Programming Languages and Paradigms

Brigitte Pientka

School of Computer Science
McGill University
Montreal, Canada

Contents

Chapter 1

Basic Concepts in Functional Programming

OCaml is a *statically typed functional* programming language. What does this mean? - In functional programming languages we write programs using *expressions*, in particular functions. These *functions compute a result* by manipulating and analyzing values recursively. OCaml as many ML-like languages (such as SML, Haskell, F#, etc.) facilitates defining recursive data structures and programming functions via *pattern matching* to analyze them. This often leads to elegant, compact programs. Functional programming is often associated with effect-free programming, i.e. there is no explicit memory allocated or stored, there are no exceptions that would divert the control flow. Functional languages that do not support effects are often called pure. OCaml supports both exception handling as well as state-full computation and hence is an impure functional language.

Functional languages compute a result (i.e. a boolean, an integer, a list, etc.). This is in contrast to procedural or imperative languages where we write methods or procedure; we typically update some state by assigning values to variables or fields, and we observe the result as an effect, i.e. we print the result on standard output or we read and lookup the state and value of a given variable. We will contrast these two approaches more later in the course.

What does statically typed mean? - Types approximate the runtime behaviour of an expression statically, i.e. before running and executing the program. We can also say "Types classify expressions by their values". Basic types are integers, booleans, floating point numbers, strings or characters. But we can in fact also reason about functions by relating the input and output of a function. And we can then check whether functions are used with the correct types of input and reason about the correct use of the results they compute. Static type checking is a surprisingly effective technique. It is quick (i.e. for most programs it is polynomial). It can be re-run every time the program changes. It gives precise error messages where the problem might lie and how it might be fixed.

Type checking is conservative. It examines the code statically purely based on its syntactic structure checking whether the given program is meaningful, i.e. its evaluation does something sensible. There are programs which the type checker rejects, although nothing would go wrong during evaluation. But more importantly, if the type checker succeeds, the we are guaranteed that the execution of the program does not lead to a core dump or crash.

Statically typed functional languages like OCaml (but also Java, ML, Haskell, Scala for example) are often called type-safe, i.e. they guarantee that if a program is well-typed, then its execution does not go wrong, i.e. either it produces a value, or it aborts gracefully by raising a runtime exception, or it continues to always steps to a well-defined state. This is a very strong guarantee. As a consequence, typed functional programs do not simply crash because of trying to access a null pointer, trying to access a field in an array that is out of bound or trying to write over and beyond a given buffer (buffer overrun). In fact, the heartbleed bug in 2014 is a classic example of a type error and could have been easily avoided.

Let's begin slowly. We begin using OCaml's interactive toplevel, aka a read-eval-print loop, that we can use to play around. To start simply type `ocaml` in a terminal:

```
[bpientka][taiko][~]$ ocaml
        OCaml version 4.02.1
#
```

We can now type expressions followed by `;;` to evaluate them.

1.1 Expressions, Names, Values, and Types

The most basic expressions are numbers, strings, and booleans.

```
# 3;;
- : int = 3
# 2;;
- : int = 2
# 3 + 2 ;;
- : int = 5
#
```

OCaml says that `3` is an integer and it evaluates to `3`. More interestingly, we can ask what the value of the expression `3 + 2` is and OCaml will return `5` and tell us that its type is `int`.

As you can see, the format of the toplevel output is

```
<name> : <type> = <value>
```

If we do not bind the value resulting from evaluating an expression to a name, OCaml will simply write _ instead. We often want to introduce a name to be able to subsequently refer to it. We come back to the issue of binding a little later.

We can not only compute with integers, but also for example with floating point numbers.

```
# 3.14;;
- : float = 3.14
# 5.0 /. 2.0;;
- : float = 2.5
#  3.2 +. 4.5;;
- : float = 7.7
#
```

Note that we have a different set of operators to compute with floating point numbers. All arithmetic operators have as a postfix a dot. You might wonder whether it is possible to simply say 5   3.2+. The answer is no as you can see below.

```
# 5 + 3.2;;
Characters 4-7:
  5 + 3.2;;
      ^^^
Error: This expression has type float but an expression was expected of type
         int
#
```

Arithmetic operators are not *overloaded* in OCaml. Overloading operators requires that during runtime we must decide what function (or method) to choose base on the type of the arguments. This requires that we keep types around during runtime and it can be expensive. OCaml does not keep any types around during runtime - this is unlike languages such as Java. In OCaml (as in many functional languages) types are purely used for statically reasoning about the code and optimizing the compilation of the code.

The example illustrate another benefit of types: precise error messages. The type checker tells us that the problem seems to be in the right argument which is passed to the operator for addition.

OCaml has other standard base types such as strings, characters, or booleans.

```
# "comp302";;
- : string = "comp302"
# 'a';;
- : char = 'a'
# true;;
- : bool = true
# false;;
- : bool = false
# true || false ;;
- : bool = true
# true && false;;
- : bool = false
#
```

Here we have used boolean operators `||` (for disjunction) and `&&` (for conjunction). We can also use if-expressions.

```
# if 0 = 0 then 1.0 else 2.2;;
- : float = 1.
#
```

As we mentioned type checking is conservative. Hence there are programs that would produce a value, but the type checker rejects them. This happens for example in the program below:

```
Characters 23-28:
  if 0 = 0 then 1.0 else "AAA";;
                         ^^^^^
Error: This expression has type string but an expression was expected of type
         float
#
```

