15-150 Fall 2013
Lecture 17

Stephen Brookes

This week

- **Lectures**: *modular programming*
  - Designing large programs
  - Information hiding
  - Abstract data types, invariants, and abstraction functions
  - Representation independence

modular

- Divide program into small units
  - manageable
  - easy to maintain
- Give an interface for each unit
  - other units rely only on interface

language support

- **Signatures**
  - specifications
- **Structures**
  - implementions
- **Functors**
  - ways to put structures together...

signatures

```
signature ARITH =
 sig
    type integer
    val rep : int -> integer
    val display : integer -> string
    val add : integer * integer -> integer
    val mult : integer * integer -> integer
end
```

decimal digits

```
structure Dec : ARITH =
 struct
   type digit = int
   type integer = digit list

   fun rep 0 = [ ]
     |   rep n = (n mod 10) :: rep(n div 10)

   ...
end
```

invariant

- To prove correctness we introduce a ***representation invariant***

inv_{10} : Dec.integer -> bool

inv_{10}(L) = true

iff

every item in L is a decimal digit

0 1 2 3 4 5 6 7 8 9

Every* Dec.integer *value used in our code satisfies this invariant

purpose

- Design Dec so that every value of type integer constructible from rep, add, mult satisfies the invariant
- Allows us to write *specialized* code for add and mult that REQUIRES the invariant and ENSURES *correct* results
- Documents any implicit assumptions built into the implementation

(* uses 0 through 9 *)

abstraction

- And we define an ***abstraction function***

eval_{10} : Dec.integer -> int

For all L such that inv_{10}(L) = true,
eval_{10} L = the int value *represented* in decimal by L

***A value* L *of type* Dec.integer *represents the int value* $\textbf{eval}_{10}$ L**

[2,4] *represents* 42

purpose

- The abstraction function tells us
  the *abstract* integer value (of type int)
  that is *represented* by
  a *concrete* integer value (of type Dec.integer)
- Useful in specifications and documentation
- Unambiguous

(only needs to make sense
when the invariant holds)

correctness

Dec implements non-negative integers *faithfully*

- Every non-negative value is ***rep****resentable*

  For $n \geq 0$, inv_{10}(rep n) = true, and $eval_{10}$ (rep n) = n

- **add** implements +

  When inv_{10}(L) and inv_{10}(R) hold, so does inv_{10}(add(L,R)), and $eval_{10}$ (add(L,R)) = $eval_{10}$ L + $eval_{10}$ R

- **mult** implements *

  Similarly

correctness

- For all $n \geq 0$, **rep(n)** satisfies $\mathbf{inv_{10}}$ and $\mathbf{eval_{10}}$**(rep n) = n**
- If **L** and **R** satisfy $\mathbf{inv_{10}}$

  so does **add(L, R)**, and

$$\mathbf{eval_{10}}(\mathbf{add(L, R)}) \\ = (\mathbf{eval_{10}\ L}) + (\mathbf{eval_{10}\ R})$$

(similarly for **mult**)

correctness of add

```
fun add ([ ], qs) = qs
  |    add (ps, [ ]) = ps
  |    add (p::ps, q::qs) =
         ((p+q) mod 10) :: carry ((p+q) div 10, add(ps, qs))
```

Lemma

If **ds** satisfies $\mathbf{inv_{10}}$ and $0 \leq \mathbf{d} \leq 9$
then **carry(d, ds)** satisfies $\mathbf{inv_{10}}$
and $\mathbf{eval_{10}}(\mathbf{carry(d, ds)}) = \mathbf{d} + \mathbf{eval_{10}}\ \mathbf{ds}$

Theorem

If **L** and **R** satisfy $\mathbf{inv_{10}}$ so does **add(L, R)**
and $\mathbf{eval_{10}}(\mathbf{add(L, R)}) = (\mathbf{eval_{10}}\ \mathbf{L}) + (\mathbf{eval_{10}}\ \mathbf{R})$

questions

- Why ***decimal***?
  - Could have used ***binary***

    [0,1,0,1,0,1] represents 42 in base 2
  - Could have used *any* positive base.

