15-150 Fall 2013
Lecture 15

Stephen Brookes

announcements

- Midterm exam 12:00-1:20 Thursday
- Covers class material up to last Thursday
- Can use ONE page of notes (we collect)
- Write proper ML code, short proofs
- Be on time, bring pen

today

- Dealing with runtime errors
- Declaring, raising and handling exceptions
- Evaluation and equality, revisited
- Referential transparency, revisited

so far

- Expressions have *types*
- Types determine sets of *values*
- Expression evaluation
  - *returns* a value, of the *correct type*
  - or *loops forever*

still true

- A well-typed expression can't go *wrong*
  - *no stupid type errors*

but also

- A well-typed expression
  may cause a runtime error

  - fact 100 uncaught exception Overflow
  - 42 div 0 uncaught exception Div
  - hd (allqueens 3) uncaught exception Empty

hd [42 div (length(allqueens 3) * fact 100)]

=>* uncaught exception Overflow

it's ok

- Type analysis is based on syntax
  - does not *evaluate* expressions
- Can't expect to prevent *all runtime errors* just by looking at *syntax*
- But we're still type-safe...

exceptions

- ML has *exceptions*
- Some built-in, e.g.   Div, Overflow, Match
- Can be *declared*, *raised*, and *handled*
- Very flexible mechanism, simple scope rules
- Fits nicely with type discipline

however...

- The potential for exceptions means we need to revise our foundational definitions!
  - Evaluation
    - Equality
      - Referential transparency

evaluation

- An expression evaluation will either

  return a *value*, of the *correct type*

  or *loop forever*

  or *raise an exception*

equality

- Expressions of type **int** are *equal* iff

    both *evaluate, to the same value*
    or both *fail to terminate*
    or both *raise the same exception*

equality

- Expressions of type **t->t'** are *equal* iff

  both *evaluate, to* ***equal*** *values*

  or both *fail to terminate*

  or both *raise the same exception*

f = g iff
for all x, y : t,
x = y implies f(x) = g(y)

ref trans

- Safe substitution for ***equal*** sub-expressions

If $e_1 = e_2$ then $E[e_1] = E[e_2]$

21 + 21 = 42,

so

(**fn** x:int => 21 + 21) = (**fn** x:int => 42)

fact 100 = fact 200,

so

(**fn** x:int => fact 100) = (**fn** x:int => fact 200)

declaring

exception Negative

exception Ring-ding-ding-ding-dingeringeding

exception Wa-pa-pa-pa-pa-pa-pow

usually,
choose an
appropriate name

scope

```
let
  exception Foo
in
  ....
end
```

```
local
  exception Foo
in
  ....
end
```

OK to **raise** and **handle** Foo here

raising

- In scope of a declaration for Foo,
  we can *raise* it to cause a runtime error

- raise Foo can be used at any type(!)

raise Foo

42 + **raise** Foo = **raise** Foo

(**fn** x:int => 0) (**raise** Foo) = **raise** Foo

fun f(x) = **if** x<0 **then** **raise** Negative **else** ...

gcd : int * int -> int

```
(* REQUIRES x>0 & y>0 *)
(* ENSURES  gcd(x, y) = the g.c.d. of x and y. *)
fun gcd (x, y) =
    case Int.compare(x, y) of
          LESS        => gcd(x, y-x)
        | EQUAL       => x
        | GREATER => gcd(x-y, y)
```

will loop forever

gcd(1, 0) =>* gcd(1-0, 0) =>* gcd(1, 0) =>* ...

gcd(1, ~1) =>* gcd(2, ~1) =>* gcd(3, ~1) =>*...

will raise Overflow

GCD : int * int ->int

(* REQUIRES true *)
(* ENSURES GCD(x,y) = the g.c.d of x and y if x > 0 and y > 0, *)
(* ENSURES GCD(x,y) = **raise** NotPositive if x ≤ 0 or y ≤ 0. *)

exception NotPositive;

fun GCD (x, y) =
  if (x <= 0 **orelse** y <= 0) **then raise** NotPositive **else**
    case Int.compare(x,y) **of**
      LESS => GCD(x, y-x)
      | EQUAL => x
      | GREATER => GCD(x-y x)

recursive calls do redundant tests

GCD(42, 72) =>* 6
GCD(1, 0) =>* raise NotPositive
GCD(1, ~1) =>* raise NotPositive

