rows in parallel (using `sum` for each row, taking time proportional to n/k) and then add the row sums (again using `sum`, in time proportional to k). The total runtime for this algorithm would be $O(k + n/k)$. The work here is still $O(n)$, because you have to do n additions altogether.The "span" is the length of the critical path, which in this case is $O(k + n/k)$: No matter how smart you are at dividing the work among processors, you have to wait until the first phase is over (time n/k) before starting the second phase (another k units of time).

Finally, if we had n integers at the leaves of a balanced binary tree and an unlimited supply of processors, we could use a parallel divide-and-conquer strategy, starting at the leaves and working up towards the root in $O(log\ n)$ phases, each phase taking $O(1)$ time. The work here is $O(n)$ again, but the span is $O(log\ n)$.

Here we have been rather informal about the precise meanings of the terms "work" and "span". Later we will develop these ideas in more detail. Throughout the course we will pay attention to the work and span of the code that we develop. This should help you to become familiar with the potential benefits of parallelism, and you should develop an appreciation for whether and where you can safely exploit parallel evaluation and you should be able to figure out what asymptotic benefits this can bring.

[1]Assuming we have primitive functions `reduce` and `map` that use parallel evaluation, we might express this algorithm as `fun parcount R = reduce (op +) (map sum R)`. Such "higher-order" functions permit very concise and elegant program designs. We will return to this topic later!