    [0,1] represents 42 in base 42

Let's consider binary...

binary digits

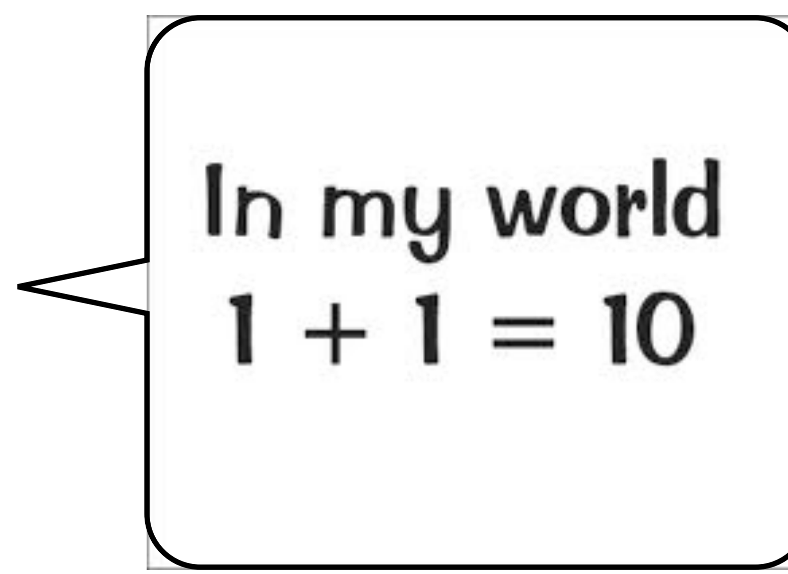
```
structure Binary : ARITH =
 struct
  type digit = int
  type integer = digit list

  fun rep 0 = [ ]
   |  rep n = (n mod 2) :: rep(n div 2)

  ...
end
```

just replace 10 *by* 2
in the code for Dec

correctness

- Binary implements non-negative integers in a way that is ***faithful*** to *standard arithmetic*
  - Every non-negative value is ***rep***_resentable_
  - **add** implements +
  - **mult** implements *

add(rep 1, rep 1) = [0,1]



invariant

- To prove correctness we introduce a ***representation invariant***

inv_2 : int list -> bool

inv_2(L) = true
iff
every item in L is a binary digit

0 1

abstraction

- And we define an ***abstraction function***

eval_2 : integer -> int

For all L : int list such that inv_2(L) = true,
eval_2 L = the int value represented
in binary by L

correctness

- For all $n \geq 0$, **rep(n)** satisfies **inv$_2$** and **eval$_2$(rep n) = n**

- If **L** and **R** satisfy **inv$_2$** so does **add(L, R)**, and

  eval$_2$(**add(L, R)**)
  = (**eval$_2$ L**) + (**eval$_2$ R**)

(similarly for **mult**)

proof

- A correctness proof for Dec,
  with 10 replaced by 2,
  yields a correctness proof for Bin

representation independence

- Both Dec and Bin implement ($\mathbb{N}$, +, *)
- Define a relation

$$\mathscr{R} \subseteq \text{Dec.integer} * \text{Bin.integer}$$

$$\mathscr{R}(\text{ds}, \text{bs}) \text{ iff } \text{eval}_{10}\ \text{ds} = \text{eval}_2\ \text{bs}$$

- For every expression e of type integer built from rep, add, mult

$$\mathscr{R}(\text{Dec.e}, \text{Bin.e}) \text{ holds, i.e.}$$

$$\text{eval}_{10}\ (\text{Dec.e}) = \text{eval}_2\ (\text{Bin.e})$$

deja deja deja vu

- This all looks very similar
  - To get octal representation, replace 10 by 8
  - To get ternary representation, replace 10 by 3
- Let's encapsulate the common design...
- What we need is a *parameterized* structure definition...
- ... with a parameter that specifies a *base*

functors

- An ML functor is a ***parameterized structure definition***
- Like a function from structures to structures
- Its argument and result have *signatures* rather than *types*

