Exceptions

COMP 320

13 CPS

Safety Checks

SML is a safe language.

Statically, type checking allows us to prevent errors

Dynamic checks allow us to rule out violations that cannot be detected statically

Static violations are signaled by type checking error.

Dynamic violations are signaled by raising an exception.

- division by zero, arithmetic overflow

For example:

3 div 0 will type-check

However it cannot be evaluated and will incur a runtime fault that is signaled by raising the `exception Div`.

Raising exceptions

An exception is a form of answer to the query:

What is the value of this expression?

If the expression cannot be evaluated at runtime, SML will report this by raising an exeption:

```
- 3 div 0;
uncaught exception Div [divide by zero]
   raised at: stdIN:1.4-1.7
```

Other Run-Time errors

Another source of dynamic run-time error is caused by non-exhaustive matches

```
- fun head (x::xs):int list = x;
stdIn:1.5-1.29 Warning: match nonexhaustive
          x :: xs => ...

val head = fn : int list list -> int list
- head [];

uncaught exception Match [nonexhaustive match failure]
  raised at: stdIn:1.22
```

User Defined Exceptions

We have seen examples of pre-defined exceptions.

It is also possible to introduce new exceptions to signal specific run time program errors

Exceptions may also carry values

Exceptions that carry values

```
exception Error of string ;
fun fact(n:int):int =
    if n < 0 then raise
         Error "attempt to call factorial on inputs < 0 \n"
    else if n = 0 then 1
         else n*fact(n-1)
```

This function checks for a negative argument on each recursive function call

If the original argument is non-negative, this is unnecessary

We can improve this code:

Exceptions that carry values

```
exception Error of string ;
fun fact(n:int):int =
  let
     fun f(n:int):int =
       if n = 0 then 1
       else n*f(n-1)
  in
    if n < 0 then raise
         Error "attempt to call factorial on inputs < 0 \n"
    else f(n)
  end;
```

Exception Handlers

Raising an exception that signals an error condition is fatal

The program terminates with the raised exception

However, it is possible to use an exception handler to “catch” the raised exception and continue evaluation along another path

That is, exceptions can be used to implement non-local transfer of control

Exception Handlers

Consider this driver for the factorial function:

```
fun runFact (n) =
  let
    val r = fact(n)
  in
    print ("Factorial of " ^ Int.toString n ^ " is "
               ^ Int.toString r ^ "\n")
  end
 handle Error msg => print ("Error: Invariant violated --" ^ msg )
```

Exception Handlers

If called with an argument of 6, the function returns 24 and the handler is ignored

If called with an argument of ~6, the function raises an exception. The exception value is matched against the handler and evaluation resumes with the body of the handler

If no pattern matches, the exception is re-raised so that outer handlers may act on it

If no handler matches the exception, the computation is aborted with the uncaught exception

More Formally

SML has handle expressions with syntax:

```
        e0 handle p1 => e1
```

Type Checking:

If e0 and e1 have type T, the expression has type T

Evaluation:

- Evaluate e0
- If e0 evaluates to v0 and no exceptions are raised, the expression evaluates to v0
- If e0 raises an exception matching p1, then e1 is evaluated
- If e1 evaluates to v1 and no exceptions are raised, the expression evaluates to v1
- If e1 raises an exception then the handle also raises that exception

User Defined Exceptions

In the following example we find the “first”, meaning the leftmost odd number in a binary tree

```
fun oddP n =
      case n mod 2 of 1 => true | _ => false

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

Using Options

```
(* return SOME(the first odd number) 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
```

Using Exceptions instead

```
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
```

Raising an Exception

Evaluate the expression:

```
 findOdd (Node(Leaf 2, Node(Leaf 3, Leaf 4)))
 |->* …
 |->
```

An exception is raised by the left subtree

The handler is discarded when the value 3 is returned

Semantics

Expression handlers have the form `exp handle case`

This allows for non-local transfer of control.

We may catch a raised exception and continue evaluating along some other path.

SML attempts to evaluate an expression.

- If this yields a value, then this value is returned.
- If evaluation of expression raises an exception exc then exc is matched against in the body of the handler to see how to proceed

If there is no handler or no successful match, the exception will re-raised and propagated to the next higher level.

If it is never caught at any higher level, the computation will abort with an uncaught exception exc

Exceptions

The primary benefit of exceptions:

1. They force you to consider the exceptional case.
2. They allow you to segregate the special case from the normal case in the code (often avoids clutter)

Propagating Values

We may want to propagate a value together with the exception!

Rules for propagating exceptions include:

u

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

Propagating Values

We may want to propagate a value together with the exception!

The SML type unit has one value, () and no operations. (Think of the type void in C or Java)

```
exception Found of int

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
```

Example

```
findOdd (Node(Leaf 2, Node(Leaf 3, Leaf 4))

|->

…

|-> raise Found 3
```

Multiple Exceptions

Usually the excessive use of exceptions is not a good idea but we are able to do things like the following

It never returns normally but always raises some exception

```
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
```

Rules for Exceptions

The values produced by exceptions are of type exn (EXteNsible) since there might be more branches than specified

The operation on values of type exn is case analysis

Type Rules:

```
raise e : ‘a if e : exn
```

Handles have the form:

```
e1 handle x => e2

type T if e1:T and e2:T and x:exn
```

Evaluation Rules

Handle-raise:

```
(raise v) handle x => e’ |-> e’[v/x]
```

Handle-value:

```
v handle x => e |-> v, if v is a value
```

Propogation:

```
let val x = raise v in f end |-> raise v

f (raise v) |-> raise v

raise (raise v) |-> raise v

(raise v, f) |-> raise v
```

Options vs Exceptions

Options:

- The type system forces you to handle failures or there is a compile-time error

Exceptions

- It’s a runtime error if you don’t handle them
- You can code as if the errors won’t happen and then handle them at the end
- But if you don’t handle them you get a runtime error

Converting between them

```
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 => v
```

Converting between them

```
(* 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
```