15-150 Fall 2019

Thursday, 5 December

Stephen Brookes

REVIEW
Final Exam

Friday Dec 13, 1:00pm -- 4:00pm

(still be divided into multiple rooms)

Study
lecture notes
and slides!
Whole semester!
Look at self-tests!
What is SML?

• A functional programming language
  • computation = expression evaluation

• A polymorphically typed language
  • only well-typed expressions are evaluated
  • syntax-directed algorithm for most general type

• A call-by-value language
  • function calls evaluate their argument
Why ML?

More difficult to shoot your foot off

Can only shoot your foot in ML if

(a) the bullets and gun are well-typed

and

(b) the bullets are the right type for the gun

Moreover, if the gun is polymorphic
there’s a most general kind of bullet
Features

- Referential transparency
  - safe substitution for equivalent code
- Functions are extensionally equivalent if they map equal arguments to equal results
- Functional programs are mathematical objects
  - use math techniques to prove correctness
  - use induction to analyze recursive code
Referential transparency

• The type of an expression depends only on the types of its sub-expressions
• The value of a (pure) expression depends only on the values of its sub-expressions
• The value and effect of an (impure) expression depends only on the values and effects of its sub-expressions
For each type we define a notion of equality (or equivalence)

- Expressions of type `int` are equal if they evaluate to the same integer, or diverge
- Expressions of type `int list` are equal if they evaluate to the same list of integers, or diverge
- Function values are equivalent if they map equal arguments to equal results
Compositionality

- In any functional program, replacing an expression by an equal expression produces an equal program
Types and values

• A well-typed expression has a most general type
  • can be used at any instance of this type
• Every type has a set of (syntactic) values
  • evaluation respects types
    If \( e : t \) and \( e \Rightarrow^* v \), then \( v : t \)
    Also: if \( e \Rightarrow^* v \), then \( e \) is equal to \( v \)

\[ ML \]
computes
most general types
(based on syntax)

\[ ML \]
evaluates
well-typed expressions
to obtain (syntactic) values
Types and values

• Find the *most general type* of an expression
• Write well-typed code
• Use *evaluational* reasoning
• Use *equational* reasoning
• Understand connections
benefits

- A type is a very high-level specification
- wrong type implies incorrect code!

```plaintext
split : 'a list -> 'a list * 'a list
merge : int list * int list -> int list

fun sort [ ] = [ ]
  | sort L = let
      val (A,B) = split L
      in
      merge(sort A, sort B)
      end;

sort [2,1] = ???

sort : 'a list -> int list
```
benefits

- A type predicts value structure
- programmer can do pattern matching
- ML can check for exhaustive coverage

fun unstutter [ ] = [ ]
|  unstutter (x::y::L) = if x=y
|    then unstutter (y::L)
|    else x:: unstutter(y::L);

Warning: match nonexhaustive
nil => ...
x :: y :: L => ...

val unstutter = fn : "a list -> "a list

unstutter [1,2,2,3] = ???
Parallelism

- Expression evaluation has \textit{no side-effects}
- So evaluation order makes no difference to the value obtained
- Can evaluate \textit{independent code in parallel}
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.
Principles

• Large programs should be designed as modules. Well-interfaced code is easier to maintain.

• Data structures algorithms. Choice of data structure can lead to better code.

• Think parallel, when feasible. Parallel programs may go faster.

• Strive for simplicity As simple as possible, but no simpler
Themes

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

• Write functional programs

• Write specifications, and use rigorous techniques to prove correctness

• Learn techniques for analyzing sequential and parallel running time

• Choose data structures and exploit parallelism to improve efficiency

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

If you can't explain it simply, you don't understand it well enough.

Albert Einstein

(* REQUIRES ??? *)
(* ENSURES ??? *)
functions are values

- Some values are integers, lists of integers, ...
- Some values do functions, lazy lists, ...

functions can be used to represent graphs, lazy lists, dictionaries, ...
higher-order functions

- Functions can take functions as arguments
- Functions can return functions as results

List.map : ('a -> 'b) -> ('a list -> 'b list)
Seq.map : ('a -> 'b) -> ('a seq -> 'b seq)
foldl, foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b
reduce : ('a * 'a -> 'a) -> 'a -> 'a seq -> 'a
higher-order functions

• Allow us to give \textit{uniform} solutions to \textit{parameterized} problems

• Write once, use many ways

\begin{verbatim}
ins : ('a * 'a -> order) -> 'a list -> 'a list
foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

fun isort cmp = foldr (ins cmp) []
\end{verbatim}
higher-order functions

• Can represent patterns of computation

• Can express control flow, e.g. as a continuation

• Let you delay, manipulate, ignore a computation

  backtracking

  failure continuations

  lazy evaluation
recursion

• ML supports recursive function definition

\[
\text{fun } f(x:t_1):t_2 = e
\]

• Use induction to prove properties

\textit{simple} induction on \{n \mid n \geq 0\}

\textit{complete} induction on \{n \mid n \geq 0\}

\textit{structural} induction on lists, trees, ...

induction on length of lists or sequences,
size or depth of trees

\textit{all are forms of well-founded induction}
How to induct?

