COMP 302: In Class Notes

Ryan Ordille

April 2, 2012

1 03 February

Regular Expression Matching

Typical patterns:

- Singleton: matching a specific character
- Alternation: choice between two patterns
- Concatenation: succession of patterns
- Iteration: repeat a certain pattern (indefinite)

Regular expressions;

- 0 and 1 are regular expressions.
- If $a \in \Sigma$ where Σ is an alphabet, then a is a regular expression.
- If r_1 and r_2 are regular expressions, then $r_1 + r_2$ (choice) and $r_1 r_2$ (concatenation) are regular expressions.
- If r is a regular expression, then r^* is a regular expression (repetition).

Examples;

- `a(p*)l(e+y)` matches against “apple”, “apply”, “ale”
- `g(1+r)(e+a)y` matches against “grey”, “gray”, “gey”, “gay” (`1` means you can either have something there or not)
- `g(1+o)*gle` matches “google”, “ggle”

Our goal is to implement a regex matcher in SML.

Regular expression algorithm:

```
s matches 1
  iff s is empty
s matches a
  iff s = a
s matches r1+r2
  iff either s matches r1
          or s matches r2
s matches r1r2
  iff s = s1s2 and s1 matches r1
               and s2 matches r2
s matches r*
  iff either s is empty
          or s = s1s2 where s1 matches r
                and s2 matches r*
```

Remember that continuations tell us what to do once an initial segment of the input `char list` has been matched.

In SML, using continuations:

```
datatype regexp = Zero | One | Char of char | Plus of regexp * regexp
                | Times of regexp*regexp | Star of regexp

fun accept r s =

(* acc r s cont = bool *)
(* acc: regexp -> char list -> (char list -> bool) -> bool *)
(* ex: a(p*)l(e+y) on [a,p,p,l,e] *)
fun acc (Char c) [] cont = false
  | acc (Char c) (c1::s) cont =
        c = c1 andalso (cont s)
  | acc (Times (r1,r2)) s cont =
        acc r1 s (fn s2 => acc r2 s2 cont)
  | acc (Plus (r1,r2) s cont =
        acc r1 s cont orelse acc r2 s cont
  | acc One s cont = cont s
  | acc (Star r) s cont =
        (* remove (1*) case - s must shrink *)
        (cont s) orelse
      acc r s (fn s' => not (s = s') orelse
                        acc (Star r) s' cont)
```

2 6 February

2.1 Exceptions

We use exceptions to quit out from the runtime stack. We have already seen some built-in exceptions - for example, SML will throw a `Div` exception if you try to divide by zero (like `3 div 0`). Exceptions like this are used to abort a program safely whenever invalid input is given.

```
(* define the exception *)
exception Error of String;

fun fact n =
    let fun fact’ n =
        if n = 0 then 1
        else n * fact’(n-1)
    in
        if n < 0 then raise Error "Invalid Input"
        else fact’ n
    end;

(* non-exhaustive warning *)
fun head (h::_) = h;
(* uncaught exception Match *)
head [];
```

Sometimes we want to handle exceptions:

```
(* runFact: int -> unit *)
fun runFact n =
    let val r = fact n
    in print ("Factorial of " ^ Int.toString n ^ " is " ^ "Int.toString r)
    end
    handle Error msg => print ("Error: " ^ msg)
```

To sequentialize expressions, use the `;` operator. `exp1;exp2` first executes `exp1` then executes `exp2`. This is equivalent to `(fn x => exp2) exp1`. This will be expanded upon in the next lecture.

We can pattern match on error codes:

```
(* define the exception *)
exception Error of int;
```

```
fun fact n =
    let fun fact' n =
        if n = 0 then 1
    else n * fact'(n-1)
    in
        if n < 0 then raise Error 00
        else fact' n
    end;

fun runFact n =
    let val r = fact n
    in
       print ("Factorial of " ^ Int.toString n ^ " is " ^ "Int.toString r)
    end
    handle Error 00 => print ("invalid input")
         | Error 11 => print ("blah")
         | Error _ => print ("Something else")
```

Exceptions cannot be polymorphic, e.g. it cannot be of type `'a list` but can be of type `int list`.

Exceptions are usually pretty powerful in managing runtime stacks, but usually continuations are more powerful.

```
(* first arg: list of coins, second: what we want to get change for *)
change [50,25,10,5,2,1] 43;
(* result:  [25,10,5,2,1] *)

exception Change
(* change: int list -> int -> int list *)

fun change _ 0 = []
  | change [] amt = raise Change
  | change (coin::coins) amt =
          if coin > amt then change coins amt
          (*ignore this coin, look at other available coins*)
          else cont change (coin::coins) (amt - coin)
          (*could raise Change exception in following recursive steps *)
          handle Change => change coins amt
```

There are some situations where we cannot give change at all, but `change` does not handle these situations. Below, we handle this situation - `change` might not be able to do anything but raise `Change`, so this must be caught.

```
fun change_top coins amt =
    let val r = change coins amt
    in print ("Change:" ^ ListToString r)
    end
    handle Change => print "Sorry, can't give change."
```

```
change [5,2] 8
=>* if 5>8 then ... else (5::change[5,2] 3 handle Change change [2] 8)

change [2] 3
=>* 2::(change [2] 1 handle Change change [] 1)

change [2] 1 => change [] 1 => raise Change
(* goes to handle Change change [] 1 *)
(* then goes to handle Change change [2] 8, which succeeds *)
```

3 8 February

3.1 References (State)

Recall the binding/scope rules from the beginning of the class:

```
let
    val pi = 3.14
    val area = fn r => pi * r * r
val a2 = area 2.0 (*a2 = 12.56 *)
val pi = 6.0
in
    area (2.0) (* a2 = 12.56 *)
end;
```

So far, we have only seen bindings like the one above. For bindings, remember we have a variable name bound to some value. Today we will look at references, which are a form of mutable storage. References allow us to to imperative programming.

Commands:

- Initialize a cell in memory

`val r : int ref = ref 0` where `r` is the name of the cell and `0` is the content of the cell
`val s : int ref = ref 0` where `s` and `r` do not point to the same cell in memory

- Read what is stored in a cell
  `val x = !r` will read from location `r` the value `0`
  `r : int ref` and `!r : int`
- Write some value into a cell (i.e update the content)
  `r := 5 + 3` where `r : int ref` and `5+3 : int`
  Previous content of cell `r` is erased when we store `8`.
  Evaluating `r:=3` returns `unit` and as an effect updates the content of the cell with `3`.

`val x = !s + !r` binds `x` to `3`.

