CSE 341
Lecture 13

signatures

slides created by Marty Stepp

http://www.cs.washington.edu/341/
Recall: Why modules?

- **organization**: puts related code together
- **decomposition**: break down a problem
- **information hiding / encapsulation**: protect data from damage by other code
- group identifiers into **namespaces**; reduce # of globals
- provide a layer of **abstraction**; allows re-implementation
- ability to rigidly enforce data **invariants**
- provides a discrete unit for **testing**
- structure Helpers = struct
  fun square(x) = x*x;
  fun pow(x, 0) = 1 | pow(x, y) = x * pow(x, y - 1);
end;
structure Helpers :
  sig
    val square : int -> int
    val pow : int * int -> int
  end

• every structure you define has a public *signature*
  - *signature*: Set of symbols presented by a module to clients
  - by default, all definitions are presented in its signature
Limitations of structures

• Ways that Java hides information in a class?
  ▪ make a given field and method private, protected
  ▪ create an interface or superclass with fewer members; refer to the object through that type (polymorphism)

• signature: A group of ML declarations of functions, types, and variables exported to clients by a structure / module.
  ▪ combines Java's concepts of private and interface
Using signatures

• 1. Define a signature SIG that declares members A, B, C.
• 2. Structure ST1 defines A, B, C, D, E.
  ▪ ST1 can specify that it wants to use SIG as its signature.
  ▪ Now clients can call only A, B, C (not D or E).

  ▪ ST2 can also specify to use SIG as its public signature.
  ▪ Now clients can call only A, B, C (not F or G).
Signature syntax

signature \texttt{NAME} =

\begin{verbatim}
sig
  \textit{definitions}
end;
\end{verbatim}

a signature can contain:

- function \textit{declarations} (using \texttt{val}, not \texttt{fun}) ... no bodies
- \texttt{val} \textit{declarations} (variables; class constants), definitions
- exceptions
- type \textit{declarations}, definitions, and datatypes
Function declarations

val name: paramType * paramType ... -> resultType;

• Example:
  val max: int * int -> int;

• signatures don't have function definitions, with fun
• they instead have declarations, with val
• lists parameter types return type (no implementation)
Abstract type declarations

type name;

• Example:
  type Beverage;

• signatures shouldn't always define datatypes
  ▪ this can lock the implementer into a given implementation

• instead simply declare an abstract type
  ▪ this indicates to ML that such a type will be defined later
  ▪ now the declared type can be used as a param/return type
Signature example

(* Signature for binary search trees of integers. *)
signature INTTREE =
sig
  type intTree;

  val add: intTree -> intTree;
  val height : intTree -> int;
  val min : intTree -> int option;
end;
Implementing a signature

structure name :> SIGNATURE = struct
    definitions
end;

• Example:

structure IntTree :> INTTREE = struct
    ...
end;
Signature semantics

• when a structure implements a signature,
  ▪ structure must implement all members of the signature
  ▪ by convention, signature names are ALL_UPPERCASE
• Modify the Rational structure to implement a RATIONAL signature.
  - In the signature, hide any members that clients shouldn't use directly.
  (What members should be in the signature?)
(* Type signature for rational numbers. *)
signature RATIONAL = sig
  (* notice that we don't specify the innards of rational type *)
  type rational;
  exception Undefined;

  (* notice that gcd and reduce are not included here *)
  val new : int * int -> rational;
  val add : rational * rational -> rational;
  val toString : rational -> string;
end;
structure Rational :> RATIONAL = struct
  datatype rational = Whole of int | Fraction of int * int;
  exception Undefined of string;

  fun gcd(a, 0) = abs(a)      (* 'private' *)
  |   gcd(a, b) = gcd(b, a mod b);
  fun reduce(Whole(i)) = Whole(i) (* 'private' *)
  |   reduce(Fraction(a, b)) =
    let val d = gcd(a, b)
    in  if b = d then Whole(a div d)
        else Fraction(a div d, b div d)
    end;

  fun new(a, 0) = raise Undefined("cannot divide by zero")
  |   new(a, b) = reduce(Fraction(a, b));

  fun add(Whole(i), Whole(j)) = Whole(i + j)
  |   add(Whole(i), Fraction(c, d)) = Fraction(i*d + c, d)
  |   add(Fraction(a, b), Whole(j)) = Fraction(a + j*b, b)
  |   add(Fraction(a, b), Fraction(c, d)) =
      reduce(Fraction(a*d + c*b, b*d));

  (* toString unchanged *)
end;
Using a structure by its signature

- ```
  val r = Rational.new(3, 4);
  val r = - : Rational.rational
  - Rational.toString(r);
  val it = "3/4" : string

  - Rational.gcd(24, 56);
   stdIn:5.1-5.13 Error: unbound variable or constructor: gcd in path Rational.gcd

  - Rational.reduce(r);
   stdIn:1.1-1.15 Error: unbound variable or constructor: ...

  - Rational.Whole(5);
   stdIn:1.1-1.15 Error: unbound variable or constructor: ...
``` 

- using the signature restricts the structure's interface
  - clients cannot access or call any members not in the sig
A re-implementation

(* Alternate implementation using a tuple of (numer, denom). *)
structure RationalTuple :> RATIONAL = struct
  type rational = int * int;
  exception Undefined;

  fun gcd(a, 0) = abs(a)
  |   gcd(a, b) = gcd(b, a mod b);

  fun reduce(a, b) =
    let val d = gcd(a, b)
    in   if b >= 0 then (a div d, b div d) else reduce(~a, ~b)
    end;

  fun new(a, 0) = raise Undefined
  |   new(a, b) = reduce(a, b);

  fun add((a, b), (c, d)) = reduce(a * d + c * b, b * d);

  fun toString(a, 1) = Int.toString(a)
  |   toString(a, b) = Int.toString(a) ^ "/" ^ Int.toString(b);

  fun fromInteger(a) = (a, 1);
end;
Another re-implementation

(* Alternate implementation using a real number; imprecise due to floating point round-off errors. *)
structure Rational :> RATIONAL = struct
  type rational = real;
  exception Undefined;

  fun new(a, b) = real(a) / real(b);
  fun add(a, b:rational) = a + b;
  fun toString(r) = Real.toString(r);
end;
Signature exercise 2

• Use the new signature to enforce these invariants:
  ▪ All fractions will always be created in reduced form.
    – (In other words, for all fractions $a/b$, $\gcd(a, b) = 1$.)
  ▪ Negative fractions will be represented as $-a / b$, not $a / -b$.
    – (In other words, for all fractions $a/b$, $b > 0$.)

• Add the ability for clients to use the Whole constructor.

• Add operations such as ceil, floor, round, subtract, multiply, divide, ...