BASE

```
signature BASE =
 sig
   val base : int
 end;
```

Digits

```
functor Digits(B : BASE) : ARITH =
struct
  val b = B.base
  type digit = int (* use 0 through b-1 *)
  type integer = digit list

  fun rep 0 = [ ]
   |    rep n = (n mod b) :: rep(n div b)

    ... as before but using b ...

end;
```

```
functor Digits(B : BASE) : ARITH =
 struct
   val b = B.base
   type digit = int (* uses 0 through b-1*)
   type integer = digit list

   fun rep 0 = [ ] | rep n = (n mod b) :: rep (n div b)

   (* carry : digit * integer -> integer *)
   fun carry (0, ps) = ps
    | carry (c, [ ]) = [c]
    | carry (c, p::ps) = ((p+c) mod b) :: carry ((p+c) div b, ps)

   fun add ([ ], qs) = qs
    | add (ps, [ ]) = ps
    | add (p::ps, q::qs) =
         ((p+q) mod b) :: carry ((p+q) div b, add(ps,qs))

   (* times : digit -> integer -> integer *)
   fun times 0 qs = [ ]
    | times k [ ] = [ ]
    | times k (q::qs) =
        ((k * q) mod b) :: carry ((k * q) div b, times k qs)

   fun mult  ([ ], _) = [ ]
    | mult (_, [ ]) = [ ]
    | mult (p::ps, qs) = add (times p qs, 0 :: mult (ps,qs))

   fun display L = foldl (fn (d, s) => Int.toString d ^ s) "" L
 end
```

using Digits

structure Dec = Digits(**struct val** base = 10 **end**);

anonymous structure expressions

structure Binary = Digits(**struct val** base = 2 **end**);

Dec.rep 42 = [2,4] : Dec.integer

Binary.rep 42 = [0,1,0,1,0,1] : Binary.integer

fun decfact(n:int) : Dec.integer =
  if n=0 **then** Dec.rep 1 **else** Dec.mult(Dec.rep n, decfact(n-1));

fun binfact(n:int) : Binary.integer =
  if n=0 **then** Binary.rep 1 **else** Binary.mult(Binary.rep n, binfact(n-1));

what's visible?

```
functor Digits(B : BASE) : ARITH =
struct
  val b = B.base;
  type digit = int (* use 0 through b-1 *)
  type integer = digit list
   ...
```

```
signature ARITH =
 sig
    type integer
    ...
end;
```

- The type Dec.integer is int list
- The type Binary.integer is int list

```
- Binary.rep 42;
val it = [0,1,0,1,0,1] : Binary.integer

- it : int list
val it = [0,1,0,1,0,1] : int list
```

oops!

```
- Binary.add(Dec.rep 42, Dec.rep 42);

val it = [0,0,1,2] : Binary.integer
```

solution (I)

- In the functor body, make **integer** a ***datatype*** whose *constructors* are ***hidden***
- Then adapt the code for **rep**, etc...

```
functor Digits(B : BASE) : ARITH =
struct
  val b = B.base;
  type digit = int (* use 0 through b-1 *)
  datatype digits = D of digit list
  type integer = digits

  fun rep 0 = D [ ]
    |   rep n = let val (D L) = rep(n div b) in D((n mod b)::L) end
...
```

