15-150 Fall 2013
Lecture 15

Stephen Brookes

Tuesday, 15 October

1 Topics

- Exceptions
- Referential transparency, revisited
- Examples to be discussed:
  - Dealing with common errors
  - Computing greatest common divisors, robustly
  - Making change
  - The n-queens, revisited

2 Introduction

So far we've been working with well-typed, *purely functional* expressions, which when evaluated either terminate and return a value, or loop forever. We've introduced the *type guarantee*: whenever an expression of type `t` has a value `v`, that value is a value of type `t`. And we've introduced a notion of equality for expressions at each type, based on extensionality and evaluation. We've exploited the principle of referential transparency: it's safe to replace a sub-expression by any other sub-expression that is "equal", and the entire program will still be "equal" to the original version.

However, when writing code even for simple problems we've already seen that this isn't the full story! We've 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`, ML says "`uncaught exception Div`" or some other similarly worded error message; no value is produced, and evaluation stops. And 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, ML will complain about `exception Overflow`.

Built-in exceptions

ML 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's 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's 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 => ...

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

uncaught exception Match [nonexhaustive match failure]
```

A "Warning: match non-exhaustive" comment isn't fatal: ML will let you keep going. Indeed, in addition to the warning, ML says that `foo` is

well-typed, with the type `'a list -> int`. But a warning like this should prompt you to examine your code to see if you've missed out 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 a non-empty list causes a runtime error. And since ML can't tell just by looking at syntax whether or not a well-typed expression of type `t list` represents an empty list, there's no reasonable way to expect the ML type system to treat this kind of error as "compile time" rather than "run time".

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]

- 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]
```

So built-in exceptions indicate runtime errors that occur during expression evaluation.

User-defined exceptions

ML 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 index out of bounds). You can also "handle" an exception by providing some expression to be evaluated "instead" if a given exception gets raised. Of course, you can also handle built-in exceptions if you need to. 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 ML syntax for these constructs, by writing some simple ML functions for which it is natural to want to build in some error handling.

3 Evaluation and equality, revisited

Functional programs that may use exceptions have a potential for a more dangerous range of behaviors: a well-typed expression may, when evaluated, either terminate (and return a value), or fail to terminate (loop forever), as before; but it may instead raise an exception. Raising an exception is a kind of *effect*.

Accordingly, we need to modify the notion of equality for expressions that may involve exceptions. To give the general idea, here are the details for expressions of type `int` and expressions of type `int -> int`.

- Two expressions of type `int` are equal iff either both fail to terminate, or they both evaluate to the same integer value, or their evaluations both raise the same exception.
- Two expressions of type `int -> int` are equal iff either both fail to terminate, or they evaluate to function values that are extensionally equal, or their evaluations both raise the same exception.
- Two function values `f,g:int -> int` are extensionally equal iff, for all integer values `n`, `f(n)=g(n)`. This means that for all integer values `n`, either `f(n)` and `g(n)` both fail to terminate, or they both evaluate to the same integer, or they both raise the same exception.

For example:

```
1 div 0 = 2 div 0
(1 div 0) + 1 = (2 div 0) + 1
(fn x:int => x div 0) = (fn x:int => (2*x) div 0)
```

For other types we define equality as before, but taking into account the potential for exceptions. As before, for function values `f` and `g` of type `t -> t'`, `f = g` if and only if for all values `x,y` of type `t`, `x=y` implies `f(x) = g(y)`. But now saying that `f(x) = g(y)` means that both evaluate to equal values, or both fail to terminate, or both raise the same exception.

Importantly, we still have referential transparency!

> Replacing a sub-expression by an equal sub-expression produces an expression that is equal to the original one (using the revised notion of equality).

So we can still do "equational reasoning", provided we take account of the potential for raising exceptions.

4 Using exceptions

Declaring and raising

You can declare and name an exception, with an exception declaration like

```
exception Negative
```

which introduces an exception named `Negative`. Note that `exception` is a keyword in ML, so don't try to use the name `exception` as an exception name! In the scope of this declaration for `Negative` you can "raise" this exception by (evaluating) the expression

```
raise Negative
```

– evaluation of this expression causes an error stop and ML reports the name of the exception. By the way, exceptions are *not* like ordinary values. You can't do `case`-analysis based on the "value" of an exception; instead there is a special syntax for "handling" exceptions, which we will get to shortly.

