Lesson 12
EXCEPTIONS

June 22, 2023

Last time, we learned about **continuation-passing style**. We learned that we could make our control flow even more explicit by, instead of implicitly using the return values of recursive functions, instead passing lambda expressions (**continuations**) denoting the computation to be done next to those recursive functions.

This separated concerns when it came to functions which had branching control flow behavior, as well as allowing us to achieve clean tail-recursive code, by drawing a distinction between **writing now** and **remembering later**.

We carried out a mechanistic process of CPS translation on various functions to achieve this.

1 - Exceptions

Recall our definition for all the possible behaviors of an expression:

- Evaluate to a value
- Loop forever
- **Raise an exception**

Until now, we've given an intentionally bare bones treatment of exceptions. Now, it is time to dive into exceptions in more detail.

One of the first examples of expressions that we ever saw was the expression `1 div 0`. This expression raises an exception, when evaluated.

```
Standard ML of New Jersey (64-bit) v110.99.3 [built: Thu Jul
   28 00:35:16 2022]
- 1 div 0;

uncaught exception Div [divide by zero]
  raised at: stdIn:1.4-1.7
```

Exceptions are baked into many common processes in SML. In fact, there are three very essential ones we will discuss today!

We see exceptions like `Div`, which are specific to certain functions.

There are more fundamental exceptions which are raised upon given patterns of behaviors in SML. These exceptions are:

1. `Match`, which is raised when a nonexhaustive match receives an input that fails to match any of its cases
2. `Bind`, which is raised when a `val` binding tries to bind something matching a particular pattern, and receives one which does not match

If you were to write the following lambda expression in SML:

```
val f = fn 1 => 2
```

you would receive the following warning:

```
stdIn:1.10-1.19 Warning: match nonexhaustive
          1 => ...
```

What happened? We didn't specify what would happen on all possible inputs! The function is only defined on the input `1`.

This is perfectly legal, and SML will let you proceed, but upon being given an invalid input:

```
- f 2;

uncaught exception Match [nonexhaustive match failure]
  raised at: stdIn:1.19
```

we get an exception `Match`.

In essence, you could think of all such nonexhaustive cases (and this function, in particular) as being defined implicitly as

```
fn 1 => 2 | _ => raise Match
```

An archaic way of writing test cases is to try to bind an expression to a constant pattern:

```
val 6 = fact 3
```

This will proceed without a hitch. If the pattern were to not match the returned value, however:

```
- val 5 = fact 3;
stdIn:3.5-3.15 Warning: binding not exhaustive
          5 = ...

uncaught exception Bind [nonexhaustive binding failure]
  raised at: stdIn:3.5-3.15
```

we would now get an exception `Bind` being raised.

We see that `Bind` and `Match` are more consequences of programming in SML, and aren't specific to any functions' logic.

For exceptions like `Div`, we use them to *escape* from having to return a value, when given some input. Instead of returning a value, we simply abort execution.

At this point in the semester, we have seen functions which return optional values, which might beg the question – why use exceptions in the first place?

We see that for any function which raises exceptions, we can produce an equivalent function which returns an optional value, which instead returns `NONE` in any exceptional cases:

```
infix safeDiv

fun n safeDiv 0 = NONE
  | n safeDiv d = SOME (n div d)
```

Why use exceptions at all, then, when they might cause a program to unexpectedly crash?

Let's take a look at the following specifications:

`div : int * int -> int`
REQUIRES: `n > 0`
ENSURES: `div (n, d)` evaluates to the floor of `n` divided by `d`

`safeDiv : int * int -> int option`
REQUIRES: `true`
ENSURES: `safeDiv (n, d)` evaluates to `NONE` if `d` is `0`, and `SOME (n div 0)` otherwise

They look rather similar! The difference is that `safeDiv` has moved the precondition into the type of its return value.

In practice, `safeDiv` turns out to be the safer option, to no surprise.

Forcing the caller to handle the exceptional case by handling the `NONE` is a **type-level distinction**, that causes code which does not acknowledge the possible failure to not compile.

This is a really strong enforcement, and leads to code which cannot fail to address the failure!

Occasionally, however, this can prove to be more of a burden than a safety net.