`val t = r` essentially makes two names for the same cell in memory. Calling `val y = !t` binds `y` to the value of `r`. This is called *aliasing*.

We can rewrite our beginning function:

```
let
   val pi = ref 3.14
   val area = fn r => !pi * r * r
   val a2 = area 2.0 (* a2 = 12.56 *)
   val _ = (pi := 6.0)
in
   area (2.0) (* 24.00*)
end
```

Now we can program mutable data structures like Linked Lists:

```
datatype 'a rlist = Empty | RCons of 'a * ('a rlist) ref;
val l1 = ref (RCons(4, ref Empty));
```

For `l1`, we now have a value `4` with a reference to some place in memory with an `Empty` list.

```
val l2 = ref (RCons(5,l1));
```

For `l2` we have a value `5` with a reference to `l1` defined above.

```
l1 := !l2;
```

The above will remove the value of `l1`, change it to `5` (`l2`'s value) and create a reference back to this element. Here, we've created a circular list.

```
type 'a reflist = ('a rlist) ref;
(* rapp: 'a reflist * 'a reflist -> unit *)
(* returns unit as all we're doing is updating space in memory *)
fun rapp (r1 as ref Empty, r2) = r1 := !r2
  | rapp (ref (RCons (x,t)), r2) = rapp t r2
```

Now we can check this with some examples:

```
val rlist1 = ref (RCons(1,ref Empty))
val rlist2 = ref (RCons(2,ref Empty))
rapp rlist1 rlist2
```

4 10 February

4.1 References and the environment diagram

4.1.1 References for modelling closures and objects

```
local
   val counter = ref 0
in
   (* tick: unit -> unit *)
   fun tick () = counter := !counter + 1
   (* reset: unit -> unit *)
   fun reset () = counter := 0
   (* read: unit -> int *)
   fun read () = !counter
end
```

We can use this to create a counter program:

```
fun newCounter () =
    let
   val counter = ref 0
   fun tick () = counter := !counter + 1
   fun reset () = counter := 0
   fun read () = !counter
    in
   {tick = tick; reset = reset; read = read}
end

val c1 = newCounter ();
val c2 = newCounter ();
```

```
#tick c1 (); (* increments c1's counter *)
#tick c2 ();
#tick c1 ();
#read c1 (); (* returns 2 *)
#read c2 (); (* returns 1 *)
```

In essence, we've created an object - every time we create a new counter, we create an instance of the object. We can now program in the object-oriented paradigm using ML (although the syntax isn't quite as built for OOP).

4.1.2 The Environment Diagram

`let val x = 5+3 in x+7 end:` will replace `x` by `8` then compute `8+7`.

So far, evaluation is driven by substitution. We substitute the value of `x` into the body. Unfortunately, the substitution model fails when we have references because substitutions cannot capture global effects.

We have three different kinds of bindings we'd like to track using the environment diagram. A binding is an association between a variable and a value.

1. `val x = 3+2` creates a "box" with the variable name `x` and its value `5`.
2. `val x = ref (8+2)` creates a box with the variable name and a location pointing to another box with the value `10`.
3. `val f = fn x => x + 3` creates a box with the function name and a location pointing to a box with the input and the function body, where this box points back to the original box.
   `val f = let val y = 8+2 in fn y => y + x end` adds another box (an extra step) with the local body.

5 13 February

5.1 Lazy Evaluation

So far, we've had an *eager evaluation* strategy. For example, `let x = e1 in e2 end` will evaluate `e1` to some value `v1` and bind `x` to the value `v1`, then evaluate `e2`. This is also known as a *call-by-value* strategy. Why should we evaluate `e1` if we never use it at all.

This is especially relevant with "harder" computations.

```
let val y = horribleComp(522)
in 3*2 end
```

With the call-by-value strategy, we always compute `horribleComp(322)`, even if we never use it.

We also have the *call-by-name* strategy. In the original example, it will bind `x` to the expression `e1`, then evaluate `e2`. However, if we use `x` multiple times in `e2`, we are evaluating `e1` multiple times.

There's a "best of both worlds" strategy we can also use – *call-by-need.* With this strategy in the original example, it will bind `x` to the expression `e1`, then evaluate `e2`, but memorize the result of evaluating `e1`.

Lazy evaluation is not only useful for saving computation time, but it also useful for evaluating infinite data structures. A stream of numbers online or interactive input/output from users would not be possible to deal with without infinite data structures.

Remember that continuations delay computation within functions. We can wrap functions around things we wish to delay.

```
datatype 'a susp = Susp of (unit -> 'a)

(* takes in a continuation and wraps it in a suspension to delay computation *)
(* delay: (unit -> 'a) -> 'a susp *)
fun delay c = Susp c
(* forces computation of inner function *)
fun force (Susp c) = c ()
```

Now we can use lazy evaluation with the `horribleComp` example.

```
(* original *)
let val x = horribleCom(522)
in x+x end

(* call-by-name model *)
let val x = Susp(fun () => horribleComp(522))
in force x + force x end

(* call-by-need*)
val memo = ref None
val x = Susp (fun () => case memo of None =>
    let val y = horribleComp(522) in memo := (*MISSED*) end
| Some y => y
```

```
(* infinite stream of 'a *)
datatype 'a stream' = Cons of 'a * 'a stream
withtype 'a stream = ('a stream') susp
(* stream' shows the first element, hides the rest, while stream hides all *)

(* create an infinite stream of, say, 1's *)
fun ones () = Susp (fun () => Cons (1, ones ()))
val o = ones() (* returns a Susp of a function *)

(* take: int -> 'a stream -> 'a list
   take': int -> 'a stream' -> 'a list
*)
fun take 0 s = []
  | take n s = take' n (force s)
and take' 0 s' = []
  | take' n (Cons(x,s)) = x::(take (n-1) s)

val l = take 5 (ones ()) (*returns [1,1,1,1,1]*)

(* numsFrom: int -> int stream *)
fun numsFrom n = Susp(fn () => Cons(n, numsFrom (n-1))

take 5 (numsFrom 0); (* returns [0,1,2,3,4] *)
```

We can compute a stream of Finonnaci numbers:

```
val fibStream =
    let
        fun fib a b = Cons(a, Susp(fn () => fib b, (a+b)))
    in
        Susp(fn () => fib 0 1)
    end

take 4 fibStream; (* [0,1,1,2] *)
```

6 15 February

6.1 Lazy programming continued

Recall from last class:

```
datatype 'a susp = Susp of (unit -> 'a)
datatype 'a stream' = Cons of 'a * 'a stream
withtype 'a stream = ('a stream') susp
```

