ML Modules and Abstract Data Types

Hiding implementation details is the most important strategy for writing correct, robust, reusable software.

Topics:
- ML structures and signatures.
- Abstraction for robust library and client-library code.
- Abstraction for easy change.
- ADTs and functions as data.

Hiding with functions

*procedural abstraction*

Hiding implementation details is the most important strategy for writing correct, robust, reusable software.

Can you tell the difference?

```ml
- double x = x*2
fun double x = x+x
val y = 2
fun double x = x*y
fun double x = let fun help 0 y = y
               | help x y = help (x-1) (y+1)
               in help x x end
val it : int = 8
```

“Private” top-level functions would also be nice...
- share a “private” helper function

```ml
fun double x = x*2
fun double x = x+x
val y = 2
fun double x = x*y
fun double x = let fun help 0 y = y
               | help x y = help (x-1) (y+1)
               in help x x end
```

structure (module)

*namespace management and code organization*

```ml
structure MyMathLib =
struct
  fun fact 0 = 1
  | fact x = x * fact (x-1)
  val half_pi = Math.pi / 2
  fun doubler x = x * 2
end
outside:
val facts = List.map MyMathLib.fact [1,3,5,7,9]
```

signature

*type for a structure (module)*

List of bindings and their types:
- variables (incl. functions), type synonyms, datatypes, exceptions

Separate from specific structure.

```ml
signature MATHLIB =
sig
  val fact : int -> int
  val half_pi : real
  val doubler : int -> int
end
```

adapted from slides by Dan Grossman
Ascription (opaque – will ignore other kinds)

Ascribing a signature to a structure
  • Structure must have all bindings with types as declared in signature.

**signature MATHLIB =**

```plaintext
val fact : int -> int
val half_pi : real
val doubler : int -> int
```

**structure MyMathLib ::> MATHLIB =**

```plaintext
fun fact 0 = 1
| fact x = x * fact (x-1)
val half_pi = Math.pi / 2
fun doubler x = x * 2
```

Hiding with signatures

**MyMathLib.doubler** unbound (not in environment) outside module.

**signature MATHLIB2 =**

```plaintext
val fact : int -> int
val half_pi : real
```

**structure MyMathLib2 ::> MATHLIB2 =**

```plaintext
fun fact 0 = 1
| fact x = x * fact (x-1)
val half_pi = Math.pi / 2.0
fun doubler x = x * 2
```

Abstract Data Type
type of data and operations on it

Example: rational numbers supporting **add** and **toString**

```plaintext
structure Rational =
struct
datatype rational = Whole of int
| Frac of int*int
exception BadFrac
(* see adts.ml for full code *)
fun make_frac (x,y) = ...
fun add (z1,z2) = ...
fun toString r = ...
end
```

Library spec and invariants

**External properties [externally visible guarantees, up to library writer]**
  • Disallow denominators of 0
  • Return strings in reduced form (“4” not “4/1”, “3/2” not “9/6”)
  • No infinite loops or exceptions

**Implementation invariants [not in external specification]**
  • All denominators > 0
  • All rational values returned from functions are reduced

Signatures help **enforce** internal invariants.
More on invariants

Our code maintains (and relies) on invariants.

Maintain:
• `make_frac` disallows 0 denominator, removes negative denominator, and reduces result
• `add` assumes invariants on inputs, calls `reduce` if needed

Rely:
• `gcd` assumes its arguments are non-negative
• `add` uses math properties to avoid calling `reduce`
• `toString` assumes its argument is in reduced form

Problem: clients can violate invariants

Create values of type `Rational.rational` directly.

```
signature RATIONAL_OPEN =
sig
datatype rational = Whole of int
| Frac of int*int
... 
end
```

Rational.Frac(1,0)
Rational.Frac(3,-2)
Rational.Frac(9,6)

Solution: hide more!

*ADT must hide concrete type definition so clients cannot create invariant-violating values of type directly.*

This attempt goes too far: type `rational` is not known to exist.

```
signature RATIONAL_WRONG =
sig
exception BadFrac
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end
structure Rational :> RATIONAL_WRONG = ...
```
Abstract the type!  

(Really Big Deal!)  

Client can pass them around, but can manipulate them only through module.