For instance, we might have occasions where we know for sure that an exceptional case cannot be reached. Suppose we are implementing the following function to collect the average grade of every student in 150:

```
fun averageGrade (grades : int list) : int =
  (List.foldr op+ 0 grades)
  div
  (List.length grades)
```

Suppose that we wanted to be rid of exceptions, however:

```
fun averageGradesSafe (grades : int list) : int option =
  (List.foldr op+ 0 grades)
  safeDiv
  (List.length grades)
```

Now, it is the responsibility of the caller to handle the `NONE` case!

The only sensible thing to do (if not raising an exception) is often just to propagate the `option`, and cause all of the dependencies to also need to return optional values.

This quickly gets messy.

The thing is, this entire mess was never really necessary.

Unless something is deeply wrong, it's a fairly safe bet that the database containing grade data for the entire class isn't empty.[1]

At a certain point, we need to be able to trust the data that we input. Although there is the *possibility* of a failure case, in realistic situations, quite often there is no reason to believe that they are possible.

So in this case, we might prefer the exceptional behavior, because it leads to cleaner code in a hypothetically impossible case. We have an *implicit precondition* on our inputs.

[1] If it is, I have bigger problems.

This is exactly the logic that has caused a million and one bugs in production code before.

This is why judicious use of exceptions is important! In some cases, it really is OK to raise an exception in a failure case, depending on how bad the failure case is.

It might seem that an unrecoverable error that completely crashes the running process might be worth no amount of code golfing. Luckily, SML does have ways of dealing with raised exceptions.

2 - Using Exceptions

We can raise exceptions ourselves, as you may have seen many times before.

What expressions are we allowed to raise? SML has a type of exception values, which is called `exn`. This may stand for "exception name".

So for instance, valid constant constructors of type `exn` include:

- `Match`
- `Bind`
- `Div`

but not `Fail : string -> exn`, which takes in an additional argument of type `string` before it can be raised.

We use the syntax `raise e` to raise exception `e`, given that `e : exn`.

Note `raise` is not a function, it merely looks like one. So `List.map raise` is not a valid expression.

We said that `raise e` is an expression which raises the expression `e`. To evaluate, however, it needs to have a type, but what should its type be?

Given that it never returns a value, it doesn't actually matter. For our purposes, it's important to use that an expression of type `t` returns a value of type `t`, *if it returns at all*.

Since `raise` never returns, we are free to give it any type. So `raise e` has type `'a`, in that it can take on any type.

SML provides a language construct called `handle`. Here's an example of how we might use it, instead:

```
fun reportGrades (grades : (string * int) list) =
  (let
    val grades = List.map snd records
  in
    "The average was " ^ Int.toString (averageGrade grades)
  end)
  handle Div => "ERROR: No grades found"
```

In this case, we use `handle` to evaluate the body of the function to an error string in the case where `Div` is raised. Otherwise, we evaluate normally.

A `handle` expression has the following behaviors:

The expression

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

has type `t` only if `e : t`, `e1 : t`, ..., `en : t`.

In addition, `p1`, ..., `pn` must all be patterns of type `exn`.

The behavior of the above `handle` expression is that:

- If the expression `e` never raises an exception, then it evaluates to `e`
- If the expression `e` raises exception `ex`, then it matches to the first `pi` that matches `ex`
- If the exception `e` raises matches none of the `handle` cases, it raises that exception again

Here are some examples of `handle` expressions, and how they evaluate:

- `raise Div` raises exception `Div`
- `(raise Div) handle Div => 2` ↪ `2`
- `(raise Div) handle Bind => 2` raises exception `Div`
- `(raise Div) handle Bind => 1 | Div => 2` ↪ `2`
- `2 handle Div => 3` ↪ `3`

The great strength of `handle` is that it can be used to handle exceptions from *anywhere inside of the enclosing expression*.

Key For non `handle` expressions, if a sub-expression raises an exception, then that expression does too.

This means that exceptions *propagate outwards* from where they were initially raised. This process continues until they reach the *nearest enclosing handler*, at which point they are possibly handled, or keep going.

This means that, essentially, exceptions allow a program to stop doing what it is doing, and resume control flow at an earlier point in time.

So in the expression `1 + (3 * (4 div 0)) handle Div => 5`:

The expression `4 div 0` raises `Div`.

The sub-expression `4 div 0` of `3 * (4 div 0)` raises `Div`, so `3 * (4 div 0)` raises it too.

The sub-expression `3 * (4 div 0)` of `1 + (3 * (4 div 0))` raises `Div`, so `1 + (3 * (4 div 0))` raises it too.

