# SML Modules and Abstract Data Types (ADTs)

These slides are lightly edited versions of Ben Wood's Fall '15 slides, some of which are based on Dan Grossman's material from the University of Washington.



### CS251 Programming Languages Fall 2017, Lyn Turbak

Department of Computer Science Wellesley College

# 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.

SML Modules and ADTS 2

structure Name =
struct bindings end

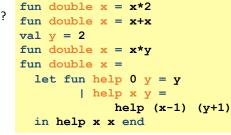
# Hiding with functions

procedural abstraction

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

SML Modules and ADTS 3

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 NAME = structure Name :> NAME = ascription sig binding-types end struct bindings end (opaque – will ignore other kinds) signature type for a structure (module) Ascribing a signature to a structure • Structure must have all bindings with types as declared in signature. List of bindings and their types: signature MATHLIB = variables (incl. functions), type synonyms, datatypes, exceptions sig val fact : int -> int Separate from specific structure. Real power: val half pi : real **Abstraction and Hiding** val doubler : int -> int val twelve : int signature MATHLIB = end sig val fact : int -> int structure MyMathLib :> MATHLIB = val half pi : real val doubler : int -> int struct fun fact 0 = 1val twelve : int | fact x = x \* fact (x-1)end val half pi = Math.pi / 2 fun doubler x = x \* 2val twelve = doubler (fact 3) end SML Modules and ADTS 5 SML Modules and ADTS 6

```
Hiding with signatures
```

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

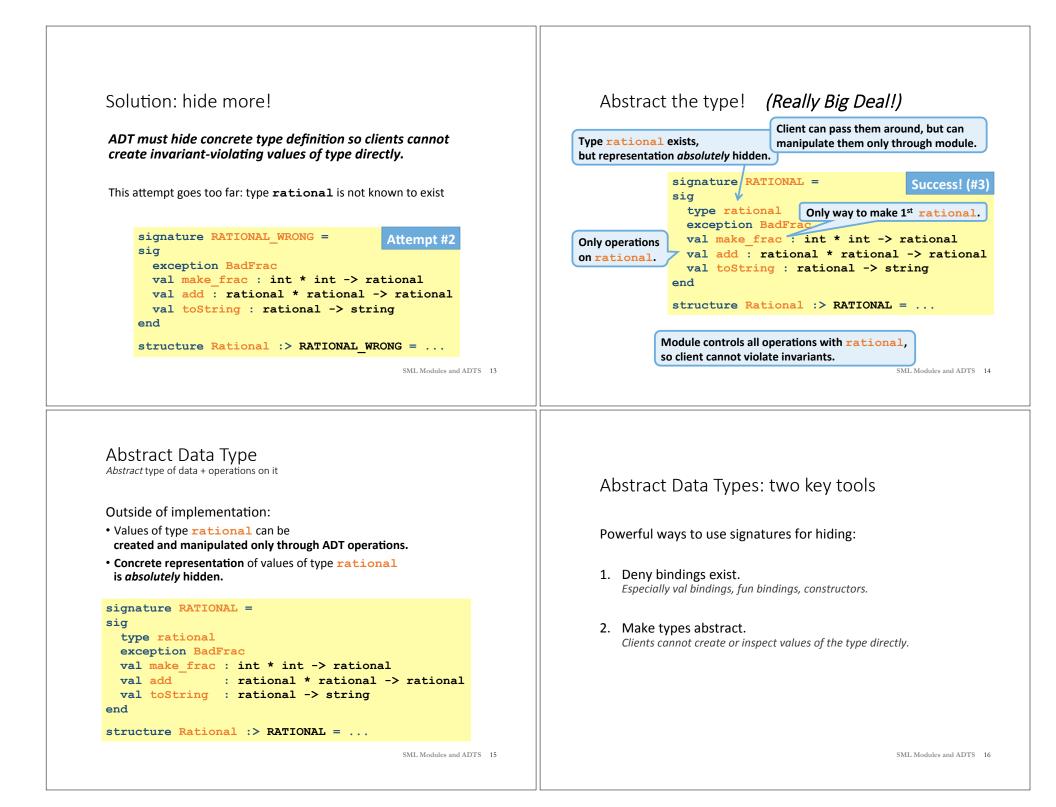
SML Modules and ADTS 7

Abstract Data Type type of data and operations on it

Example: rational numbers supporting add and toString

#### Library spec and invariants More on invariants External properties [externally visible guarantees, up to library writer] Our code maintains (and relies) on invariants. Disallow denominators of 0 • Return strings in reduced form ("4" not "4/1", "3/2" not "9/6") · No infinite loops or exceptions Maintain: • make frac disallows 0 denominator, removes negative denominator, and Implementation invariants [not in external specification] reduces result add assumes invariants on inputs, calls reduce if needed • All denominators > 0 • All rational values returned from functions are reduced Rely: • gcd assumes its arguments are non-negative Signatures help enforce internal invariants. add uses math properties to avoid calling reduce toString assumes its argument is in reduced form SML Modules and ADTS 9 SML Modules and ADTS 10 Problem: clients can violate invariants A first signature With what we know so far, this signature makes sense: • Helper functions gcd and reduce not visible outside the module. Create values of type **Rational.rational** directly. signature RATIONAL CONCRETE = signature RATIONAL CONCRETE = Attempt #1 sig sig datatype rational = Whole of int datatype rational = Whole of int | Frac of int\*int | Frac of int\*int . . . exception BadFrac end val make frac : int \* int -> rational val add : rational \* rational -> rational Rational.Frac(1,0) val toString : rational -> string Rational.Frac(3,~2) end Rational.Frac(40,32) structure Rational :> RATIONAL OPEN = ...

