OCaml Modules

Part 1: Simple Structures

Speaker: David Walker

COS 326

Princeton University

The Reality of Development

We rarely know the *right* algorithms or the *right* data structures when we start a design project.

– When implementing a search engine, what data structures and algorithms should you use to build the index? To build the query evaluator?

Reality is that *we often have to go back and change our code*, once we've built a prototype.

– Often, we don't even know what the *user wants* (requirements) until they see a prototype.
– Often, we don't know where the *performance problems* are until we can run the software on realistic test cases.
– Sometimes we just want to change the design -- come up with *simpler* algorithms, architecture later in the design process

Engineering for Change

Given that we know the software will change, how can we write the code so that doing the changes will be easier?

The primary trick: use *data and algorithm abstraction*.

- *Don't* code in terms of *concrete representations* that the language provides.
- *Do* code with *high-level abstractions* in mind that fit the problem domain.
- Implement the abstractions using a *well-defined interface*.
- Swap in *different implementations* for the abstractions.
- *Parallelize* the development process.

Example

Goal: Implement a query engine.

Requirements: Need a scalable *dictionary* (a.k.a. index)

- maps words to *set* of URLs for the pages on which words appear.
- want the index so that we can efficiently satisfy queries
  - e.g., all links to pages that contain “Dave” and “Jill”.

Wrong way to think about this:

- Aha! A *list* of pairs of a word and a *list* of URLs.
- We can look up “Dave” and “Jill” in the *list* to get back a *list* of URLs.

Example

```
type query =
  Word of string
| And of query * query
| Or of query * query

type index = (string * (url list)) list

let rec eval(q:query)(h:index) : u
  match q with
   | Word x ->
       let (_,urls) = List.find                          h in
       urls
   | And (q1,q2) ->
       merge_lists (eval q1 h) (e
   | Or (q1,q2) ->
       (eval q1 h) @ (eval q2 h)
```

Concrete data structure chosen as index representation

merge expects to be passed sorted lists.

Oops!

Example

```
type query =
  Word of string
| And of query * query
| Or of query * query

type index = string (url list) hashtable

let rec eval(q:query)(h:index) : url list =
  match q with
  | Word x ->
       let i = hash_string x in
       let l = Array.get h [i] in
       let urls = assoc_list_find l x in
       urls
  | And (q1,q2) -> ...
  | Or (q1,q2) -> ...
```

I find out there's a better data structure to use

A Better Way

```
type query =
  Word of string
| And of query * query
| Or of query * query

type index = (string, Url.t Set.t) Dict.t

let rec eval(q:query)(d:index) : Url.t Set.t
  match q with
   | Word x -> Dict.lookup d x
   | And (q1,q2) -> Set.intersect (eval q1 h) (eval q2 h)
   | Or (q1,q2) -> Set.union (eval q1 h) (eval q2 h)
```

The problem domain talked about an abstract type of *dictionari*

Once we've written the client, we know what operations we need on these abstract types.

Later on, when we find out linked lists aren't so good for sets, we can replace them with balanced trees.

So we can define an interface, and send a pal off to implement the *abstract types* dictionary and set.

Abstract Data Types

Barbara Liskov
Assistant Professor, MIT
1973

Invented CLU language
that enforced data abstraction

Barbara Liskov
Professor, MIT
Turing Award 2008

"For contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing."

ADTS IN OCAML

Building Abstract Types in OCaml

OCaml has mechanisms for building new abstract data types:

- ***signature***: an interface.
  - specifies the abstract type(s) without specifying their implementation
  - specifies the set of operations on the abstract types
- ***structure***: an implementation.
  - a collection of type and value definitions
  - notion of an implementation matching or satisfying an interface
    - gives rise to a notion of subtyping
- ***functor***: a parameterized module
  - really, a function from modules to modules
  - allows us to factor out and re-use modules

The Abstraction Barrier

Rule of thumb: Use the language to enforce the abstraction barrier.

- Reveal little information about *how* something is implemented
- Provide maximum flexibility for change moving forward.
- *Murphy's Law: What is not enforced, will be broken*

But rules are meant to broken: Exercise judgement.

- may want to reveal more information for debugging purposes
  - eg: conversion to string so you can print things out

ML gives you precise control over how much of the type is left abstract

- different amounts of information can be revealed in different contexts
- type checker helps you detect violations of the abstraction barrier

Simple Modules

Recall assigment #2:

query.ml

```
type movie = { ... }

let sort_by_studio  = ...
let sort_by_year = ...
```

main.ml

```
open Io
open Query

let main () = ... sort_by_studio ...

let _ = main ()
```

Simple Modules

Recall assigment #2:

query.ml

```
type movie = { ... }

let sort_by_studio  = ...
let sort_by_year = ...
```

main.ml

```
open Io
open Query

let main () = ... sort_by_studio ...

let _ = main ()
```

Each .ml file actually defines an ML module.

