CSE341: Programming Languages Lecture 10 ML Modules Dan Grossman Autumn 2017	<text><text><text><text><text><text><text></text></text></text></text></text></text></text>
<section-header> Example structure MyMathLib = struct fgt = f</section-header>	<section-header><section-header><section-header><section-header><section-header><list-item><list-item><list-item><list-item><list-item></list-item></list-item></list-item></list-item></list-item></section-header></section-header></section-header></section-header></section-header>
<section-header><section-header><section-header><section-header><section-header><section-header><section-header><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item><list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></list-item></section-header></section-header></section-header></section-header></section-header></section-header></section-header>	<section-header><section-header><section-header><section-header><section-header><section-header><section-header><text><list-item><list-item><list-item><equation-block></equation-block></list-item></list-item></list-item></text></section-header></section-header></section-header></section-header></section-header></section-header></section-header>

In general

Autumn 2017

Hiding things

Real value of signatures is to to *hide* bindings and type definitions - So far, just documenting and checking the types

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

So first remind ourselves that functions already do well for some forms of hiding...

Autumn 2017

7

CSE341: Programming Languages

Hiding with functions

These three functions are totally equivalent: no client can tell which we are using (so we can change our choice later):

CSE341: Programming Languages

fun double x = x*2
fun double x = x+x
val y = 2
fun double x = x*y

Defining helper functions locally is also powerful

Can change/remove functions later and know it affects no other code

Would be convenient to have "private" top-level functions too

- So two functions could easily share a helper function
- ML does this via signatures that omit bindings...

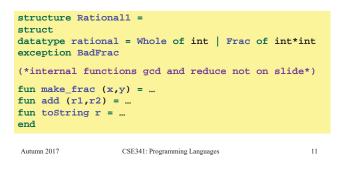
Autumn 2017

CSE341: Programming Languages

A larger example [mostly see the code]

Now consider a module that defines an Abstract Data Type (ADT) – A type of data and operations on it

Our example: rational numbers supporting add and toString



Example

Outside the module, MyMathLib.doubler is simply unbound

- So cannot be used [directly]
- Fairly powerful, very simple idea

```
signature MATHLIB =
sig
val fact : int -> int
val half_pi : real
end
structure MyMathLib :> MATHLIB =
struct
fun fact x = ...
val half_pi = Math.pi / 2.0
fun doubler x = x * 2
end
Autumn 2017 CSE341: Programming Languages
```

8

Library spec and invariants

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

Invariants [part of the implementation, not the module's spec]

- All denominators are greater than 0
- All rational values returned from functions are reduced

More on invariants

Our code maintains the invariants and relies on them

Maintain:

- make_frac disallows 0 denominator, removes negative denominator, and reduces result
- add assumes invariants on inputs, calls reduce if needed

Rely:

- gcd does not work with negative arguments, but no denominator can be negative
- add uses math properties to avoid calling reduce
- toString assumes its argument is already reduced

CSE341: Programming Languages

13

15

17

A first signature

```
With what we know so far, this signature makes sense:
   - gcd and reduce not visible outside the module
signature RATIONAL_A =
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 Rational1 :> RATIONAL_A = ...
```

Autumn 2017

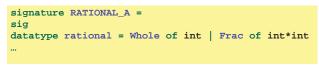
CSE341: Programming Languages

14

The problem

By revealing the datatype definition, we let clients violate our invariants by directly creating values of type Rational1.rational

- At best a comment saying "must use Rational1.make_frac"



Any of these would lead to exceptions, infinite loops, or wrong results, which is why the module's code would never return them

- Rational1.Frac(1,0)
- Rational1.Frac(3,~2)
- Rational1.Frac(9,6)

Autumn 2017

CSE341: Programming Languages

So hide more

Key idea: An ADT must hide the concrete type definition so clients cannot create invariant-violating values of the type directly

Alas, this attempt doesn't work because the signature now uses a type rational that 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 Rational1 :> RATIONAL_WRONG = ...
```

Autumn 2017

CSE341: Programming Languages

16

Abstract types

So ML has a feature for exactly this situation:

In a signature:

Autumn 2017

type foo means the type exists, but clients do not know its definition

