

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

# Programming Languages

Dan Grossman

Signatures for Our Example

# *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 = ...
```

# *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`”

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

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)`

## *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 = ...
```

# *Abstract types*

So ML has a feature for exactly this situation:

In a signature:

`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)*

```
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
```