Last class we saw how to create infinite streams of real numbers, natural numbers, etc..

```
(* shd: 'a stream -> 'a *)
fun shd (Susp s) = shd' (s ())
(* shd': 'a stream' -> 'a *)
and shd' (Cons (h,s)) = h
```

The first line is equivalent to `fun shd s = shd' (force s)`.

```
(* ltail: 'a stream -> 'a stream *)
fun ltail s = ltail'
(* ltail': 'a stream' -> 'a stream *)
and ltail' (Cons (h,s)) = s

(* smap: ('a -> 'b) -> 'a stream -> 'b stream *)
(* mapStr: 'a stream -> 'b stream *)
(* mapStr': 'a stream' -> 'b stream *)
fun smap f s =
let fun mapStr s = mapStr' (force s)
    and mapStr' (Cons (x, xs)) = Cons(f x, Susp (fn () => mapStr xs))
in mapStr s
end
```

```
(* addStreams: int stream * int stream -> int stream *)
fun addStreams (s1, s2) = addStreams' (force s1, force s2)
(* addStreams': int stream' * int stream' -> int stream *)
and addStreams' (Cons (x,xs), Cons (y,ys)) = Susp(fn () => Cons(x+y, addStreams (xs,ys)))

(* zipStreams: 'a stream * 'a stream -> 'a stream *)
fun zipStreams (s1, s2) = zipStream' (force s1, s2)
(* zipStreams': 'a stream' * ['a stream] -> 'a stream *)
and zipStreams' (Cons (x,xs), s2) =
    Susp(fn () => Cons (x, zipStreams (s2, xs)))
```

We don't need to force both streams for the `zipStreams` function to save work.

```
(* filter: ('a->bool)*'a stream -> 'a stream *)
fun filter (p, s) = filter' (p, foce s)
(* filter': ('a -> bool) * 'a stream' -> 'a stream *)
and filter' (p, (Cons(x,xs))) =
    if p x then
   Susp(fn () => Cons (x, filter (p,xs)))
    else
       filter (p, xs)
```

7 27 February

7.1 Midterm Review

Midterm exam – Wednesday, 29 February in Leacock 26.

Crib sheet allowed – one page, back and front.

Three questions – proofs, programs, covering all material up until the break.

No continuation, lazy evaluation, exceptions, or environment diagram questions.

7.1.1 Example 1: Proofs

```
fun sum [] = 0
  | sum (h::t) = h + sum t

fun sum_tl [] acc = acc
  | sum_tl (h::t) acc = sum_tl t (h + acc)
```

For the above code, we wish to prove that `sum l = sum_tl l 0`. We can do this using structural induction on `l`.

The base case is trivial. We'll start with the step case where `l = h::t`. Our induction hypothesis states that `sum t = sum_tl t 0`. We'll need to show that `sum (h::t) = sum_tl (h::t) 0`.

```
sum (h::t)
=> h + sum t

sum_tl (h::t) 0
=> sum_tl t (h+0)
=> sum_tl t h
```

This attempt will not work, since we want to use the IH. We'll need to generalize the theorem:

Lemma: For all lists `t` and for all accumulators `acc`, `sum t + acc = sum_tl t acc` is true. We'll also need to prove this using structural induction on `t`.

Base case: where `t = []`:

```
sum [] + acc
=> 0 + acc
=> acc

sum_tl [] acc
=> acc
```

Both sides are equal, so our base case checks out.

Step case: where `t = h::t'`:

```
IH: for all acc', sum t' + acc' = sum_tl t' acc'

sum (h::t') + acc'
=> (h + sum t') + acc' [by program]
=>* sum t' + (h + acc') [by associativity and commutativity]

sum_tl (h::t') acc'
=> sum_tl t' (h + acc') [by program]
```

By the induction hypothesis using `(h + acc)` for `acc'`, we know these are equal.

Now that we've proved the lemma, we need to prove the main theorem.

By the lemma, using `l` for `t` and `0` for `acc`:

```
sum l + 0
=> sum l
```

7.1.2 Example 2: Rewriting library functions

We have a library function:

```
tabulate f n returns  [f0, f1, ..., fn]
```

We want to write this in a tail-recursive manner.

```
fun tabulate f 0 acc = (f 0)::acc
  | tabulate f n acc = tabulate f (n-1) ((f n)::acc)
```

Another example:

```
foldr f b [x1, ..., xn] returns f (x1, ..., f (xn, b))
```

Now, we want to write a list append function using `foldr`.

```
fun append l1 f2 =
    foldr (fn (x,r) => x::r) l2 l1
```

We can also write a `filter` function:

```
fun filter p l =
   foldr (fn (x,r) => if (p x) then x::r else r) [] l
```

```
(* all: (’a -> bool) -> ’a list -> bool *)
fun all p [] = true
  | all p (h::t) =
        p h andalso all p t
```

8 02 March - Post-Midterm

8.1 Midterm review

8.1.1 Question 1

Dot product of two vectors $a\dot{b} = \sum_{i=1}^{n} a_1 \times b_1$

Use `pair_foldr` $= f(x_n, y_n, f(x_{n-1}, y_{n-1}, \ldots, f(x_1, y_1, init)) \ldots)$.

```
(* pair_foldr (’a * ’b *’c -> ’c) -> ’c -> (’a list * ’b list) -> ’c *)
fun prod_vect v w =
    pair_foldr (fn (a,b,c) => a*b + c)
      0 (v, w)
```

8.1.2 Question 2

Matrices question:

```
[ [ 1,3,-5],
  [2, 0, 4]
]
```

```
fun emptyMatrix B =
    all (fn l => l = []) B

(* multiply a vector times a matrix *)
fun sm (v, B) =
if emptyMatrix B then []
```

```
else let
   val c = map (fn (x::xs) => x) B
   val B' = map (fn (x::xs) => xs) B
in
   prod_vect (v,c)::sm(v,B')
end
```

8.1.3 Question 3

Proofs question: similar structure to past proofs

8.1.4 Question 4

References question:

```
(* mon_ref: 'a -> (unit -> int) * (unit -> 'a) * ('a -> unit) *)
fun mon_ref a =
let
   val r = ref a
   val c = ref 0
in
   (
     (fn () => !c),
     (fn () => (c := !c + 1; !r)),
     (fn a => (c := !c + 1; r := a)
   )
end
```

9 05 March – Post-Midterm Material

9.1 Introduction to Language Design

Homework 4 will be handed out on Friday, 09 March, to be due two weeks from then.

"A good designer must rely on experience, on precise, logical thinking, and on pedantic exactness. No magic will do." – N. Worth

Goal: a precise foundation for answering questions such as:

- How will a program execute?