solution (1)

```
functor Digits(B:BASE) : ARITH =
 struct
   val b = B.base
   type digit = int (* uses 0 through b-1 *)
   datatype digits = D of digit list
   type integer = digits

   fun rep 0 = D [ ]
   | rep n = let val (D L) = rep(n div b) in D ((n mod b) :: L) end

   (* carry : digit * digit list -> digit list *)
   fun carry (0, ps) = ps
    | carry (c, [ ]) = [c]
    | carry (c, p::ps) = ((p+c) mod b) :: carry ((p+c) div b, ps)

   (* adder : digit list * digit list -> digit list *)
   fun adder ([ ], qs) = qs
    | adder (ps, [ ]) = ps
    | adder (p::ps, q::qs) =
        ((p+q) mod b) :: carry ((p+q) div b, adder(ps,qs))

   (* add : integer * integer -> integer *)
   fun add (D L, D R) = D (adder(L, R))

   (* times : digit -> digit list -> digit list *)
   fun times 0 qs = [ ]
    | times k [ ] = [ ]
    | times k (q::qs) =
        ((k * q) mod b) :: carry ((k * q) div b, times k qs)

   (* multer : digit list * digit list -> digit list *)
   fun multer  ([ ], _) = [ ]
    | multer (_, [ ]) = [ ]
    | multer (p::ps, qs) = adder (times p qs, 0 :: multer (ps,qs))

   (* mult : integer * integer -> integer *)
   fun mult(D L, D R) = D(multer(L,R))

   fun display (D L) = foldl (fn (d, s) => Int.toString d ^ s) "" L
 end
```

```
structure Dec = Digits(struct val base = 10 end);

structure Binary = Digits(struct val base = 2 end);
```

```
- Dec.rep 42;
val it = D [2,4] : Dec.integer

- Bin.rep 42;
val it = D [0,1,0,1,0,1] : Bin.integer

- D [1+1] = D [2];
Error: unbound variable or constructor: D

- Bin.add(Dec.rep 42, Dec.rep 42);
Error:operator and operand don't agree
[tycon mismatch]
```

solution (2)

- Leave the functor body as is,
  but *ascribe* the signature ***opaquely***

```
functor Digits(B : BASE) :> ARITH =
struct
  val b = B.base;
  type digit = int (* use 0 through b-1 *)
  type integer = digit list

  fun rep 0 = [ ]
   |    rep n = (n mod b) :: rep (n div b)
...
```

problem solved

- With either of these solutions,
  the code fragment

```
Binary.add(Dec.rep 42, Dec.rep 42)
```

  is **not well-typed**,
  so cannot be evaluated.

transparency

```
- Binary.rep 42;
val it = [0,1,0,1,0,1] : Binary.integer

- Dec.rep 42;
val it = [2,4] : Dec.integer
```

Dec.integer = int list
Binary.integer = int list

opacity

```
- Binary.rep 42;
val it = - : Binary.integer

- Dec.rep 42;
val it = - : Dec.integer
```

testing

With *opaque* ascription

```
- Dec.rep 42;
val it = - : Dec.integer
```

How can we test if the value is *correct*?

```
- it = [2,4];
```
type error

Convert it to a visible type!

```
- Dec.display(Binary.rep 42);
val it = "42" : string
```

correctness

- Suppose B:BASE and B.base > 1
- Let S = Digits(B) and b = B.base
- Define $\mathbf{inv_b}$ and $\mathbf{eval_b}$ as before
- For all $n \geq 0$, S.rep(n) satisfies $\mathbf{inv_b}$
- If v_1, v_2 : S.integer satisfy $\mathbf{inv_b}$ so does S.add(v_1, v_2) and $\mathbf{eval_b}$(S.add(v_1, v_2)) = ($\mathbf{eval_b}$ v_1) + ($\mathbf{eval_b}$ v_2)
- Similarly for S.mult

what happens if B.base = 1?

unary

```
structure Unary : ARITH = Digits(struct val b = 1 end);
open Unary;
display(add(rep 3, rep 2));
```

What goes wrong? Why?
Didn't we *prove* correctness?

```
fun rep 0 = [ ]
  | rep n = (n mod 1) :: rep(n div 1)
```

$$\text{eval}_{10}(\text{rep } n) \neq n$$

Figure out where the correctness proof breaks!

Unary

```
structure Unary : ARITH =
struct
   datatype mark = X
   type integer = mark list
   fun rep 0 = [ ]
   |     rep n = X :: rep(n-1)
   fun add (L,R) = L@R
   fun mult([ ], R) = [ ]
   |    mult(_::L, R) = add(mult(L,R), R)
   fun display L = foldr (fn (_,s) => "1"^s) "" L
end
```

fun inv_I _ = **true**
fun eval_I L = length L

exercise

- Try using the Unary structure in ML
- Try the same structure but make it opaque
- Understand what's visible and what's not, and contrast transparent with opaque
- Say what "correctness" should mean
- And prove that the structure implements arithmetic correctly