We can use exception declarations to introduce locally available bindings, with limited scope, by using `let`-expressions and `local`-declarations. For example:

```
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
```

The type of `f` says nothing about the function's propensity for raising an exception! That's OK, because the type serves to tell us what happens when the function is applied to an argument for which the exception doesn't occur.

```
- f 2;

val it = 1 : int

- f (~2);

uncaught exception Neg

- f (1 div 0);

uncaught exception Div
```

Although the type of a function doesn't indicate whether or not it has any *effect* such as raising an exception, we should provide such a function with a *specification* that describes the circumstances in which an exception gets triggered. For example, consider the following code fragment:

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

fun inc (n:int) : (int -> int) =
      if n<0 then raise Negative else (fn x:int -> x+n)
```

Incidentally, the sub-expression `raise Negative` in the body of `inc` has type `int -> int`.

The following specification accurately describes the applicative behavior and effect of this function.

```
(* inc : int -> int -> int *)
(* REQUIRES true *)
(* ENSURES For all x:int, inc n x = x+n             if n >= 0 *)
(*                        inc n x = raise Negative  if n < 0  *)
```

Alternatively we could use a pair of specifications: one that describes what happens when the function behaves "normally", and one that describes what happens in error cases.

```
(* inc : int -> int -> int *)
(* REQUIRES n >= 0 *)
(* ENSURES For all x:int, inc n x = x+n *)
```

```
   (* REQUIRES n < 0 *)
   (* ENSURES For all x:int, inc n x = raise Negative *)
```

The first spec here is the "normal" part. The second spec tells us that if `n<0` then `inc n x = raise Negative`. So the spec says that applying `inc n` to any integer value `x` will raise the exception. In fact the exception gets raised even when we apply `inc` to an expression whose value is negative, not just when we apply it to a negative value. For example:

```
 inc (0 - 42)
   => inc (~42)
   => (fn n:int => if n<0 then raise Negative else (fn x:int -> x+n)) (~42)
   => if (~42) < 0 then raise Negative else (fn x:int => x+(~42))
   => if true then raise Negative else (fn x:int => x+(~42))
   => raise Negative
```

Here is a transcript of a session in the ML REPL, to show you what the ML system's responses are.

```
- exception Negative
exception Negative

- fun inc (n:int) : int -> int =
        if n<0 then raise Negative else (fn x:int => x+n);
val inc = fn : int -> int -> int

- inc 4;
val it = fn : int -> int

- inc (~4);

uncaught exception Negative

- inc (~4) 42;

uncaught exception Negative
```

As the example shows, `inc (~4)` is a well-typed expression (but not a *value*!) of type `int -> int` whose evaluation raises `Negative`.

To allow the user freedom to raise exceptions from anywhere inside a well-typed code fragment, the ML type system allows `raise Negative` to be used at any type.

```
- (raise Negative) : int;

uncaught exception Negative

- (raise Negative) : bool;

uncaught exception Negative
```

We needed parentheses in the lines above, because if we enter

```
- raise Negative : int;
```

the expression gets parsed as `raise(Negative:int)`,
which is not well-typed.

```
(* prod : int list -> int *)
   fun prod [ ] = 0
   |   prod (x::L) = if x<0 then raise Negative else x * prod L
(* Uses (raise Negative) at type int *)

(* prod : real list -> real *)
   fun prod [ ] = 0.0
   |   prod (x::L) = if x < 0.0 then raise Negative else x * prod L
(* Uses (raise Negative) at type real *)
```

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.

An example: gcd

To illustrate the design issues involved when we plan to write code that uses exceptions, let's discuss the gcd function. There is a simple way to compute the greatest common divisor (g.c.d.) of two positive integers, using *Euclid's algorithm*:

```
(* gcd : int * int -> int *)
fun gcd (x, y) =
      case Int.compare(x,y) of
          LESS    => gcd(x, y-x)
        | EQUAL   => x
        | GREATER => gcd(x-y, y);