- What is the meaning of a program?
- What are legal expressions?
  e.g. `fun foo x x = x+2` and `foo 3 5` returns 7 instead of 5
- What concept of a variable do we have?
- Where is a variable bound?
- When is an expression well-typed?
- Does every expression have a unique type?
- What exactly is an expression?
  Code -¿ Parser (syntax checker) -¿ Type Checker (static semantics) -¿ Interpreter (operational semantics)

In this class, we'll go through the different stages of running an ML program – parsing, type checking, and interpreting. To ensure a language will produce "correct" programs, we have to ensure that each of these stages produce correct results.

9.2 Nano ML

We'll start with a small subset of ML.

Definition: the set of expressions is inductively defined as follows:

1. A number is an expression.
2. The boolean `true` and `false` are expressions.
3. If `e1` and `e2` are expressions, then `e1 op e2` is an expression, where `op` ∈ { `+, -, *, =, <` }.
4. If `e0`, `e1` and `e2` are expressions, then `if e0 then e1 else e2` is an expression.

A more compact way of defining expressions is the BNF grammar (Backus-Naur-Form):

```
Operator op := + | - | * | = | < | orelse
Expression e := n | true | false | e1 op e2 | if e0 then e1 else e2
Value v := n | true | false
```

Examples of syntatically illegal expressions:

- `true false`
- `+3`
- `5-`

This does not type check, however – for example, `true + 3` is syntactically legal but ill-typed.

"An expression `e` evaluates to a value `v`" is equivalent to a *judgement* "$\mathtt{e} \Downarrow \mathtt{v}$"

An expression `if e0 then e1 else e2` evaluates to some value `v` if:

1. `e0` evaluates to `true` and
2. `e1` evaluates to `v`.

This is equivalent to:

```
premise1 .... premise2        ==     e0 evaluates to true    e1 evaluates to v
----------------------               ----------------------------------------
     conclusion               ==       if e0 then e1 else e2 evaluates to v
```

10 07 March

10.1 Language Design and Nano ML continued

Today we want to add variables and `let` expressions to our BNF grammar.

```
Expression e := n | true |false |
                ... | x | let x = e in e' end
```

`x` here represents a class of variables `x,y,z,...`. For example:

```
let z = if true then 2
                else 43
in z + 123
end;
```

The following is not well-formed:

```
let z = 302
in -z end;
let x = 3 in x
let x =3 x+2 end
(* etc *)
```

When is a variable bound? When is a variable free?

Free variables: variables that are not bound. `FV(e)` is the set of free variable names.

- `FV(n) = {}` where `n` is a number.

- `FV(x) = {x}`
- `FV(e1 op e2) = FV(e1) U FV(e2)`
- `FV(let x = e1 in e2 end) = FV(e1) U F(e2) \ {x}`
  `let x=5 in let y=x+2 in y+x end end`
  `let x=x+2 in x+3 end`

Bound variable names don't matter – `let y=x+2 in y+3 end`.

10.2 Substitution:

```
e evalsto v0       [v / x] e' evalsto v
---------------------------------------
     let x=e in e' end evalsto v
```

To evaluate `let x=e in e' end`:

1. Evaluate `e` to `v0`.
2. Substitute `v0` for `x` in `e'`.
3. Evaluate `[v0/x]e'` to `v`.

Substitution: `[e/x]e' = e''` – in `e'`, replace every free occurence of `x` with `e`.

Examples:

- `[e/x] n = n`
- `[e/x] x = e`
- `[e/x] y = y`
- `[e/x] (e1 op e2) = [e/x] e1 op [e/x] e2`
- `[e/x] (let y=e1 in e2 end) =`
  `let y = [e/x] e1 in [e/x] e2 end`
  if `y` $\notin$ `FV(e)`

A problem: free variables in `e` may be bound `y` (captured) if we don't guarantee that the free variables of `e` and the bound variable `y` don't clash or overlap.

Renaming is a special case of substitution – `[y1/y] e = e'`

Our substitution has to be capture-avoiding.

We can also add functions to our BNF grammar:

```
Expression e := ... | fn x => e
```

Substitution for (nameless) functions: (MISSED)

Evaluating functions: `(fn x => e)`

Functions are themselves values – we can extend our values definition:

```
Values v := n | true | false | fn x => e
```

```
------------------------------
(fn x => e) evalsto (fn x => e)

e1 evalsto (fn x => e)    e2 evalsto v2    [v2/x] e evalsto v
-------------------------------------------------------------
                  e1 e2 evalsto v
```

11 09 March

1. Evaluating expressions (recursion)
2. Turing theory into code
3. Modules

11.1 Evaluation

```
Expression e := ... | fn x => | e1 e2
         | rec f => e
Values v := ... | fn x => e
```

Evaluation rules:

```
-------------------------------
(fn x => e) evalsto (fn x => e)


e1 evalsto (fn x => e)  e2 evalsto v2  [v2/x] e evalsto v
---------------------------------------------------------
                  e1 e2 evalsto v
```

In SML, you might write:

```
fun f (x) = if x=0 then 0
            else x + f (x-1)

(* equivalent to in Nano ML *)
rec f => fn x => if x=0 then 0
                 else x + f (x-1)

f 3
=> if 3=0 then 0
   else 3 + f (3-1)
(* instead *)
( rec f => fn x => if x=0 then 0
         else x + f (x-1) ) 3

(rec f => fn x => if x=0 then 0
     else x + (rec f => fn x ....)) 3
=> if 3=0 then 0
   else 3 + (rec f => fn x ...) (3-1)
=> ...

[e'/x] (fn y => e) = fn y => [e'/x] e
provided y is not in FV(e')

[e'/x] (rec f => e) = rec f => [e'/x] e
where f is not in FV(e)
```

11.2 Modules

ML: Core language and the Module Language

Modules: two parts – signature and structure

Signature: interface of a structure

Structure: program consisting of declarations

When does a structure implement a signature?

`:>` makes the implementation of the structure opaque.

A structure can provide more components, but it cannot have fewer.

Structures may provide more general types (e.g. using `'a` instead of `int`).

Structures may also implement concrete datatypes (e.g. in Queues, lists, etc.), but the signature keeps the type abstract. This is important for information hiding.

The order of declarations does not matter.

12 12 March

12.1 Types

Many ill-typed expressions will get "stuck". For example:

```
if 0 then 3 else 3+2
```

```
0+2
```

```
(fn x => x+1) true
```

Our evaluator will accept these expressions, but typing should rule out these ill-formed expressions. These should lead to run-time errors. Typing will ensure that we never evaluate these expressions.There will be fewer run-time errors as a consequence.

