Modulear programming

Stephen Brookes
Write an ML function

\[
\text{factzeros} : \text{int} \rightarrow \text{int}
\]

such that for all \(n > 0\),
\[
\text{factzeros } n = \text{the number of zeros on the end of } n! \text{ (in decimal)}
\]

\[
5! = 120
\]

\[
\text{factzeros } 5 = 1
\]
modularity principle

- A large program should be organized as a collection of small components
  - manageable size
  - easy to maintain
- Give an interface for each component
  - others should rely only on this

Motivation: to facilitate separate development
language support

Signatures
  • interfaces

Structures
  • implementations

Functors
  • ways to combine structures...
A signature is a collection of names with specified types

```signature
SIGNAME =
  sig
  . names for types,
  . functions,
  . exceptions
end
```
a signature

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

A type named integer
A function rep : int -> integer
A function display : integer -> string
A function add : integer * integer -> integer
A function mult : integer * integer -> integer
just an interface

- Just declaring this signature doesn’t create any types, values, exceptions…
- To implement it we must define a structure
- Defining a structure with this signature will create instances of the types, values and exceptions in the signature

one signature may have many implementations
an implementation

```
structure Ints =
struct
  type integer = int
  fun rep n = n
  fun display n = Int.toString n
  fun add(x, y) = x + y
  fun mult(x, y) = x * y
end
```

This structure implements \texttt{ARITH} and defines \texttt{integer} as the type \texttt{int}
ML says

structure Ints :
  sig
    type integer = int
    val rep : 'a -> 'a
    val display : int -> string
    val add : int * int -> int
    val mult : int * int -> int
  end
**Ints implements ARITH**

- a type named `integer`
- values `rep`, `display`, `add`, `mult` of types `consistent` with ARITH

```plaintext
structure Ints : sig
  type integer = int
  val rep : 'a -> 'a
  val display : int -> string
  val add : int * int -> int
  val mult : int * int -> int
end

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

rep is OK because `integer = int` and `int -> integer` is an instance of `'a -> 'a`
ML says

structure Ints : ARITH
struct
  type integer = int
  fun rep n = n
  fun display n = Int.toString n
  fun add(x, y) = x + y
  fun mult(x, y) = x * y
end

ascribing the signature ARITH constrains what the structure provides
ascription requirements

Ints : ARITH

• For every type named in ARITH, Ints must have a type definition

• For every value named in ARITH, Ints must have a suitable declaration

these rules must be obeyed by implementers
ascription
requirements

Ints : ARITH

• *Ints* defines *integer* as *int*

• *Ints* declares the values named in *ARITH*, with types at least as general as required, given that *integer* is *int*

  add : integer * integer -> integer    add : int * int -> int
  rep : int -> integer                 rep : int -> int
what can go wrong

signature CIRCLE =
sig
  val radius : real
  val center : real * real
end

structure Square : CIRCLE =
struct
  val side = 3.0
  val center = (0.0, 0.0)
end

> Error: unmatched value specification: radius
ascription constraints

Ints : ARITH

• Users of Ints can only use data named in ARITH

• Users of Ints “know” that integer is int

• Users of Ints can call rep, add, mult, display
  and can do arithmetic on values of type integer

  \[ 40 + \text{rep} \ 2 = \Rightarrow \ 42 : \text{int} \]

this rule is imposed on all clients or users
open

opening a structure reveals the types and values in its signature

- open Ints;

ML says

opening Ints

type integer = int
val rep : int -> integer
val add : integer * integer -> integer
val mult : integer * integer -> integer
val display : integer -> string

- add(rep 2, rep 3);
  val it = 5 : integer
using a structure
for ARITH

Write a function

\[
\text{fact} : \text{int} \rightarrow \text{integer}
\]

such that for \( n \geq 0 \),

\[
\text{fact} \ n = \text{an integer value representing } n!\]

Available for use:

\[
\text{rep} : \text{int} \rightarrow \text{integer}
\]
\[
\text{add} : \text{integer} \times \text{integer} \rightarrow \text{integer}
\]
\[
\text{mult} : \text{integer} \times \text{integer} \rightarrow \text{integer}
\]
\[
\text{display} : \text{integer} \rightarrow \text{string}
\]
factorial implementation

- fun fact(n:int):integer = 
  if n=0 then rep 1 else mult(rep n, fact(n-1));
val fact = fn : int -> integer

- fact 3;
val it = 6 : integer

this should work for ANY structure that implements ARITH
using Ints

open Ints

fun fact(n:int):integer =
    if n=0 then rep 1 else mult(rep n, fact(n-1))

fact 3 =>* 6 : int

fact 100 raises Overflow
qualified names

The built-in ML structures Int and String both define compare

Use the structure name to disambiguate

• Int.compare : int * int -> order
• String.compare : string * string -> order
using qualified names

fun fact(n:int): Ints.integer =
  if n=0 then Ints.rep 1
  else Ints.mult(Ints.rep n, fact(n-1))

type Ints.integer = int

fact 3 =>* 6 : int

fact 100

raises Overflow
transparency

We ascribed the signature *transparently*

