Overview of Modules and ADTs

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

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

Can you tell the difference?

- double 4;
- val it : int = 8

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

```
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

structure MyMathLib =
  struct
    fun fact 0 = 1
    | fact x = x * fact (x-1)
    val half_pi = Math.pi / 2
    fun doubler x = x * 2
    val twelve = doubler (fact 3)
  end
outside:
  val facts = List.map MyMathLib.fact [1,4,MyMathLib.doubler 3, MyMathLib.twelve]
```
signature 

type for a structure (module)

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

Separate from specific structure.

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

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 =
sig
  val fact : int -> int
  val half_pi : real
  val doubler : int -> int
  val twelve : int
end

structure MyMathLib :> MATHLIB =
struct
  fun fact 0 = 1
    | fact x = x * fact (x-1)
  val half_pi = Math.pi / 2.0
  fun doubler x = x * 2
  val twelve = doubler (fact 3)
end

Hiding with signatures

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

signature MATHLIB2 =
sig
  val fact : int -> int
  val half_pi : real
  val twelve : int
end

structure MyMathLib2 :> MATHLIB2 =
struct
  fun fact 0 = 1
    | fact x = x * fact (x-1)
  val half_pi = Math.pi / 2.0
  fun doubler x = x * 2
  fun twelve = doubler (fact 3)
end

Abstract Data Type

type of data and operations on it

Example: rational numbers supporting add and toString

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 (x1,x2) = ...
  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

A first signature

With what we know so far, this signature makes sense:
- Helper functions `gcd` and `reduce` not visible outside the module.

```sml
signature RATIONAL_CONCRETE =
  sig
    datatype rational = Whole of int |
      Frac of int*int
    exception BadFrac
    val make_frac : int * int -> rational
    val add : rational * rational -> rational
    val toString : rational -> string
  end
structure Rational :> RATIONAL_OPEN = ...
```

Problem: clients can violate invariants

Create values of type `Rational.rational` directly.

```sml
signature RATIONAL_CONCRETE =
  sig
    datatype rational = Whole of int |
      Frac of int*int
  end

Rational.Frac(1,0)
Rational.Frac(3,~2)
Rational.Frac(40,32)
```
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!)*

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

Success! (#3)

Only way to make 1st rational.

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

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

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.

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

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 RATIONAL_WHOLE =
sig
  type rational
  exception BadFrac
val Whole : int -> rational
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end
```

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 type synonym
• 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**.

```
structure PairRational =
struct
  type rational = int * int
  exception BadFrac
  fun make_frac (x,y) = ...
  fun Whole i = (i,1) (* for RATIONAL_WHOLE *)
  fun add ((a,b),(c,d)) = (a*d + b*c, b*d)
  fun toString r = ... (* reduce at last minute *)
end
```
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!
- ... and 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.

Set ADT (set.sml)

```sml
signature SET =
sig
  type 'a t
  val empty : 'a t
  val singleton : 'a -> 'a t
  val isEmpty : 'a t -> bool
  val size : 'a t -> int
  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 intersection : 'a t -> 'a t -> 'a t
  val difference : 'a t -> 'a t -> '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

end
```

Side Note: Equality Types

**Double-tick types** like `'a range over so-called equality types, which are types over which the polymorphic equality operator `=` is defined.

Sadly, the semantics of IEEE 754 floating point arithmetic standard prevents the `real` type from being an equality type. It includes Nan (not-a-number) values that represent the results of certain operations, such as subtracting positive infinity from itself. According to the IEEE standard, testing two Nan values for equality must return `false`, but that would break the **reflexivity property** that is required for an equality type (i.e., for any value `v` in an equality type, `v = v` must be `true`). See the examples below.

```sml
- val myNan = Real.posInf - Real.posInf; - myNan = myNan;
  - Real.isNan myNan;
  - val it = true : bool
- Real.== (myNan, myNan); - Real.compare (myNan, myNan);
  - uncaught exception Unordered
```

- `Real.compare (myNan, myNan)`
- `Error: operator and operand don't agree [equality type required]`
Implementing the SET signature

**ListSet structure**
Represent sets as unordered list.
- Invariant: no duplicates
- What about ordering? Can’t use it, since not part of signature!

**FunSet structure (PS7)**
Represent sets as predicate functions

**OperationTreeSet structure (PS7)**
Represent sets as trees of set operations

---

**Opening Modules**

- ListSet.isEmpty (ListSet.empty);
  val it = true : bool
- ListSet.size (ListSet.singleton 17);
  val it = 1 : int
- open ListSet;
  opening ListSet
  type 'a t
  val empty : 'a t
  ... lots of bindings omitted ...
  val toString : ('a -> string) -> 'a t -> string
- isEmpty (empty);
  val it = true : bool
- size (singleton 17);
  val it = 1 : int
- List.size (singleton 17);
  val it = 1 : int

**Testing ListSet**

- val s1 = fromList [1,2,1,2,3,2,3,1,4];
  val s1 = - : int t
- toList s1;
  val it = [4,3,2,1] : int list
- toString Int.toString s1;
  val it = "[4,3,2,1]" : string
- val s2 = fromList [3,4,5,6];
  val s2 = - : int t
- toList (union s1 s2);
  val it = [1,2,6,5,4,3] : int list
- toList (intersection s1 s2);
  val it = [4,3] : int list-
- toList (difference s1 s2);
  val it = [2,1] : int list-
- toList (difference s2 s1);
  val it = [6,5] : int list
FunSet (PS7)
Specifying sets with predicates is fun!

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

SML: fn x => x mod 3 = 0

structure FunSet :> SET =
struct
  type 'a t = 'a -> bool
  val empty = fn _ => false
  fun singleton x = fn y => x=y
  fun member x pred = pred x
  fun fromPred pred = pred
  ... Flesh out the rest in PS7 ...
end

• Which set operations are unimplementable in FunSet?
• Is fromPred implementable in ListSet?

OperationTreeSet (PS7)
(delete 4 (difference (union (insert 1 empty)
  (insert 4 empty))
  (union (insert 7 empty)
  (insert 4 empty)))
  (intersection (insert 1 empty)
  (union (insert 1 empty)
  (insert 6 empty))))