Convention: the file foo.ml or Foo.ml defines the module named Foo.

Simple Modules

Recall assigment #2:

open gives
direct access to
module components

query.ml

```
type movie = { ... }

let sort_by_studio  = ...
let sort_by_year = ...
```

main.ml

```
open Io
open Query

let main () = ... sort_by_studio ...

let _ = main ()
```

Simple Modules

Recall assigment #2:

redacted

query.ml

```
type movie = { ... }

let sort_by_studio  = ...
let sort_by_year = ...
```

main.ml

```
open Io
```
~~open Query~~

```
let main () =
  ... Query.sort_by_studio ...
```

Can refer to module components using dot notation

Simple Modules

query.ml

```
type movie = { ... }

let sort_by_studio  = ...
let sort_by_year = ...
```

main.ml

```
open Io
~~open Query~~

let main () =
  ... Query.sort_by_studio ...
```

query.mli

```
type movie

val sort_by_studio : movie list -> movie list
val sort_by_year : movie list -> movie list
```

You can add interface files (.mli) (also called *signatures* in ML)

These interfaces can hide module components or render types abstract.

Simple Modules

query.ml

type movie = { ... }

let sort_by_studio = ...
let sort_by_year = ...

main.ml

open Io
~~open Query~~

let main () =
  ... Query.sort_by_studio ...

query.mli

type movie

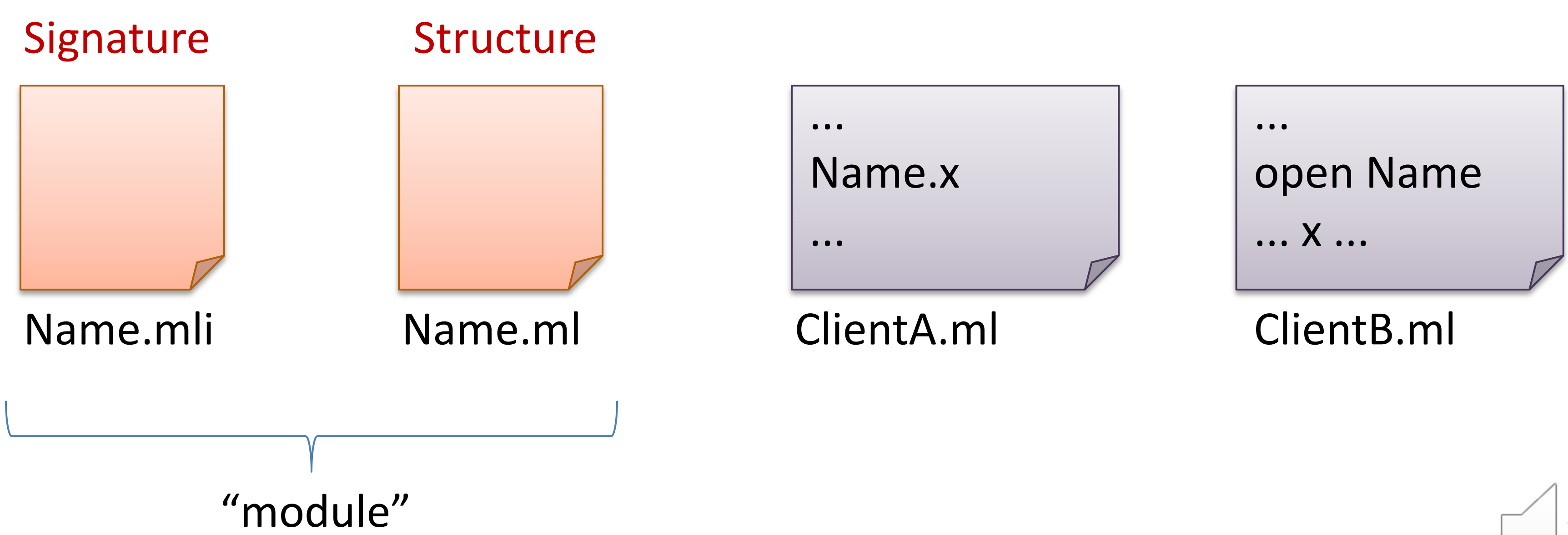
val sort_by_studio : movie list -> movie list
val sort_by_year : movie list -> movie list

If you have no signature file, then the default signature is used: all components are fully visible to clients.