Typing approximates what happens during run-time. Typing allows us to detect errors early and gives us precise error messages. As a consequence, programmers can spend more time developing and less time testing their programs.

When we type-check an expression, we prove the absense of certain program behaviors.

Safety: If a program is well-typed, then every intermediate state during evaluation is defined and well-typed.

Types classify expressions according to their value. If we know what values there are in a language, we know what types there are.

Recall `Values v := n | true | false`. So, for now:

`Types T := int | bool`

The shorthand `e : T` can be read as "expression `e` has type `T`".

Axioms:

- `n : int`
- `false : bool`

- `true : bool`

For `if`-expression `if e then e1 else e2`:

- `e : bool`
- `e1 : T`
- `e2 : T`

We need to check that `e1 : T` and `e2 : T`.

```
e1 : int  e2 : int
-----------------
  e1 + e2 : int

e1 : T   e2 : T
---------------
 e1=e2 : bool
```

We can add tuples to our expressions: `Expressions e := ... | (e1, e2) | fst e | snd e`. We can then add tuples to possible values:

```
Values v := n | false | true | (v1, v2)
Types T := int | bool | T1 x T2

e1 : T1      e2 : T2
--------------------
 (e1, e2) : T1 x T2

e   T1 x T2      (ditto)
-----------   ------------
fst e : T1     snd e : T2
```

Now for let-expressions:

```
let x=5 in x+3 end : int
as 5:int
and (assuming x : int) x+3 : int
```

Note that we need to reason about the type of variables.

$\Gamma \vdash e : T$ reads "Given assumtion Γ, expression `e` has type `T`". I'm going to replace Gamma with `G` in any verbatim blocks.

```
Context G := | G1 x T
```

```
G |- e : bool  G |- e1 : T  G |- e2 : T
---------------------------------------
  G |- if e then e1 else e2 : T

G(x) = T
-------
G |- x : t

G |- e1 : T1  G1 x: T1 |- e2 : T  |
-------------------------------   } x is new
G |- let x=e1 in e2 end : T       |
```

Each assumption is unique implies that each variable has a unique type! We'll come back to let expressions with assumptions later.

```
               ...
            ----------------
            x:int |- x+2:int  x:int |-...
            ------------------------------
|- 5 : int  x : int |- let y ... end : int
-----------------------------------------
let x=5 in let y=x+2 in x+y end end : int

Axioms:
--------   -------------  ------------
G|-n:int   G|-false:bool  G|-true:bool
```

We can infer types:

```
G |- e : T
+    +   -
```

Typing rules lend themselves to be interpreted as type-inference rules. They will infer a unique type.

13 14 March

13.1 Typing rules continued

When is an expression well-typed?

```
e := n | true | false | e1 op e1
```

```
   | if e then e1 else e1 | (e1, e2)
   | x | let x=e in e' end
T := int | bool | T1 x T2
```

`e : T` = "expression e has type T"

`G |- e : T` = typing assumption about variables (e.g. Given assumptions in T, expression e has type T)

```
G|-e:T'         GxT'|-e':T
--------------------------
G |- let x=e in e' end : T



           ------------ ------------
           x:int|-x:int x:int|-3:int
-------    ---------------------
|-5:int         x:int|-x+3:int
------------------------------
    |- let x=5 in x+3 end
```

For every expression, we can infer a type. Every expression has a unique type.

13.2 Extensions

Today we will look at extensions – functions, applications, recursion, and references.

```
T := ... | T1 -> T2
V := ... | fn x => e


assume x:int, verify that if x=0 else x+2 has type int
------------------------------------
fn x => if x=0 then 4 else x+2 : int

(* this rule cannot be interpreted as inferring a type *)
G1xT1|-e:T2
-----------------------
G|-fn x => e : T1 -> T2

(* solution: annotate x with its type *)
fn x : T1 => e
fn x => if fst x=0 then snd x
                   else 5
         : int x int -> int
```

Without type annotations, `fn x => x` has infinitely many types. With type annotations, we can infer a unique type.

What is the most general type of an expression? `'a` "principle type"

Extremely important rule:

```
G|-e1:T1->T2  G|-e2:T1
-------------------------
G |- e1 e2 : T2
```

New reference type:

```
T := ... | T1 ref | unit

e := ... | !e | e := e' | ref e
```

```
G|-e : T ref
------------
G |- !e : T

G|-e:T
---------
G |- ref e : ref T

G|-e:T ref  G|-e':T
-----------------
G |- e := e' : unit

G1 f:T |- e : T
--------------------------
G |- ref f => e : T


(1)        (2)       sum: int->int, x:int |- sum(x-1)+x : int
-----------------------------------------------------------------
sum int->int, x:int |- if x=0 then 0 else sum(x-1) + x : int
----------------------------------------------------------------
sum int->int |- fn x => if x=0 then 0 else sum(x-1) + x : int->int
```

```
----------------------------------------------------------------
rec sum => fn x => if x=0 then 0 else sum (x-1) + x : int -> int
```

14 16 March

14.1 Type inference and polymorphism

Can we infer a type for an expression?

Recall that we needed type annotations on functions. `fn x : int => x`

Does an expression have a unique type? `fn x => x` could have the type `int->int` or `bool->bool`, etc., however this function does have one principle type `'a->'a`.

The question: how can we infer the principle type of an expression *without* given type annotations?

Example:

```
double = fn f => fn x => f(f(x))
: ('a -> 'a) -> 'a -> 'a
```

Intuitively, we can now use this function in multiple ways.

```
double (fn x => x+2) 3 : int
```

```
f         ( f         x )
^           ^         ^
'a->'b     'a->'b    'a
               'b
```

Because of this contradiction, we cannot pass a β to a function which expects an α.

```
double (fn x => x) false : bool
```

Crucial in this: type variables

```
T := int | bool | T1 x T2 | T1 -> T2 | 'a
```

How do we instantiate type variables? Substitution!

```
[T/'a] (int) = int
[T/'a] ('a) = T
[T/'a] ('b) = 'b (where 'b != 'a)
[T/'a] (T1 -> T2) = [T/'a] T1 -> [T/'a] T2
```

Two views:

1. Are *all* substitution instances of `e` well-typed?
   If e has some type T and constraints some type variables $\alpha_1, \ldots, \alpha_n$, then e has type `[T1/'a1 ... Tn/'an] T` for every $T_1, \ldots T_n$?
2. Does there exist some substitution instance such that e has type T?

For example: `fn x => x+1` has type β? Choose for $\beta =$ `int->int`.

`fn x => x` has type β? Choose for β =`int->int` or `bool->bool` or, most generally, `'a->'a`.

Will `fn x => x+1` have type `'b->bool`? No. there is instantiation for `'b`.