```ml
structure Ints : ARITH = ...
open Ints;

fun fact(n:int):integer =
  if n=0 then ~1 else mult(rep n, fact(n-1));
  : int : integer
val fact = fn - : int -> integer
```

transparency allows users to *exploit*
the fact that *integer* is *int*
and execute *dangerous* code
transparency in action

- fun fact(n:int):integer =
  if n=0 then rep 1 else mult(rep n, fact(n-1));
val fact = fn : int -> integer

- fact 3;
val it = 6 : integer GOOD!

- fun fact(n:int):integer =
  if n=0 then \textcolor{red}{\sim}1 else mult(rep n, fact(n-1));
val fact = fn : int -> integer

- fact 3;
val it = \textcolor{red}{\sim}6 : integer BAD!!!!
opacity

We can ascribe the signature *opaquely* using `::>

```haskell
structure Ints ::> ARITH = ...
open Ints;

fun fact(n:int):integer =
    if n=0 then ~1 else mult(rep n, fact(n-1))
        : int        : integer
```

Error: types of if branches do not agree

opacity *prevents* users from *exploiting* the fact that *integer* is *int*
opacity in action

- fun fact(n:int):integer =
  if n=0 then rep 1 else mult(rep n, fact(n-1));
val fact = fn : int -> integer

- fact 3;
val it = 6 : integer

GOOD!

- fun fact(n:int):integer =
  if n=0 then \textcolor{red}{\sim} 1 \textcolor{red}{\else} else mult(rep n, fact(n-1));

Error: types of if branches do not agree

GOOD!
ascription
requirements

\[ \text{Ints} :> \text{ARITH} \]

- For every type named in \text{ARITH}, \text{Ints} must have a type definition.

- For every value named in \text{ARITH}, \text{Ints} must have a \textit{suitable} declaration with \textit{same name, at least as general type}.

\hspace{1em} (\textit{same as with transparent ascription})
ascription constraints

Ints :> ARITH

• Users of Ints can only use the type names and value bindings named in ARITH

• Users of Ints cannot “see inside” values of the types named in ARITH

• Users of Ints do not “know” that integer is int

• Users of Ints can call rep, add, mult, display but cannot do arithmetic on values of type integer

40 + rep 2 is not well-typed!

these rules are imposed on all clients or users
opaque use

structure Ints => ARITH = ... 
open Ints;

ML says 

opening Ints 
type integer 
val add : integer * integer -> integer
val mult : integer * integer -> integer
val rep : int -> integer
val display : integer -> string

- add(rep 2, rep 3);
val it = - : integer

users cannot see inside values of type integer

NOT THIS!
- add(rep 2, rep 3);
val it = 5 : integer
implementing ARITH

- fun fact(n:int):integer =
  if n=0 then rep 1 else mult(rep n, fact(n-1));

fact 100;

unciaught exception Overflow [overflow]
  raised at: <file stdIn>

**Ints** isn’t the only way...

Let’s find a way to deal with **larger** integers, like 100!
the plan

• Represent an integer as a list of decimal digits (with least significant digit first)

• Implement digitwise addition and multiplication
  • grade school algorithms
  • using `div 10, mod 10`, carry propagation

• Use `Int.toString` to produce a string
structure Dec : ARITH =
struct
  type digit = int
  type integer = digit list

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

fun display [ ] = "0"
     | display L = foldl (fn (d, s) => Int.toString d ^ s) "" L

.........
decimal digits

.... continued ...

(* carry : digit * integer -> integer *)

fun carry (0, ps) = ps

| carry (c, [ ]) = [c]
| carry (c, p::ps) =
    ((p+c) mod 10) :: carry ((p+c) div 10, ps)

fun add ([ ], qs) = qs

| add (ps, [ ]) = ps
| add (p::ps, q::qs) =
    ((p+q) mod 10) :: carry ((p+q) div 10, add(ps, qs))
(* times : digit * integer -> integer *)

fun times (0, qs) = [ ]
 -times (p, [ ]) = [ ]
  | times (p, q :: qs) =
