COMP 302 Crib Sheet
by Julian Lore Side 1 of 2

1 Types

Defining a type:

```
type suit = Clubs |
  Spades | Hearts
  | Diamonds
```

Order declared does not matter. Clubs, Spades, etc are **constructors** and must begin with a **capital letter**

Recursively defined Define hand inductively. Empty is of type hand. If c is a card and h is a hand, then Hand(c, h) is a hand. Nothing else is a hand.

```
type hand = Empty |
  Hand of card * hand
let hand0:hand = Empty
let hand1:hand =
  Hand((Ace, Hearts),
  Empty)
```

Mutual recursive data type

```
type 'a forest = Forest
  of ('a tree) list
and 'a tree = Empty |
  Node of 'a * 'a forest
```

Option Data Type

```
type 'a option = None
  | Some of 'a
```

Used in case a function might not return something.

Types of following exprs

- 3+2, type = int, val = 5, no effect
- 55, int, 55, no
- `fun x -> x+3*2`, `int -> int`, <fun>, no
- `((fun x->match x with [] -> true | y::ys -> false), 3.2 *. 2.0)`, `(a' list -> bool) * float`, (<fun>,6.4), no
- `let x = ref 3 in x := !x + 2`, unit, (), updates val of x to 5
- `fun x -> x := 3`, `ref 'a -> unit`, <fun>, no
- `fun x -> (x := 3; x)`, `ref 'a -> ref 'a`, fun, no
- `fun x -> (x := 3; !x)`, `ref 'a -> 'a`, fun, no

2 Pattern Matching

```
match <exp> with
| <pattern> -> <exp>
| <pattern> -> <exp>
...
```

exp we're analyzing is called *scrutinee.*

3 Arguments

Passing all args at same time: 'a * 'b -> 'c

One arg at a time: 'a -> 'b -> 'c

curry: (('a * 'b -> 'c) -> 'a -> 'b -> 'c)

```
let curry f =
  (fun x y -> f (x,y))
```

uncurry: (('a -> b' -> 'c) -> ('a * b' -> 'c))

```
let uncurry f =
  (fun (x,y) -> f x y)
```

Note that function types are right associative: 'a -> 'b -> 'c = 'a -> ('b -> 'c) and function application is left associative: f 1 2 = (f 1) 2

4 Proofs

$e \Downarrow v$: e evals to v in multiple steps (Big-Step).

$e \Rightarrow e'$: e evals in one step to e' (small-step (single))

$e \Longrightarrow^{*} e'$: e evals in multiples steps to e' (small-step (multiple))

Structural induction

```
type 'a list =
  nil | :: of 'a *
        'a list
```

To inductively prove about lists, prove for empty list, assume it holds for lists t and then show for lists h :: t.

```
type 'a tree =
  Empty | Node of 'a *
          'a tree *
          'a tree
```

To inductively prove about trees, prove for empty tree, assume for trees l and r and show for tree Node(a, l, r)

Inductive Proof

```
let rec r_app l1
  l2 = match l1 with
  | [] -> l2
  | x::xs -> r_app xs
(x::l2)
let r_app' l1
l2 =
let rec rev l =
match l with |[]->[]
|x::xs -> xs @ [x] in
let rec app l1 l2
= match l1 with
|[]->l2
|x::l1'->x::(app l1' l2
  )
in app
(rev l1) l2
```

For all lists l1 l2, r_app l1 l2= r_app' l1 l2

Proof by structural induction on l1

Base: l1 = [].

r_app l1 l2 = r_app [] l2 = l2 (by rev_app) = app [] l2 (by def of app) = app (rev []) l2 (def of rev) = app (rev l1) l2 = rev_app' l1 l2 (by def of rev_app')

Step: l1 = h :: t,

IH: For all l2, rev_app t l2 = rev_app' t l2.

rev_app' l1 l2 = rev_app (h :: t) l2 = rev_app = t (h :: l2) (by def of rev_app) = rev_app' t (h :: l2) (by IH) = app (rev t) (h :: (app [] l2)) (def app) = app (rev t) (app [h] l2) (def app) = app (app (rev t) [h]) l2 (ass of app) = app (rev (h :: t) l2) (def rev_app') = rev_app' l1 l2

5 Higher Order Functions

Used to abstract over common functionality.

Non-generic sum: int * int -> int

Generic sum with fun as arg: (int -> int) -> int * int -> int

Common hofs:

```
List.map: ('a -> 'b) ->
  'a list -> 'b list
List.filter: ('a
  -> bool)
  -> 'a list -> 'a list
(* Folds from l -> r *)
List.fold_right:
  ('a -> 'b -> 'b) ->
  'a list -> 'b -> 'b
(* Folds from r -> l *)
List.fold_left:
  ('a -> 'b -> 'a) ->
  'a -> 'b list ->'b
List.for_all:
  ('a -> bool) ->
  'a list -> bool
List.exists:
  ('a -> bool) ->
  'a list -> bool
```

Anonymous functions: (fun x -> x+1)

(function -> |_ |else), equiv to matching argument

Implementing them: map -> apply fun to head and prepend. Return empty for empty list. filter -> Prepend on tail called if true, else call again on tail.