Only operations on rational.

Type rational exists, but representation absolutely hidden.

Abstract Data Type

Abstract type of data + operations on it

Outside of implementation:

• Values of type rational can be created and manipulated only through ADT operations.
• Concrete representation of values of type rational is absolutely hidden.

Module controls all operations with rational, so client cannot violate invariants.

Abstract Data Types: two key tools

Powerful ways to use signatures for hiding:

1. Deny bindings exist.  
   Especially val bindings, fun bindings, constructors.

2. Make types abstract.  
   Clients cannot create or inspect values of the type directly.

A cute twist

In our example, exposing the Whole constructor is no problem

In SML we can expose it as a function since the datatype binding in the module does create such a function
• Still hiding the rest of the datatype
• Still does not allow using Whole as a pattern
Signature matching rules

structure Struct := SIG type-checks if and only if:
• Every non-abstract type in SIG is provided in Struct, as specified
• Every abstract type in SIG is provided in Struct in some way
  * Can be a datatype or a typesynonym
• Every val-binding in SIG is provided in Struct, possibly with a more general and/or less abstract internal type
  * 'a list -> int more general than string list -> int
  * example soon
• Every exception in SIG is provided in Struct.

Of course Struct can have more bindings (implicit in above rules)

Allow *different implementations* to be *equivalent*

A key purpose of abstraction:
• No client can tell which you are using
• Can improve/replace/choose implementations later
• Easier with more abstract signatures (reveal only what you must)

UnreducedRational in adts.sml.
• Same concrete datatype.
  * Different invariant: reduce fractions only in toString.
  * Equivalent under RATIONAL and RATIONAL_WHOLE, but not under RATIONAL_OPEN.

PairRational in adts.sml.
• Different concrete datatype.
  * Equivalent under RATIONAL and RATIONAL_WHOLE, but cannot ascribe RATIONAL_OPEN.

Some interesting details

• Internally make_frac has type int * int -> int * int, externally int * int -> rational
  * Client cannot tell if we return argument unchanged

• Internally Whole has type 'a -> 'a * int externally int -> rational
  * Specialize 'a to int
  * abstract int * int to rational
  * Type-checker just figures it out

• Whole cannot have types 'a -> int * int or 'a -> rational (must specialize all 'a uses)
Cannot mix and match module bindings

Modules with the same signatures still define different types

These do not type-check:

* Rational.toString(UnreducedRational.make_frac(9, 6))
* PairRational.toString(UnreducedRational.make_frac(9, 6))

Crucial for type system and module properties:

* Different modules have different internal invariants!
  * Different type definitions:
    * UnreducedRational.rational looks like Rational.rational, but clients and the type-checker do not know that
    * PairRational.rational is int*int not a datatype!

Will return and contrast with Object-Oriented techniques.

Implementing the SET signature

ListSet structure
Represent sets as lists.

Invariants?
* Duplicates?
* Ordering?

FunSet structure
Represent sets as function closures (!!!)

Set ADT (set.sml)

signature SET =
  sig
  type 'a t
  val empty : 'a t
  val singleton : 'a -> 'a t
  val fromList : 'a list -> 'a t
  val toList : 'a t -> 'a list
  val fromPred : ('a -> bool) -> 'a t
  val toPred : 'a t -> 'a -> bool
  val toString : ('a -> string) -> 'a t -> string
  val isEmpty : 'a t -> bool
  val member : 'a -> 'a t -> bool
  val insert : 'a -> 'a t -> 'a t
  val delete : 'a -> 'a t -> 'a t
  val union : 'a t -> 'a t -> 'a t
  val intersect : 'a t -> 'a t -> 'a t
  val diff : 'a t -> 'a t -> 'a t
end

Common idiom: if module provides one externally visible type, name it t
Then outside references are Set.t

Sets are fun!

Math: \{ x \mid x \mod 3 = 0 \}

SML: \( \text{fn} \ x \Rightarrow x \mod 3 = 0 \)

structure FunSet :> SET =
  sig
  type 'a t
  val empty = \_ => false
  fun singleton x = fn y => x=y
  fun member x set = set x
  fun insert x set = fn y => x=y |orelse set y ...
end

Are all set operations possible?