GCD : int * int -> int

(* REQUIRES true *)
(* ENSURES GCD(x,y) = the g.c.d of x and y if x > 0 and y > 0, *)
(* ENSURES GCD(x,y) = **raise** NotPositive if x ≤ 0 or y ≤ 0. *)

exception NotPositive
fun gcd (x, y) =
    case Int.compare(x, y) **of**
        LESS => gcd(x, y-x)
      | EQUAL => x
      | GREATER => gcd(x-y, y)

one test per call

fun GCD (x, y) =
  if (x > 0 **andalso** y > 0) **then** gcd(x,y) **else raise** NotPositive

GCD(42, 72) =>* 6
GCD(1, 0) =>* raise NotPositive
GCD(1, ~1) =>* raise NotPositive

GCD': int * int -> int

```
exception NotPositive

local
  fun gcd (m,n) =
    case Int.compare(m,n) of
         LESS    => gcd(m, n-m)
       | EQUAL   => m
       | GREATER => gcd(m-n, n)
in
  fun GCD' (x, y) =
    if x>0 andalso y>0 then gcd(x, y)
                              else raise NotPositive
end;
```

even better: the dangerous gcd function is *not available outside*

GCD = GCD’

GCD : int * int -> int
GCD’ : int * int -> int

are **extensionally equal**, because:
for all integer values x and y,

EITHER x>0 & y>0, and
GCD(x,y) and GCD’(x,y) both evaluate
to the g.c.d of x and y,

OR not(x>0 & y>0), and
GCD(x,y) and GCD’(x,y) both raise NotPositive

handling

e_1 **handle** Foo => e_2

- Has type t if e_1 and e_2 have type t
- If e_1 =>* v, so does e_1 **handle** Foo => e_2
- If e_1 raises Foo,
  e_1 **handle** Foo => e_2 =>* e_2
- If e_1 raises Bar, so does e_1 **handle** Foo => e_2
- If e_1 loops, so does e_1 **handle** Foo => e_2

norris : int * int -> int

Chuck says "For all values n:int, n div 0 = n."

```
fun norris(x:int, y:int) : int =
        (x div y) handle Div => x
```

scope

- The *scope* of the handler for Ringerding in

  e **handle** Ringerding => e′

  is e

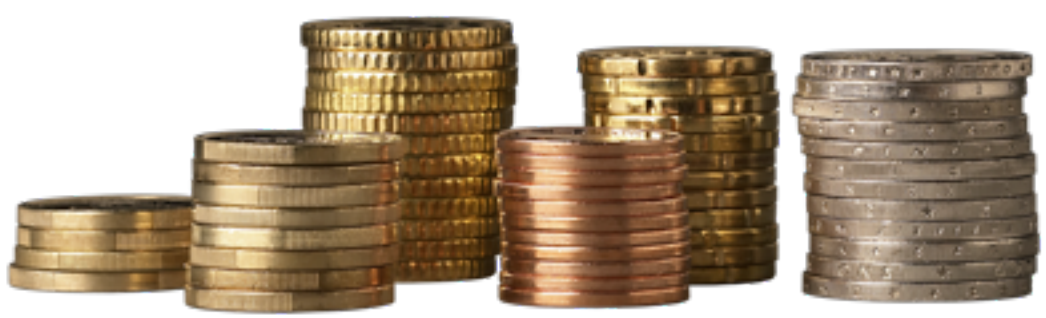
- Can also *combine* handlers

```
e handle  Ringerding => e1′
        | Hatee-hatee-ho => e2′
        | Wa-pow-pow => e3′
        | _ => raise NotFox
```

(e, e_1', e_2', e_3' must have the same type)

making change

- Given a list L of positive integers (*coin sizes*) and a non-negative integer a (an *amount*)
- Find a list of items from L that adds up to a
  - ***can use coin sizes more than once***



idea

- If $a = 0$, make change with the empty list.
- If $a > 0$ and the coins list is c::R,
  - if $c > a$ you can't use it;
  - otherwise use c and make change for $a - c$.