fold_right f l b-> ret base if empty, else f h (fold_right f t b)

fold_left f l b-> ret base if nil, else fold_left on f t and (f h b), new base is f h b

6 Partial Evaluation

```
let plus x y = x + y
let plus3 = (plus 3)
let plus3' = (fun x ->
  plus x 3)
```

plus: int -> int -> int

plus3: int -> int, although it's really a fun y -> 3 + y

plus3' is really a fun x -> x + 3

Partial evaluation doesn't evaluate inside the function, just plugs in the value you gave it. If you want to force it to evaluate a certain part, you must define the part to evaluate using let z = x in (fun y -> z + y)

7 Effects

`let x = ref 0`, compare addr with `t == s`, compare content with `t=s`, read value with `!x`, update val `x := 3`, pattern match val with `let {contents = x} = ref 0` gives `x = 0`.

```
type counter_object = {
    tick : unit -> int ;
    reset: unit -> unit}

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

8 Exceptions

Force to consider exceptional case, can segregate special case from others (less clutter), **divert control flow**

3/0 has type int, but no val, has an **effect**, raises run-time exception Division_by_zero

```
let head (x::t) = x in
  head []
```

Raises Match_failure

```
(*Make a new ex* )
exception Domain
raise Domain
```

Backtrack through tree using exceptions

```
let rec find_gen  t k =
    match t with
| Empty -> raise NotFound
| Node (l, (k',d), r) ->
if k = k' then d else
try find_gen l k with
    NotFound -> find_gen
    r k

let rec change coins amt
  =
 if amt = 0 then [] else
  begin match coins with
    | [] -> raise Change
    | coin::cs ->
     if coin > amt then
        change cs amt
     else
      try
       coin :: change
          coins (amt -
          coin)
      with Change ->
          change cs amt
  end
```

9 Modules

sig for defining signature, struct for actual implem

```
 module type CURRENCY =
sig
 type t
 val unit : t
 val plus : t -> t -> t
 val prod : float -> t ->
    t
 val toString : t ->
    string
end;;

module Float =
struct
 type t = float
 let unit = 1.0
 let plus = (+.)
 let prod = ( *. )
end;;
```

module Euro = (Float : CURRENCY), module Dollar = (Float : CURRENCY). Euro.t and Dollar.t incompatible.

```
module type CLIENT =
sig
...
end;;

module type BANK =
sig
 include CLIENT
end;;
```

include will inherit all vals declared in CLIENT sig. Can overshadow by redeclaring same name.

Functor, takes in module as input

```
module Old_Bank (M :
   CURRENCY) : (BANK
   with type currency =
   M.t) =
struct
 type currency = M.t
 type t = {mutable
    balance : currency}
end;;
```

10 Continuations

Representation of execution state of a program (ex. call stack) at a pt in time. Save state and restore later.

Every function can be written tail-recursively

To re-write a fun tail-recursively, add additional arg, a continuation (like acc). Base case calls continuation, recursive builds up cont.

Tail-recursion Continuation is a functional accumulator, represents call stack built when recursively calling func and builds final result

Failure Continuation Continuation tells you what to do upon **failure**

```
let rec find_tr p t cont
   = match t with
 | Empty -> cont ()
 | Node(l, d, r) ->
  if (p d) then Some d
   else find_tr p l (fun
     () -> find_tr p r
     cont)

let find' p t = find_tr p
   t (fun () -> None)
```

Success Continuation Continuation keeps track of what to do on **success**, builds final result

```
let rec findAll' p t sc =
   match t with
 | Empty -> sc []
 | Node(l,d,r) ->
  findAll' p l
   (fun el -> findAll' p
     r
    (fun er -> if (p d)
       then sc (el@(d::
       er)) else sc (
       el@er)))
```

11 Lazy Programming

Eager: Eval by call-by-value, variables bound to vals

let x = horribleComp (345) in 5, this evals horribleComp (345), binds xx to it and evals 5

It is easier to reason about eager comp, clear when things eval but may eval things **never needed**

Lazy Computation: Suspend computation until needed

Memoize results, demand-driven

Good for **infinite data and interactive data**

Ex lists are finite. Can pattern match lists. Reason about lists via induction.

We **don't** construct infinite data, we define **observations** we make about them

Given stream we can ask for head of stream and tail (rest of stream)

Can we always make an observation? Does it eventually terminate? No, prog may run forever. Prog should remain **productive** though, i.e. can make observation at each step

We can suspend eval of an expr

```
type 'a susp = Susp of
   unit -> 'a
let force (Susp f) = f ()
let x = Susp(fun () -> 1
   + 2) in
force x
```

Infinite data streams