14.1.1 Type inference

(1) use typing rules to generate constraints. ←will always succeed **(2)** solve constraints. ←will sometimes fail

What is a constraint? For example, `T=bool`.

`Constraint C := T=T' | tt | C1 ^ C2`

How do we collect constraints? Informally, to infer a type for `if e then e1 else e2`, we do the following:

1. Infer a type T for e (and C).
2. Infer a type T_1 for e_1 (and C_1).
3. Infer a type T_2 for e_2 (and C_2).

Constraints: `T = bool ^ T1 = T2`.

To infer a type for `fn x => e`:

1. Assume x has type α_1.
2. Infer a type $T_2 4$ (provided the constraints C can be solved).
3. We then know that `fn x => e` has type `'a1 -> T2` (provided the constraints C).

`G |- e => T  C` means "given the assumptions Γ for an expression e, the type T, and the constraints C", or "the expression e has type T provided I can solve C".

14.1.2 Solving constraints

```
G |- e => T/C   G |- e1 => T1/C1   G |- e2 => T2/C2
----------------------------------------------------
G |- if e then e1 else e2 => T2 / T=book ^ T1=t2 ^ C ^ C1 ^ C2

G1 x:'a1 |- e => T2/C
-------------------------
G |- fn x => e => a1 -> T2 / C

input: left of =>
output: right of =>
x:'a1 means 'a1 is new

fn x => if 3=1 then 55 else x : 'a1 ->
assuming x:'a1
constraint: int = 'a1 ^ bool=bool
```

Solving the constraints, we learn that `'a1 = int`, and therefore

```
fn x => if 3=1 then 55 else x : int->int

------------------
G |- n => int / tt

----------------------
G |- true => bool / tt

G(x) = T
----------------
G |- x => T / tt
```

15 19 March

Some small mistakes on HW4 – see WebCT for more details.

15.1 Type inference

Examples:

```
fn f => fn x => f (x) + 1
    : 'a1 -> 'a2 -> int
f: 'a1, x: 'a2 |- f(x) + 1 : int
f(x) : int
'a1 = 'a2 -> int
.: the type of the fn is ('a2->int) -> 'a2 -> int
    as f : 'a2->int
```

A slightly more complicated example:

```
fn f => fn g => fn x => f( g x) + g x
    : 'a1 -> 'a2 -> 'a3 -> int
f: 'a1, g: 'a2, x: 'a3 |- f(g(x)) + g(x) : int

Constraints to solve:
'a2 = 'a3 -> int (from last g(x))
'a2 = 'a3 -> 'a4 (from first g(x))
'a1 = 'a4 -> int
'a2 = 'a3 -> 'a4, 'a2 = 'a3 -> int (intuitively)
'a1 = 'a4 -> int
.: 'a4 = int, 'a1 = int -> int
No constraint on 'a3

.: the type we infer is:
    (int -> int) -> ('a3->int) -> 'a3 -> int
         f               g         x
```

Third example:

```
fn f => fn x => f x + x f

f: 'a1, x: 'a2 |- f x + x f
'a1 = 'a2 -> int
'a2 = 'a1 -> int

'a1 = ('a1 -> int) -> int

Does there exist an instantiation for 'a1 such that
    'a1 = ('a1 -> int) -> int ?
There is no solution to make both sides equal.
Reason: variable on the LHS occurs embedded on the RHS (circular).
```

15.2 Unification Algorithm

Input: set of constraints

Question: Does there exist an instantiation such that all constraints are true?

```
C := tt | C1 ^ C2 | T = T’
```

`C =====> C’` until we reach our goal where `C =>* tt`.

```
C ^ tt => C
C ^ int = int => C
C ^ bool = bool => C
C ^ T1->T2 = S1->S2 => C ^ T1=S2 ^ T2=S2
C ^ ’a = T => [T/’a] C (provided ’a is not in FV(T)) (occurs check -- prevents circular ter
C ^ T = ’a => [T/’a] C (same provisions)
```

```
unify (T1, T2) =
    (* pattern match on T1, T2 *)
    (* return bool *)
```

15.3 Examples (part 2)

```
double = fn f => fn x => f (f x)
    : ’a1 -> ’a2 -> ’a3
f:’a1, x:’a2 |- f (fx) : ’a3
inter (f x) ---> ’a1 = ’a2 -> ’a4
outer f (f x) -> ’a1 = ’a4 -> ’a3
’a4 -> ’a3 => ’a2 -> ’a1
’a4 = ’a2, ’a3 = ’a4
.: ’a2 = ’a3 = ’a4
.: we infer:
    (’a2 -> ’a2) -> ’a2 -> ’a2

double (fn x => x+1) 5;
double (fn x => x) true;

let
    d = fn f => fn x => f (f x)
    x = d (fn x => x+1) 5
    y = d (fn x => x ) true
in
```

```
    (y, x)
end
(* this will type-check in SML, but would not type check in our definition so far,
 * as we don’t have a way to reuse a function with different types as we’ve
 * given it a type. SML will abstract over polymorphic type variables, i.e.
 * it can reuse the type over multiple types by saying "for all ’a2:(’a2->’a2)->’a2->’a"
 *)
```

16 21 March

16.1 Type inference continued

16.1.1 Warm-up examples

Example 1:

```
fn f => fn g => fn x => if gx then f(g x) else f(f(g x))

f : ’a1, g : ’a2, x : ’a3 |- if (g x) then f(g x) else f(f(g x)) : ’a4

’a1 -> ’a2 -> ’a3 -> ’a4
```

We are looking for instantiations for `’a1, ’a2, ’a3, ’a4` such that the expression is well-typed.

```
(g x) == ’a2 = ’a3 -> bool
f(g x) == ’a1 = bool -> ’a4
f(f(g x)) == ’a1 = ’a4 -> ’a4
.: ’a4 = bool
```

This given function will have type `(bool->bool)->(’a3->bool)->’a3->bool`.

Example 2:

```
fn f => if f true then f 5 else f 4

f:’a1 |- if f true then f 5 else f 4 : ’a2

f true == ’a1 = bool -> bool
f 5, f 4 == ’a1 = int -> ’a2

bool->bool != int -> ’a2
```

```
(* won't type check? *)

(* in SML *)
let val f = fn x => x
in if (f true) then (f 5) else (f 4)
end
(* will type int -- why? *)
```

In our algorithm:

```
f : 'a -> 'a |- if (f true) then f 5 else f 4
(f true) == 'a -> 'a = bool -> bool
f 4 == 'a -> 'a = int -> int
f 5 == ""

'a = int
'a = bool
```

Our algorithm does not do what SML does – it cannot truly reuse `f` in multiple ways.

