Formal Methods

Lecture 1

Grading

Seminar activity :**-- 40%**

- Programming Assignment (group of 1 or 2 students)

Final exam: **-- 60%**

- Final Written Exam (about 2 hours, open books)

Course References

-The course is based on the following book(mainly Chapters 1-17):

Benjamin C. Pierce, Types and Programming Languages
(https://www.cis.upenn.edu/~bcpierce/tapl/)

- the lecture notes are based on Pierce slides from
http://www.seas.upenn.edu/~cis500/cis500-f06/

What is "software foundations"?

Software foundations (or "theory of programming languages") is the mathematical study of the meaning of programs.

The goal is finding ways to describe program behaviors that are both precise and abstract.

- precise so that we can use mathematical tools to formalize and check interesting properties
- abstract so that properties of interest can be discussed clearly, without getting bogged down in low-level details

Why study software foundations?

- To prove specific properties of particular programs (i.e., program verification)
  - Important in some domains (safety-critical systems, hardware design, security protocols, inner loops of key algorithms, ...), but still quite difficult and expensive
- To develop intuitions for *informal* reasoning about programs
- To prove general facts about all the programs in a given programming language (e.g., safety or isolation properties)
- To understand language features (and their interactions) deeply and develop principles for better language design *(PL is the "materials science" of computer science...)*

What you can expect to get out of the course

- A more sophisticated perspective on programs, programming languages, and the activity of programming
  - How to view programs and whole languages as formal, mathematical objects
  - How to make and prove rigorous claims about them
  - Detailed study of a range of basic language features
- Deep intuitions about key language properties such as type safety
- Powerful tools for language design, description, and analysis

Most software designers are language designers!

What this course is not

- An introduction to programming
- A course on functional programming (though we'll be doing some functional programming along the way)
- A course on compilers (you should already have basic concepts such as lexical analysis, parsing, abstract syntax, and scope under your belt)
- A comparative survey of many different programming languages and styles (boring!)

Approaches to Program Meaning

- *Denotational semantics* and *domain theory* view programs as simple mathematical objects, abstracting away their flow of control and concentrating on their input-output behavior.
- *Program logics* such as *Hoare logic* and *dependent type theories* focus on logical rules for reasoning about programs.
- *Operational semantics* describes program behaviors by means of *abstract machines*. This approach is somewhat lower-level than the others, but is extremely flexible.
- *Process calculi* focus on the communication and synchronization behaviors of complex concurrent systems.
- *Type systems* describe *approximations* of program behaviors, concentrating on the shapes of the values passed between different parts of the program.

Overview

In this course, we will concentrate on operational techniques and type systems.

- Part O: Functional Programming
  - A taste of OCaml
  - Functional programming style
  - Implementing programming languages

- Part I: Modelling programming languages
  - Syntax and operational semantics
  - Inductive proof techniques
  - The lambda-calculus
  - Syntactic sugar; fully abstract translations

Overview

- Part II: Type systems
  - Simple types
  - Type safety
  - References
  - Subtyping

A Tour of OCaml

OCaml and this course

The material in this course is mostly conceptual and mathematical. However:

- some of the ideas we will encounter are easier to grasp if you can "see them work"
- experimenting with small implementations of programming languages is an excellent way to deepen intuitions

For these purposes, we will use the OCaml language.

OCaml is a large and powerful language. For present purposes, though, we can concentrate just on the "core" of the language, ignoring most of its features. In particular, we will not need modules or objects.

Functional Programming

OCaml is a *functional* programming language — i.e., a language in which the *functional programming style* is the dominant idiom.

Other well-known functional languages include Lisp, Scheme, Haskell, and Standard ML.

Functional Programming

The functional style can be described as a combination of...

- *persistent* data structures (which, once built, are never changed)
- *recursion* as a primary control structure
- heavy use of *higher-order functions* (functions that take functions as arguments and/or return functions as results)

Imperative languages, by contrast, emphasize...

- *mutable* data structures
- *looping* rather than recursion
- *first-order* rather than higher-order programming (though many object-oriented design patterns involve higher-order idioms—e.g., Subscribe/Notify, Visitor, etc.)

Computing with Expressions

OCaml is an *expression language*. A program is an expression. The "meaning" of the program is the value of the expression.

```
# 16 + 18;;
- : int = 34

# 2*8 + 3*6;;
- : int = 34
```

The top level

OCaml provides both an interactive *top level* and a *compiler* that produces standard executable binaries. The top level provides a convenient way of experimenting with small programs.

The mode of interacting with the top level is typing in a series of expressions; OCaml *evaluates* them as they are typed and displays the results (and their types). In the interaction above, lines beginning with # are inputs, and lines beginning with − are the system's responses. Note that inputs are always terminated by a double semicolon.