Clearly the guard `0 = 0` is always true and we would never reach the string `''AAA''`. We can also say the second branch is dead code. While this is obvious when our guard is of the form `0 = 0`, in general this is hard to detect statically. In principle, the guard could be a very complicated call to a function which always happens to produce true for a given input. The type checker makes no assumption about the actual value a guard has, but only verifies that the guard would evaluate to a boolean. Hence the type checker does not know what branch will be taken during runtime and it must check that both branches produce the same kind of value. Why? - Because the if-expression may be part of a larger expression. Reasoning about the type of expressions is compositional.

```
# (if 0 = 0 then 1.0 else 2.2) +. 3.3;;
- : float = 4.3
#
```

As mentioned earlier, if the type checker accepts the program, executing it either yields a value or the program is aborted gracefully, if it reaches a state that is identified as a run-time error. An example of such a run-time error is division by zero.

```
# 3 / 0;;
Exception: Division_by_zero.
```

Attempting to divide `3` by `0` will pass the type checker, because the type checker simply checks whether the two arguments given to the division operator are integers. This is the case. During runtime we note that we are dividing by zero, and the evaluator raises a built-in exception `Division-by-zero`.

1.2 Variables, Bindings, and Functions

A central concept in a programming language are variables and the scope of variables. In OCaml we can declare a variable at the top-level (i.e. a global declaration) using a let-expression of the form `let <name> = <expression>` as follows:

```
# let pi = 3.14;;
val pi : float = 3.14
#
```

Here `pi` is the name of the variable and we bind to the name the value `3.14`. Note, because OCaml is a call-by-value language, we bind *values* to variable names - not expressions. For example as a result of evaluating the following expression, we bind the variable name `x` to the value `9`.

```
# let x = 3 * 3;;
val x : int = 9
#
```

A variable binding is not an assignment. We simply establish a relation between a name and a value. There is no state being allocated and this association or relation cannot be updated or changed. Once the relationship is done it is fixed. We can only *overshadow* a given binding, not update it.

```
# let x = 42;;
val x : int = 42
#
```

For example by establishing again a binding between the variable name x and the value 42, we now have two bindings for x on the binding stack. The last binding we pushed onto that stack bind x to 42. When we look up the value of a variable, we look it up in the stack and we pick the one that was establish most recently, i.e. the binding that is at the top. The earlier binding is *overshadowed*.

A garbage collector might decide to remove the earlier binding for efficiency reasons, it determines there is no other code that still uses it.

We can also introduce scoped variable bindings or local bindings as illustrated in the following example.

```
# let m = 3 in
  let n = m * m in
  let k = m * m in
    k*n
;;

        - : int = 81
#
```

We use a let-expression that has the following structure:

```
let <name> = <expression 1> in <expression 2>.
```

The `<name>` can be used in the body, i.e. `<expression 2>`, to refer to the value of `<expression 1>`, i.e. we bind the value of `<expression 1>` to the variable `<name>` and continue evaluating `<expression 2>` using this new binding. The binding of variable `<name>` to the value of `<expression 1>` ends after the let-expression has finished evaluating and the binding is removed from the binding stack.

We also say the scope of the variable `<name>` ends after `<expression 2>`.

How do global and local bindings interact? - Local bindings are only temporary. Hence, they may overshadow temporarily a given global binding. Here is an example:

```
# let k = 4;;
val k : int = 4
# let k = 3 in k * k ;;
- : int = 9
# k;;
- : int = 4
#
```

When we evaluate the body `k * k` in the second let-expression, we have two bindings for the name `k`: the first one bound `k` to 4; the second one and the most recent one that is on top of the binding stack says `x` is bound to 3. When we evaluate `k * k`, we look up the most recent binding for `k` and obtain 3. Hence we return the value 9. When we then ask what is the value of `k` we obtain 4, as the local binding between `k` and 3 was removed from the binding stack. This clearly illustrates that variables bindings are not updated - once they are made they persist.

[Explanation about functions and tail recursion to be added]

1.3 Datatypes and Pattern Matching

Often it is useful to define a collection of elements or objects and not encode all data we are working with using our base types of integers, floats or strings. For example, we might want to model a simple card game. As a first step, we want to define the suit and rank present in the game. In OCaml, we define a finite, unordered collection of elements using a (non-recursive) data type definition. Here we define a new type `suit` that contains the elements `Clubs`, `Spades`, `Hearts`, and `Diamonds`.

```
type suit = Clubs | Spades | Hearts | Diamonds
```

We also say `Clubs`, `Spades`, `Hearts`, and `Diamonds` *inhabit* the type `suit`. Often, we simply say `Clubs`, `Spades`, `Hearts`, and `Diamonds` *is* of type `suit`.

When we define a collection of elements the order does not matter. Further, OCaml requires that each element begins with a capital letter.

Elements of a given type are defined by *constructors*, i.e. constants that allow us to construct elements of a given type. In our previous definition of our type `suit` we have four constructors, namely `Clubs`, `Spades`, `Hearts`, and `Diamonds`.

How do we write programs about elements of type `suit`? - The answer is *pattern matching* using a match-expression.

```
match <expression> with
| <pattern> -> <expression>
| <pattern> -> <expression>
 ...
| <pattern> -> <expression>
```

We call the expression that we analyze the *scrutinee*. Patterns characterize the shape of values the scrutinee might have by considering the type of the scrutinee. Let's make this more precise by looking at an example. We want to write a function `dom` that takes in a pair `s1` and `s2` of `suit`. If `s` beats of is equal to suit `s2` we return true – otherwise we return false. We will use the following ordering on suits:

`Spades ¿ Hearts ¿ Diamonds ¿ Clubs`

The type of the function `dom` is `suit * suit -> bool`.

```
let rec dom (s1, s2) = match (s1, s2) with
  | (Spades, _)        -> true
  | (Hearts, Diamonds) -> true
  | (Hearts, Clubs)    -> true
  | (Diamonds, Clubs)  -> true
  | (s1, s2)           -> s1 = s2
```

Please read the `datatypes.ml` and `trees.ml` for a longer and more detail ed description of the examples discussed in class.