```
signature RATIONAL_B =
sig
type rational
exception BadFrac
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end
structure Rational1 :> RATIONAL_B = ...
```

This works! (And is a Really Big Deal)

	<pre>signature RATIONAL_B = sig</pre>	
	type rational	
	exception BadFrac	
	<pre>val make_frac : int * int -> rational</pre>	
	<pre>val add : rational * rational -> rational</pre>	
	<pre>val toString : rational -> string</pre>	
	end	
	ng a client can do to violate invariants and properties:	_
	Only way to make first rational is Rational1.make_frac	C
-	After that can use only Rational1.make_frac, Rational1.add, and Rational1.toString	
-	Hides constructors and patterns – don't even know whether or not Rational1.rational is a datatype	ər
-	But clients can still pass around fractions in any way	

CSE341: Programming Languages

Two key restrictions

So we have two powerful ways to use signatures for hiding:

- 1. Deny bindings exist (val-bindings, fun-bindings, constructors)
- 2. Make types abstract (so clients cannot create values of them or access their pieces directly)

(Later we will see a signature can also make a binding's type more specific than it is within the module, but this is less important)

Autumn 2017

CSE341: Programming Languages

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_C =
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
Autumn 2017 CSE341: Programming Languages
```

Signature matching

Have so far relied on an informal notion of, "does a module typecheck given a signature?" As usual, there are precise rules...

structure Foo :> BAR is allowed if:

- Every non-abstract type in BAR is provided in Foo, as specified
- Every abstract type in BAR is provided in Foo in some way
- Can be a datatype or a type synonym
- Every val-binding in BAR is provided in Foo, possibly with a more general and/or less abstract internal type
 - Discussed "more general types" earlier in course
 - Will see example soon
- Every exception in BAR is provided in Foo

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

Autumn 2017

CSE341: Programming Languages

Equivalent implementations

A key purpose of abstraction is to allow *different implementations* to be *equivalent*

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

Now:

Another structure that can also have signature **RATIONAL_A**,

RATIONAL_B, OF RATIONAL_C

 But only equivalent under RATIONAL_B or RATIONAL_C (ignoring overflow)

Next:

A third equivalent structure implemented very differently

Autumn 2017 CSE341: Programming Languages

22

20

Equivalent implementations

Example (see code file):

- structure Rational2 does not keep rationals in reduced form, instead reducing them "at last moment" in tostring
 Also make gcd and reduce local functions
- -
- Not equivalent under RATIONAL_A
 - Rational1.toString(Rational1.Frac(9,6)) = "9/6"
 - Rational2.toString(Rational2.Frac(9,6)) = "3/2"
- Equivalent under RATIONAL_B or RATIONAL_C
 - Different invariants, but same properties
 - Essential that type rational is abstract

Autumn 2017

19

21

More interesting example

Given a signature with an abstract type, different structures can:

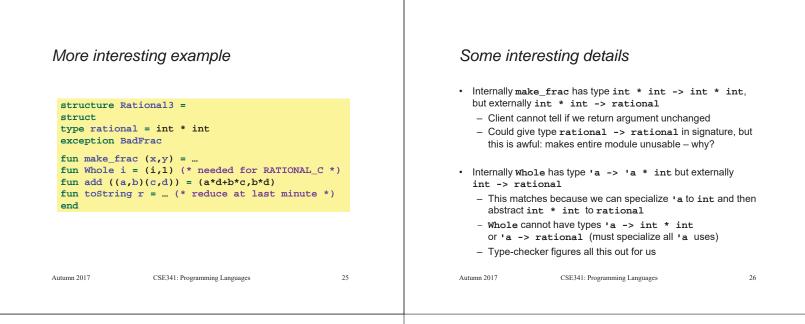
- Have that signature
- But implement the abstract type differently

Such structures might or might not be equivalent

Example (see code):

- type rational = int * int
- Does not have signature RATIONAL_A
- Equivalent to both previous examples under RATIONAL_B or RATIONAL_C

_



Can't mix-and-match module bindings

Modules with the same signatures still define different types

So things like this do not type-check:

- Rational1.toString(Rational2.make_frac(9,6))
- Rational3.toString(Rational2.make_frac(9,6))

This is a crucial feature for type system and module properties:

- Different modules have different internal invariants!
- In fact, they have different type definitions
 - Rational1.rational looks like Rational2.rational, but clients and the type-checker do not know that
 - Rational3.rational is int*int not a datatype!

Autumn 2017

CSE341: Programming Languages

27