Use program structure as guide!

fun X n = if n = 1 then 0 else X(n-2)

fun Y n = if n < 1 then 0 else Y(n-2)

fun Z n = if n < 1 then 0 else Z(Z(n-2))

fun S (0, n) = n
  | S (m, 0) = m
  | S (m, n) = 1 + S(m, n-1)

fun G n = if n>100 then n-10 else G(G(n+11))

Recognize what to induct on, what can be proven, and when it’s impossible!
datatypes

• represent your problem, your way
• extend the type discipline, seamlessly
• can be recursive and parametric
  ’a list ’a tree

use invariants
  to enforce desired properties

binary trees ➔ binary search trees
                balanced trees
structural induction

• The set of values of a recursive datatype can be characterized *inductively*

• For every recursive datatype definition there is a *principle of structural induction*
  
  • Use to prove properties of values...

For all types $t$ and values $T : t$ tree, $\text{inord}(T)$ evaluates to a value...
parallelism

- Independent code can be evaluated in parallel
- Result doesn’t depend on evaluation order
  *determinism*
data, structured

• Sequences and trees may be better than lists

• Sequences support parallel application of mathematical transformations

   *map, reduce, mapreduce*
staging

- A *curried* function may do useful work before getting all of its arguments
- May improve efficiency by doing this work once, early, rather than in every function call
  - Choose argument order wisely
    - e.g. comparison-based sorting
modules

allow separate development

• signatures as interfaces

• structures as implementations

• can hide or reveal as much as you want, and no more

signature DICT =
sig
  structure Key : ORDER
type ’a dict
val empty : ’a dict
...
end

structure Bst : DICT =
struct
  structure Key = ...
datatype ’a dict = Empty | ...
val empty = Empty
...
end
modules

support abstract code design

• **abstract data types** with limited operations

  ordered sets, dictionaries, ...

  representation invariants

  support **localized reasoning**

• **type classes**: families of types with operations, not necessarily exclusive

```plaintext
signature ORD =
sig
  type t
  val compare : t * t -> order
end
```
functors

support separate development

- build implementations from implementations
- encapsulate common constructions
- allow code re-use

minimax

red-black trees
design

• functional (persistent) code is all about value
• imperative (ephemeral) code is all about effect
• Some code is benign in sequential contexts only
  \[ \text{step}(s,m); \text{F}(s); \text{undo}_\text{step}(s,m) \]
• Some code is benign even in parallel contexts
  \[ \text{Seq.map (reachable G)} \left( (x,y_1),..., (x,y_k) \right) \]
  graph reachability
design

• can design an **abstract data type** to be **persistent** or **ephemeral**

• **persistence** facilitates backtracking and supports parallelism without any problem

• **ephemeral** may be more efficient but may not work well in parallel

**ephemeral vs. persistent games**
work and span

• work = sequential complexity
• span = parallel complexity
• Can extract *recurrence relations* for W and S from a recursive function definition
  • W combines with *sum*
  • S combines with *max* for independent code, with *sum* for dependent
• Can *solve* or find *asymptotic* approximation
work and span

- Can use a cost graph for an expression evaluation
- Work = size
- Span = depth

abstracts away from scheduling details

Brent’s Theorem
An expression with work w and span s can be evaluated on a p-processor machine in time $O(\max(w/p, s))$. 
exceptions

• Useful for signalling and handling errors
• Extend the type discipline seamlessly
• Can be used to implement backtracking
• exceptions ≈ options
math

• Code is math

• Need math sophistication to get code right

• Use induction to prove correctness

• Use induction to analyze work and span
harder?

• A harder (more general) problem may be easier
  • easier to solve (recursively)
  • easier to prove correct (inductively)
• In proofs: choose good induction hypothesis
• In programs: pick good helper functions
  rev, helped by revver
• **complexity** of type inference

  most general types...  **priceless?**

  the type of an expression of length \( n \)
  can have length proportional to \( 2^{2^n} \)

• the **value restriction**

  polymorphic references are dangerous

  - val x = ref [ ];
  ...
  ...  Warning: type vars not generalized because of
  value restriction are instantiated to dummy types (\( X_1, \ldots \))
  val x = ref [ ] : ?.\( X_1 \) list ref
most general types

Try this...

```haskell
fun pair x y = fn z => z x y
val x1 = fn y => pair y y
val x2 = fn y => x1(x1 y)
val x3 = fn y => x2(x2 y)
val x4 = fn y => x3(x3 y)
```
val x4 = fn
  : 'a -> ((((((((((('a -> 'a -> 'b) -> 'b) -> 'b) -> ('b -> 'c) -> 'c) -> (((('a -> 'a -> 'b) -> 'b)
-> ('b -> 'c) -> 'c) -> 'd) -> 'd) -> 'd) -> 'e) -> 'e) -> 'f) -> 'f) -> 'g) -> 'g) -> 'h) -> 'h) -> 'i
-> 'yada -> 'yada -> 'yada
He who learns but does not think, is lost!
He who thinks but does not learn is in great danger.

Confucius

Do not worry about your difficulties in Mathematics. I can assure you mine are still greater.

Albert Einstein