```
type 'a str =
{hd:'a ; tl ('a str) susp
  }

let rec numsFrom n =
{hd = n ;
 tl = Susp (fun () ->
      numsFrom (n+1))}
```

Get head using observation hd, getting tail gives suspended stream. Want more elements, ask for more

12 Language Design

3 key qs: Syntactically legal exprs? **Grammar.** Well-typed exprs? **Static semantics.** How is expr executed? **Dynamic semantics.**

Syntactically Legal exprs Defined inductively by: number n is an expr, bools true and false are exprs, e_1, e_2 exprs then so is $e_1\ op\ e_2$ with op $= \{+, =, -, *, <\}$, if e, e_1 and e_2 are exprs then if e then e1 else e2 is an expr

Alternatively, Backus-Naur Form (BNF)

operations op ::== $+|-|*|<|=$

expressions e ::== $n|e_1\ op\ e_2|true|false|$if e then e_1 else $e_2|x|$ let $x = e_1$ in e_2 end $|fn\ y \Rightarrow e|e_1\ e_2|fn\ y \Rightarrow e|e_1 e_2 \Downarrow v$

In ocaml:

```
type primop = Equals |
LessThan | Plus | Minus |
   Times
type exp =
| Int of int
| Bool of bool
| If of exp * exp * exp
| Primop of primop * exp
   list
| Let of dec * exp
and dec = Val of exp *
   name
```

Values Values v ::== $n|true|false$

$e \Downarrow v$ means expr e evals to val v **Formalizing**

$$\frac{}{v \Downarrow v}\ \text{B-VAL}$$

$$\frac{e \Downarrow true \qquad e_1 \Downarrow v}{\text{if } e \text{ then } e_1 \text{ else } e_2 \Downarrow v}\ \text{B-IFTRUE}$$

$$\frac{e \Downarrow false \qquad e_2 \Downarrow v}{\text{if } e \text{ then } e_1 \text{ else } e_2 \Downarrow v}\ \text{B-IFFALSE}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1\ op\ e_2 \Downarrow \overline{v_1\ op\ v_2}}\ \text{B-OP}$$

$$\frac{e_1 \Downarrow v_1 \qquad [v_1/x]e_2 \Downarrow v}{\text{let } x = e_1 \text{ in } e_2 \text{ end} \Downarrow v}\ \text{B-LET}$$

$$\frac{\text{x}}{fn\ x \Rightarrow e \Downarrow fn\ x \Rightarrow e}\ \text{B-FN}$$

```
let rec eval e = match
   e with
| Int _ -> e
| Bool _ -> e
| If (e , e1 , e2 ) ->
( match eval e with
| Bool true -> eval e1
| Bool false -> eval e2
| _ -> raise ( Stuck "
  guard is not a bool "
   ) )
```

Formal description advantages: Coverage: $\forall$ expressions $e \exists$ eval rule. Determinacy: if $e \Downarrow v_1$ and $e \Downarrow v_2$ then $v_1 = v_2$. Value soundness, if $e \Downarrow v$ then v is a value

Well-typed expressions Static type checking. Types classify exprs based on what they compute.

Types T ::= $int|bool|T_1 \to T_2|T_1 \times T_2|\alpha$

Typing in context Given assumptions $x_1 : T_1, \ldots, x_n : T_n$ -> Γ, expr e has type T. i.e. $\Gamma \vdash e : T$

$$\frac{}{\Gamma \vdash \text{true} : \text{bool}}\ \text{T-T}$$

$$\frac{}{\Gamma \vdash \text{n} : \text{int}}\ \text{T-NUM}$$

$$\frac{}{\Gamma \vdash \text{false} : \text{bool}}\ \text{T-F}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \qquad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}\ \text{T-PLUS}$$

$$\frac{\Gamma \vdash e_1 : \text{T} \qquad \Gamma \vdash e_2 : \text{T}}{\Gamma \vdash e_1 = e_2 : \text{bool}}\ \text{T-EQ}$$

$$\Gamma \vdash e : \text{bool} \qquad \Gamma \vdash e_1 : \text{T} \qquad \Gamma \vdash e_2 : \text{T}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \text{T-Var}$$

$$\frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 \text{ end} : T}\ \text{T}$$

$$\frac{\Gamma, x : \alpha \vdash e : T_2 \Rightarrow T/C}{\Gamma \vdash fn\ x \Rightarrow \alpha \to T/C}\ \text{T-FN}$$

Type checking e:T Given exp e and type T, check that e has type T. Are all substitution instances of e well-typed? **Type Inference** e:T Given expr e, infer its type T. Is some substitution instance of e well-typed?

Free Variables $FV(e)$ returns the set of free var names occurring in expr e. Defined inductively based on structure of expr e.

Substitution [e'/x]e Replace all **free** occurrences of x in expr e by e'

Type Inference How to get type of expr e given Γ? Analyze e following typing rules. Analyze e recursively, if lacking type info, introduce type var and possible constraints. T may contain type vars. Solve constraints to see if e is well-typed. Solving gives a type substitution σ.

- fn x => x + 1 type $\alpha \to int/\{\alpha = int\}$
- fn f => fn x => f (f x) type $\alpha \to \beta \to \alpha_1, \{\alpha = \beta \to a_0, \alpha = \alpha_0 \to \alpha_a\}$

Constraint Solving via Unification Two types unifiable if exists instantiation σ for type vars in both types such that they are syntactically equal. Simplify set of contrainsty by rewriting constraints and getting rid of useless ones