To make this function type check in our language, we'll need different copies of `f`.

Observation: `if (fn x => x) true then (fn x => x) 5 then (fn x => x) 4` will type check – each "sub-function" will be assigned its own relevant type. By writing this function three times, we will type check it three times, however we will have no constraints to carry over, so our language will type check this correctly.

16.1.2 Modifying our typing rules

As a solution to this problem, we can modify our old rule, which is not suitable to handle polymorphic functions. Our new rule is as follows:

```
G|-[e1/x]e2 : T
-----------------------
G |- let x=e1 in e2 : T
```

Note: type checks e_1 multiple times – not very efficient, but at least this will work for polymorphism. In practical languages, this is not what happens. Instead, we'll need to make the assumption $\forall \alpha - > \alpha$, so whenever we look up the type of `f`, we get a "fresh", unique copy of its type.

In ML: (generalize – abstract over the free variables and quantify over them, so we can reuse them as often as we'd like)

```
G|- e1 : S   G1 x generalize(S) |- e2 : T
-----------------------------------------
G |- let x=e1 in e2 : T
```

In example 2, `(forall 'a, 'a -> 'a) -> int` – full polymorphism. In ML, we have parametric polymorphism, where all quantification over type variables is at the outside. Some languages are slowly moving towards allowing full polymorphism.

16.1.3 More examples

```
let val r = ref (fn x => x)
in r := (fn x => x+1); !r true
end
```

In SML, this will not type check; however, from the rules we have, it will.

```
ref (fn x => x) == ('a -> 'a) ref
ref (fn x => x) := (fn x => x+1); (ref (fn x => x))! true
'a = int
then
'a = bool

"happy" but WRONG
```

In general, we can reuse functions and values in a polymorphic way, but we cannot make any general polymorphic assumptions about expressions which are not values! Recall that references are not values, so we cannot make any general polymorphic assumptions about references.

This is called *value restriction* – we can only reuse values in a polymorphic way. If something is not a value, we cannot make any polymorphic assumptions about it, since there can be side effects and such.

```
let val r = ref (fn x => x)
in r
end

(* SML *)
Warning : type variables are not generalized because of value restriction are instantiated
```

17 23 March

HW 5 out – due 11 April, 2012

17.1 Bi-directional type checking

So far, we have type inference rules.

Hindley-Milner (type inference) – *Goal:* infer the most general type of an expression without declaring any types.

Disadvantages of not writing type annotations:

- Gives better error messages (programmer communicates his or her intent)
- High-grade (high-quality) documentation – does not get out of sync with the code
- I can refuse types to force programs to be used in a specific way
- Hindley-Milner type inference does not scale to richer types such as sub-typing, full polymorphism, and dependent types.
- Hindley-Milner type inference is complicated and difficult to implement (unification is needed).

Basic idea behind bi-directional type checking – instead of inferring a type:

$$\Gamma \vdash e \Rightarrow \tau$$

$\forall e$ where τ is the output, we'll use two judgements (functions), "check" and "synthesize". First, check $\Gamma \vdash e \Leftarrow \tau$ for inputs Γ, e, τ (check that expression e has type τ under the assumptions Γ). Then, synthesize $\Gamma \vdash e \Rightarrow \tau$ (synthesize a type τ for expression e under the assumptions Γ for *some* e). Synthesize basically means infer.

This algorithm is based on two observations. First, we can't use information that we don't have. Second, we should try to use information that we do have.

Type variables:

$$\frac{}{\Gamma_i x : \tau \vdash x \Rightarrow \tau}$$

Functions

$$\frac{\tau_i x : \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \texttt{fn x => e} \Rightarrow \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_2 \to \tau \quad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash e_1 e_2}$$

Example:

```
twice: (int -> int) -> int -> int)
    |- (twice (fn x => x+1)) 3
        ----
        ^ infer type (int->int)->int->int
.: int
```

Conversion:

$$\frac{\Gamma \vdash e \Rightarrow \tau' \quad \tau' = \tau}{\Gamma \vdash e \Leftarrow \tau}$$

Example:

$$\frac{\vdash 3 \Rightarrow \texttt{int} \quad \texttt{int = int}}{\vdash 3 \Leftarrow \texttt{int}}\ byconversion$$

18 26 March

18.1 Subtyping

```
fun area r = 3.14 * r * r

area : real -> real
area 2 (* type error *)
```

We want to pass an int whenever a real is required (int ≤ real)

Subtyping principle: S < T. S is a subtype of T if we can provide a value of type S whenever a value of T is required.

$$\frac{\Gamma \vdash e : S \quad S \leq T}{\Gamma \vdash e : T}$$

$$\frac{T \leq R \quad R \leq S}{T \leq S}$$

$$\frac{T_1 \leq S_1 \quad T_2 \leq S_2}{T_1 \times T_2 \leq S_1 \times S_2}\ covariant$$

Records are a generalization of tuples (n-ary tuples where each element has a label).

$$\frac{\forall i\ T_i \leq s_1}{\{x_1 : T_1, \ldots, x_n : T_1\} \leq \{x_1 : S_1, \ldots, x_n : S_1\}}\ \textit{depth subtyping}$$

Two things we want for records:

1. Permutations of elements should be allowed!

$$\frac{\phi \text{ is a permutation}}{\{x_1 : T_1, \ldots, x_n : T_n\} \leq \{x_{\phi(1)} : T_{\phi(1)}, \ldots, x_{\phi(n)} : T_{\phi(n)}\}}$$

(MISSED FROM FAR BOARD)

$$\frac{k > n}{\{x_1 : T_1, \ldots, x_k : T_k\} < \{x_1 : T_1, \ldots, x_n : T_n\}}$$

18.1.1 Function example

```
let
    (* areaSqr: real -> real *)
    fun areaSqr (r : real) = r*r
    (* areaFake: real -> int *)
    fun areaFake (r : real) = 3
in
    areaSqr 2.2 + 3.2
end
```

Question: Can we provide `areaFake: real -> int` whenever `areaSqr: real -> real` is required? YES, but not in SML – our discussion of subtyping is purely theoretical.

Therefore,

$$\frac{T_2 \leq S_2}{T \rightarrow T_2 \leq T \rightarrow S_2}$$

19 28 March

19.1 Subtyping continued

19.1.1 Review

Basic subtyping principle: $S \leq T$ “S is a subtype of T” if we can provide a value of type S whenever a value of type T is required.

Products: covariant `int * real <= real * real` `real * int <= real * real`

Functions: contravariant `int -> int <= int -> real` as `int <= real` and functions are co-variant in the output type.

