OCaml Cheat Sheet

by advitya via cheatography.com/20757/cs/3643/

The Environment Model Semantics

```
env :: n || n
env :: e1 + e2 || n
       if env :: e1 || n1 and env :: e2 || n2
       and n is the result of applying
        primitive operation + to n1 and n2
env :: (e1, e2) || (v1, v2)
       if env :: e1 || v1 and env :: e2 || v2
env :: fst e || v1
       if env :: e || (v1,v2)
env :: Left e || Left v
       if env :: e || v
env :: match e with Left x -> e1
             | Right y -> e2 || v1
     if env :: e || Left v and
        env +{x=v} :: e1 || v1
env :: let x = e1 in e2 || v2
     if env :: e1 || v1 and
      env +{x=v1} :: e2 || v2
env :: (fun x -> e) || <<fun x -> e, env>>
env :: e1 e2 || v
     if env :: e1 || <<fun x -> e, env'>>
     and env :: e2 || v2
     and env' + {x=v2} :: e || v
env :: let rec f x = e1 in e2 || v
   if env + {f = <<f, fun x -> e1, env>>}
   :: e2 || v
env :: e1 e2 || v
   if env :: e1 || <<f, fun x -> e, env'>>
```

The Environment Model Semantics (cont)

```
>   and env :: e2 || v2
  and env' + {x=v2,f=<<f, fun x -> e, env'>>}
  :: e || v
```

Environment Model Semantics Rule with Lexical Scoping

Technique to Generalize Folding

```
1. Write a recursive fold function that takes in one argument for each variant of the datatype.
2. That fold function matches against the datatype variants, calling itself recursively on any instance of the datatype that it encounters.
3. When a variant carries data of a type other than the datatype being folded, use the appropriate argument to fold to incorporate that data.
4. When a variant carries no data, use the appropriate argument to fold to produce an accumulator.
let rec fold_left (f : 'a -> 'b ->'a) (acc : 'a) (lst : 'b list): 'a =
    match lst with
       [] -> acc
   | x :: xs -> fold_left f (f acc x) xs
fold_left : 'a -> 'b -> 'a -> 'a -> 'b list -> 'a
let rec fold_right (f : 'a -> 'b -> 'b) (l : 'a list) (acc : 'b) : 'b =
    match l with
       [] -> acc
   | x :: xs -> f x (List.fold_right f xs acc)
fold_right: 'a -> 'b -> 'b -> 'a list -> 'b ->'b
```

Technique to Generalize Folding (cont)

```
> Example of Generalized fold:
type
 'a exprTree =
| Val of 'a
| Unop of ('a -> 'a) * 'a exprTree
| Binop of ('a -> 'a -> 'a) 'a exprTree 'a exprTree
let rec exprTree_fold (foldVal) (foldUnop) (foldBinop) = function
     | Val x -> foldVal x
     | Unop (f, t) -> foldUnop f (exprTree_fold foldVal foldUnop foldBinop t) | Binop (f, t1, t2) -> foldBinop f (exprTree_fold foldVal foldUnop foldBinop t1) (exprTree_fold foldVal foldUnop foldBinop t2)
  ;;
```

Generalized fold and List folding functions

Function Type Inferrence

Infer the type of functions from operations nested within the function. Start off by labeling all of the bindings and parameters with a random type Tn. And, then find out the type for each of them. Use patterns like the branches of an if and else statements are the same type and same goes for match statements.

Cheatography

OCaml Cheat Sheet

by advitya via cheatography.com/20757/cs/3643/

Function Type Inferrence (cont)

Points to note are that the failure ("blah") and Exception Not_found have type 'a (just something random), so they can be restricted to whatever the other type is in a match expression. Also, let rec f x= f x in f has type 'a -> 'b

Documenting Abstractions

A specification is a contract between an implementer of an abstraction and a client of an abstraction. An implementation satisfies a specification if it provides the described behavior.
Locality: abstraction can be understood without needing to examine implementation
Modifiability: abstraction can be reimplemented without changing implementation of other abstractions
Good Specs:
Sufficiently restrictive: rule out implementations that wouldn't be useful to clients
Sufficiently general: do not rule out implementations that would be useful to clients
Sufficiently clear: easy for clients to understand behavior

Documenting Abstractions (cont)

