1 Exception Basics

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

1.1 Built-In Exceptions

SML has a number of built-in exceptions, which we’ve been using without really talking much about them. For example, dividing by zero:

```sml
- fun divby x = 42 div x;
val divby = fn : int -> int
- divby 0;
  uncaught exception Div [divide by zero]
  raised at: stdIn:1.19-1.22
```

or overflowing integers:

```sml
- fun pow (n, 0) = 1
  = | pow (n, m) = n * (pow (n, m - 1));
val pow = fn : int * int -> int
- pow (2, 30);
  uncaught exception Overflow [overflow]
  raised at: <file stdIn>
```

There’s also a built-in exception `Fail` that takes a string, which lets us give more information on what went wrong:

---

*based on notes by Michael Erdmann, Stephen Brookes and Brandon Bohrer*
fun unimplemented () = raise Fail "Unimplemented";
val unimplemented = fn : unit -> 'a

fun lazy_pow (n, 0) = 1
  | lazy_pow (n, m) = (* eh, it’s probably too big *) raise Overflow;
val lazy_pow = fn : 'a * int -> int

fun unimplemented ();

val unimplemented = fn : unit -> 'a

fun lazy_pow (n, 0) = 1
  | lazy_pow (n, m) = (* eh, it’s probably too big *) raise Overflow;
val lazy_pow = fn : 'a * int -> int

uncaught exception Fail [Fail: Unimplemented]
  raised at: stdIn:2.6-2.26

Above is an example of raising an exception: this is done using the keyword raise. This lets us raise exceptions in our code. Exceptions like Div and Overflow are raised by certain built-in SML functions, though we can also raise them.

fun lazy_pow (n, 0) = 1
  | lazy_pow (n, m) = (* eh, it’s probably too big *) raise Overflow;
val lazy_pow = fn : 'a * int -> int

uncaught exception Overflow [overflow]
  raised at: stdIn:7.61-7.69

1.2 User-Defined Exceptions

SML also allows programmers to declare their own exceptions (and give them names), and to design code that will “raise” a specific exception in order to signal a particular kind of runtime error. You can also “handle” an exception by providing some expression to be evaluated “instead” if a given exception gets raised. In short, exceptions can be declared (given names, available for use in a syntactically determined scope), raised, and handled. We will introduce you to the SML syntax for these constructs, by writing some simple SML functions for which it is natural to want to build in some error handling.

2 Using Exceptions

2.1 Declaring and Raising

You can declare and name an exception, with a declaration such as

exception Negative

which introduces an exception named Negative. In the scope of this declaration you can then “raise” this exception by (evaluating) the expression

raise Negative

– evaluation of this expression causes an error stop and SML reports the name of the exception. Just like any other declaration, we can use exception declarations in let-expressions, e.g.,

let exception Foo in e end
In the let-expression above, e may contain occurrences of raise Foo.

- exception Negative;
exception Negative
- fun f x = if x < 0 then raise Negative else 1;
val f = fn : int -> int
- f 2;
val it = 1 : int
- f (~2);

uncaught exception Negative
  raised at: stdIn:9.31-9.39

The specification of a function that may raise an exception should explain the circumstances in which the exception gets triggered. For example, consider the following code fragment:

```
exception Negative
(* an exception named Negative is now in scope *)

(* inc : int -> int -> int *)
(* REQUIRES: true *)
(* ENSURES: If n >=0, then for all x:int, (inc n x) returns x+n. *)
(* If n < 0, inc n raises Negative. *)
fun inc (n:int) : (int -> int)=
  if n<0 then raise Negative else (fn x:int -> x+n)
```

Raising an exception is not like returning a value. Raising an exception signals a runtime error. So for example, the expression

```
0 * (raise Negative)
```

is well-typed (with type int), but its evaluation raises the exception, and doesn’t magically return 0.

### 2.2 Handling Exceptions

Now that we have shown you how to declare your own exceptions and raise them to signal runtime errors, you may be looking for a way to design code that recovers gracefully from a runtime error by performing some expression evaluation. For instance, maybe if there is a runtime error the code might return a “default” value.

The SML syntax for “handling” an exception named Foo is

```
e1 handle Foo => e2
```

We informally explain the evaluation behavior of this expression as follows:

- If e1 evaluates to a value v, so does e1 handle Foo => e2, without even evaluating e2.
• If \( e_1 \) fails to terminate (loops forever), so does \( e_1 \text{ handle Foo => e2} \).