Chapter 2

Induction

"On theories such as these we cannot rely. Proof we need. Proof!"
Yoda, Jedi Master

In this chapter we will briefly discuss how to prove properties about ML programs using induction. Proofs by induction are ubiquitous in the theory of programming languages, as in most of computer science. Many of these proofs are based on one of the following principles: mathematical induction and structural induction. In this chapter we will discuss these most common induction principles.

2.1 Mathematical induction

Mathematical induction is the simplest form of induction. When we try to prove a property for every natural number n, we first show the property holds for 0 (induction basis). Then we assume the property holds for n and establish it for $n+1$ (induction step). Basis and step together ensure that the property holds for all natural numbers.

There are small variations of this scheme which can be easily justified. For example, we may start by proving the property holds for 1, if we want to prove a property for all positive integers. There may also be two base cases, one for 0 and one for 1.

As an example, let us consider the following program `power`.

```
(* Invariant: power:int >=0 *)

let rec power(n, k)=
  if k=0 then 1  else n * power(n, k-1)
```

How can we prove that this program indeed computes n^k? – To clearly distinguish between the natural number n in our on-paper formulation and its representation and use in a program, we will write $\overline{n}$ to denote the latter. Moreover, we will use the following notation

$e \Downarrow v$	expression e evaluates in multiple steps to the value v.
$e \Rightarrow e'$	expression e evaluates in one steps to expression e'.
$e \Rightarrow^* e'$	expression e evaluates in multiple steps to expression e'.

In this example, we want to prove that $\texttt{power}(\overline{n}, \overline{k})$ evaluates in multiple steps to the value $\overline{n^k}$.

Theorem 1. $\texttt{power}(\overline{n}, \overline{k}) \Downarrow \overline{n^k}$ *for all* $k \geq 0$

Proof. By induction on k.

Base Case $k = 0$

	$\texttt{power}(\overline{n}, \overline{0})$	
$\Rightarrow$	$\texttt{if 0 = 0 then 1 else } \overline{n} \texttt{ *power(}\overline{n}\texttt{, 0-1)}$	by program
$\Rightarrow$	$\texttt{if true then 1 else } \overline{n} \texttt{ *power(}\overline{n}\texttt{,0-1)}$	by evaluation rules for equality
$\Rightarrow$	$\texttt{1} = \overline{1} = \overline{n^0}$	by evaluation rules for if-expressions

Step Case Assume that $\texttt{power}(\overline{n},\overline{k}) \Downarrow \overline{n^k}$. We have to show that $\texttt{power}(\overline{n}, \overline{k+1}) \Downarrow \overline{n^{k+1}}$.

	$\texttt{power}(\overline{n}, \overline{k+1})$	
$\Rightarrow$	$\texttt{if } \overline{k+1} \texttt{ = 0 then 1 else } \overline{n} \texttt{ * power(} \overline{n}, (\overline{k+1}) \texttt{ - 1)}$	by program
$\Rightarrow$	$\texttt{if false then 1 else } \overline{n} \texttt{ * power(}\overline{n}, (\overline{k+1}) \texttt{ - 1)}$	by evaluation rules for equality
$\Rightarrow$	$\overline{n} \texttt{ * power(}\overline{n}, (\overline{k+1}) \texttt{ - 1)}$	by evaluation rules for if-expressions
$\Rightarrow$	$\overline{n} \texttt{ * power(}\overline{n}, \overline{k}\texttt{)}$	by evaluation rules for $-$
$\Rightarrow^*$	$\overline{n} * \overline{n^k}$	by induction hypothesis
$\Rightarrow$	$\overline{n * n^k} = \overline{n^{k+1}}$	by evaluation rules for $*$

$\square$

This proof emphasizes each step in the evaluation of the program[1]. Often, we may not want to go through each single step in that much detail. However, it illustrates that when reasoning about programs, we must know about the underlying operational semantics of the programming language we are using, i.e. how will a given program be executed.

2.2 Complete induction

The principle of complete induction formalizes a frequent pattern of reasoning. To prove a property by complete induction we first need to establish the induction basis for $n = 0$. Then we prove the induction step for $n \geq 0$ by assuming the property for all $n' < n$ and establishing it for n. One can think of it like mathematical induction, except that we are allowed to appeal to the induction hypothesis for any $n' < n$ and not just the immediate predecessor.

These two principles should be familiar to you and are the basis for proving some fundamental properties about programs. As an example, we define a program for `power` which is more efficient and defined via pattern matching.

```
let rec power (n,k) = match k with
| 0 -> 1
| _ -> if even(k) then
        let r = power(n, k / 2) in r * r
       else n * power (n, k-1)
```

To prove that this program works correctly, we rely on the following properties which we state as lemmas without proofs.

Lemma 1. *For all* n, $\overline{n^k} * \overline{n^k} \Downarrow \overline{n^{2k}}$.

Theorem 2. `power`$(\overline{n}, \overline{k}) \Downarrow \overline{n^k}$ *for* $k \geq 0$.

Proof. By complete induction on k.

Base Case $k = 0$

`power`$(\overline{n}, \overline{0})$
$\Rightarrow 1$ by program

[1]Note, in class we assumed the property holds for $\overline{k-1}$ and showed the property for $\overline{k}$. Both are fine

Step Case $k > 0$

Assume that $\texttt{power}(\overline{n}, \overline{k'}) \Downarrow \overline{n^{k'}}$ for any $k' < k$. We have to show that $\texttt{power}(\overline{n}, \overline{k}) \Downarrow \overline{n^{k}}$.

$$\begin{array}{lll}
 & \texttt{power}(\overline{n}, \overline{k}) & \\
\Rightarrow & \texttt{if even } (\overline{k}) \texttt{ then (let r = power}(\overline{n}, \overline{k} \texttt{ / 2) in r * r)} & \\
 & \quad \texttt{else } \overline{n} \texttt{ * power } (\overline{n}, \overline{k}\texttt{-1)} & \text{by program}
\end{array}$$

Now we will distinguish subcase, whether k is even or odd.