`real -> int <= int -> int` as `int <= real` and functions are contra-variant in the input type.

Invalid: `int -> int !<= real -> real`

$$\frac{S_1 \leq T_1 \quad T_2 \leq S_2}{T_1 \rightarrow T_2 \leq S_1 \rightarrow S_2}$$

19.1.2 References

```
let
    val x = ref 2.0
    val y = ref 3
in
    !x + 3.14
end
```

This is a perfectly valid program, as `x : real ref` and `y : int ref`. Can we supply an `int ref` (e.g. `y`) whenever a `real ref` is required? YES (but, again, not in SML).

$$\frac{S \leq T}{\texttt{S ref} \leq \texttt{T ref}}$$

```
let
    val x = ref 2.0
    val y = ref 2
in
    y := 4
```

```
end
```

Should we be able to supply a `real ref` (e.g. `x`) whenever an `int ref` is required? YES. This seems almost contradictory, since we said yes to the previous question. The following rule seems incompatible to the one before:

$$\frac{T \leq S}{\texttt{S ref} \leq \texttt{T ref}}$$

Therefore, there is *no subtyping* on references (locations) – references are *invariant*.

$$\frac{S \leq T \quad T \leq S}{\texttt{S ref} \leq \texttt{T ref}}$$

Of course, if $S \leq T$ and $T \leq S$, then $S = T$.

19.1.3 More subtyping

Recall the typing rule for subtyping:

$$\frac{\Gamma \vdash e : S \quad S \leq T}{\Gamma \vdash e : T}$$

Upcasting is always safe with type annotations – it is safe to “forget”!

$$\frac{\Gamma \vdash e : S \quad S \leq T}{\Gamma \vdash (T)e : T}\ \textit{Upcast}$$

Below, there can be no relationship between S and T. This is not really safe in general. Most often, S is a supertype of T. You must trust that this is safe in order to use it.

$$\frac{\Gamma \vdash e : S}{\Gamma \vdash (T)e : T}$$

E.g. `fn (x:real) => (int) x + 1` downcasts `x` to an `int`.

20 30 March

20.1 Dependent types

```
fun append [] l = l
  | append (h::t) l = h::(append t l)

(* append: ’a list -> ’a list -> ’a list *)
```

What else do we know?

```
length (append l1 l2) = length l1 + length l2
```

How can types track information about the length of lists? Index a type by an object/expression which stands for an integer.

```
for all n : int, for all m : int:
list 'a n -> list 'a m -> list 'a (plus n m)
```

Agda (a dependently typed functional language), DML (Dependent ML), Omega, Epigram, Coq, ...

Staring with simple types in Agda:

```
data Nat : Set where
     zero : Nat
     succ : Nat -> Nat
data Bool : Set where
     true : Bool
     false : Bool

data List (A : Set) : Set where
     [] : List A
     _::_ : A -> List A -> List A
```

In Agda, we have to explicitly declare the type of a program:

```
plus : Nat -> Nat -> Nat
plus zero m = m
plus (succ n) m = succ (plus n m)

rev_tl : {A : Set} -> List A -> List A -> List A
rev_tl [] acc = acc
rev_tl (h::t) acc = rev_tl t (h::acc)
```

The `{A : Set}` clause is read as "for all `A`".

Dependent types: Index a boolean list with its length

```
    data BoolList : Nat -> Set where
         nil : BoolList zero
         cons : {n : Nat} -> Bool -> BoolList n -> BoolList (succ n)

    append : {n : Nat} -> {m : Nat} ->
        BoolList n -> BoolList m -> BoolList (plus n m)
    append nil l = l
    append (cons h t) l = cons h (append t l)
```

Surprisingly, the program didn't change at all! However, the type checker in Agda will do a lot more work for you than it did in SML, as our type declaration is much, much richer in our Agda code.

Type checker's tasks:

`(1) l : BoolList m |- l : BoolList (plus n m)`

We need to prove that `BoolList m = BoolList (plus zero m)`. Show that `plus zero m => m`.

To check that two types are equal means we need to show that the index arguments are equal. This is based on evaluation!

$$\frac{m \Rightarrow m'}{\texttt{BoolList m = BoolList m'}}$$

Agda works with total functions only – i.e. those that are defined to work with all possible inputs. This way, the functions are provably terminating.

21 02 April

21.1 Dependent Types continued

```
    cons: {n : Nat} -> Bool -> BoolList n -> BoolList (succ n)
```

For our append method (from Friday), consider the base case to be case 1 and the induction case to be step 2. To check case 1, can we show that `BoolList m = BoolList (plus zero m)`? `BoolList m` is the inferred type for `l` and the RHS is the expected type. Because `plus zero m => m`, we know that the two types are equal.

For case 2, we know `cons h t : BoolList (succ n')`, and therefore `t : BoolList n'`. `append t l : BoolList (plus n' m)`, so `cons h (append t l) : BoolList (succ (plus n' m))`.

Expected type: `BoolList (plus (succ n') m)`, because `plus (succ n') m => suc (plus n' m)`, so we know that the two types are equal.

Vectors: polymorphic lists

```
    data Vec (A : Set) : Nat -> Set where
         [] : Vec A zero
         _::_ : {n : Nat} -> A -> Vec A n -> Vec A (succ n)
    (* by using (succ n), we’re prevented from ever getting a list of length zero *)
    vhead : {A : Set} -> {n : Nat} -> Vec A (succ n) -> A
    vhead h::t = h
```

Note that there is no case necessary for the empty list, so this `vhead` function is total.

```
    vtail : {A : Set} -> {n : Nat} -> Vec A (succ n) -> Vec A n
    vtail h::t = t

    vmap : {A : Set} -> {B : set} -> {n : Nat} ->
        (A -> B) -> Vec A n -> Vec B n
    vmap f [] = []
    vmap f (h :: t) = (f h)::(vmap f t)

    vzip : {A : Set} -> {B : Set} -> {n : Nat} ->
        Vec A n -> Vec B n -> Vec (A * B) n
    vzip [] [] = []
    vzip (x::xs) (y::xs) = (x,y)::(vzip xs ys)
```

We’d also like to have a safe look-up function of the kth element in a vector:

```
    data _ <= _ : Nat -> Nat -> Set where
         leq_zero : {n : Nat} -> zero <= n
         leq_succ : {n,m : Nat} -> n <= m -> succ n <= succ m

    kth : {A : Set} -> {n : Nat} -> {k : Nat} -> k <= n -> Vec A (succ n) -> a
    kth zero leq_zero (x::xs) = x
    kth (succ k) (leq_succ p) (x::xs) =
        kth k p xs
```