(* REQUIRES x>0, y>0 *)
(* ENSURES gcd(x, y) returns the g.c.d of x and 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. And we proved 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 *)
fun GCD (x, y) =
      if x <=0 orelse y <= 0 then raise NotPositive else gcd(x,y)

(* 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. *)
```

It's easy to check, using evaluational reasoning, that when `x>0` and `y>0`,

```
GCD(x, y) =>* gcd(x, y)
```

and that if `x<=0` or `y<=0`,

```
GCD(x, y) =>* raise NotPositive
```

And according to our expanded notion of equality, we have

- If `x>0` and `y>0` then `GCD(x,y) = gcd(x,y)`.
- If `x<=0` or `y<=0` then `GCD(x,y) = raise NotPositive`.

What we've just done here 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 by raising a specific exception, which means that we can "handle" this situation in whatever manner we deem appropriate.

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, but it's easy to see that if the original `x` and `y` are positive the arguments in all recursive calls will also be positive.

Even better

Another way to define an exceptional version of gcd is given below.This version (which we name `GCD'` so that we can contrast it with `GCD` as defined above) keeps the dangerous `gcd` function local, so it isn't available for use elsewhere – only `GCD'` itself makes calls to this function and we have shown that it only ever makes "safe" calls to `gcd`:

```
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, using our expanded notion of equality, because for all integer values `x` and `y`,

```
EITHER x>0 and y>0, in which case
      GCD(x,y) and GCD'(x,y) both evaluate to the g.c.d of x and y,
OR not(x>0 and y>0), in which case
      GCD(x,y) and GCD'(x,y) both raise NotPositive *)
```

An advantage of `GCD'` is that here we don't make the original `gcd` function available for use outside, so users are prevented from applying `gcd` to bad arguments.

Handling exceptions

Now that we've shown you how to declare your own exceptions and raise ether 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's a runtime error we'd like to return a "default" value.

The ML 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` have type `t`.
Write down the obvious template describing the evaluational behavior of this expression, based on how `e` evaluates.

(And we can even define handlers with more than 2 alternatives, or use a wildcard pattern to match "all exceptions".)

A simple example, going back to 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;
```

5 Case study: making change

To put all these ingredients together, let's look for 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. And we also want to know whether or not it is possible: 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. (Unlike our previous discussion of the subset-sum problem, we allow repeated use of the same coin.)

It would be silly to write two separate functions, one which returns a truth value indicating whether or not it is possible, and another which, when applied to arguments for which it is possible will find a solution. This would be inefficient and clumsy, because it's pretty obvious that we can design an algorithm that tries to compute a list `L`, where `L` is a suitable way to make change for our amount, and raises an exception when it discovers that the task is (going to be) impossible.

Before starting, it's reasonable to assume that coins denominations are *positive integers*. A coin worth 0 dollars is (literally) useless, and it's very rare to find coins whose worth is negative. So we'll require that our function only every gets used on lists of positive coin sizes. Similarly, we can realistically expect to only ever use our function to make change for a non-negative amount.

It will be convenient to recall the function `sum : int list -> int` that adds up the integers in a list.

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

We'll use this function *only* in specifications!

Let's start by defining

```
exception Change
```

and deciding to use this exception to signal an inability to make change.

So we'll give the specification

```
(*   change : int list * int -> int list *)
(* REQUIRES L is a list of positive integers & a >= 0 *)
(* ENSURES change(L, a) = C,
                    where C is a list of items from L with sum C = a,
                    if there is one;
           change(L, a) = raise Change
                    if there is no such list C       *)
```

First attempt

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

- If `a` is zero, we can make change for `a` with the empty list.
  In fact, that's the only way!
- If `a>0` and the coins list is `c::R`,
  - if `c` is bigger than `a` you can't use it;
  - otherwise use `c` and a coin list making change for `a-c`.

The following function definition is based very closely on the above sketch of an algorithm. It's called a greedy algorithm because in the interesting case it assumes that you 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, ML 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 ML 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 it should be easy to spot that there's no clause for `change ([ ], n)` except for `n=0`. But we'd expect the function to raise the exception when asked to give change for a non-zero amount using no coins!

(And actually, we should have realized this earlier – what was the point of introducing the exception `Change` if we weren't planning to raise it?)

Just to confirm that this function is bad, note that it seems to "work" on some arguments, such as

```
- change ([2,3], 8);
val it = [2,2,2,2] : int list

- change ([3,2], 8);
val it = [3,3,2] : int list
```

and it does *raise an exception* in some cases where it's impossible to make change, e.g. but it goes wrong on others, e.g.

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

uncaught exception Match [nonexhaustive match failure]
```

the exception that gets raised is *not* the one mentioned in the spec! Moreover, in other cases where there *is* a way to make change we may get an exception instead, e.g.

