15–150: Principles of Functional Programming

Exceptions

Michael Erdmann*

Spring 2023

1 Topics

- Exceptions
- Extensional equivalence, revisited
- Examples to be discussed:
  - Dealing with arithmetic errors
  - Making change

2 Introduction

So far we have for the most part dealt only with well-typed expressions, which when evaluated either terminate (by reducing to a value), or loop forever. However, when writing code even for simple problems we have already seen that this isn't the full story. We have seen examples such as `1 div 0` and `fact 100`. If we try to evaluate `1 div 0` (a well-typed expression of type `int`) SML says "`uncaught exception Div`"; no value is produced and evaluation stops. If we try to evaluate `fact 100` (again an expression of type `int`, assuming we have defined `fact : int -> int` as we did a while ago) SML will complain about `exception Overflow`.

2.1 Built-In Exceptions

SML has some "built-in" exceptions with suggestive names that signal runtime errors caused by common problems – e.g., division by zero, or integer value too large. There is also an exception that happens when you try to apply a function to an argument value for which the function's definition doesn't include a suitable clause (so there is no pattern that matches the argument value). Here is an example, albeit a pretty silly function.

```
- fun foo [ ] = 0;
stdIn:1.5-1.16 Warning: match nonexhaustive
          nil => ...
```

*Adapted from a document by Stephen Brookes.

```
val foo = fn : 'a list -> int
- foo [1,2,3];

uncaught exception Match [nonexhaustive match failure]
```

A "Warning: match nonexhaustive" comment isn't fatal, i.e., SML will let you keep going. Indeed, SML also says that `foo` is well-typed, with the type `'a list -> int`.

But it should prompt you to examine your code to see whether you have missed some cases and your function isn't going to work for all values of its intended argument type. As we show above, using `foo` on any non-empty list causes a runtime error. (And since SML can't tell just by looking at syntax whether or not a well-typed expression of type `t list` represents an empty list, there is no reasonable way to expect the SML type system to treat this kind of error as "compile time" rather than "runtime".)

Here again are some familiar functions and some uses that cause runtime errors.

```
- fun fact n = if n>0 then n*fact(n-1) else 1;
val fact = fn : int -> int
- fact 100;

uncaught exception Overflow [overflow]
```

Note: We generally ignore `Overflow` in our proofs but you might see it for some of your programs.

```
- fun divmod(x,y) = (x div y, x mod y);
val divmod = fn : int * int -> int * int
- divmod(2,0);

uncaught exception Div [divide by zero]
```

2.2 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 (e.g., an inappropriate index into an array of some kind). 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.

3 Using Exceptions

3.1 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 and `local`-declarations, e.g.,

```
let exception Foo in e end
local exception Foo in fun f(x) = e end
```

In the let-expression above, `e` may contain occurrences of `raise Foo`. In the local-declaration above, `e` may contain occurrences of `raise Foo`, and the function `f` defined by this declaration may raise `Foo` when it is applied to an argument.

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

uncaught exception Neg
```

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) evaluates to 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.

For example, the expression

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

is well-typed (with type `int`), but its evaluation raises the exception `Negative`. It does *not* magically return 0.

Typing Why does the previous expression have type `int`? Because of the following typing rules:

- An exception has type `exn`, which is short for "exception", though some people prefer to say it is short for "extensible". Extensible means that one can add constructors (i.e., new exceptions) at runtime. So `Negative : exn`.
- *Raising* an exception has most general type `'a`. For instance, the most general type of the expression `raise Negative` is `'a`.
  That type specializes to `int` in the expression `0 * (raise Negative)` above. Thus that expression is well-typed, with type `int`.

Exceptions that carry data It is possible to declare exceptions that take arguments. For instance, the exception `Fail` is predefined in SML as:

```
exception Fail of string
```

We then have the following types:

```
Fail : string -> exn
Fail "oops" : exn
raise Fail "oops" : 'a
```

3.2 An Example: gcd, revisited

To illustrate the design issues involved when we plan to write code that uses exceptions, let's go back to the gcd function. Here is one possible implementation, based on subtraction:

```
(* gcd : int * int -> int                                              *)
(* REQUIRES: x>0, y>0                                                  *)
(* ENSURES: gcd(x, y) computes the greatest common divisor 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))
```

If the values of `x` and `y` are positive integers, `gcd(x,y)` evaluates to the g.c.d. of `x` and `y`. The spec says this, and exactly this. Once can prove that the function satisfies this spec. But if `x` or `y` is 0 or less, `gcd(x, y)` either loops forever (e.g., `gcd(1,0)`) or causes integer overflow (e.g., `gcd(100, ~1)`). Since this bad behavior is easy to predict, simply by checking the sign of `x` and `y` before doing any arithmetic, we could introduce an exception with a suitably suggestive name, and modify the function and spec accordingly:

```
exception NotPositive
(* GCD : int * int -> int                                          *)
(* REQUIRES: true                                                  *)
(* ENSURES: GCD(x,y) ≅ gcd(x,y)            if x>0 and y>0.  *)
(*          GCD(x,y) ≅ raise NotPositive   if x<=0 or y<=0. *)
fun GCD (x, y) =
      if x <=0 orelse y <= 0 then raise NotPositive else gcd(x,y)
```

What we have done is to take a function `gcd` which only works "properly" on positive arguments, and built a "bullet-proof" version `GCD` that also deals gracefully with bad arguments. Given bad arguments, the code raises a specific exception which surrounding code will be able to "handle", as we will see shortly.

Question

What is bad about the following function definition?