• If \( e_1 \) raises Foo, then

\[
(e_1 \text{ handle Foo => e2}) \Rightarrow e2
\]

and \( e_1 \text{ handle Foo => e2} \) behaves like \( e2 \) in this case.

• If \( e_1 \) raises an exception other than Foo, so does \( e_1 \text{ handle Foo => e2} \).

In \( e_1 \text{ handle Foo => e2} \) we refer to “\( \text{handle Foo => e2} \)” as a “handler” for Foo attached to \( e_1 \). This handler has a “scope” – it is only effective for catching raises of Foo that occur inside \( e_1 \).

In the following examples, figure out the evaluation behavior:

```ml
exception Foo

fun loop x = loop x

val a = (42 handle Foo => loop 0)
val b = (loop 0) handle Foo => 42
val c = ((20 + (raise Foo)) handle Foo => 22)
```

We can also build handlers for several differently named exceptions, as in

```ml
e handle Foo => e1
  | Bar => e2
```

3 Example

Recall shrubs:

```ml
datatype 'a shrub =
  Leaf of 'a
  | Node of 'a shrub * 'a shrub
```

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

```ml
(* REQUIRES: true
 * ENSURES: findOdd t ==> SOME n where n is the leftmost odd number in t,
 * or NONE if there isn't one
 *)
fun findOdd (t : int shrub) : int option =
  case t of
    Leaf x => if x mod 2 = 1 then SOME x else NONE
  | Node(l,r) =>
    (case findOdd l of
      NONE => findOdd r
      ...
```
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

(* returns the leftmost odd number in the shrub, *
* or raises NoOdd if there is no odd number *)
fun findOdd (t : int shrub) : int =
  case t of
    Leaf x => if x mod 2 = 1 then x else 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 the code:

findOdd (Node(Leaf 2, Node(Leaf 3, Leaf 4)))
  |->* (findOdd (Leaf 2)) handle NoOdd => findOdd (Node(Leaf 3, Leaf 4))
  |->* (case (2 mod 2) = 1 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)) [1]
  => findOdd (Leaf 3) handle NoOdd => findOdd(Leaf 4)
  => 3 handle NoOdd => findOdd(Leaf 4) [2]
  |-> 3

In step [1], we note that a raise that is met by a handler steps to the body of the handler. In step [2], we use the fact 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!
4 Value-Carrying Exceptions

Here’s another way to write the same code:

```hsaskell
exception Found of int

(* raises Found v where v is the leftmost odd number in the shrub, *
  * or returns if there is no odd number * )

fun findOdd (t : int shrub) : unit =
  case t of
    Leaf x => if x mod 2 = 1 then raise Found x else ()
  | Node(l,r) => let val () = findOdd l in findOdd r end

Let’s step the code:

findOdd (Node(Leaf 3, Leaf 4))
== let val () = findOdd (Leaf 3) in findOdd (Leaf 4) end
== let val () = if 3 mod 2 = 1 then raise Found 3 else ()
in findOdd (Leaf 4) end
== let val () = raise Found 3 in findOdd (Leaf 4) end
|-> raise Found 3

The propagation step [3] says 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

5 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 shrub) : 'a =
  case t of
    Leaf x => if x mod 2 = 1 then raise Found x else raise NoOdd
  | Node(l,r) => (findOdd 1) 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 the code:

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

If a handler doesn’t have a case for the exception that is being raised, then that exception is re-raised (step 4).

5.1 Comparison to CPS

The previous example may seem a lot like our use of failure continuations yesterday. Indeed, it’s pretty similar. In both cases, the function doesn’t return normally but does something different depending on whether the result is a failure or a success. The main difference is that, like the options of the first implementation, exceptions propagate up so you can handle them. The failure continuation immediately returns out to the top level.

6 Exception Packets

What exactly are Fail "spec", NoOdd, and 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. For example,

```
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 (not EXCEPTION), which means there always might be more branches. Thus, a case-analysis on exn will never be exhaustive, unless you have a variable or an underscore as a catch-all.