Abstraction function (AF) captures designer's intent in choosing a particular representation of a data abstraction. Not actually OCaml function but an abstract function. Maps concrete values to abstract values. Think about Set example, where implementer sees Set as 'a list [1;2] but user sees it as {1,2}.
Many-to-one: many values of concrete type can map to same value of abstract type.
[1;2] & [2;1] both map to {1,2}
Partial: some values of concrete type do not map to any value of abstract type
[1;1;2] because no duplicates
opA(AF(c)) = AF(opC(c)). AF commutes with op!
You might write:
– Abstraction Function: comment – AF: comment
– comment
Representation invariant characterizes which concrete values are valid and which are invalid.
-Valid concrete values will be mapped by AF to abstract values
-Invalid concrete value will not be mapped by AF to abstract values

Substitution Model of Evaluation

```
e1 + e2 --> e1' + e2
       if e1 --> e1'
v1 + e2 --> v1 + e2'
       if e2 --> e2'
n1 + n2 --> n3
        where n3 is the result
of applying primitive operation
+
       to n1 and n2
(e1, e2) --> (e1', e2)
       if e1 --> e1'
(v1, e2) --> (v1, e2')
       if e2 --> e2'
fst (v1,v2) --> v1
Left e --> Left e'
       if e --> e'
match e with Left x -> e1 |
Right y -> e2
--> match e' with Left x -> e1 |
Right y -> e2
       if e --> e'
match Left v with Left x -> e1 |
Right y -> e2
--> e1{v/x}
match Right v with Left x -> e1
| Right y -> e2
--> e2{v/y}
let x = e1 in e2 --> let x = e1'
in e2
       if e1 --> e1'
let x = v in e2 --> e2{v/x}
e1 e2 --> e1' e2
       if e1 --> e1'
v e2 --> v e2'
```

Substitution Model of Evaluation (cont)

> if e2 --> e2'

Capture Avoiding Substitution
(fun x -> e) v2 --> e{v2/x}
(Left e'){e/x} = Left e'{e/x}
(Right e'){e/x} = Right e'{e/x}
(match e' with Left y -> e1 | Right z -> e2){e/x}
= match e'{e/x} with Left y -> e1{e/x} | Right z -> e2{e/x}
(match e' with Left x -> e1 | Right z -> e2){e/x}
= match e'{e/x} with Left x -> e1 | Right z -> e2{e/x}
(match e' with Left y -> e1 | Right x -> e2){e/x}
= match e'{e/x} with Left y -> e1{e/x} | Right x -> e2
(match e' with Left x -> e1 | Right x -> e2){e/x}
= match e'{e/x} with Left x -> e1 | Right x -> e2
(let x = e1 in e2){v/x} = let x = e1{v/x} in e2
(let y = e1 in e2){v/x} = let y = e1{v/x} in e2{v/x}
(e1,e2){e/x} = (e1{e/x}, e2{e/x})
(fst e'){e/x} = fst e'{e/x}

Substitution Model Evaluation- Capture-avoiding substitution

Example Module & Functor example

Start off with this functor for Intervals.

```
module Make_interval :
    functor (Endpoint : Comparable) ->
        sig
            type t = Interval of Endpoint.t * Endpoint.t | Empty
            val create : Endpoint.t -> Endpoint.t -> t
            val is_empty : t -> bool
            val contains : t -> Endpoint.t -> bool
            val intersect : t -> t -> t
        end
```

Now, the functor does not have an abstract type. Because, the user can see the type in the functor. So, we have to hid that type t impelementation. There's a problem with Make_interval. The invariant is enforced by the create function, but because Interval.t is not abstract, we can bypass the create function. So you do something like this with sharing constraints: :

```
module Make_interval(Endpoint : Comparable) :
Interval_intf with type endpoint = int struct
        type endpoint = Endpoint.t
        type t = | Interval of Endpoint.t * Endpoint.t
                 | Empty
```

Modules Signatures, Structures and Functors

Basically, signature is the interface that we must follow for a certain module. The Structure of a module is the implementation of the given signature of the module. Furthermore, the functors go ahead and parameterize modules: that is, they will take in a module or multiple modules as inputs and return a new module that is parameterized with the input module. So, suppose you have a given Set module and you want this module to applicable to all types not only ints. So, you will need the notion of equality in your module, but this notion of equality is different between Ints and Strings, so you can parameterize by having a functor that has a type sig of EQUAL as its input. With functors remember to do the sharing constraints..