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

uncaught exception Match [nonexhaustive match failure]
```

There's definitely a defect in the function above. And it's very bad style to try to rely on `Match` exceptions to signal a user-specific runtime issue (here, a discovery that it's impossible to make change).

Second attempt

So 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` with the empty list.
  In fact, that's the only way!
- If `a>0` and the coins list is `c::R`,
  - if `c` is bigger than `a` you can't use it;
  - otherwise use `c` and a coin list making change for `a-c`.
- If `a>0` and the coins list is empty, there's no way to make change for `a`, so raise `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 we get

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

uncaught exception Change
```

so the intended exception is getting raised.
However, we *still* get wrong results, e.g.

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

uncaught exception Change
```

when we wanted to get a list such as `[3,3,3]` or `[2,2,2,3]` instead (both of these are sublists of `[2,3]` that add up to 9). The problem is that the algorithm is too greedy. If it turns out that there *is* a way too make change for `a` but only *without* using the first coin, this code won't discover it. We need to use a *handler* to make this recovery happen.

Third and final attempt

Here is a third amended outline of the algorithm:

- If `a` is zero, we can make change for `a` with the empty list.
  In fact, that's the only way!
- If `a>0` and the coins list is `c::R`,
  - if `c` is bigger than `a` you can't use it;
  - otherwise *try* using `c` and a coin list making change for `a-c`; if this works, fine; if not, handle the exception raised from this attempt by trying to make change without using `c`.
- If `a>0` and the coins list is empty, there's no way to make change for `a`, so raise `Change`.

Here is the 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. Actually the `else` branch expression is

```
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'll 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 (changer (coins, a)) handle Change => NONE;
```

Exercise: give a suitable spec for `mkchange`.

Question

What is wrong with the following version of the `change` function, in which the handler for `Change` has a different scope (the entire `if-then-else` expression, not just the `else` branch)?

```
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);
```

Answer: double dipping. It computes same results but less efficiently! Can show that for all lists L of positive integers, and all non-negative integers `a`,

```
change' (L, a) = change (L, a)
```

As usual the proof uses induction, and again it is on `length L + a`. The only interesting case (!) is when `L` is `c::R`, `a<c`, and `change'(R, a)` raises the exception `Change`. [This case is not trivial, because the then-branch of the third clause is NOT inside the handler's scope for `change` but it IS inside the handler's scope for `change'`.] If we assume as induction hypothesis that the recursive call on `(R, a)` is correctly dealt with by `change'`, i.e. there really is no way to make change for `a` using `R`, then there won't be any way to make change for `a` using `c::R`. And we get

```
change' (c::R, a)
= (if a<c then change'(R, a) else . . .) handle Change => change'(R, a)
= change'(R, a) handle Change => change'(R, a)
= raise Change handle Change => change' (R, a)
= change' (R, a)
= raise Change
= change (c::R, a)
```

as required.

Question

Write a version of `change` that doesn't allow repeated use of coin sizes. What is the connection with the subset-sum problem?

Question

Write an ML function

```
bestchange : int list * int -> int list
```

that computes "optimal" ways to make change; it should find a shortest way to make change, whenever it is possible to make change for the given amount. More formally: for all lists `L` of positive integers, and all non-negative integers `a`,

(a) `bestchange(L, a) = C`, where `C` is a list of items from `L` with shortest length such that `sum C = a`, if such a list exists.

(b) `bestchange(L, a) = raise Change` if there is no list of items from `L` with sum equal to `a`.

6 Correctness proof

Since the `change` function uses exceptions, it's worth writing out the proof that it meets the specification. In this proof we will see the expanded notions of equality and referential transparency in action.

