Modules and Abstract Data Types

COS 326
David Walker
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?

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*.

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) : url list =
  match q with
  | Word x ->
       let (_,urls) = List.find (fun (w,urls) -> w = x) in
       urls
  | And (q1,q2) ->
       merge_lists (eval q1 h) (eval q2 h)
  | Or (q1,q2) ->
       (eval q1 h) @ (eval q2 h)
```

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                 w = x) in
       urls
  | And (q1,q2) ->
       merge_lists (eval q1 h) (eval q2 h)
  | Or (q1,q2) ->
       (eval q1 h) @ (eval q2 h)
```

merge expects to be passed sorted lists.

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                        in
      urls
  | And (q1,q2) ->
      merge_lists (eval q1 h) (e
  | Or (q1,q2) ->
      (eval q1 h) @ (eval q2 h)
```

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 h in
      let l = Array.get h [i] in
      let urls = assoc_list_find ll x in
      urls
  | And (q1,q2) -> ...
  | Or (q1,q2) -> ...
```

I find out there's a better hash-table implementation

A Better Way

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

type index = string url_set dictionary ;;

let rec eval(q:query)(d:index) : url_set =
  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)
```

A Better Way

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

type index = string url_set dictionary ;;

let rec eval(q:query)(d:index) : url_set =
  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 *dictionaries* and *sets of URLs*.

A Better Way

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

type index = string url_set dictionar

let rec eval(q:query)(d:index) : url_s
  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 ***dictionaries*** and ***sets of URLs***.

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

A Better Way

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

type index = string url_set dictionar

let rec eval(q:query)(d:index) : url_s
  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 *dictionaries* and *sets of URLs*.

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.

A Better Way

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

type index = string url_set dictionar

let rec eval(q:query)(d:index) : url_s
  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 *dictionaries* and *sets of URLs*.

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

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

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

Building Abstract Types in Ocaml

- We can use the module system of Ocaml to build 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 sub-typing
  - ***functor***: a parameterized module
    - really, a function from modules to modules
    - allows us to factor out and re-use modules

functor kitten

Example Signature

```
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
```

Example Signature

```
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.

Example Signature

```
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 -
  end
```

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

Example Signature

```
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
```

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

Example Signature

```
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
```

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

A Better 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 None
       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
  end
```

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
    let pop (s:stack) =
      match s with
       | [] -> None
       | _::t -> Some t
    let top (s:stack) =
      match s with
       | [] -> None
       | h::_ -> Some h
  end
```

Example Structure

```
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) =
      match s with
       | [] -> None
       | _::t -> Some t
    let top (s:stack) =
      match s with
       | [] -> None
       | h::_ -> Some h
  end
```

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

Example Structure

```
module ListIntStack : INT_STACK =
  struct
    type stack = int list
    let empty () : stack = []
    let push (i:int) (s:stack) =
    let is_empty (s:stack) =
      match s with
        | [] -> true
        | _::_ -> false
    let pop (s:stack) =
      match s with
        | [] -> None
        | _::t -> Some t
    let top (s:stack) =
      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;;
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;;
ListIntStack.top s2 ;;
```

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

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;;
ListIntStack.top s2;;
- : option int = Some 4
```

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;;
ListIntStack.top s2 ;;
- : option int = Some 4
ListIntStack.top (ListIntStack.pop s2) ;;
- : option int = Some 3
```

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;;
ListIntStack.top s2 ;;
- : option int = Some 4
ListIntStack.top (ListIntStack.pop s2) ;;
- : option int = Some 3
open ListIntStack ;;
```

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;;
ListIntStack.top s2 ;;
- : option int = Some 4
ListIntStack.top (ListIntStack.pop s2) ;;
- : option int = Some 3
open ListIntStack ;;
top (pop (pop s2)) ;;
- : option int = None
```

An Example Client

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

module ListIntStack : INT_STACK

let s2 = ListIntStack.push 4 s1;;
…
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
       | [] -> raise EmptyStack
       | _::t -> t
    let top (s:stack) =
      match s with
       | [] -> raise EmptyStack
       | h::_ -> h
  end
```

Note that when you are debugging, you may want to comment out the signature ascription so that you can access the contents of the module.

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;;

…
List.rev s2 ;;
- : 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 ListIntStack : INT_STACK =
  struct
    type stack = int list
    let empty () : stack = []
    let push (i:int) (s:stack) =
    let is_empty (s:stack) =
      match s with
       | [ ] -> true
       | _::_ -> false
    exception EmptyStack
    let pop (s:stack) =
      match s with
       | [] -> raise EmptyStack
       | _::t -> t
    let top (s:stack) =
      match s with
       | [] -> raise EmptyStack
       | h::_ -> h
  end
```

When you put the signature on here, you are restricting client access to the information in the signature (which does *not* reveal that stack = int list.) So clients can *only* use the stack operations on a stack value (not list operations.)

Summary

- Design in terms of *abstract* types and algorithms.
  - think “sets” not “lists” or “arrays” or “trees”
  - think “document” not “strings”
- In OCaml, we have a powerful *module system* with:
  - *signatures* (interfaces)
  - *structures* (implementations)
  - *functors* (functions from modules to modules)
- We can use the module system
  - to support name spaces
  - to hide information (concrete types, local value definitions)
  - to make it easy to reuse code (via parameterization, functors)

END