Giving things names

The `let` construct gives a name to the result of an expression so that it can be used later.

```
# let inchesPerMile = 12*3*1760;;
val inchesPerMile : int = 63360

# let x = 1000000 / inchesPerMile;;
val x : int = 15
```

Functions

```
# let cube (x:int) = x*x*x;;
val cube : int -> int = <fun>

# cube 9;;
- : int = 729
```

We call `x` the *parameter* of the function `cube`; the expression `x*x*x` is its *body*. The expression `cube 9` is an *application* of `cube` to the *argument* `9`.
The *type* printed by OCaml, `int->int` (pronounced "`int` arrow `int`") indicates that `cube` is a function that should be applied to an integer argument and that returns an integer.
Note that OCaml responds to a function declaration by printing just `<fun>` as the function's "value."

Here is a function with two parameters:

```
# let sumsq (x:int) (y:int)  = x*x + y*y;;
val sumsq : int -> int -> int = <fun>

# sumsq 3 4;;
- : int = 25
```

The type printed for sumsq is int->int->int, indicating that it should be applied to two integer arguments and yields an integer as its result.
Note that the syntax for invoking function declarations in OCaml is slightly different from languages in the C/C++/Java family: we write cube 3 and sumsq 3 4 rather than cube(3) and sumsq(3,4).

The type boolean

There are only two values of type `boolean`: `true` and `false`. Comparison operations return boolean values.

```
# 1 = 2;;
- : bool = false
#
# 4 >= 3;;
- : bool = true
```

`not` is a unary operation on booleans.

```
# not (5 <= 10);;
- : bool = false
#
# not (2 = 2);;
- : bool = false
```

Conditional expressions

The result of the conditional expression `if B then E1 else E2` is either the result of `E1` or that of `E2`, depending on whether the result of `B` is `true` or `false`.

```
# if 3 < 4 then 7 else 100;;
- : int = 7
-
# if 3 < 4 then (3 + 3) else (10 * 10);;
- : int = 6
-
# if false then (3 + 3) else (10 * 10);;
- : int = 100

# if false then false else true;;
- : bool = true
```

Defining things inductively

In mathematics, things are often defined inductively by giving a "base case" and an "inductive case." For example, the sum of all integers from 0 to n or the product of all integers from 1 to n:

$$\begin{aligned} \text{sum}(0) &= 0 && \text{if } n \geq 1 \\ \text{sum}(n) &= n + \text{sum}(n-1) \end{aligned}$$

$$\begin{aligned} \text{fact}(1) &= 1 && \text{if } n \geq 2 \\ \text{fact}(n) &= n * \text{fact}(n-1) \end{aligned}$$

It is customary to extend the factorial to all non-negative integers by adopting the convention $\text{fact}(0) = 1$.

Recursive functions

We can translate inductive definitions directly into *recursive* functions.

```
# let rec sum(n:int) = if n = 0 then 0 else n + sum(n-1);;
val sum : int -> int = <fun>

# sum(6);;
- : int = 21

# let rec fact(n:int) = if n = 0 then 1 else n * fact(n-1)
val fact : int -> int = <fun>

# fact(6);;
- : int = 720
```

The `rec` after the `let` tells OCaml this is a recursive function — one that needs to refer to itself in its own body.

Making Change

Another example of recursion on integer arguments.
Suppose you are a bank and therefore have an “infinite” supply of coins (pennies, nickles, dimes, and quarters, and silver dollars), and you have to give a customer a certain sum. How many ways are there of doing this?
For example, there are 4 ways of making change for 12 cents:

12 pennies
1 nickle and 7 pennies
2 nickles and 2 pennies
1 dime and 2 pennies

We want to write a function change that, when applied to 12, returns 4.

Making Change – continued

To get started, let's consider a simplified variant of the problem where the bank only has one kind of coin: pennies.
In this case, there is only one way to make change for a given amount: pay the whole sum in pennies!

```
# (* No. of ways of paying a in pennies *)
  let rec changeP (a:int) = 1;;
```

That wasn't very hard.

Making Change – continued

Now suppose the bank has both nickels and pennies.
If a is less than 5 then we can only pay with pennies. If not, we can do one of two things:

- Pay in pennies; we already know how to do this.
- Pay with at least one nickel. The number of ways of doing this is the number of ways of making change (with nickels and pennies) for a-5.

```
# (* No. of ways of paying in pennies and nickels *)
  let rec changePN (a:int) =
    if a < 5 then changeP a
    else changeP a + changePN (a-5);;
```

Making Change – continued

Continuing the idea for dimes and quarters:

```
# (* ... pennies, nickels, dimes *)
  let rec changePND (a:int) =
    if a < 10 then changePN a
    else changePN a + changePND (a-10);;

# (* ... pennies, nickels, dimes, quarters *)
  let rec changePNDQ (a:int) =
    if a < 25 then changePND a
    else changePND a + changePNDQ (a-25);;
```