"greedy"

change

```
(* change : int list * int -> int list *)
fun change (_, 0) = [ ]
 |    change (c::R, a) =
    if a<c then change(R, a)
              else c :: (change(c::R, a-c));
```

results

```
stdIn:60.5-63.41 Warning: match nonexhaustive
          (_,0) => ...
          (c :: R,a) => ...
val change = fn : int list * int -> int list
```

```
- change ([2,3], 8);
val it = [2,2,2,2] : int list
```

✓

```
- change ([3,2], 8);
val it = [3,3,2] : int list
```

✓

```
- change([ ], 42);
uncaught exception Match
```

WRONG!

```
- change([2,3], 9);
uncaught exception Match
```

WRONG!

bad style

- It's not good to use a built-in exception like Match to signal a problem-specific error
- Instead let's declare our own exception
- Give it a *suggestive* name like Impossible

idea 2

- If a = 0, make change with the empty list.

- If a > 0 & coins list is c::R
  - if c > a you can't use it;
  - otherwise use c & make change for a - c.

- If a > 0 & coins list is [ ], raise Impossible.

change 2

```
exception Impossible;

(* change : int list * int -> int list *)
fun change (_, 0) = [ ]
 | change ([ ], _) = raise Impossible
 | change (c::R, a) =
    if a<c then change (R, a)
              else c :: (change (c::R, a-c))
```

results

val change = fn : int list * int -> int list

- change ([2,3], 8);
val it = [2,2,2,2] : int list ✓

- change([ ], 42);
uncaught exception Impossible ✓

- change ([2,3], 9);
uncaught exception Impossible

idea 3

- If a = 0, make change with the empty list.

- If a > 0 & coins list is c::R
  - if c > a you can't use it;
  - otherwise *try to* use c and make change for a-c; if this fails, *handle the exception* by making change for a without c.

- If a > 0 & coins list is [ ], raise Impossible.

change 3

```
exception Impossible;

(* change : int list * int -> int list *)
fun change (_, 0) = [ ]
 |    change ([ ], _) = raise Impossible
 |    change (c::R, a) =
    if a<c then change (R, a)
          else c :: (change (c::R, a-c))
                 handle Impossible => change(R, a)
```

results

```
- change ([2,3], 9);
val it = [2,2,2,3] : int list
```

equations

- The change definition yields these equations, for all *values* L, c, R, a

```
(1) change (L, 0) = [ ]
(2) change ([ ], a) = raise Impossible

  change (c::R, a) =
     if a<c then change (R, a)
           else c :: (change (c::R, a-c))
                handle Impossible => change(R, a)
```

equations

- The change definition yields these equations, for all *values* L, c, R, a

(1) change (L, 0) = [ ]
(2) change ([ ], a) = **raise** Impossible

(3) change (c::R, a) = change (R, a) if $a < c$
(4) change (c::R, a) = c :: (change (c::R, a-c))
handle Impossible => change(R, a)
if $a \geq c$

```
change([5,2], 6)
    = (5 :: change([5,2], 1))
            handle Impossible => change([2],6)
    = (5 :: change([2], 1))
            handle Impossible => change([2],6)
    = (5 :: change([ ], 1))
            handle Impossible => change([2],6)
    = (5 :: raise Impossible)
            handle Impossible => change([2],6)
    = (raise Impossible)
            handle Impossible => change([2],6)
    = change([2],6) = [2,2,2]
```

change spec

(* change : int list * int -> int list *)

(* REQUIRES
L is a list of positive integers & a≥0 *)

(* ENSURES
change(L, a) = a list of items from L
with sum equal to a, if there is one;
change(L, a) = **raise** Impossible, otherwise. *)

mkchange

mkchange : int list * int -> (int list) option

```
fun mkchange (coins, a) =
   SOME (change (coins, a))
      handle Impossible => NONE
```

what's wrong?

```
(* change : int list * int -> int list *)
fun change (_, 0) = [ ]
 |   change ([ ], _) = raise Impossible
 |   change (c::R, a) =
   (if a<c then change (R, a)
              else c :: change (c::R, a-c) )
                 handle Impossible => change(R, a)
```