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
  - safe to use parallel evaluation for independent code
  - strive to avoid unnecessary (re)-evaluation
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!*

Some of these (all?) may sound trite or obvious, but (we intend to demonstrate and convince you that) these are fundamental and important principles that can improve your programming skills.
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!

This write-up may not cover exactly what was shown in class. It serves to supplement the classroom slides and the material introduced during the lecture.

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.

\[
(* \text{sum} : \text{int list} \rightarrow \text{int} *)
\]
\[
\text{fun sum \[\] = 0}
\]
\[
| \text{sum (x::L) = x + (sum L)}
\]

(* 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:

\[
\text{sum [1,2,3]}
\]
\[
= 1 + \text{sum [2,3]}
\]
Next, a function for adding the integers in a list of integer lists. Again recursive, again by cases. Uses \texttt{sum} from above.

\begin{verbatim}
(* count : (int list) list -> int *)
fun count [] = 0
  | count (r::R) = (sum r) + (count R)
\end{verbatim}

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

Use equational reasoning to show that \( \text{count } [[1], [2,3,4], [5]] = 15 \).

**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 \( \Rightarrow^* \) for “evaluates in a finite number of steps to”.

\begin{verbatim}
(* 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. *)
\end{verbatim}

Similarly,

\begin{verbatim}
(* 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 *)
\end{verbatim}
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.

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

(* Specification: for all L:int list, a:int, \text{sum'}(L,a) = \text{sum}(L)+a. *)

Use equational reasoning to show that `\text{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.

\[
(* \text{sum'} ([1,2,3], 0) \Rightarrow \text{sum'} ([2,3], 1) \Rightarrow \text{sum'} ([3], 3) \Rightarrow \text{sum'} ([ ], 6) \Rightarrow 6 *)
\]

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

\[
(* \text{Sum} : \text{int list} \to \text{int} *)
\]
\[
\text{fun Sum L = sum'} (L, 0);
\]
Equivalence and Referential Transparency

It is easy to show by induction on the length of L that for all integer lists L, \( \text{Sum } L = \text{sum } L \). Hence we say the functions \( \text{Sum} \) and \( \text{sum} \) are “extensionally equivalent”, or just “equivalent”, and we write \( \text{Sum} = \text{sum} \).

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 parallelism 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 \texttt{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 \texttt{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 \texttt{sum} for each row, taking time proportional to \( n/k \)) and then add the row sums (again using \texttt{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 \texttt{reduce} and \texttt{map} that use parallel evaluation, we might express this algorithm as \texttt{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!
7 Self-test 1

These questions test your understanding of the lecture slides and these notes. We haven’t yet gone into the details behind the notation, so use your intuition and common sense to fill the gaps. Try to understand the reason behind each answer. (No credit, but it’s still a good idea to do this self-test!)

1. What values belong to the following types? In each case, describe the set of values and give an ML expression having the given type. For example, for the type \texttt{int * real} the values are pairs of an integer and a real number; for instance the ML expression \((0.12,42)\) is a (syntactic) value of this type.

   (a) \texttt{real * int}
   (b) \texttt{real -> real}
   (c) \texttt{int list -> int}
   (d) \texttt{int -> int list}

2. Which of the following assertions about equivalence, if any, are true?

   (a) \texttt{fn x:int => (21+21)} is equivalent to \texttt{fn x:int => 42}
   (b) \texttt{fn x:int => 42} is equivalent to \texttt{fn y:int => 42}
   (c) \((\texttt{fn x:int => (21+21)})(3+3) = 42\)
   (d) \((\texttt{fn x:int => (21+21)})(6) = (\texttt{fn y:int => 42})(3+3)\)

3. Write a recursive ML function \texttt{product} of type \texttt{int list -> int} such that for all integer lists \(L\), \texttt{product \ L} evaluates to the product of the items in \(L\). The product of the empty list of integers is defined to be 1.

4. Which, if any, of the following are true?

   (a) \texttt{product [1,2,3] = 6}
   (b) \texttt{product [1,2,3] = product [3,2,1]}
   (c) \texttt{product [0,42,42,42,42] = product [42,42,42,42,0]}
5. Which, if any, of the following are true?
   (a) \text{product} [1,2,3] \Rightarrow 6
   (b) \text{product} [1,2,3] \Rightarrow \text{product} [3,2,1]
   (c) \text{product} [0,42,42,42,42] \Rightarrow 0
   (d) \text{product} [42,42,42,42,0] \Rightarrow 0

6. Investigate the different forms of syntax allowed in ML for real numbers. As you have seen already, 0.12 is legal syntax. You can also about use E-notation, such as 2E10. Get familiar with the notation, what it means, and what happens when you add or multiply. This is really an open-ended exploration, but will help later when we solve some non-trivial problems concerning real numbers.

7. We can represent a point as a pair \((x,y)\) of type \texttt{real*real}. Two points whose x-coordinates are unequal determine a straight line: there is a unique line that passes through the points. A line has a slope and a y-intercept, both of which are real numbers. The line with slope \(a\) and y-intercept \(b\) is often written in math notation as “the line \(y = ax + b\)”. The line determined by \((x_1, y_1)\) and \((x_2, y_2)\) has slope \((y_2 - y_1)/(x_2 - x_1)\) and y-intercept \((y_2 x_1 - x_2 y_1)/(x_2 - x_1)\).