Sub-Case 1 $k = 2k'$ for some $k' < k$.

$$\begin{array}{lll}
\Rightarrow & \texttt{let r = power}(\overline{n}, \overline{2k'}\texttt{/ 2) in r * r} & \text{by evaluation rules for } \texttt{if} \\
\Rightarrow & \texttt{let r = power}(\overline{n}, \overline{k'}\texttt{) in r * r} & \text{by evaluation rules for } \texttt{/} \\
\Rightarrow^* & \texttt{let r = } \overline{n^{k'}} \texttt{ in r * r} & \text{by i.h. on } k' \\
\Rightarrow & (\overline{n^{k'}})*(\overline{n^{k'}}) & \text{by Lemma 1} \\
= & \overline{n^{2k'}} = \overline{n^{k}} &
\end{array}$$

Sub-Case 2 $k = 2k' + 1$ for some $k' < k$.

$$\begin{array}{lll}
\Rightarrow & \overline{n} \texttt{ * power } (\overline{n}, \overline{k}\texttt{-1)} & \text{by evaluation rules for } \texttt{if} \\
\Rightarrow & \overline{n} \texttt{ * power } (\overline{n}, \overline{k-1}\texttt{)} & \text{by evaluation rules for } \texttt{-} \\
\Rightarrow & \overline{n} * \overline{n^{k-1}} & \text{by i.h. } k-1 \\
\Rightarrow & \overline{n * n^{k-1}} = \overline{n^{k}} & \text{by evaluation rules for } *
\end{array}$$

$\square$

2.3 Structural induction

When proving properties about ML programs, we typically need to reason not only about numbers but about defined inductively data-structures, such as lists, trees etc. Structural induction allows us to reason about the structure of the objects we are considering. This is best illustrated by considering an example of an inductive data-type such as lists.

```
type 'a list = nil | :: of 'a * 'a list
```

To inductively prove a property about lists, we first prove it for the empty list `nil`. Then we assume the property holds for lists `t` and establish it for lists `h::t`.

Similarly, for trees:

```
type 'a tree = Empty | Node of 'a * 'a tree * 'a tree
```

To inductively prove a property about trees, we first prove it for the empty tree `Empty`. Then we assume the property holds for trees `l` and `r`, and establish it for the tree `Node(a, l, r)`.

Inductive data-structures make it easy to reason about them inductively, since they directly give rise to induction principles. Typically, we reason directly about their structure. For example, we consider `l` and `r` to be sub-trees of the tree `Node(a, l, r)`. Let us consider the following two programs. The first one allows us to insert an element, which consists of a key `x` and the data `d`, into a binary search tree. The second one allows us to lookup the data `d` associated with some key `x` in a binary search tree `t`.

```
let rec insert ((x,d) as e) t = match t with
  | Empty                 -> Node(e, Empty, Empty)
  | Node ((y,d'), l, r) ->
      if x = y then Node(e, l, r)
      else
  (if x < y then Node((y,d'), insert e l, r)
   else
    Node((y,d'), l, insert e r))
```

```

let rec lookup x t = match t with
  | Empty                -> None
  | Node ((y,d), l, r) ->
  if x = y then Some(d)
    else
      (if x < y then lookup x l
   else lookup x r)
```

Let's try to prove that when we have inserted an element `(x,d)` into a binary search tree `t`, and then look up the data corresponding to the key `x`, we will get back the date `d`. In the proof below we will write $\Rightarrow *$ when we skip over some intermediate steps.

Theorem 3. *If* `t` *is a binary search tree, then* `lookup x (insert (x,d) t)` $\Downarrow$ `Some(d)`

Proof. By structural induction on `t`.

Base Case `t = Empty`

	`lookup x (insert (x,d) Empty)`	
$\Rightarrow$	`lookup x (Node((x,d), Empty , Empty))`	by program insert
$\Rightarrow^*$	`Some(d)`	by program lookup

Step Case `t = Node((y,d'), l, r)`

We can assume the property holds for the sub-trees `l` and `r`.

1. `lookup x (insert (x,d) l)` $\Downarrow$ `Some(d)`
2. `lookup x (insert (x,d) r)` $\Downarrow$ `Some(d)`

Sub-case: `x = y`

	`lookup x (insert (x,d) (Node((y,d'), l, r)))`	
$\Rightarrow^*$	`lookup x (Node((x,d), l, r))`	by program insert
$\Rightarrow^*$	`Some(d)`	by program lookup

Sub-case: `x < y`

	`lookup x (insert (x,d) (Node((y,d'), l, r)))`	
$\Rightarrow^*$	`lookup x (Node((y,d'), insert (x,d) l, r))`	by program insert
$\Rightarrow^*$	`lookup x (insert (x,d) l)`	by program lookup
$\Rightarrow$	`Some(d)`	by i.h.

Sub-case: `y < x`

	`lookup x (insert (x,d) (Node((y,d'), l, r)))`	
$\Rightarrow^*$	`lookup x (Node((y,d'), l, insert (x,d) r))`	by program insert
$\Rightarrow^*$	`lookup x (insert (x,d) r)`	by program lookup
$\Rightarrow$	`Some(d)`	by i.h.