```
(* 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)

(* REQUIRES L is a list of positive integers & a >= 0 *)

(* ENSURES change(L, a) = C,
              where C is a list of items from L with sum C = a
              if there is one;
           change(L, a) = raise Change
             otherwise        *)
```

Looking at the function definition, we can see that when applied to arguments that satisfy the REQUIRES assumptions, `change(L, a)` makes recursive calls to `change(L', a')` in which either `a'=a` and `L'` is a proper suffix of `L` (so a shorter list, and still a list of positive integers), or `L'` is the same list as `L` but `a'` is smaller than `a` (and still non-negative). This means that we can use *induction* on `length L + a` to prove properties of `change` on arguments that obey the requirements. In every recursive call `change(L' a')` we will have

```
length L + a > length L' + a'
```

and `length L' + a' >= 0`.

Theorem
For all lists `L` of positive integers, and all non-negative integers `a`,

EITHER there is a list of items from `L` with sum equal to `a`,
and `change(L, a)` returns such a list;

OR there is no list of items from `L` with sum equal to `a`,
and `change(L, a) = raise Change`.

Proof: by (strong) induction on `length L + a`.
Our case analysis follows the structure of the function definition.

- Case for `a = 0`.
  For any list `L`, `[ ]` is a sublist with sum equal to 0.
  By definition of `change`, `change(L, 0)` returns `[ ]`.
  So the required property holds, for `a=0` and all `L`.

- Case for `a>0`, `L = [ ]`.
  There is no way to make change for a non-zero amount using no coins.
  By definition, `change(a, [ ]) = raise Change` if `a>0`.
  This is the required result for this case.

- Case for `a>0`, `L` of form `c::R`.
  By function definition (and since `c, R, a` are values),

```
change (c::R, a) =
if a<c then change (R, a)
       else c :: (change (c::R, a-c))
             handle Change => change(R, a)
```

  If `a<c`, then there is a way to use `c::R` to make change for `a` if and only if there is a way to use `R` to make change for `a`. (Remember: all items in `L` are positive integers.) By induction hypothesis for `(R, a)`, `change(R, a)` returns a way to use `R` to give change for `a`, if there is one, and raises `Change` otherwise. So `change(R, a)` returns a way to use `L` to make change for `a`, if there is one, and raises `Change` otherwise. So `change(c::R, a)` behaves as intended if `a<c`.

  Otherwise, `a` is not less than `c`. Then there is a way to make change for `a` using `c::R` if and only if *either* there is a way to make change for `a-c` using `c::R`, *or* there is a way to make change for `a` using `R`.
  By the induction hypothesis for `(L, a-c)`, if there is a way to make change for `a-c` using `c::R` then `change(c::R, a-c)` returns a sublist `C` of `c::R` such that `sum C = a-c`; otherwise `change(c::R, a-c) = raise Change`.
  If `sum C = a-c`, then `sum(c::C) = a`. So if there is a way to make change for `a` using `c`, `c::(change(c::R, a-c))` returns a sublist of `c::R` with sum equal to `a`. That is the required result for this subcase.

  Finally, if there's no way to make change for `a` using `c`, we get

```
c :: (change(c::R, a-c)) = raise Change
```

so

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

By induction hypothesis for `(R,a)`, this call returns a sublist of `R` with sum equal to `a` if there is one, and raises `Change` otherwise. By the previous analysis, this outcome is the required result for making change for `a` using `c::R`.

7 Exception laws

Assuming that `e` and `e1, ..., ek` are well-typed expressions of type `t`, and `v` is a value of type `t`, and `E1,..., Ek` are (distinct) exception names currently in scope, the following equational laws hold:

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

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

(3) If E is an exception distinct from E1, . . ., Ek then
         (raise E) handle E1 => e1 | . . . | Ek => ek   = raise E
```

Also, just to remind you that we still have referential transparency, note the following laws:
If `e`, `e'`, `e1,...,ek,e1',...,ek'` are all well-typed expressions of type `t`, then

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

(5) If e1 = e1' and . . . and ek = ek'  then
      (e handle E1 => e1 | . . . | Ek => ek)
    = (e handle E1 => e1' | . . . | Ek => ek')
```

8 n-queens, revisited

Here is an exception-based version of the backtracking n-queens algorithm. We leave it up to you to try this code out and verify that it works as desired.

