OCaml Modules
Part 2: Design Choices

Speaker: David Walker

COS 326

Princeton University

A Stack Interface

```
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
```

Interface design

```
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 -> st

    (* returns true iff                    y *)
    val is_empty : stack -> boo

    (* 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
```

Is this a good idea?

Design choices

```
sig
  type stack
  (* pops top element;
     returns empty if empty
   *)
   val pop : stack -> stack
end
```

```
sig
  type stack
  (* pops top element;
     returns arbitrary stack
     if empty *)
   val pop : stack -> stack
end
```

```
sig
  type stack
  (* pops top element;
     returns option *)
   val pop:
          stack -> stack option
end
```

```
sig
  type stack
  exception EmptyStack
  (* pops top element;
     raises EmptyStack if empty
   *)
   val pop : stack -> stack
end
```

Principle: Fail as Early as Possible

```
sig
 type stack
 (* pops top element;
     returns empty if empty
   *)
  val pop : stack -> stack
end
```

Returning a stack when pop receives a "bad" argument like the empty stack keeps the program going

But the fact that pop has received a "bad" stack may indicate something is busted in code calling stack

It is a good idea to find out *now* instead of having to track down a bug that arose through some *longer, more complex, more torturous path*

Principle: Nondeterminism is tough for clients

```
sig
 type stack
 (* pops top element;
    returns arbitrary stack
    if empty *)
  val pop : stack -> stack
end
```

Pro: This is easy for the module to implement.

Pro: It is easy for the module to change – no promises!

Con: It is tough for the client! Somewhere down the line, some other code is processing an arbitrary stack.

Unfortunately, C programs make this choice all the time or worse!

Principle: Recoverable Errors are Better

```
sig
  type stack
  (* pops top element;
     crashes the program
     if empty *)
   val pop : stack -> stack
end
```

It would be nice to have an error one can recover from (and analyze) as opposed to an error one cannot recover from ...

This interface imposes *a proof obligation* on the user: Prove you do not supply pop an empty stack. Programmers often forget to do their proofs! (Or make mistakes.)

Principle: More Non-determinism is Worse

```
sig
 type stack
 (* pops top element;
    Demonic choice between:
       - crashing
       - corrupting a different data structure
    if empty *)
  val pop : stack -> stack
end
```

A common C idiom that I don't approve of!

Very difficult to debug.

The source of an enormous number of security vulnerabilities.

Principle: Errors We Can Recover From Are Better

```
sig
 type stack
 exception EmptyStack
 (* pops top element;
     raises EmptyStack if empty
   *)
  val pop : stack -> stack
end
```

Exceptions with their handlers are errors we can recover from.

But sometimes callers forget to catch and handle exceptions ...

Principle: Require Programmers Recover for Safety

```
sig
 type stack
 (* pops top element;
    returns option *)
  val pop:
       stack -> stack option
end
```

Using an option has the advantage of forcing the caller to consider what to do on the “error” condition every time the function is called. *They can’t forget* to handle this situation.

But it imposes overhead on every call. At every call, one must make an explicit check for empty, even if you know it isn't empty!

Design choices

```
sig
 type stack
 (* pops top element;
    returns empty if empty
  *)
  val pop : stack -> stack
end
```

```
sig
 type stack
 (* pops top element;
    returns arbitrary stack
   if empty *)
  val pop : stack -> stack
```

All of these are reasonable design choices!

```
sig
 type stack
 (* pops top element;
    returns option *)
  val pop:
       stack -> stack option
end
```

```
 type stack
 exception EmptyStack
 (* pops top element;
    raises EmptyStack if empty
   *)
  val pop : stack -> stack
end
```

Design choices

But use these two with extreme care

```
sig
  type stack
  (* pops top element;
     returns empty if empty
   *)
   val pop : stack -> stack
end
```

```
  type stack
  (* pops top element;
     returns arbitrary stack
   if empty *)
   val pop : stack -> stack
```

All of these are reasonable design choices!

```
sig
  type stack
  (* pops top element;
     returns option *)
   val pop:
         stack -> stack option
end
```

```
  type stack
  exception EmptyStack
  (* pops top element;
     raises EmptyStack if empty
   *)
   val pop : stack -> stack
end
```

The bottom two are more common. Options are the “safest.” They force consideration of the error condition every time.

Summary

Designing good interfaces is one of the most difficult aspects of software engineering.

It's important to understand your options*

And it takes a lot of experience and practice.

* Some pun intended.