□

2.4 Generalizing the statement

From the examples, it may seem that induction is always straightforward. Often this is indeed the case. Sometimes however we will encounter functions whose correctness property is more difficult to prove. This is often because we need to prove something more general than the final result we are aiming for. This is also referred to as *generalizing the induction hypothesis*. There is no general recipe for generalizing the induction hypothesis, but one common case is the following.

Consider the following two programs for reversing a list. The first one is the naive version, while the second one is the tail-recursive version.

```
let rec rev l = match l with
  | []    -> []
  | x::l -> (rev l) @ [x]
```

```
let rec rev' l acc = match l with
| []    -> acc
| h::t  -> rev' t (h::acc)
```

We would like to prove that both programs yield the same result. Essentially we would like to say `rev l` returns the same result as calling `rev' l []`.

$$\texttt{rev l} \Downarrow \texttt{l'} \text{ iff } \texttt{rev' l []} \Downarrow \texttt{l'}$$

We will simplify this statement a little bit, and try to prove

$$\texttt{rev l} = \texttt{rev' l []}$$

The problem arises in the step-case, when we attempt to prove

$$\texttt{rev (x::t)} = \texttt{rev' (x::t) []}$$

On the left, the program `rev'` evaluates as follows:

```
   rev' (x::t) []
⇒  rev' t  (x::[])
⇒  rev' t   [x]
```

But now we are stuck. We cannot apply the induction hypothesis, because the statement we attempt to prove requires that the second argument to `rev'` is the empty list! The solution is to generalize the statement in such a way that the desired result follows easily.

The following theorem generalizes the problem appropriately.

Theorem 4. *For any list* `l`, `rev(l)@acc` $=$ `rev' l acc`

Proof. Induction on `l`.

Base Case `l = []`

```
   rev([])@acc
⇒  []@acc                          by program rev
⇒  acc                             by program @
⇐  rev' [] acc                     by program rev'
```

Step Case `l = x::t`
Assuming, for any `acc'`, `rev(t)@acc'` = `rev' t acc'`,
we must prove `rev(x::t)@acc'` = `rev' (x::t) acc'` .

	`rev(x::t)@acc`	
$\Rightarrow$	`(rev(t)@[x])@acc`	by program `rev`
$\Rightarrow$	`rev(t)@([x]@acc)`	by associativity of `@`
$\Rightarrow$	`rev(t)@(x::acc)`	by unrolling the definition of `@`
$=$	`rev' (t) (x::acc)`	by i.h.
$\Leftarrow$	`rev' (x::t) acc`	by program

□

We want to emphasize that you should always state lemmas (i.e. properties) you are relying on. In the previous example, we need to know properties of append for example.

2.5 Conclusion

We presented several important induction principles and examples of induction proofs. While in practice, we will rarely verify programs completely, we may want to prove certain properties about them in practice, for example we may want to prove that some confidential data is not leaked, or only some designated principals will have access to a given resource. These properties will typically follow the same induction principles we have seen in these notes.

There is a wide spectrum of properties we would like to enforce about programs. Types, as we encounter them in a language such as OCaml, SML, or Haskell enforce fairly simple properties. For example, the type of the lookup function is `'a * ('a * 'b) tree -> 'b option`. While this gives us a partial correctness guarantee, it does for example not ensure that the tree passed is a binary search tree. On the other hand, type systems are great because they enforce a property statically. When you change your program, the type-checker will verify if it still observes this type property. If it doesn't the type checker will give precise error messages, so the programmer can fix the problem. Inductive proofs can typically enforce stronger properties about programs than types. In fact, we can prove full correctness. However, inductive proofs have to be redone every single time your program changes. Doing them by hand is time-consuming. What is it we actually need to prove? How do we know when to generalize an induction hypothesis? What happens if a proof fails? Can we give meaningful error messages in this case? A key question is therefore how we can make type systems stronger so they can check stronger properties statically, while retaining all their good properties. Haskell Curry and William Howard discovered that there is an isormophic relationship between propositions and types; moreovoer, the proof that a proposition is valid corresponds to a program. This relationship is generally known as “propositions-as-types” and “proofs-as-programs”. Fundamentally the Curry-Howard correspondance, allows us to write programs with very rich types (i.e. types which correspond to first-order formulas and encode the full specification of a given function); if the program type checks, it is correct by construction. But that’s a question for a different course :–).

Chapter 3

Fun with Higher-Order Functions

"Computer science is the science of abstraction - creating the right model for a problem and devising the appropriate mechanizable technique to solve it." A. Aho, J. Ullman

In this chapter, we cover a very powerful programming paradigm: Higher-order functions. Higher-order functions are one of the most important mechanisms in the development of modular, well-structured, and reusable programs. They allow us to write very short and compact programs, by abstracting over common functionality. This principle of abstraction is in fact a very important software engineering principle. Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying part. The use of higher-order functions allows you to easily abstract out varying parts.

Since abstracting over varying parts to allow code reuse is such an important principle in programming, many languages support it in various disguises. Recently, the concept of higher-order functions has received particular attention since it features prominently in Google's MapReduce framework for distributed computing and in the Hadoop project, an open-source implementation to support large-scale data processing which is used widely. These frameworks are used to process 20 to 30 petabytes of data a day and simplify data processing on large clusters. Although mostly written in C++, the philosophy behind MapReduce and Hadoop is functional:

"Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages."
-Jeffrey Dean and Sanjay Ghemawat

Higher-order functions also play an important role in code generation, i.e. programs which generate other programs. Later on in the course, we will discover that higher-order functions also are key to understanding lazy computation and handling infinite data. Finally, we will see that they allow us to model closures and objects as you know them in object-oriented programming.

3.1 Passing Functions as Arguments

Functions form the powerful building blocks that allow developers to break code down into simple, more easily managed steps, as well as let programmers break programs into reusable parts. As we have seen early on in the course, functions are values, just as numbers and booleans are values.