SML Modules and ADTS 11



## 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

SML Modules and ADTS 17

## 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)

SML Modules and ADTS 18

## PairRational (alternative concrete type)

```
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
```

## Allow *different implementations* to be *equivalent*

#### A key purpose of abstraction:

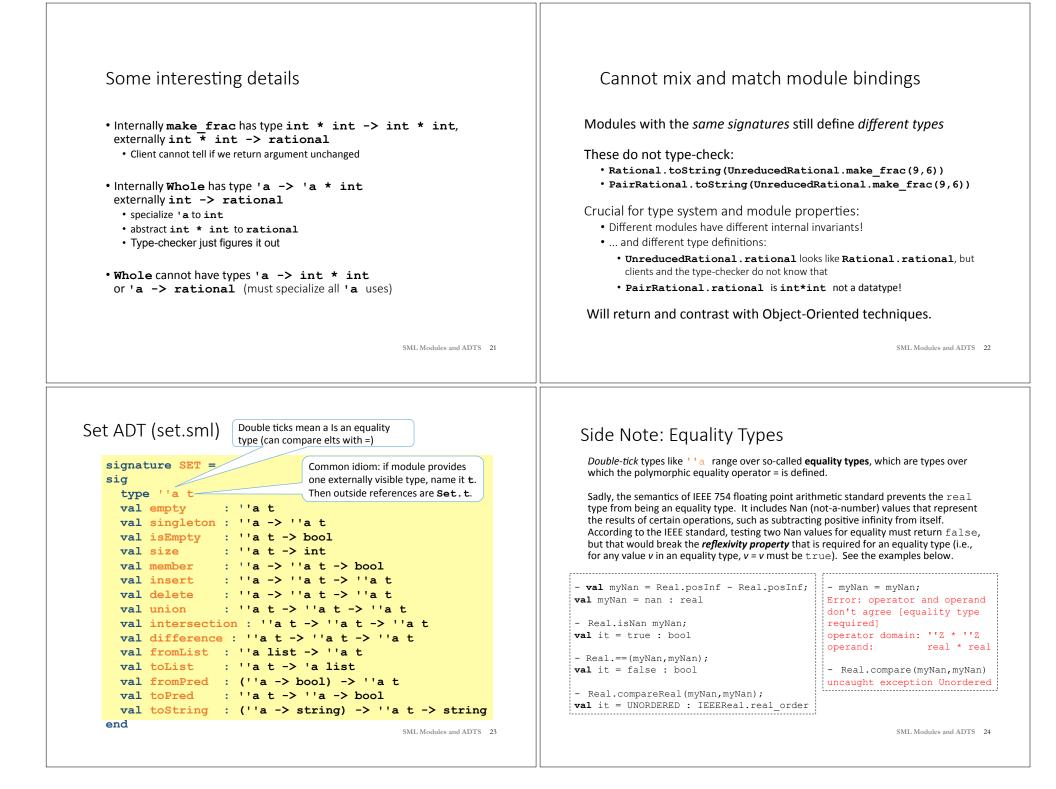
- No client can tell which you are using
- Can improve/replace/choose implementations later
- Easier with more abstract signatures (reveal only what you must)

#### UnreducedRational in adts.sml.

- Same concrete datatype.
- Different invariant: reduce fractions only in toString.
- Equivalent under RATIONAL and RATIONAL\_WHOLE, but not under RATIONAL CONCRETE.

#### PairRational in adts.sml.

- Different concrete datatype.
- Equivalent under **RATIONAL** and **RATIONAL\_WHOLE**, but cannot ascribe **RATIONAL\_CONCRETE**.



## Implementing the SET signature

## ListSet structure (in class)

Represent sets as unordered list.

- Invariant: no duplicates
- What about ordering? Can't use it, since not part of signature!

## ListSetDups structure (in class)

Represent sets as unordered list, \*allowing\* duplicates

## FunSet structure (PS8)

Represent sets as predicate functions

## **OperationTreeSet structure (PS8)**

Represent sets as trees of set operation

```
SML Modules and ADTS 25
```

# ListSet (in class)

```
structure ListSet :> SET =
struct
  type ''a t = ''a list
  val empty = []
  fun singleton x = [x]
  fun insert x ys =
      if member x ys then ys else x :: ys
   ... flesh out the rest in class ...
end
```

- Represent sets as unordered list without duplicates
- Can't use ordering, since not part of signature!
- The following are helpful in implementation: foldr, List.filter, List.exists, String.concatWith

SML Modules and ADTS 26

# ListSetDups (in class)

```
structure ListSetDups :> SET =
struct
  type ''a t = ''a list
  val empty = []
  fun singleton x = [x]
  fun insert x ys = x :: ys (* Allow dups *)
  ... flesh out the rest in class ...
end
```

- Represent sets as unordered lists of elements, possibly containing duplicates. This simplifies some operations and complicates others. Which?
- When must duplicates be removed?
- A removeDups helper function is handy.

SML Modules and ADTS 27

# **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
```

```
SML Modules and ADTS 29
```

```
FunSet (PS8)
Specifying sets with predicates is fun!
Math: { x | 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?

SML Modules and ADTS 30

# OperationTreeSet (PS8)

(delete 4 (difference	(union	<pre>(insert 1 empty) (insert 4 empty)) (insert 7 empty) (insert 4 empty))) (insert 1 empty) (union (insert 1 empty)</pre>
4 Union Union Insert I Empty 4 Empty 7	Union Insert Empty 4 Empty	Intersection Insert I Empty Insert Insert