15-150 Lecture 14:
Exceptions

Lecture by Dan Licata

March 6, 2012

1 Exception Basics

For a while, you've been doing things like `raise Fail "error"` and `raise Unimplemented`. What do these really mean?

Recall trees:

```
datatype 'a tree =
    Empty
  | Leaf of 'a
  | Node of 'a tree * 'a tree
```

and the function to find the leftmost odd number in a tree:

```
(* returns SOME(the first odd number) if any,
   or NONE if there isn't one
*)
fun findOdd (t : int tree) : int option =
    case t of
        Empty => NONE
      | Leaf x => (case oddP x of true => SOME x | false => NONE)
      | Node(l,r) =>
            (case findOdd l of
                 NONE => findOdd r
               | SOME x => SOME x)

val SOME 3 = findOdd (Node(Leaf 2, Node(Leaf 3, Leaf 4)))
val NONE   = findOdd (Node(Leaf 2, Node(Leaf 6, Leaf 4)))
```

Here is another way to write the same code:

```
exception NoOdd
fun findOdd (t : int tree) : int =
    case t of
        Empty => raise NoOdd
      | Leaf x => (case oddP x of true => x | false => raise NoOdd)
      | Node(l,r) => (findOdd l) handle NoOdd => findOdd r
```

In the node case, we try running `findOdd l`, and then, if that raises `NoOdd`, run `findOdd r`.

Let's step through

```
     findOdd (Node(Leaf 2, Node(Leaf 3, Leaf 4)))
|->* (findOdd (Leaf 2)) handle NoOdd => findOdd (Node(Leaf 3, Leaf 4))
|->* (case oddP 2 of true => 2 | false => raise NoOdd)
     handle NoOdd => findOdd (Node(Leaf 3, Leaf 4))
|->  (raise NoOdd) handle NoOdd => findOdd (Node(Leaf 3, Leaf 4))
|->  findOdd (Node(Leaf 3, Leaf 4))                                 [handle-raise]
|->* findOdd (Leaf 3) handle NodOdd => findOdd(Leaf 4)
|->* 3 handle NodOdd => findOdd(Leaf 4)                             [handle-value]
|->  3
```

The rule `handle-raise` says that a raise that is met by a handler steps to the body of the handler. The rule `handle-value` says that when a value reaches a handler, you discard the handler, because the expression it is around returns normally.

What happens when you have a `raise` in a context other than an immediate handler? E.g. `1 + (raise NoOdd)`? The raise *propagates up*: `1 + raise NoOdd -`¿ raise NoOdd—.

What is the type of `raise e`? Recall that an expression's type is a prediction about the value it will return, and this expression never returns a value! That is, the fact that an expression may raise an exception is not marked in its type. Consequently, `raise e : 'a` for any `'a`. `e1 handle NoOdd => e2` has type `T` if both `e1` and `e2` do: if `e1` returns normally, the expression will compute a `T`, and if it raises `NoOdd`, it will compute a `T`.

Here are some tests for `findOdd`:

```
val 3 = findOdd (Node(Leaf 2, Node(Leaf 3, Leaf 4)))
val 0 = (findOdd (Node(Leaf 2, Node(Leaf 6, Leaf 4)))) handle NoOdd => 0
val NONE = SOME (findOdd (Node(Leaf 2, Node(Leaf 6, Leaf 4))))
           handle NoOdd => NONE
```

The last shows how to convert an exception to an option!

2 Value-Carrying Exceptions

Here's another way to write the same code:

```
exception Found of int

(* raises Found v where v is the leftmost odd number in the tree,
   or returns if there is no odd number *)
fun findOdd (t : int tree) : unit =
    case t of
        Empty => ()
      | Leaf x => (case oddP x of true => raise Found x | false => ())
      | Node(l,r) => let val () = findOdd l in findOdd r end
```

The type `unit` is a nullary tuple: it has one value, `()`, and no operations (cf. the misnamed `void` in C/Java).

Let's step through

```
     findOdd (Node(Leaf 3, Leaf 4))
|->* let val () = findOdd (Leaf 3) in findOdd (Leaf 4) end
|->* let val () = (case oddP 3 of true => raise Found 3 | false => ())
      in findOdd (Leaf 4) end
|->* let val () = raise Found 3 in findOdd (Leaf 4) end
|->  raise Found 3                                           [propagation]
```

The *propagation* rules that say that "raise percolates up to the outside".

```
let val x = raise e in f end |-> raise e
(raise v + e) |-> raise v
(f (raise v)) |-> raise v
raise (raise v) |-> raise v
```

Here are some tests:

```
val () = (findOdd (Node(Leaf 2, Node(Leaf 6, Leaf 4))))
val SOME 3 = (let val () = (findOdd (Node(Leaf 2, Node(Leaf 3, Leaf 4))))
              in NONE end)
             handle Found x => SOME x
```

3 Two Exceptions at Once

Here's a final, somewhat twisted, way to write the code:

```
(* raises Found x with the leftmost odd number,
   or raises NoOdd otherwise *)
fun findOdd (t : int tree) : 'a =
    case t of
        Empty => raise NoOdd
      | Leaf x => (case oddP x of true => raise Found x | false => raise NoOdd)
      | Node(l,r) => (findOdd l) handle NoOdd => findOdd r
```

Thus function *never* returns normally: it always raises one exception or the other. Because of this, it is polymorphic: it has any result type you want.