Finally:

```
# (* Pennies, nickels, dimes, quarters, dollars *)
  let rec change (a:int) =
    if a < 100 then changePNDQ a
    else changePNDQ a + change (a-100);;
```

Some tests:

```
# change 5;;
- : int = 2
-
# change 9;;
- : int = 2
-
# change 10;;
- : int = 4
```

...

```
# change 29;;
- : int = 13
# change 30;;
- : int = 18

# change 100;;
- : int = 243

# change 499;;
- : int = 33995
```

Lists

One handy structure for storing a collection of data values is a *list*. Lists are provided as a built-in type in OCaml and a number of other popular languages (e.g., Lisp, Scheme, and Prolog—but not, unfortunately, Java).
We can build a list in OCaml by writing out its elements, enclosed in square brackets and separated by semicolons.

```
# [1; 3; 2; 5];;
- : int list = [1; 3; 2; 5]
```

The type that OCaml prints for this list is pronounced either “integer list” or “list of integers”.
The empty list, written `[]`, is sometimes called “nil.”

The types of lists

We can build lists whose elements are drawn from any of the basic types (`int`, `bool`, etc.).

```
# ["cat"; "dog"; "gnu"];;
- : string list = ["cat"; "dog"; "gnu"]
-
# [true; true; false];;
- : bool list = [true; true; false]
```

We can also build lists of lists:

```
# [[1; 2]; [2; 3; 4]; [5]];;
- : int list list = [[1; 2]; [2; 3; 4]; [5]]
```

In fact, for *every* type `t`, we can build lists of type `t list`.

Lists are homogeneous

OCaml does not allow different types of elements to be mixed within the same list:

```
# [1; 2; "dog"];;
Characters 7-13:
This expression has type string list but is here used
with type int list
```

Constructing Lists

OCaml provides a number of built-in operations that return lists. The most basic one creates a new list by adding an element to the front of an existing list. It is written :: and pronounced “cons” (because it *cons*tructs lists).

```
# 1 :: [2; 3];;
- : int list = [1; 2; 3]

# let add123 (l: int list) = 1 :: 2 :: 3 :: l;;
val add123 : int list -> int list = <fun>

# add123 [5; 6; 7];;
- : int list = [1; 2; 3; 5; 6; 7]

# add123 [];;
- : int list = [1; 2; 3]
```

Some recursive functions that generate lists

```
# let rec repeat (k:int) (n:int) =(* A list of n copies of k
    if n = 0 then []
    else k :: repeat k (n-1);;

# repeat 7 12;;
- : int list = [7; 7; 7; 7; 7; 7; 7; 7; 7; 7; 7; 7]

# let rec fromTo (m:int) (n:int) = (* The numbers from m to n
    if n < m then []
    else m :: fromTo (m+1) n;;

# fromTo 9 18;;
- : int list = [9; 10; 11; 12; 13; 14; 15; 16; 17; 18]
```

Constructing Lists

Any list can be built by "consing" its elements together:

```
-# 1 :: 2 :: 3  :: 2 :: 1 :: [] ;;;
- : int list = [1; 2; 3; 2; 1]
```

In fact, [x_1; x_2; ...; x_n] is simply a shorthand for

$$x_1 :: x_2 :: \ldots :: x_n :: []$$

Note that, when we omit parentheses from an expression involving several uses of ::, we associate to the right—i.e., 1::2::3::[] means the same thing as 1::(2::(3::[])). By contrast, arithmetic operators like + and − associate to the left: 1−2−3−4 means ((1−2)−3)−4.

Taking Lists Apart

OCaml provides two basic operations for extracting the parts of a list.

- `List.hd` (pronounced "head") returns the first element of a list.

```
# List.hd [1; 2; 3];;
- : int = 1
-
-
```

- `List.tl` (pronounced "tail") returns everything *but* the first element.

```
# List.tl [1; 2; 3];;
- : int list = [2; 3]
```

More list examples

```
# List.tl (List.tl [1; 2; 3]);;
- : int list = [3]
-
# List.tl (List.tl (List.tl [1; 2; 3]));;
- : int list = []
-
# List.hd (List.tl (List.tl [1; 2; 3]));;
- : int = 3
```

More list examples

```
# List.hd [[5; 4]; [3; 2]];;
- : int list = [5; 4]
-
# List.hd (List.hd [[5; 4]; [3; 2]]);;
- : int = 5
-
# List.tl (List.hd [[5; 4]; [3; 2]]);;
- : int list = [4]
```

Modules – a brief digression

Like most programming languages, OCaml includes a mechanism for grouping collections of definitions into *modules*.