But if they are values, can we write functions which take other functions as arguments? In other words, can we write a function `f:int * (int -> int) -> int`? Would this be useful? The answer is YES! It is in fact extremely useful.

Consider the following implementation of the function which computes $\sum_{k=a}^{k=b} k$.

```
let rec sumInts (a,b) = if (a > b) then 0 else a + sumInts(a+1,b)
```

Similarly, we can compute the function $\sum_{k=a}^{k=b} k^2$ or $\sum_{k=a}^{k=b} k^3$ or $\sum_{k=a}^{k=b} 2^k$

```
let rec sumSquare(a,b) = if (a > b) then 0 else square(a) + sumSquare(a+1,b)
let rec sumCubes (a,b) = if (a > b) then 0 else cubes(a)  + sumCubes(a+1,b)
let rec sumExp   (a,b) = if (a > b) then 0 else exp(2, a) + sumExp(a+1, b)
```

All these functions look very similar, and the code is almost the same. But obviously, the sum depends on what function we are summing over! It is natural to ask, if we can write a generic sum function where we only give a lower bound `a`, and upper bound `b`, and a function `f` which describes what needs to be done in each iteration. The answer is, YES, we can! Here is how it works:

```
(* sum: (int -> int) -> int * int -> int *)
let rec sum f (a,b) =
  if (a > b) then 0
  else (f a) + sum(f,a+1,b)
```

We can then easily obtain the previous functions as follows:

```
let rec sumInts'  (a,b) = sum (fun x -> x)        (a, b)
let rec sumCubes' (a,b) = sum cube                (a, b)
let rec sumSquare'(a,b) = sum square              (a, b)
let rec sumExp'   (a,b) = sum (fun x -> exp(2,x)) (a, b)
```

Note that we can create our own functions using `fun x -> e`, where `e` is the body of the function and `x` is the input argument, and pass them as arguments to the function `sum`. Or we can pass the name of a previously defined function like `square` to the function `sum`. In general, this means we can pass functions as arguments to other functions.

What about if we want to sum up all the odd numbers between a and b?

```
(* sumOdd: int -> int -> int *)
let rec sumOdd (a, b) =
  let rec sum' (a,b) =
      if a > b then 0
      else a + sum'(a+2, b)
  in
    if (a mod 2) = 1 then
      (* a was odd *)
      sum'(a,b)
    else
      (* a was even *)
      sum'(a+1, b)
```

This seems to suggest we can generalize the previous `sum` function and abstract over the increment function.

```
let rec sum' f (a, b) inc =
    if (a > b) then 0
    else (f a) + sum' f (inc(a),b) inc
```

Isn't writing products instead of sums similar? – The answer is also YES. Consider computing the product of a function $f(k)$ for k between a and b.

```
(* product : (int -> int) -> int * int -> (int -> int) -> int *)
let rec product f (lo,hi) inc =
  if (lo > hi) then 1
  else (f lo) * product f (inc(lo),hi) inc

(* Using product to define factorial. *)
let rec fact n = product (fun x -> x) (1, n) (fun x -> x + 1)
```

The main difference is two-folded: First, we need to multiply the results, and second we need to return 1 as a result in the base case, instead of 0.

So how could we abstract over addition and multiplication and generalize the function further? – We could define such a function `series`? – Our goal is to write this function tail-recursively and we use an accumulator `{acc:int}` in addition to a function `f:int -> int`, a lower bound `a:int`, an upper bound `b:int`, increment function `inc:int -> int`. We also need parameterize the function `series` with a function `comb:int -> int -> int` to combine the results. By instantiating `comb` with addition (i.e. `(fun x y -> x + y)`) we obtain the function `sum` and by instantiating it with multiplication (i.e. `(fun x y -> x * y)`) we obtain the function `prod`.

```
(* series: (combiner : int -> int -> int) ->
           (    f       : int -> int) ->
           ( (a,b)    : int * int ) ->
           ( inc      : int -> int) ->
           ( acc      : int)
        -> ( result   : int)
*)
let rec series comb f (a,b) inc acc =
  let rec series' (a, acc) =
      if (a > b) then acc
      else
  series' (inc(a), comb acc (f a))
  in
    series'(lo, r)

let rec sumSeries  f (a,b) inc = series (fun x y -> x + y) f (a, b) inc 0
let rec prodSeries f (a,b) inc = series (fun x y -> x * y) f (a, b) inc 1
```

Ok, we will stop here. Abstraction and higher-order functions are very powerful mechanisms in writing reusable programs.

3.1.1 Example 1: Integral

Let us consider here one familiar problem, namely approximating an integral (see Fig. 3.1). The integral of a function $f(x)$ is the area between the curve $y = f(x)$ and the x-axis in the interval $[a, b]$. We can use the rectangle method to approximate the integral of $f(x)$ in the interval$[a, b]$, made by summing up a series of small rectangles using midpoint approximation.

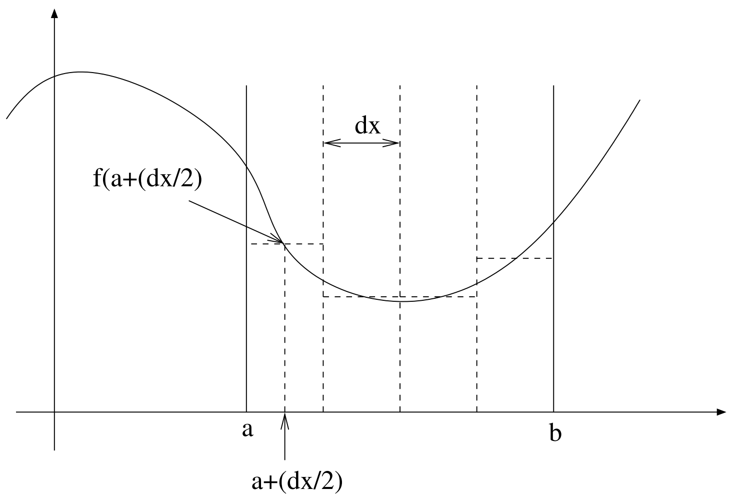


Figure 3.1: Integral between a and b

To approximate the area between a and b, we compute the sum of rectangles, where the width of the rectangle is dx. The height is determined by the value of the function f. For example, the area of the first rectangle is $dx * f(a + (dx/2))$.