```
exception NotPositive

(* GCD : int * int -> int *)
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))
```

Answer: it pointlessly re-checks the signs in every recursive call.

Another way to define an exceptional version of gcd is given below. In this version (which we name `GCD'` so that we can contrast it with `GCD` as defined above) the "original" `gcd` function is kept local, so isn't exported for use elsewhere – only `GCD'` is available in the scope of this declaration:

```
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
```

The two functions

```
GCD : int * int -> int
GCD' : int * int -> int
```

are extensionally equivalent.

3.3 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
```

This is well-typed, with type `t`, if `e1` and `e2` have the same type `t`. We 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 `e1` fails to terminate (loops forever), so does `e1 handle Foo => e2`.
- If `e1` raises `Foo`, then

  ```
  (e1 handle Foo => e2) ==> e2
  ```

  and `e1 handle Foo => e2` behaves like `e2` in this case.
- If `e1` raises an exception other than `Foo`, so does `e1 handle Foo => e2`.

In `e1 handle Foo => e2` we refer to "`handle Foo => e2`"
as a "handler" for `Foo` attached to `e1`. This handler has a "scope" – it is only effective for catching raises of `Foo` that occur inside `e1`.

In the following examples, figure out the evaluation behavior:

```
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

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

Again, this is well-typed with type `t` if `e`, `e1`, and `e2` all have type `t`.
Exercise: describe the evaluational behavior of the previous expression, based on how `e` evaluates. (Comment: We may define handlers with more than two alternatives, using pattern matching on exceptions, much like in a case statement.)

As an example, consider again the `gcd` code. We might want to handle an argument error by returning the default value 0 (although this doesn't qualify as a "greatest common divisor"), and we could do this by defining

```
fun GCD'' (x, y) = GCD(x,y) handle NotPositive => 0
```

4 Case Study: Making Change

To put all these ingredients together, we will implement a function to find a way to make change for a given amount of money, using (arbitrarily many) coins from a fixed list of denominations (coin sizes). This is a classic example of a problem ("How can you make change for $\$A$ using coins of sizes $[\$c_1, \ldots, \$c_n]$?") We use the word "coins" even though in reality we might be thinking of paper money. When it is impossible to make change we want a definite "no", and when it is possible we want *some* combination of coins that adds up to the desired amount.

We assume that coin denominations are strictly positive integers and that we make change for nonnegative amounts.

Recall the function `sum : int list -> int` that adds up the integers in a list. Here is one possible implementation:

```
fun sum (C:int list):int = foldr (op +) 0 C
```

So let's start by defining

```
exception Change
```

and giving the specification

```
(* change : int list * int -> int list                                    *)
(* REQUIRES: L is a list of strictly positive integers and A >= 0.        *)
(* ENSURES:                                                               *)
(*     EITHER change(L, A) ==> C,                                         *)
(*            where C is a list of items drawn (possibly with duplicates) *)
(*            from the list L, such that    sum C ≅ A;                    *)
(*     OR     change(L, A) raises Change and there is no such list C.     *)
```

First attempt

Before starting to write our function, we make some simple suggestions. (We are suggesting here a *greedy algorithm*.) Remember that we assume `L` is a list of strictly positive integers, and `A` is non-negative.

- If `A` is zero, we can make change for `A` using no coins.
- If `A>0` and the coin list is `c::R`,
  - if `c` is bigger than `A`, then we can't use it;
  - otherwise, we use `c` and make change for `A-c`.

The following function definition is based very closely on the above sketch of an algorithm. It is called a greedy algorithm because in the interesting case it assumes that we can use the first coin in the list.

```
(* 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))
```

Do you see a problem? Well, SML does. We get this message in the REPL:

```
stdIn:60.5-63.41 Warning: match nonexhaustive
          (_,0) => ...
          (c :: R,A) => ...

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

The function does have the intended type, but SML warns us that the clauses are not exhaustive, i.e., there might be some coin lists and/or amounts for which the function won't be applicable. Looking again at the function definition, we see that there is no clause for `change ([ ], n)` except for `n=0`. We forgot to deal with the situation in which the function is supposed to make change for a positive dollar amount using no coins.

Second attempt

Here is a second attempt, based on the following extension of the algorithm outline from above:

- If `A` is zero, we can make change for `A` using no coins.
- If `A>0` and the coin list is `c::R`,
  - if `c` is bigger than `A`, then we can't use it;
  - otherwise, we use `c` and try to make change for `A-c`.
- If `A>0` and the coin list is empty, then there is no way to make change for `A`, so we raise the previously declared exception `Change`.