    (((p * q) mod 10) :: carry (((p * q) div 10, times(p, qs)))

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

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

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

fun carry (0, ps) = ps l ...

fun add ([ ], qs) = qs l ....

fun times (0, qs) = [ ] l ....

fun mult ([ ], _) = [ ] l ....

fun display L = foldl (fn (d, s) => Int.toString d ^ s) "" L
end;
• **Dec implements** the signature ARITH

• It’s OK to define extra data, such as
  
  carry : digit * integer -> integer
  times : digit * integer -> integer

• These are not in the signature, so not visible to users of Dec

• The type **integer** is **int list**

• The only *relevant* lists contain *decimal digits*
using Dec

open Dec

define fact(n:int):integer =
  if n=0 then rep 1 else mult(rep n, fact(n-1))
  : integer : integer

fact 3 =>* [6] : int list

fact 100 =>* [0,0,0,0,...] : int list
results

display(fact 100) =
"9332621544394415268169923885626670049071596826438162146859296389521759999322991560894146397615651828625369792082722375825118521091686400000000000000000000000000000"

24 trailing zeros
reflection

• We have given two different structures

   Ints Dec

that implement the same signature

   ARITH

• The Dec implementation uses values of type int list built from decimal digits

  • All lists constructible from ARITH functions are decimal
importance

• A decimal list \textit{represents} an integer
• \texttt{rep} produces a decimal list from an int
• \texttt{carry, add, mult, …} produce a decimal list from decimal arguments
• So every value constructible from \texttt{rep, carry, add, mult, …} is a decimal list and \textit{represents} an integer
correctness

The ARITH functions defined in Dec are *arithmetically accurate*

- **rep n** = a decimal list representing *n*

- If *xs* represents *x* and *ys* represents *y*, then **add(xs, ys)** represents *x*+*y*

- If *xs* represents *x* and *ys* represents *y*, then **mult(xs, ys)** represents *x*×*y*
key concepts

• ARITH is a signature for an abstract data type
  • An abstract type (“integers”) with a limited collection of operations (“addition,…”)

• Dec implements the abstract type, using values that satisfy an invariant

• Values satisfying the invariant represent an abstract value, and the functions in Dec correctly implement the operations, when used on values satisfying the invariant
invariance

- For $n \geq 0$, $\text{rep } n$ evaluates to a decimal list
  
  **Induction on** $n$

- If $0 \leq c \leq 9$ and $ps$ is a decimal list, then $\text{carry}(c, ps)$ evaluates to a decimal list
  
  **Induction on** $\text{length}(ps)$

- If $ps$ and $qs$ are decimal lists, then $\text{add}(ps, qs)$ evaluates to a decimal list
  
  **Induction on** $\text{length}(ps) \times \text{length}(qs)$
representation

• A decimal list $[d_0, d_1, \ldots, d_n]$ represents the integer written (in decimal notation) as $d_n \ldots d_1 d_0$

fun eval [ ] = 0
| eval (d::R) = d + 10 * (eval R)

eval : int list -> int
REQUIRES L is a decimal list
ENSURES eval L = the integer represented by L
accuracy

• For $n \geq 0$, $\text{rep } n$ represents $n$
  
  Induction on $n$

• If $0 \leq c \leq 9$ and $ps$ represents $p$, then $\text{carry}(c, ps)$ represents $c+p$
  
  Induction on $\text{length}(ps)$

• If $ps$ and $qs$ represent $p$ and $q$, then $\text{add}(ps, qs)$ represents $p+q$
  
  Induction on $\text{length}(ps) \times \text{length}(qs)$
yada yada yada

• Similarly for times and mult

  If ps and qs represent p and q, then mult(ps, qs) represents p*q

• And for display…

  If L represents n, display L evaluates to the string for n