The sub-expression `1 + (3 * (4 div 0))` of `1 + (3 * (4 div 0)) handle Div => 5` raises `Div`. Since this entire expression is a `handle`, we match `Div` to the handler's first case, and evaluate to `5`.

This is an example of *nonlocal control flow*, because when evaluating the expression `1 + (3 * (4 div 0))`, our `handle` expression lets us skip directly from a deeply nested `div` call directly back up to the handled expression!

Before, we saw there was a relationship between functions which returned options, and functions who instead raised exceptions.

Let's generalize. Suppose we have a function `f : t1 -> t2`, which is possibly exception-raising. Then, we can define `f_opt` with the following spec:

`f_opt : t1 -> t2 option`
REQUIRES: `true`
ENSURES: For all values `x : t1`:

$$\texttt{f\_opt x} \cong \begin{cases} \texttt{SOME res}, & \text{if } \texttt{f x} \hookrightarrow \texttt{res} \\ \texttt{NONE}, & \text{if } \texttt{f x} \text{ raises an exception} \\ \text{loops forever}, & \text{otherwise} \end{cases}$$

Using `handle`, we can quickly go between these functions. Given `f`, as previously described, we define `f_opt : t1 -> t2` as:

```
fun f_opt x = (SOME (f x)) handle _ => NONE
```

Check your understanding Why did we have to surround `f x` with `SOME`? What would happen to the type if we didn't?

Warning It is a really bad idea to use `_` in a `handle` clause. This is because you might swallow up *any* possible exception, rather than just the one you are interested in. This could obfuscate certain errors.

If you know the precise exception that `f` raises, it's better to case on that here.

Let's look at a particular example. Suppose we have the following function:

```
fun f x =
  ("Divided to " ^ Int.toString x)
  handle Div => "Divided by zero!"
```

Check your understanding Is it true that `f (1 div 0)` ≅ `"Divided by zero!"`?

Answer: No, it is not! SML is an eagerly evaluated language, so the exception `Div` is raised before we enter the body of `f`. The handler never applies, because the handler is only within the definition of `f`.

Exceptions are what we call a **side effect**.

Def A **side effect** is an effect of an expression which is not just computing a value. For instance, reading from a file, printing to the console, or raising an exception.[2]

Side effects tend to make our definitions of equivalence fuzzy. For instance, with exceptions, we cannot freely exchange the order of unrelated `val` bindings without possibly changing the behavior of the program:

```
val _ = raise Bind
val _ = raise Match
```

These two bindings are unrelated, but the order in which they happen matters! This can make mathematical reasoning annoying.

[2] One can also consider infinite loops a side effect.

Recall that our definition of extensional equivalence maintains that `e1 : t` and `e2 : t` are only extensionally equivalent if they evaluate to equivalent values, both loop forever, or **both raise the same exception**.

A fun fact that goes along with that is that, in a world with exceptions, it is no longer true that `e1 + e2` $\cong$ `e2 + e1`, where `e1, e2` are both expressions of type `int`.

This is because `(raise Div) + (raise Bind)` raises `Div`, but `(raise Bind) + (raise Div)` raises `Bind`! This is another reason why *valuability* (and by extension, totality) is important.

3 - Custom Exceptions

Suppose we wanted to revisit our old friend, `fact`.

```
fun fact 0 = 1
  | fact n = n * fact (n - 1)
```

We know that `fact` loops forever on negative inputs. Generally, this is undesirable, because it can be difficult to discern an infinite loop from a program which is just taking a really long time.

Let's define `fact_exn`[3], which raises an exception on negative inputs!

...But what exception should we raise?

[3] Labeling functions which possibly raise an exception with the suffix `_exn` is a common practice in the OCaml language, and in my opinion, a really good practice.

We could define the following:

```
fun fact_exn 0 = 1
  | fact_exn n =
      if n < 0 then
        raise Bind
      else
        n * fact_exn (n - 1)
```

But this isn't actually a case where we failed to produce a binding, which is what the `Bind` exception is supposed to be for. If we handled `Bind` elsewhere outside callers to this function, we might end up in a handler we didn't mean to!

We could raise `Fail "negative number"`, but this becomes problematic to pattern match on. How can we do better?

It turns out, we can! `exn` is a special type, because while it can be thought of as the `datatype` declared via

```
datatype exn = Match | Bind | Div | Fail of string | ...
```

it's actually more special than that! `exn` is the only **extensible type**.[4]

Def An **extensible type** is one where constructors can be added to it *after the type is declared*. `exn` is the only example of this in Standard ML.