```
type row = int; type col = int;
type pos = row * col;
type sol = pos list;
fun threat((x,y), (i,j)) =
   (x+y = i+j) orelse (x-y = i-j) orelse (x=i) orelse (y=j);
fun conflict(p, nil)=false
 |  conflict(p, q::qs) = threat(p, q) orelse conflict(p, qs);

exception Impossible;

(* try : int * row * col list * sol -> sol *)
fun try(n, i, A, qs) =
   case A of
        [ ]    =>  raise Impossible
     |  (j::B) =>  if conflict((i, j), qs)
                   then try(n, i, B, qs)
                   else if i = n
                        then (i, j)::qs
                        else try(n, i+1,  upto 1 n, (i, j)::qs)
                               handle Impossible => try(n, i, B, qs);

(* REQUIRES 1<=i<=n and qs is a partial solution on rows 1 through i-1   *)
(* ENSURES  try(n,i,qs,A) = qs', where qs' is a solution that extends qs
                                  with a queen in row i at column in A,
                                  if there is one;
            try(n,i,A,qs) = raise Impossible
                                  otherwise.     *)

fun queens n = SOME(try (n, 1, upto 1 n, nil)) handle Impossible => NONE;
(* REQUIRES n>0 *)
(* ENSURES  queens n = SOME(qs),  where qs is a solution to n-queens,
                      if there is one;
                   queens n = NONE,  otherwise   *)
```

9 Exercises

1. The built-in ML functions

```
hd : 'a list -> 'a
tl : 'a list -> 'a list
```

raise the built-in exception `Empty` when applied to the empty list:

```
hd [ ] = raise Empty
tl [ ] = raise Empty
hd [1,2,3] = 1
tl [1,2,3] = [2,3]
```

The function `null : 'a list -> bool` returns `true` when applied to the empty list, `false` when applied to a non-empty list.

Consider the following function definitions:

```
fun length L = if null L then 0 else 1 + length (tl L)
fun len L = (1 + len (tl L)) handle Empty => 0
fun len' L = 1 + ((len' (tl L)) handle Empty => 0)
```

(a) What happens when ML evaluates `len [ ]`?

(b) What happens when ML evaluates `len' [ ]`?

(c) Prove by induction on the structure of `L`, that for all list values `L`:

```
len L  = length L
len' L = 1 + length L.
```

(You can do separate proofs for the two properties here.)

2. Consider the following ML definitions:

```
exception TooSmall
exception TooBig

fun nth 1 (x::_) = x
|   nth n (_::xs) = if n>1 then nth (n-1) xs else raise TooSmall
|   nth _ [ ] = raise TooBig

fun entry n L = if n>0 then nth n L else raise TooSmall
```

(a) What happens when ML evaluates the following expressions?

```
nth 4 [1,2,3]
nth (~42) [1,2,3]
nth 2 [1,2,3]
entry 4 [1,2,3]
entry (~42) [1,2,3]
```

(b) Give specifications describing the applicative behavior of `nth` and `entry`, and prove that the functions satisfy these specifications.

3. Using `entry` from above, write an ML function

```
sumchain : int list -> (int -> int) -> int -> int
```

such that for all suitably typed `L, f, i`, `sumchain L f i` evaluates to the sum of the entries of `L` whose indices are reachable from `i` by iterating `f`. You can assume that `f` is a strictly increasing function. For example,

```
sumchain [1,2,3,4,5] (fn x => x+2) 1 = 1+3+5 = 9
sumchain [1,2,3,4,5] (fn x => x+2) 2 = 2 + 4 = 6
sumchain [1,2,3,4,5] (fn x => x+42) 2 = 2
```

4. Consider the following ML definitions:

```
exception Foo
exception Bar
fun silly 1 = raise Foo
|   silly 2 = raise Bar
|   silly n = (let
                 val x = (silly(n-2) handle Bar => 2)
               in
                 silly(n-1) + x
               end) handle Foo => 1
```

What is the value (if any) of `silly 5`?