Then we can approximate the area between a and b by the following series, where $l = a + (dx)/2$ is our starting point.

$$\begin{aligned}\int_a^b f(x)\,dx &\approx f(l) * dx + f(l + dx) * dx + f(l + dx + dx) * dx + \ldots \\ &= dx * (f(l) + f(l + dx) + f(l + 2 * dx) + f(l + 3 * dx) \ldots)\end{aligned}$$

Assuming we have an iterative sum function `iter_sum` for reals, we can now compute the approximation of an integral as follows:

```
let rec integral f (a,b) dx =
   dx *. iter_sum f (a +. (dx /. 2.0) , b) (fun x -> x +. dx)
```

This is very short and elegant, and directly matches our mathematical theory. Alternatively, we could directly sum up over the area without factoring out dx. In other words, we could directly implement the definition

$$\int_a^b f(x)\,dx \approx f(l) * dx + f(l + dx) * dx + f(l + dx + dx) * dx + \ldots$$

as follows:

```
let rec integral' f (a,b) dx =
  iter_sum (fun x -> f x *. dx) (a +. dx /. 2.0, b) (fun x -> x +. dx)
```

3.1.2 Example 2: Half interval method

The half-interval method is a simple but powerful technique for finding roots of an equation $f(x) = 0$, where f is a continuous function. The idea is that, if we are given points a and b such that $f(a) < 0 < f(b)$, then f must have at least one zero between a and b. To locate a zero, let x be the average of a and b and compute $f(x)$. If $f(x) > 0$, then f must have a zero between a and x. If $f(x) < 0$, then f must have a zero between x and b. Continuing in this way, we can identify smaller and smaller intervals on which f must have a zero. When we reach a point where the interval is small enough, the process stops. Since the interval of uncertainty is reduced by half at each step of the process, the number of steps required grows as $\Theta(\log(l/t))$, where l is the length of the original interval and t is the error tolerance (that is, the size of the interval we will consider "small enough"). Here is a procedure that implements this strategy:

```
let rec halfint (f:float -> float) (a, b) t =
  let mid = (a +. b) /. 2.0   in
    if  abs_float (f mid) < t then mid
    else
      if (f mid) < 0.0
        then halfint f (mid, b) t
      else halfint f (a, mid) t
```

3.1.3 Combining Higher-order Functions with Recursive Data-Types

We can also combine higher-order functions with recursive data-types. One of the most common uses of higher-order functions is the following: We want to apply a function `f` to all elements of a list.

```
(* map: ('a -> 'b) -> 'a list -> 'b list *)
let rec map f l = match l with
  | []   -> []
  | h::t -> (f h)::(map f t)
```

This is the first part in the MapReduce framework. What is second part "Reduce" referring to? - It is referring to another popular higher-order function, often called `fold` in functional programming. Essentially fold applies a function `f` over a list of elements and produces a single result by combining all elements using `f`. For example, let's say we have a list of integers `[1;2;3;4]` and we want to add all of them. Then we essentially want to use a function `add` which cumulatively is applied to all elements in the list and adds up all the elements resulting in

```
add(1, add(2, add(3, add(4, ?) ) ) )
```

To stop the successive application of the function we also need a base. In our case where we add all the elements, the initial element would be `0` (i.e. we replace `?` with `0`).

We can observe that cumulatively applying a given function to all elements in a list is useful for many situations: we might want to create a string "1234" from the given list (where the initial element would be the empty string), or we might want to multiply all elements (where the initial element would be `1`), etc.

We also observe that we sometimes want to gather all the results in the reverse order. While the former function is often referred to as fold-right the latter one is often described as fold-left.

```
add 4 (add 3 (add 2 (add 1 0 ) ) )
```

Implementing a function `fold_right` which folds elements from right to left is straightforward.

```
(* fold_right  ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b *)
(* fold_right  f [x1; x2; ...; xn] init
    returns

    f x1 (f x2 ... (f xn init)...)

    or init if the list is empty.
*)
let rec fold_right  f l b = match l with
  | []    -> b
  | h::t -> f h (fold_right f t b)
```

Finally, we revisit another popular higher-order function, called `filter`. This function can be used to filter out all the elements in a list which fulfill a certain predicate `p:'a -> bool`.

```
(* filter: ('a -> bool) -> 'a list -> 'a list *)
let rec filter (p : 'a -> bool) l = match l with
  | []    -> []
  | x::l ->
    if p x then x::filter p l
    else filter p l
```

You will find many similar functions already defined in the OCaml basis library.

3.2 Returning Functions as Results

In this section, we focus on returning functions as results. We will first show some simple examples demonstrating how we can transform functions into other functions. This means that higher-order functions may act as *function generators*, because they allow *functions to be returned as the result from other functions*.

Functions are very powerful. In fact, using a calculus of functions, the lambda-calculus, we can cleanly define what a computable function is. The lambda calculus is universal in the sense that any computable function can be expressed and evaluated using this formalism. It is thus equivalent to the Turing machine formalism and was originally conceived by Alonzo Church (1903-1995).