Write an ML function based on these ideas, by filling in this template:

\[
\text{fun line ((x1:real,y1:real), (x2:real,y2:real)) : (real -> real) =}
\]
\[
\text{let}
\]
\[
\text{.....}
\]
\[
\text{val a = ...}
\]
\[
\text{val b = ...}
\]
\[
\text{in}
\]
\[
\text{fn (x:real) => a * x + b}
\]
\[
\text{end}
\]

Use the first ..... to insert any useful declaration you need to avoid duplication of effort!

What are the (syntactic) values of the following expressions?
   (a) \text{line} ((0.0,1.0), (1.0,2.0))
8. To convert a temperature, a real number of degrees, from Fahrenheit to Celsius: subtract 32.0 then multiply by 5.0/9.0. To convert from Celsius to Fahrenheit, multiply by 9.0/5.0 then add 32.0. For example, 212.0 degrees Fahrenheit converts to 100.0 Celsius, and 100.0 degrees Celsius converts back to 212.0 degrees Fahrenheit. Write ML functions

```ml
  c_to_f : real -> real
  f_to_c : real -> real
```

that implement these conversions: e.g. \( c_{\text{to}\_f} \) goes from Celsius to Fahrenheit. What are the values of:

(i) \( f_{\text{to}\_c} \ 451.0 \)
(ii) \( c_{\text{to}\_f} \ (~273.15) \)
(iii) \( f_{\text{to}\_c} \ (~40.0) \)

9. The greatest common divisor (g.c.d.) of two positive integers \( x \) and \( y \) is defined to be the largest integer \( z \) such that \( z \) divides \( x \) and \( z \) divides \( y \), i.e. \( x \mod z = 0 \) and \( y \mod z = 0 \). Explain why every pair of positive integers has a g.c.d. (This is a math issue, not programming — yet.)

10. Here is a recursive function for computing the g.c.d. of positive integers:

```ml
  fun gcd(x:int, y:int):int =
    if x>y then gcd(x-y, y) else
      if y>x then gcd(x, y-x) else x
```

Answer these questions by first running ML (what does the interpreter say?) and then explaining the order in which the function calls happen. Example: for \( \text{gcd}(2,3) \) ML says the value is \( 1:\text{int} \), and

\( \text{gcd}(2,3) \) calls \( \text{gcd}(2,1) \), which calls \( \text{gcd}(1,1) \), returns 1.

(i) What is the value of \( \text{gcd}(24,15) \)?
(ii) What is the value of \( \text{gcd} ( \text{gcd}(24,15), \text{gcd}(2,3)) \)?
(iii) What about \( \text{gcd}(1000000,1) \) and \( \text{gcd} (~1,42) \) and \( \text{gcd}(0,42) \)?

Redefine \( \text{gcd} \) so that it returns 0 if either of the arguments is 0.
8 Comments

It’s a bit early to be preaching about style, as we haven’t really covered enough of the ML language for you to be able to make an informed and tasteful choice of syntax. And program style is a highly contentious topic, so that one person’s elegant one-liner can be viewed as horribly obscure by others. Nevertheless in later weeks we will return to some of the example programs introduced in the above Self Test, and see how we could have expressed the same algorithmic content in alternative syntax, sometimes with better taste.

For example, the gcd function was defined with a nested if-then-else:

if x>y then ... else if y>x ...

The else branch is itself an if-then-else expression. Some people get all upset when you do that, and admittedly it is very easy to write weirdly nested code that’s hard to unravel. (The gcd function is actually acceptable to me exactly as written, since the flow of control is so clear and simple.) But really the function needs to do one of three things, depending on the result of comparing the values of x and y (which will either be “less-than”, or “greater-than”, or equal). ML provides a built-in function

Int.compare : int * int -> order

where order is a built-in type. The values of type order are

LESS, EQUAL, and GREATER

and can be used to indicate the result of such an integer comparison. Here is a version of gcd written with a case expression rather than nested if-then-else:

fun gcd(x:int, y:int) : int =
case Int.compare(x, y) of
    | LESS => gcd(x, y-x)
    | GREATER => gcd(x-y, y)
    | EQUAL => x

Note also that layout on the page is worth paying attention to. We aligned the => symbols in the three clauses of the case expression, to make it easier to understand the syntactic structure. Do NOT write something as badly laid out as
fun gcd(x:int, y:int) : int = 
  case Int.compare(x, y) of LESS => gcd(x, y-x) 
  | GREATER => 
    gcd(x-y, y) | EQUAL => x 

A common stylistic mistake that also may signal a failure to think things through before writing your code is to use redundant syntactic structure: write something long or verbose where there’s a more succinct and clear way. For example:

fun both(x:bool, y:bool) :bool = 
  if x then (if y then true else false) else false

This is better expressed (equivalently!) as

fun both(x:bool, y:bool) :bool = 
  if x then y else false

and even more succinctly as

fun both(x:bool, y:bool) :bool = (x andalso y)

Never write if <boolean> then true else false, as it would be better to just use <boolean>. (Here <boolean> can be any expression of type bool.) Similarly

if <boolean> then false else true

is just a long-winded way of saying not <boolean>.

Another common detrimental plan is to introduce (and give names to) all sorts of helper functions. Sometimes it’s a good idea to use one, but it isn’t always. We will try to teach you to make sensible helpers and avoid irrelevant ones.

Style Advice

Study the syntax and style used in class and in lecture slides and notes. This is (usually!) chosen to illustrate and demonstrate taste and clarity of structure. Don’t use exotic pieces of ML syntax that you pick up on the internet or elsewhere, especially ones that have NOT yet been shown in class or labs. If you have questions, ask the professor or a TA! We’re always (usually) willing to pontificate.