

CS 251 Fall 2019 **Principles of Programming Languages** Ben Wood



Structures, Signatures, and Abstract Types

https://cs.wellesley.edu/~cs251/f19/

Topics

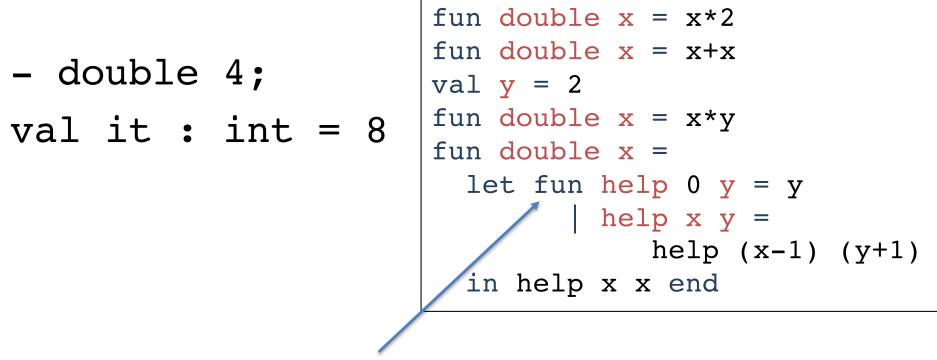
Hiding implementation details is the most important strategy for writing correct, robust, reusable software.

- 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

Can you tell the difference?



"Private", but can't be shared among functions.

structure (module)

structure Name =
struct bindings end

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
end
```

outside:

val facts = List.map MyMathLib.fact [1,3,5,7,9]

signature NAME =
sig binding-types end

signature

type for a structure (module)

List of bindings and their types:

variables, type synonyms, datatypes, exceptions

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

structure Name :> NAME =
struct bindings end

ascription

(opaque – will ignore other kinds)

Structure must have all bindings with types as declared in signature.

```
signature MATHLIB =
siq
  val fact : int -> int
                                            Real power:
  val half pi : real
                                        Abstraction and Hiding
  val doubler : int \rightarrow int
end
structure MyMathLib :> MATHLIB =
struct
  fun fact 0 = 1
    fact x = x * fact (x-1)
  val half pi = Math.pi / 2
  fun doubler x = x * 2
end
```

Hiding with signatures

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

```
signature MATHLIB2 =
siq
 val fact : int -> int
 val half pi : real
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
end
```

Abstract Data Type

type of data and operations on it

Example: rational numbers supporting add and toString

Library spec and invariants

External properties [externally visible guarantees, up to library writer]

- Disallow 0 denominators
- 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

Helper functions gcd and reduce not visible outside module.

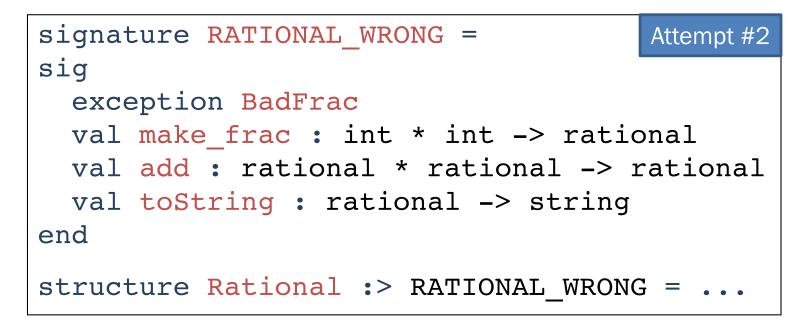
Problem: clients can violate invariants

Create values of type Rational.rational directly.

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

Solution: hide more!

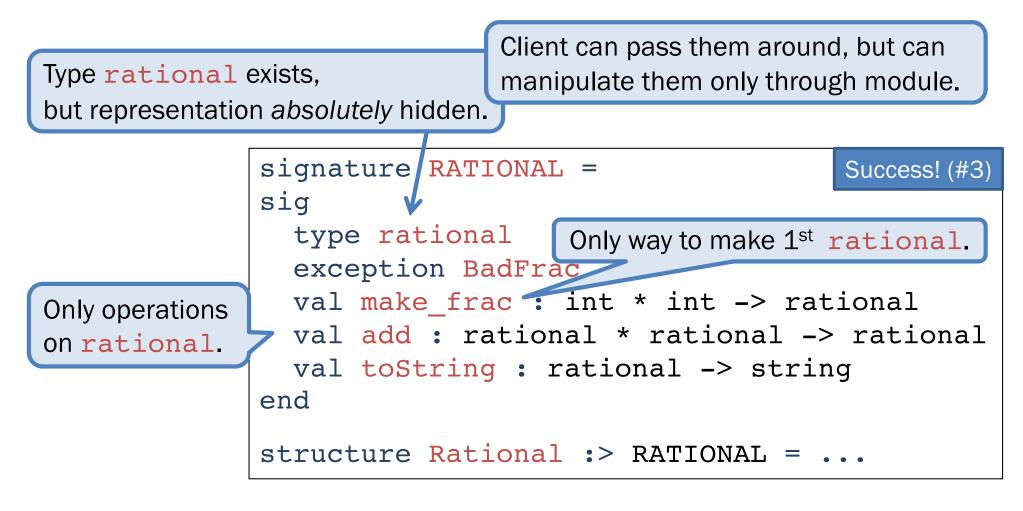
ADT must hide concrete type definition so clients cannot create invariant-violating values of type.



Too far: type rational is not known to exist!

Abstract the type!

(Really Big Deal!)



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 I
```

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

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
```

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)

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
```

Some interesting details

make_frac

Internally: int * int -> int * int Externally: int * int -> rational

• Client cannot tell if we return argument unchanged

Whole

- Internally: 'a -> 'a * int
- Externally: int -> rational
 - Specialize 'a to int
 - abstract int * int to rational
 - Type-checker just figures it out

Cannot have types

- 'a -> int * int
- 'a -> rational

Cannot mix and match module bindings

Modules with the same signatures 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 type-checker do not know
- PairRational.rational is int*int not a datatype!

Later: contrast with Object-Oriented techniques.

Set ADT (set.sml)

Common idiom: if module provides signature SET = one externally visible type, name it t. siq Then outside references are Set.t. type ''a t : ''a t val empty 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

Implementing the SET signature

ListSet structure

Represent sets as lists.

Invariants?

- Duplicates?
- Ordering?

FunSet structure

Represent sets as function closures (!!!)

Sets are fun!

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

SML: $fn x \Rightarrow mod 3 = 0$

```
structure FunSet :> SET =
struct
  type ''a t = ''a -> bool
  val empty = fn _ => false
  fun singleton x = fn y => x=y
  fun member x set = set x
  fun insert x set = fn y => x=y orelse set y
  ...
end
```

Are all set operations possible?