WELLESLEY CS 251 Fall 2019 Principles of Programming Languages **Topics** Ben Wood Hiding implementation details is the most important strategy Structures, Signatures, for writing correct, robust, reusable software. and Abstract Types ML structures and signatures. Abstraction for robust library and client+library code. Abstraction for easy change. ADTs and functions as data. https://cs.wellesley.edu/~cs251/f19/ Abstract Types 1 Abstract Types 2 structure Name = struct bindings end structure (module) **Hiding with functions** procedural abstraction namespace management and code organization Can you tell the difference? structure MyMathLib = struct fun double x = x\*2fun fact 0 = 1fun double x = x+x- double 4; | fact x = x \* fact (x-1) val y = 2fun double x = x\*yval it : int = 8val half pi = Math.pi / 2 fun double x = let fun help 0 y = yfun doubler x = x \* 2help x v = end help (x-1) (y+1)in help x x end outside: val facts = List.map MyMathLib.fact [1,3,5,7,9] "Private", but can't be shared among functions. Abstract Types 4 Abstract Types 5

```
signature NAME =
                                                                                                           structure Name :> NAME =
                                      sig binding-types end
                                                                                                           struct bindings end
signature
                                                                       ascription
type for a structure (module)
                                                                       (opaque - will ignore other kinds)
List of bindings and their types:
                                                                       Structure must have all bindings with types as declared in
                                                                       signature.
  variables, type synonyms, datatypes, exceptions
                                                                         signature MATHLIB =
                                                                         sia
                                                                           val fact : int -> int
                                                                                                                     Real power:
                signature MATHLIB =
                                                                           val half pi : real
                                                                                                                 Abstraction and Hiding
                siq
                                                                           val doubler : int -> int
                  val fact
                              : int -> int
                                                                         end
                  val half pi : real
                  val doubler : int -> int
                                                                         structure MyMathLib :> MATHLIB =
                end
                                                                         struct
                                                                           fun fact 0 = 1
                                                                             fact x = x * fact (x-1)
                                                                           val half pi = Math.pi / 2
                                                                           fun doubler x = x * 2
                                                                         end
                                                    Abstract Types 6
                                                                                                                           Abstract Types 7
```

## Hiding with signatures

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

**Abstract Data Type** 

type of data and operations on it

Example: rational numbers supporting add and toString

<ul> <li>External properties [externally visible guarantees, up to library writer]</li> <li>Disallow 0 denominators</li> <li>Return strings in reduced form     ("4" not "4/1", "3/2" not "9/6")</li> <li>No infinite loops or exceptions</li> </ul> Implementation invariants [not in external specification] <ul> <li>All denominators &gt; 0</li> <li>All rational values returned from functions are reduced</li> </ul> Signatures help enforce internal 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
Abstract Types 10	Abstract Types 11

# A first signature

Helper functions gcd and reduce not visible outside module.

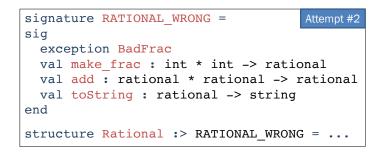
## **Problem: clients can violate invariants**

Create values of type Rational.rational directly.

```
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.



Too far: type rational is not known to exist!

Abstract Types 14

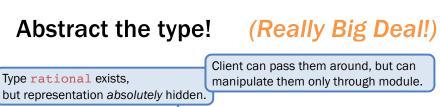
# Abstract Data Type

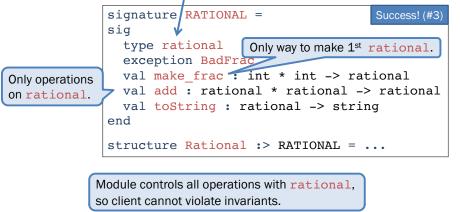
Abstract type of data + operations on it

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 Types 16
```





## 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.

Abstract Types 15

## A cute twist

Exposing the Whole constructor is no problem.

Expose it as 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
```

Abstract Types 18

# **Signature matching rules**

structure Struct :> SIG type-checks if and only if **all** of the following hold:

- 1. Every non-abstract type in SIG is provided in Struct, as specified
- 2. Every abstract type in SIG is provided in Struct in some way
- 3. Every val-binding in SIG is provided in Struct, possibly with a more general and/or less abstract internal type
- 4. Every exception in SIG is provided in Struct.

Struct can have more bindings (implicit in above rules)

Abstract Types 19

# 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.

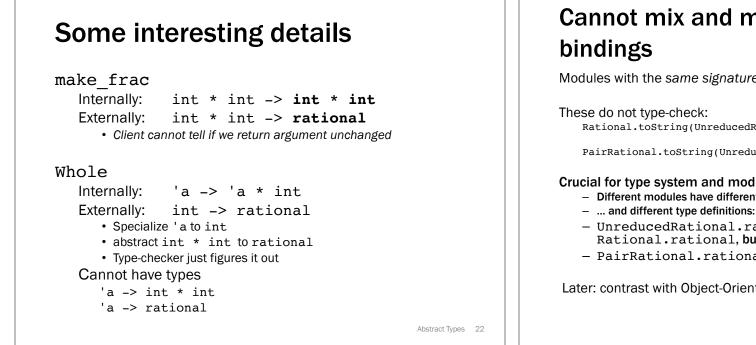
#### PairRational in adts.sml.

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

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
```

Abstract Types 20



## Set ADT (set.sml)

signature SET =	Common idiom: if module provides one externally visible type, name it t.
sig	Then outside references are Set.t.
type ''a t 🥌	mein outside relefences are bee . c.
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	Abstract Types

# Cannot mix and match module

Modules with the same signatures define different types.

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!
- UnreducedRational.rational looks like Rational.rational.but clients and type-checker do not know
- PairRational.rational is int\*int not a datatype!

Later: contrast with Object-Oriented techniques.

Abstract Types 23

# Implementing the SET signature

### ListSet structure

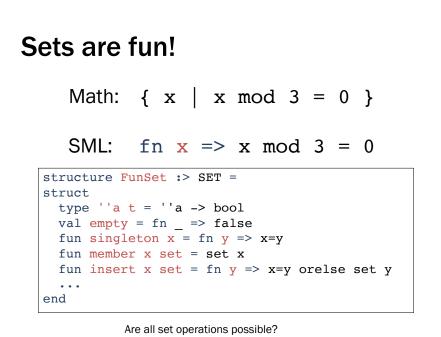
Represent sets as lists.

Invariants?

- Duplicates?
- Ordering?

## FunSet structure

Represent sets as function closures (!!!)



Abstract Types 26