Simple Modules

Simple summary:

- file Name.ml is a *structure* implementing a module named Name
- file Name.mli is a *signature* for the module named Name
  - if there is no file Name.mli, OCaml infers the default signature



At first glance: OCaml modules = C modules?

C has:

- .h files (signatures) similar to .mli files?
- .c files (structures) similar to .ml files?

But ML also has:

- tighter control over type abstraction
  - define abstract, transparent or translucent types in signatures
    - i.e.: give none, all or some of the type information to clients
- more structure
  - modules can be defined within modules
  - i.e.: signatures and structures can be defined inside files
- more reuse
  - multiple modules can satisfy the same interface
  - the same module can satisfy multiple interfaces
  - modules take other modules as arguments (functors)
- fancy features: dynamic, first class modules

Signature Definitions Inside Files

```
module type INT_STACK =
  sig
    type stack
    val empty : unit -> stack
    val push  : int -> stack -> stack
    val is_empty : stack -> bool
    val pop : stack -> stack
    val top : stack -> int option
  end
```

Signature Definitions Inside Files

```
module type INT_STACK =
  sig
    type stack
    val empty : unit -> stack
    val push  : int -> stack -> stack
    val is_empty : stack -> bool
    val pop : stack -> stack option
    val top : stack -> int option
  end
```

empty and push are abstract ***constructors***: functions that build our abstract type.

Signature Definitions Inside Files

```
module type INT_STACK =
  sig
    type stack
    val empty : unit -> stack
    val push  : int -> stack -> stack
    val is_empty : stack -> bool
    val pop : stack -> stack
    val top : stack ->
  end
```

is_empty is an ***observer*** – useful for determining properties of the ADT.

Signature Definitions Inside Files

```
module type INT_STACK =
  sig
    type stack
    val empty : unit -> stack
    val push  : int -> stack -> stack
    val is_empty : stack -> bool
    val pop : stack -> stack
    val top : stack -> int option
  end
```

pop is sometimes called a ***mutator*** (though it doesn't really change the input)

Signature Definitions Inside Files

```
module type INT_STACK =
  sig
    type stack
    val empty : unit -> stack
    val push  : int -> stack -> stack
    val is_empty : stack -> bool
    val pop : stack -> stack
    val top : stack -> int option
  end
```

top is also an *observer*, in this functional setting since it doesn't change the stack.

Put comments in your signature!

```
module type INT_STACK =
  sig
    type stack

    (* create an empty stack *)
    val empty : unit -> stack

    (* push an element on the top of the stack *)
    val push  : int -> stack -> stack

    (* returns true iff the stack is empty *)
    val is_empty : stack -> bool

    (* pops top element off the stack;
       returns empty stack if the stack is empty *)
    val pop : stack -> stack

    (* returns the top element of the stack; returns
        None if the stack is empty *)
    val top : stack -> int option
  end
```

Signature Comments

Signature comments are for clients of the module

- explain what each function should do
  - how it manipulates abstract values (stacks)
- ***not*** how it manipulates concrete values
- don't reveal implementation details that should be hidden behind the abstraction

Don't copy signature comments into your structures

- your comments will get out of date in one place or the other
- an extension of the general rule: don't copy code

Place implementation comments inside your structure

- comments about implementation invariants hidden from client
- comments about helper functions

https://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html

Previous Up Next

Module List

```
module List: sig .. end
```

List operations.

Some functions are flagged as not tail-recursive. A tail-recursive function uses constant stack space, while a non-tail-recursive function uses stack space proportional to the length of its list argument, which can be a problem with very long lists. When the function takes several list arguments, an approximate formula giving stack usage (in some unspecified constant unit) is shown in parentheses.

The above considerations can usually be ignored if your lists are not longer than about 10000 elements.

```
val length : 'a list -> int
```

Return the length (number of elements) of the given list.

```
val compare_lengths : 'a list -> 'b list -> int
```

Compare the lengths of two lists. `compare_lengths l1 l2` is equivalent to `compare (length l1) (length l2)`, except that the computation stops after itering on the shortest list.
Since 4.05.0

```
val compare_length_with : 'a list -> int -> int
```

Compare the length of a list to an integer. `compare_length_with l n` is equivalent to `compare (length l) n`, except that the computation stops after at most `n` iterations on the list.
Since 4.05.0

```
val cons : 'a -> 'a list -> 'a list
```

`cons x xs` is `x :: xs`
Since 4.03.0

```
val hd : 'a list -> 'a
```

Return the first element of the given list. Raise `Failure "hd"` if the list is empty.

```
val tl : 'a list -> 'a list
```

Return the given list without its first element. Raise `Failure "tl"` if the list is empty.

```
val nth : 'a list -> int -> 'a
```

