Modules

For larger programs, one “top-level” sequence of bindings is poor
   – Especially because a binding can use all earlier (non-shadowed) bindings

So ML has structures to define modules

structure MyModule = struct bindings end

Inside a module, can use earlier bindings as usual
   – Can have any kind of binding (val, datatype, exception, ...)

Outside a module, refer to earlier modules’ bindings via
ModuleName.bindingName
   – Just like List.foldl and String.toUpper; now you can define your own modules
Example

structure MyMathLib =
  struct

  fun fact x =
    if x=0
    then 1
    else x * fact(x-1)

  val half_pi = Math.pi / 2

  fun doubler x = x * 2

end
Namespace management

• So far, this is just namespace management
  – Giving a hierarchy to names to avoid shadowing
  – Allows different modules to reuse names, e.g., map
  – Very important, but not very interesting

• Can use open ModuleName to get “direct” access to a module’s bindings
  – Never necessary; just a convenience; often bad style
  – Often better to create local val-bindings for just the bindings you use a lot, e.g., val map = List.map
    • But doesn’t work for patterns
    • And open can be useful, e.g., for testing code
Signatures

• A signature is a type for a module
  – What bindings does it have and what are their types
• Can define a signature and ascribe it to modules – example:

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

structure MyMathLib :> MATHLIB =
 struct
 fun fact x = ...
 val half_pi = Math.pi / 2
 fun doubler x = x * 2
 end
```
In general

- Signatures

\[
\text{signature SIGNAME =}
\]

\[
\text{sig types-for-bindings end}
\]

- Can include variables, types, datatypes, and exceptions defined in module

- Ascribing a signature to a module

\[
\text{structure MyModule :> SIGNAME =}
\]

\[
\text{struct bindings end}
\]

- Module will not type-check unless it matches the signature, meaning it has all the bindings at the right types

- Note: SML has other forms of ascription; we will stick with these [opaque signatures]
Hiding things

Real value of signatures is 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…
Hiding with functions

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

\[
\begin{align*}
\text{fun double } x & = x \times 2 \\
\text{fun double } x & = x + x \\
\text{val y } & = 2 \\
\text{fun double } x & = x \times y
\end{align*}
\]

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…
Example

Outside the module, `MyMathLib.doubler` is simply unbound
  - So cannot be used [directly]
  - Fairly powerful, very simple idea

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

structure MyMathLib :> MATHLIB =
struct
  fun fact x = ...
  val half_pi = Math.pi / 2
  fun doubler x = x * 2
end
```
A larger example [mostly see lec12.sml]

Now consider a module that defines an Abstract Data Type (ADT)
  – A type of data and operations on it
Our example: rational numbers supporting \texttt{add} and \texttt{toString}

\begin{verbatim}
structure RationalAll =
struct
datatype rational = Whole of int | Frac of int*int
exception BadFrac
(*internal functions gcd and reduce not on slide*)

fun make_frac (x,y) = ...

fun add (r1,r2) = ...

fun toString r = ...
end
\end{verbatim}
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
A first signature

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

```ml
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 Rational11 :> RATIONAL_A = ...
```
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)`

signature `RATIONAL_A` =

sig
datatype `rational` = `Whole` of `int` | `Frac` of `int*int` ...

Fall 2011

CSE341: Programming Languages 14
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:

```plaintext
signature RATIONAL_WRONG =
    sig
    exception BadFrac
    val make_frac : int * int -> rational
    val add : rational * rational -> rational
    val toString : rational -> string
end

structure Rational1l :> RATIONAL_WRONG = ...
```
**Abstract types**

So ML has a feature for exactly this situation:

In a signature:

```ml
type foo
```

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

```ml
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)

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

Nothing a client can do to violate invariants and properties:

- Only way to make first rational is `Rational1.make_frac`
- 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
- But clients can still pass around fractions in any way
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)
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```
Signature matching

Have so far relied on an informal notion of, “does a module type-check 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
- Every val-binding in BAR is provided in Foo, possibly with a more general and/or less abstract internal type
  - This is the interesting part: a little more to come
- …

Of course Foo can have more bindings
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)

Example (see lec12.sml):

– structure Rational2 does not keep rationals in reduced form, instead reducing them “at last moment” in toString
– This approach is not equivalent under RATIONAL_A, but is under RATIONAL_B or RATIONAL_C
– Different invariants, same properties: can’t tell difference if type rational is abstract
More interesting example

structure Rational3 is more interesting: it implements RATIONAL_B and RATIONAL_C using a different definition for type rational

– So of course all the functions change too
– Works great: representation is abstract to clients
– Does not implement RATIONAL_A

structure Rational3 =
struct
type rational = int*int
exception BadFrac

fun make_frac (x,y) = ...
fun Whole i = (i,1) (* needed for RATIONAL_C *)
fun add ((a,b)(c,d)) = (a*d+b*c,b*d)
fun toString r = ... (* reduce at last minute *)
end
Some interesting details

- Internally \texttt{make_frac} has type \texttt{int * int -> int * int}, but externally \texttt{int * int -> rational}
  - Client can’t tell if we return argument unchanged
  - Could give type \texttt{rational -> rational} in signature, but this is awful: makes entire module unusable – why?

- Internally \texttt{Whole} has type 'a -> 'a * int but externally \texttt{int -> rational}
  - This matches because we can specialize 'a to \texttt{int} and then abstract \texttt{int * int} to \texttt{rational}
  - \texttt{Whole} cannot have types 'a -> int * int or 'a -> rational (must specialize all 'a uses)
  - Type-checker figures all this out for us
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(2,3))`
- `Rational3.toString(Rational2.make_frac(2,3))`

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 don’t know that nor does the type-checker
  - `Rational3.rational` is `int*int` not a datatype!