Let's step

```
     findOdd (Node(Leaf 3, Leaf 4))
|->* findOdd (Leaf 3) handle NoOdd => findOdd (Leaf 4)
|->  raise (Found 4) handle NoOdd => findOdd (Leaf 4)
|->  raise (Found 4)                                     [reraise]
```

If a handler doesn't have a case for the exception that is being raised, then that exception is *re-raised*.

4 Exception Packets

What exactly are `Fail "spec`, `NoOdd`, `Found 3`? They are *exception packets*, which have type `exn`.

Like a datatype, the values of type `exn` are made by applying a constructor to an appropriate argument. E.g.

```
exception Found of int
```

declares a new `exn` constructor `Found : int -> exn`. The operation on `exn` is case-analysis:

```
case (e : exn) of
   Found x => x
 | NotFound => 0
```

However, unlike a datatype, where all the constructors are known, `exn` is *EXteNsible*, which means there always might be more branches. Thus, a case-analysis on `exn` will never be exhausitive, unless you have a variable or an underscore as a catch-all.

5 Rules

Know that we know about exception packets, we can state the rules for exceptions in full generality:

Typing `raise e : 'a` if `e : exn`. If you wish, you can compute the exception you raise:

```
raise (case oddP x of true => Found x | false => NoOdd)
```

The general form of `handle` is `e1 handle x => e2`, which has type `T` if `e1 : T` and `e2 : T`, assuming `x:exn`.

Evaluation

- Handle-raise: `(raise v) handle x => e' |-> e'[v/x]` if `v` is a value.
- Handle-value: `v handle x => e |-> v` if `v` is a value
- Stepping: `raise e` steps if e does. `e handle x => e'` steps if e does.
- Propagation:

  ```
  (raise v + e) |-> raise v
  (f (raise v)) |-> raise v    [if f is a value]
  raise (raise v) |-> raise v
  (raise v, f) |-> raise v
  (v1, raise v2) |-> raise v     [if v1,v2 are values]
  ```

  etc. The latter two rules say that the *leftmost* exception gets raised.

Syntactic sugar:

```
e handle p1 => e1 | ... | pn => en
```

is syntactic sugar for

```
e handle x => case x of p1 => e1 | ... | pn => en | _ => raise x
```

For example,

```
raise (Found 4) handle NoOdd => findOdd (Leaf 4)
```

is syntactic sugar for

```
raise (Found 4) handle x =>
   (case x of
      NoOdd => findOdd (Leaf 4)
    | _ => raise x)
```

which explains why it re-raises `Found 4`.

5.1 Puzzles

- Raise inside of raise: what happens with

  ```
      raise Fail (Int.toString (4 div 0))
  ```

  Answer:

  ```
      raise Fail (Int.toString (4 div 0))
  |-> raise Fail (Int.toString (raise Div))
  |-> raise Fail (raise Div)
  |-> raise (raise Div)
  |-> (raise Div)
  ```

- Handle inside of handle:

  ```
       findOdd (Node(Node (Leaf 2, Leaf 3), Leaf 4))
  |->* (findOdd (Node (Leaf 2, Leaf 3))) handle NoOdd => findOdd (Leaf 4)
  |->* ( (findOdd (Leaf 2)) handle NoOdd => findOdd (Leaf 3) )
       handle NoOdd => findOdd (Leaf 4)
  |-> ( (raise NoOdd) handle NoOdd => findOdd (Leaf 3) )
       handle NoOdd => findOdd (Leaf 4)
  ```

  What happens next?

  ```
  |-> ( findOdd (Leaf 3) )
       handle NoOdd => findOdd (Leaf 4)
  ```

  That is, you get the nearest dynamically enclosing handler. If you got the wrong one, we wouldn't search `Leaf 3` in this case.

6 When to use Exceptions

What's the difference between options and exceptions?

Options: you are forced to handle them, or it's a compile time-error Pros: type system forces you to handle failures at each step Cons: type system forces you to handle failures at each step

Exceptions: you are forced to handle, or it's a runtime error. Pros: can code as if the failures don't happen, and handle them at the end Cons: you forget to handle them at the end.

Thus, exceptions are options that subvert the type system.

It's generally good practice to use options if you ever expect someone else to run into the failures—use the type system to communicate to them what might happen! Use exceptions for spec violations. But if you advertize what exceptions you use, then clients can pretend that you wrote the "checked" version, by handling your exceptions.

Indeed, you can convert back and forth:

```
exception Failed
(* creates a function that
   raises Failed if f returns NONE
   returns x if f returns SOME x *)
fun toexn (f : 'a -> 'b option) : 'a -> 'b =
    fn x => case f x of NONE => raise Failed | SOME v => x

(* creates a function that
   returns NONE if g raises Failed
   returns SOME x if f returns x *)
fun toopt (f : 'a -> 'b) : 'a -> 'b option =
    fn x => SOME (f x) handle Failed => NONE
```