Return the `n`-th element of the given list. The first element (head of the list) is at position 0. Raise `Failure "nth"` if the list is too short. Raise `Invalid_argument "List.nth"` if `n` is negative.

```
val nth_opt : 'a list -> int -> 'a option
```

Return the `n`-th element of the given list. The first element (head of the list) is at position 0. Return `None` if the list is too short. Raise `Invalid_argument "List.nth"` if `n` is negative.
Since 4.05

```
val rev : 'a list -> 'a list
```

Example Structure Inside a File

```
module ListIntStack : INT_STACK =
  struct
    type stack = int list
    let empty () : stack = []
    let push (i:int) (s:stack) : stack = i::s
    let is_empty (s:stack) =
      match s with
       | [] -> true
       | _::_ -> false
    let pop (s:stack) : stack =
      match s with
       | [] -> []
       | _::t -> t
    let top (s:stack) : int option =
      match s with
       | [] -> None
       | h::_ -> Some h
  end
```

Example Structure Inside a File

```
module ListIntStack : INT_STACK =
  struct
    type stack = int list
    let empty () : stack = []
    let push (i:int) (s:stack) = i:
    let is_empty (s:stack) =
      match s with
       | [] -> true
       | _::_ -> false
    let pop (s:stack) : stack =
      match s with
       | [] -> []
       | _::t -> t
    let top (s:stack) : int option =
      match s with
       | [] -> None
       | h::_ -> Some h
  end
```

Inside the module, we know the ***concrete type*** used to implement the abstract type.

Example Structure Inside a File

```
module ListIntStack : INT_STACK =
  struct
    type stack = int list
    let empty () : stack = []
    let push (i:int) (s:stack) = i
    let is_empty (s:stack) =
      match s with
       | [] -> true
       | _::_ -> false
    let pop (s:stack) : stack =
      match s with
       | [] -> []
       | _::t -> t
    let top (s:stack) : int option =
      match s with
       | [] -> None
       | h::_ -> Some h
  end
```

But by giving the module the INT_STACK interface, which does not reveal how stacks are being represented, we prevent code outside the module from knowing stacks are lists.

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let x = ListIntStack.top s2
```

An Example Client

```
module ListIntStack : INT_STACK =
  struct
    …
  end

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push 4 s1
let x = ListIntStack.top s2
```

```
s0 : ListIntStack.stack
s1 : ListIntStack.stack
s2 : ListIntStack.stack
```

An Example Client

```
module type INT_STACK =
  sig
    type stack
    val push  : int -> stack -> stack
    …
  end
module ListIntStack : INT_STACK

let s0 = ListIntStack.empty ()
let s1 = ListIntStack.push 3 s0
let s2 = ListIntStack.push   s1
let _ = List.rev s2
Error: This expression has type stack but an
expression was expected of type 'a list.
```

Notice that the client is not allowed to know that the stack is a list.

Example Structure

```
module ListIntStack (* : INT_STACK *) =
  struct
    type stack = int list
    let empty () : stack = []
    let push (i:int) (s:stack) = i::s
    let is_empty (s:stack) =
      match s with
       | [ ] -> true
       | _::_ -> false
    exception EmptyStack
    let pop (s:stack) =
      match s with
       | [] -> []
       | _::t -> t
    let top (s:stack) =
      match s with
       | [] -> None
       | h::_ -> Some h
  end
```

The Client without the Signature

```
module ListIntStack (* : INT_STACK *) =
  struct
    …
  end

let s = ListIntStack.empty()
let s1 = ListIntStack.push 3 s
let s2 = ListIntStack.push 4 s1

…
let x = List.rev s2
x : int list = [3; 4]
```

If we don't seal the module with a signature, the client can know that stacks are lists.

Example Structure

```
module type INT_STACK =
  sig
    type stack
    ...

    val inspect : stack -> int list
    val run_unit_tests : unit -> unit

  end


module ListIntStack : INT_STACK =
  struct
    type stack = int list
    ...

    let inspect (s:stack) : int list = s
    let run_unit_tests () : unit = ...
  end
```

Another technique:

Add testing components to your signature.

Or have 2 signatures, one for testing and one for the rest of the code)

Summary

ML modules support development of abstract data types

- client programs help define the operations needed
  - it is often useful to write them first
- signatures (ie, interfaces, .mli files) specify:
  - abstract types
  - names of operations and their types
  - names of abstract values
- structures (ie, implementations, .ml files) provide:
  - the concrete implementation types
  - the function implementations
  - the values to implement signatures
- when a signature is omitted, OCaml assumes the *default signature*, which allows clients to see all implementation details
  - over time, clients are going to depend upon details you don't want them to, making it hard to change ADT implementations