For example, the built-in module `List` provides the `List.hd` and `List.tl` functions (and many others). That is, the name `List.hd` really means "the function `hd` from the module `List`."

Recursion on lists

Lots of useful functions on lists can be written using recursion. Here's one that sums the elements of a list of numbers:

```
# let rec listSum (l:int list) =
    if l = [] then 0
    else List.hd l + listSum (List.tl l);;

# listSum [5; 4; 3; 2; 1];;
- : int = 15
```

Consing on the right

```
# let rec snoc (l: int list) (x: int) =
    if l = [] then x::[]
     else List.hd l  :: snoc(List.tl l) x;;

    -: val snoc : int list -> int -> int list = <fun>

# snoc [5; 4; 3; 2] 1;;
- : int list = [5; 4; 3; 2; 1]


let rec app (l: int list) (x: int list) =
    if l = [] then x
     else List.hd l  :: app(List.tl l) x;;
```

Reversing a list

We can use snoc to reverse a list:

```
# (* Reverses l -- inefficiently *)
  let rec rev (l: int list) =
    if l = [] then []
    else snoc (rev (List.tl l)) (List.hd l);;
val rev : int list -> int list = <fun>

# rev [1; 2; 3; 3; 4];;
- : int list = [4; 3; 3; 2; 1]
```

Why is this inefficient? How can we do better?

A better rev

```
# (* Adds the elements of l to res in reverse order *)
  let rec revaux (l: int list) (res: int list) =
    if l = [] then res
    else revaux (List.tl l) (List.hd l :: res);;
val revaux : int list -> int list -> int list = <fun>

# revaux [1; 2; 3] [4; 5; 6];;
- : int list = [3; 2; 1; 4; 5; 6]

# let rev (l: int list) = revaux l [];;
val rev : int list -> int list = <fun>
```

Tail recursion

The `revaux` function

```
let rec revaux (l: int list) (res: int list) =
  if l = [] then res
else revaux (List.tl l) (List.hd l :: res);;
```

has an interesting property: the *result* of the recursive call to `revaux` is also the result of the whole function. I.e., the recursive call is the *last* thing on its "control path" through the body of the function. (And the other possible control path does not involve a recursive call.)
Such functions are said to be *tail recursive*.

Tail recursion

It is usually fairly easy to rewrite a recursive function in tail-recursive style. For example, the usual factorial function is not tail recursive (because one multiplication remains to be done after the recursive call returns):

```
# let rec fact (n:int) =

    if n = 0 then 1
    else n * fact(n-1);;
```

We can transform it into a tail-recursive version by performing the multiplication *before* the recursive call and passing along a separate argument in which these multiplications "accumulate":

```
# let rec factaux (acc:int) (n:int) =

    if n = 0 then acc
    else factaux (acc*n) (n-1);;
# let fact (n:int) = factaux 1 n;;
```

Basic Pattern Matching

Recursive functions on lists tend to have a standard shape: we test whether the list is empty, and if it is not we do something involving the head element and the tail.

```
# let rec listSum (l:int list)=
    if l = [] then 0
    else List.hd l + listSum (List.tl l);;
```

OCaml provides a convenient *pattern-matching* construct that bundles the emptiness test and the extraction of the head and tail into a single syntactic form:

```
# let rec listSum (l: int list) =
    match l with
      [] -> 0
    | x::y -> x + listSum y;;
```

Pattern matching can be used with types other than lists. For example, here it is used on integers:

```
# let rec fact (n:int)
    =match n with
      0 -> 1
   |  _ -> n * fact(n-1);;
```

The _ pattern here is a *wildcard* that matches any value.

Complex Patterns

The basic elements (constants, variable binders, wildcards, `[]`, `::`, etc.) may be combined in arbitrarily complex ways in `match` expressions:

```
# let silly l =
    match l with
      [_;_;_] -> "three elements long"
    | _::x::y::_::_::rest ->
       if x>y then "foo" else "bar"
    | _ -> "dunno";;
val silly : int list -> string = <fun>
# silly [1;2;3];;
- : string = "three elements long"
# silly [1;2;3;4];;
- : string = "dunno"
# silly [1;2;3;4;5];;
- : string = "bar"
```

Type Inference

One pleasant feature of OCaml is a powerful *type inference* mechanism that allows the compiler to calculate the types of variables from the way in which they are used.

```
# let rec fact n =
 match n with
      0 -> 1
   | _ -> n * fact(n-1);;
   val fact : int -> int =<fun>
```

The compiler can tell that `fact` takes an integer argument because `n` is used as an argument to the integer `*` and `-` functions.

Similarly:

```
# let rec listSum l
    =match l with
      []    -> 0
   |  x::y  -> x + listSum y;;
val listSum : int list -> int = <fun>
```