We can write something like

```
exception Fact
```

to declare a new constructor `Fact : exn`, for the `exn` type.

[4] In fact, some say that `exn` stands for *extensible*. Some people also say this is gaslighting, and it doesn't stand for that all, though.

So now, we can define `fact_exn` better:

```
exception Fact

fun fact_exn 0 = 1
  | fact_exn n =
      if n < 0 then
        raise Fact
      else
        n * fact_exn (n - 1)
```

After this, the expression

```
(SOME (fact_exn (~1))) handle Fact => NONE
```

will evaluate to `NONE`.

We can also define exceptions which take in arguments:

```
exception Error of string

fun runProcess (f : unit -> string) : string =
  ("OUTPUT: " ^ f ())
  handle Error s => ("ERROR: " ^ s)
```

such that `runProcess (fn () => "foo")` ↪ `"OUTPUT: foo"`

and `runProcess (fn () => raise Error "bad")` ↪ `"ERROR: bad"`

4 - Exceptional Control Flow

Exceptions are useful for writing code when we want to quickly be able to escape from some error case!

We can also use them for their control-flow abilities. We saw in the last lecture how we can use continuations to make our control flow explicit, by passing around instructions on what to do in certain cases.

With exceptions, we can go the other way and make our control flow even more implicit, by relying on casing on exceptions are raised.

For instance, let's do the `search` function using exceptions:

```
searchEHS : ('a -> bool) -> 'a tree -> 'a
```
REQUIRES: `p` is total
ENSURES: `searchEHS p T` raises `NotFound` if there is no element in `T` that satisfies `p`. Otherwise, it returns the first element in its preorder traversal which does.

```
exception NotFound

fun searchEHS p Empty = raise NotFound
  | searchEHS p (Node (L, x, R)) =
      if p x then
        x
      else
        (searchEHS p L) handle NotFound => searchEHS p R
```

In this case, we handle the `NotFound` exception, which should be raised by our specification *only* in the case where we fail to find a satisfying element in that subtree.

This is known as **exception-handling style**.

We can compare with the implementation of `search` using CPS we saw last lecture:

```
fun searchCPS p Empty sc fc = fc ()
  | searchCPS p (Node (L, x, R)) sc fc =
      if p x then
        sc x
      else
        searchCPS p L
          (fn res => sc res)
          (fn () => searchCPS p R sc fc)'
```

We see that the `fc` cases correspond to our raising of the `NotFound` exception, and the success continuation corresponds simply to returning a value.

Implicit control flow is generally a bad thing. While functions written in CPS *can* be also written with exceptions, there is little reason to do this in actual code.

This is largely because exceptions are a nightmare to debug. Nonlocal control flow transfer introduces precisely the problem that `goto` statements had, which is a lack of transparency!

Business logic is generally better done without exceptions, but exceptions are great for error cases, when the program just needs to find a way to exit gracefully.

We can go even more crazy with it. We can return data *only* using exceptions.

Consider the following specification:

```
listmax_exn : int list -> 'a
```
REQUIRES: `true`
ENSURES:

$$\texttt{listmax\_exn L} \cong \begin{cases} \texttt{raise (Max m)}, & \text{if } \texttt{m} \text{ is the max element of } \texttt{L} \\ \texttt{raise NoMax}, & \text{if } \texttt{L} \cong \texttt{[]} \end{cases}$$

This function never evaluates to a value.

We can then implement `listmax_exn` like so:

```
exception NoMax
exception Max of int

fun listmax_exn []  = raise NoMax
  | listmax_exn [x] = raise (Max x)
  | listmax_exn (x::xs) =
      (listmax_exn xs) handle
        Max y => raise (Max (Int.max (x, y)))
```

There's very little reason to do this. But it sure looks fun.

There is nothing inherently wrong with exceptions. Because of the fact that they are not represented in the type system whatsoever, generally there is a preference to avoid them where possible, but there are use cases.

When dealing with cases where preconditions really are very clear, it is not wrong to use functions which can possibly be exception-raising, *so long as to do otherwise would be a significant detriment*. Clean code comes first, but not at the risk of compromising the safety of our software!

A very common use case for exceptions is to implement error handling, when the software definitely does not need to resume, but can just print out information, and then exit.

Thank you!