7 Rules

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

**Typing** raise e : 'a if e : exn.

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

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

is syntactic sugar for

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

- Handle-raise: \((\text{raise } v) \ \text{handle} \ x \Rightarrow e' \ |\rightarrow e'[v/x]\) if \(v\) is a value.
- Handle-value: \(v \ \text{handle} \ x \Rightarrow e \ |\rightarrow v\) if \(v\) is a value
- Stepping: \(\text{raise } e\) steps if \(e\) does. \(e \ \text{handle} \ x \Rightarrow e'\) steps if \(e\) does.
- Propagation:

\[
(\text{raise } v + e) \ |\rightarrow \text{raise } v \\
(f \ (\text{raise } v)) \ |\rightarrow \text{raise } v \quad \text{[if } f \text{ is a value]}
\]

\[
\text{raise } (\text{raise } v) \ |\rightarrow \text{raise } v \\
(\text{raise } v, e) \ |\rightarrow \text{raise } v \\
(v_1, \text{raise } v_2) \ |\rightarrow \text{raise } v_2 \quad \text{[if } v_1,v_2 \text{ are values]}
\]

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

7.1 Puzzles

- Raise inside of raise: what happens with
  \[
  \text{raise Fail } (\text{Int.toString } (4 \ \text{div } 0))
  \]
  Answer:
  
  \[
  \text{raise Fail } (\text{Int.toString } (4 \ \text{div } 0)) \\\n  |\rightarrow \text{raise Fail } (\text{Int.toString } (\text{raise Div})) \\
  |\rightarrow \text{raise Fail } (\text{raise Div}) \\
  |\rightarrow \text{raise } (\text{raise Div}) \\
  |\rightarrow (\text{raise Div})
  \]

- Handle inside of handle:
  
  \[
  \text{findOdd } (\text{Node}(\text{Node } (\text{Leaf } 2, \text{Leaf } 3), \text{Leaf } 4)) \\
  |\rightarrow* (\text{findOdd } (\text{Node } (\text{Leaf } 2, \text{Leaf } 3))) \ \text{handle } \text{NoOdd } \Rightarrow \text{findOdd } (\text{Leaf } 4) \\
  |\rightarrow* (\text{findOdd } (\text{Leaf } 2)) \ \text{handle } \text{NoOdd } \Rightarrow \text{findOdd } (\text{Leaf } 3) \\
  \quad \text{handle } \text{NoOdd } \Rightarrow \text{findOdd } (\text{Leaf } 4) \\
  |\rightarrow (\text{raise NoOdd}) \ \text{handle } \text{NoOdd } \Rightarrow \text{findOdd } (\text{Leaf } 3) \\
  \quad \text{handle } \text{NoOdd } \Rightarrow \text{findOdd } (\text{Leaf } 4)
  \]

  What happens next?
  
  \[
  |\rightarrow (\text{findOdd } (\text{Leaf } 3)) \\
  \quad \text{handle } \text{NoOdd } \Rightarrow \text{findOdd } (\text{Leaf } 4)
  \]

  That is, you get the nearest dynamically enclosing handler. If you got the wrong one, we wouldn’t search \text{Leaf } 3 in this case.
7.2 Frustrating Puzzles

Some caveats about exception handling:

- Handling conditionals:

  ```sml
  if raise NoOdd then 1 else 2 handle NoOdd => 3
  |-> raise NoOdd
  ```

  Why? Because

  ```sml
  if raise NoOdd then 1 else 2 handle NoOdd => 3
  ```

  parenthesizes as

  ```sml
  if raise NoOdd then 1 else (2 handle NoOdd => 3)
  ```

  Use parentheses liberally when using exception handlers unless you want to learn all of the precedence rules.

- Type carefully:

  ```sml
  let fun foo () = raise Found 5
  in
  (foo ()
  handle NOdd => "Hi!"
  end
  ==> "Hi!"
  ```

  Why? We misspelled NoOdd, so SML matched Found 5 against the variable NOdd, which matches anything. It’s harder to catch this kind of error with exception handlers because SML doesn’t give you non-exhaustive match warnings for them.

8 When to use Exceptions

What’s the difference between options and exceptions? Essentially, 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 advertise 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:

```sml
exception Failed

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

(* returns NONE if g raises Failed
 * returns SOME v if f returns v
*)

fun toopt (f : 'a -> 'b) : 'a -> 'b option =
    fn x => SOME (f x) handle Failed => NONE