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

Now the "match nonexhaustive" warning goes away. And we get the following:

```
- change([ ], 42);

uncaught exception Change
```

so the intended exception is getting raised.

However, we *still* get wrong results. For example,

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

also produces

```
uncaught exception Change
```

We really wanted to get a list such as `[3,3,3]` or `[2,2,2,3]` instead. The problem is that the algorithm is too greedy. If it turns out that there *is* a way to make change for `A` but only by *not* using the first coin, the existing code will not discover it. Since we are focused here on exceptions, let's consider compensating by using a *handler* that allows the code to recover from these unintended exceptions.

Third and final attempt

Here is a third amended outline of the algorithm:

- If `A` is zero, we can make change for `A` using no coins.
- If `A>0` and the coin list is `c::R`,
  - if `c` is bigger than `A`, then we can't use it;
  - otherwise, we *try* to use `c` and try to make change for `A-c`. If that works, fine; if not, we handle the exception that winds up being raised, now by trying to make change without using `c`.
- If `A>0` and the coin list is empty, then there is no way to make change for `A`, so we raise `Change`.

Here is the re-revised function definition:

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

The handler is attached to the `else` branch of the third clause:

```
    c :: (change (c::R, A-c)) handle Change => change(R, A)
```

Would it make any difference if we used instead

```
   c :: (change (c::R, A-c) handle Change => change(R, A))?
```

Yes, this would give incorrect results. Figure out why.

If we test this function on the examples from before, we will get the correct responses. In particular,

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

Now we have, finally, a function that satisfies the intended specification. And we can use it to obtain a function

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

as follows:

```
fun mkchange (coins, A) =
    SOME (change (coins, A)) handle Change => NONE
```

Exercise: give a suitable spec for `mkchange`.

5 Extensional Equivalence Revisited

Functional programs that may use exceptions have a potential for a more dangerous range of behaviors than completely pure programs: a well-typed expression may, when evaluated, either terminate (reduce to a value), or fail to terminate (loop forever), or it may instead raise an exception.

We define two expressions of type `exn` to be extensionally equivalent in the usual way we would for datatypes: Two exception expressions are extensionally equivalent if either (i) they are the same constant exception or (ii) one is of the form `E(e)` and the other is of the form `E(e')`, with `E` an exception constructor of type `t -> exn` and `e` and `e'` expressions of type `t` such that `e` $\cong$ `e'`.

At runtime, in order to raise exception `E(e)`, one must first reduce `e` to some value `v` of type `t`. Similarly, in order to raise `E(e')`, one must first reduce `e'` to some value `v'` of type `t`. Then two raised exceptions of the form `raise E(v)` and `raise E(v')` are extensionally equivalent if and only if `v` $\cong$ `v'`.

We have already included the possibility of raising exceptions in our definition of extensional equivalence, so let us recall the definition:

- Two expressions, `e` and `e'`, of type `t`, with `t` not a function type, are *extensionally equivalent* (written `e` $\cong$ `e'`) iff either both `e` and `e'` evaluate to extensionally equivalent values, or both loop forever, or both raise extensionally equivalent exceptions.
- Two expressions of type `t -> t'` are extensionally equivalent iff either they evaluate to function values that are extensionally equivalent, or both loop forever, or both raise extensionally equivalent exceptions.
- Two function values `f,g: t -> t'` are extensionally equivalent (again written as `f` $\cong$ `g`) iff, for all values `v` of type `t`, `f(v)` $\cong$ `g(v)`. This means that for all values `v` of type `t`, either `f(v)` and `g(v)` both evaluate to extensionally equivalent values, or both loop forever, or both raise extensionally equivalent exceptions.

Importantly, we have referential transparency:

> Replacing a sub-expression by an extensionally equivalent sub-expression produces an expression that is extensionally equivalent to the original one.

6 Exception Equivalences

Assume that `e` and `e1, ..., ek` are well-typed expressions of type `t`, that `v` is a value of type `t`, and that `E1,..., Ek` are (distinct) exceptions currently in scope. (We assume these are constant exception constructors, i.e., exceptions that take no arguments, for simplicity. The discussion here generalizes naturally.)

The following equivalences hold:

(1) If `e` $\cong$ `v`, then
`e handle E1 => e1 | . . . | Ek => ek` $\cong$ `v`

(2) For `1 <= i <= k`,
`(raise Ei) handle E1 => e1 | . . . | Ek => ek` $\cong$ `ei`

(3) If `E` is an exception not in {`E1, . . ., Ek`} then
`(raise E) handle E1 => e1 | . . . | Ek => ek` $\cong$ `raise E`

We further have these equivalences:

If `e`, `e'`, `e1,...,ek`, `e1',...,ek'` are all well-typed expressions of type `t`, then

(4) If `e` $\cong$ `e'` then
`(e handle E1 => e1 | . . . | Ek => ek)`
$\cong$ `(e' handle E1 => e1 | . . . | Ek => ek)`

(5) If `e1` $\cong$ `e1', . . ., ek` $\cong$ `ek', then`
`(e handle E1 => e1 | . . . | Ek => ek)`
$\cong$ `(e handle E1 => e1' | . . . | Ek => ek')`