Matching Mechanics & Type Declarations

A type synonym is a new kind of declaration. The type and the name are interchangeable in every way.
Matching: Given a pattern p and a value v, decide
– Does pattern match value?
– If so, what variable bindings are introduced?
If p is a variable x, the match succeeds and x is bound to v.
If p is _, the match succeeds and no bindings are introduced

Matching Mechanics & Type Declarations (cont)

If p is a constant c, the match succeeds if v is c. No bindings are introduced

If p is C p1, the match succeeds if v is C v1 (i.e., the same constructor) and p1 matches v1. The bindings are the bindings from the sub-match.

If p is (p1,..., pn) and v is (v1,..., vn), the match succeeds if p1 matches v1, and ..., and pn matches vn. The bindings are the union of all bindings from the sub-matches.

1. If Expressions are just pattern matches
2. Lists and options are just datatypes
3. Let expressions are also pattern matches.
4. A function argument can also be a pattern.

Type Checking Rules

```
Syntax:
e1 + e2
Type-c hec king:
If e1 and e2 have type int, then
e1 + e2 has type int
Syntax: e1 < e2
Type-c hec king: if e1 has type
int and e2 has type
int then e1<e2 has type bool
Syntax: if e1 then e2 else e3
Type-c hec king: if e1 has type
bool and, for some type t, both
e2 and e3 have type t, then if
e1 then e2 else e3 has type t
Simplified syntax:
let x = e1 in e2
```

Type Checking Rules (cont)

> Type-checking:

If e1:t1, and if e2:t2 under the assumption that

x:t1, then let x = e1 in e2 : t2

Syntax: e0 (e1,...,en)

Type-checking:

If: e0 has some type (t1 ... tn) -> t

and e1 has type t1, ..., en has type tn

Then e0 (e1,...,en) has type t

Syntax: {f1=e1;...;fn=en}

Type-checking:

If e1:t1ande2:t2 and ... en:tn, and if t is a declared type of the form {f1:t1, ..., fn:tn} , then{f1 = e1; ...; fn = en}:t

Syntax: e.f

Type-checking:

If e:t1 and if t1 is a declared type of the form {f:t2, ...} , then e.f: t2

None has type ‘a option

– much like [] has type ‘a list – None is a value

Some e :t option ife:t

– much like e::[] has type t list if e:t – If e-->v then Some e-->Some v

Note- Datatype VS Records Table

Type Checking Rules part of Semantics

Key Points about Modules

Other key points with modules:

1. Difference between include and open is that include just sort of extends a module/signature when its called. In general, opening a module adds the contents of that module to the environment that the compiler looks at to find the definition of various identifiers.While opening a module affects the environment used to search for identifiers, including a module is a way of actually adding new identifiers to a module proper. The difference between include and open is that we've done more than change how identifiers are searched for: we've changed what's in the module. Opening modules is usuallly not a good thing in top level as you are getting rid of the advantage of a new namespace and if you want to do it, do it locally.. 2. Don't expose the type of module especially in the signature, it is smart to hid from your user as they may abuse your invariant and don't have any idea on the implementation. So, you can also change the implementation without them knowing.

Key Points about Modules (cont)

3. We can also use sharing constraints in the context of a functor. The most common use case is where you want to expose that some of the types of the module being generated by the functor are related to the types in the module fed to the functor

Data Types VS Record VS Tuple

	Declare	Build/Construct	Access/Destruct
DataType	type	Constructor name	Pattern matching with match
Record	type	Record expression with {...}	Pattern matching with let OR field selection with dot operator .
Tuple	N/A	Tuple expression with (...)	Pattern matching with let OR fst or snd

Records are used to store this AND that. Datatypes represent this OR that. Also, a tuple is just a record with its fields referred to by position, where as with records it is by name.

Algebraic Dataypes of form <Datatype: Name Student> of String

Dynamic VS Lexical Scoping

Rule of dynamic scope: The body of a function is evaluated in the current dynamic environment at the time the function is called, not the old dynamic environment that existed at the time the function was defined.

Dynamic VS Lexical Scoping (cont)

Rule of lexical scope: The body of a function is evaluated in the old dynamic environment that existed at the time the function was defined, not the current environment when the function is called.

Functions as First Class Citizens

Functions are values
Can use them anywhere we use values
First-class citizens of language, afforded all the “rights” of any other values
– Functions can take functions as arguments – Functions can return functions as results
...functions can be higher-order
Map: let rec map f xs = match xs with
[] -> []
| x::xs’ -> (f x)::(map f xs’)
map: ('a->'b)->'alist->'blist
Filter, Map, folds are iterators basically. They can iterate through structures just like normal loops can.