The question of whether two lambda calculus expressions are equivalent cannot be solved by a general algorithm, and this was the first question, even before the halting problem, for which undecidability could be proved. The lambda calculus is also the foundation for many programming languages in particular functional programming.

3.2.1 Example 1: Currying and uncurrying

Currying has its origin in the mathematical study of functions. It was observed by Frege in 1893 that it suffices to restrict attention to functions of a single argument.

For example, for any two parameter function $f(x, y)$, there is a one parameter function f' such that $f'(x)$ is a function that can be applied to y to give $(f'(x))(y) = f(x, y)$.

This idea can be easily implemented using higher-order functions. We will show how we can translate a function `f : 'a * 'b -> 'c` which expects a tuple `x:'a * 'b` into a function `f' : 'a -> 'b -> 'c` to which we can pass first the argument of type `'a` and then the second argument of type `'b`. The technique was named by Christopher Strachey (1916 - 1975) after logician Haskell Curry (1900-1982), and in fact any function can be curried or uncurried (the reverse process).

In general, we call *currying* the transformation of a function which takes multiple arguments in form of a tuple in such a way that it can be called as a chain of functions each with a single argument

```
(* curry : (('a * 'b) -> 'c) -> ('a -> 'b -> 'c) *)
let curry f = (fun x y -> f (x,y))

(* uncurry: ('a -> 'b -> 'c) -> (('a * 'b) -> 'c) *)
let uncurry f = (fun (y,x) -> f y x)
```

We say that `f: 'a -> 'b -> 'c` is the curried form of `f': 'a * 'b -> 'c`. To create functions we use the nameless function definition `fun x -> e` where `x` is the input and `e` denotes the body of the function.

A word about parenthesis and associativity of function types Given a function type `'a -> 'b -> 'c`, this is equivalent to `'a -> ('b -> 'c)`; *function types are right-associative*. This in fact corresponds to how we use a function `f` of type `'a -> 'b -> 'c`. Writing `f arg1 arg2` is equivalent to writing `(f arg1) arg2`; *function application is left-associative*.

Let's look at why this duality between the function type and function application makes sense; `f` has type `'a -> 'b -> 'c` and `arg1` has type `'a`. Therefore `f arg1` must have type `'b -> 'c`. Moreover, applying `arg2` to the function `f arg1`, will return a result of type `'c`.

$$\underbrace{\texttt{(f arg1)}}_{\texttt{'b -> 'c}}\ \texttt{arg2} :' c$$

Some more examples of higher-order functions Recall that the following two function definitions are equivalent:

```
let id = fun x -> x

(* OR *)

let id x = x
```

Another silly function we can write is a function which swaps its arguments.

```
(* swap : ('a * 'b -> 'c) -> ('b * 'a -> 'c) *)
let swap f = (fun (x,y) -> f (y,x))
```

3.2.2 Example 2: Derivative

A bit more interesting is to implement the derivative of a function f. The derivative of a function f can be computed as follows:

$$\frac{df}{dx} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

We can approximate the result of the derivative by choosing ϵ to be some small number. Then given a function `f:float -> float` and a small number `dx`, we can compute a function `f'` which describes the derivative of `f`.

```
let deriv (f, dx) = fun x -> (f (x +. dx) -. f x) /. dx
```

3.2.3 Example 3: Smoothing

The idea of smoothing a function is an important concept in signal processing. If f is a function and dx is some small number, then the smoothed version of f is the function whose value at a point x is the average of f(x − dx), f(x), and f(x + dx). The function smooth takes as input f and a small number dx and returns a function that computes the smoothed f.

```
let smooth(f,dx) = fun x -> (f(x -. dx)+. f(x) +. f(x+dx)) /. 3.0
```

It is sometimes valuable to repeatedly smooth a function (that is, smooth the smoothed function, and so on) to obtained the n-fold smoothed function.

3.2.4 Example 4: Partial evaluation and staged computation

An important application of higher-order functions and the ability to return functions lies in partial evaluation, staged computation and code generation.

Partial evaluation is best illustrated by considering an example. For example, given the following generic function,

```
(* funkyPlus : int -> int -> int *)
let funkyPlus x y = x * x + y
```

we can first pass it `3`. This generates a function `fun y -> 3 * 3 + y` which is partially evaluated. So, we have generated a function which will increment its input by `9`.

```
(* plus9 : int -> int *)
let plus9 = funkyPlus
```

We only partially evaluate the `funkyPlus` function by passing only one of the arguments to it. This yields again a function! We can see that templates, which occur in many programming languages are in fact nothing more than higher-order functions. Note that you cannot actually see what the new function `plus9` looks like. Functions are first-class values, and the OCaml interpreter (as any other interpreter for languages supporting functions) never prints functions back to your screen. It will simply return to you the type.

Nevertheless, given our understanding of the underlying operational semantics, we can try to understand what happens when we execute `funkyPlus 3`.

`funkyPlus 3 = (fun x -> fun y -> x * x + y) 3` $\Longrightarrow$ `fun y -> 3 * 3 + y`

To evaluate the application `(fun x -> fun y -> x * x + y) 3`, we substitute `3` for `x` in the body of the function, i.e. `fun y -> x * x + y`. This results in the expression `fun y -> 3 * 3 + y`.

Note, the result is **not** `fun y -> 9 + y`. We never evaluate inside the function body. This is important because in effect this will allow us to deliberately delay the evaluation of some expression.

How would we generate a function which fixes `y`, but awaits still the input `x`?

```
let plus3' = (fun x -> funkyPlus x 3)
```

The idea of partial evaluation is quite powerful to achieve code which can be configured during run-time and code which can achieve considerable efficiency gain, especially if we stage our computation properly.

Staged computation refers to explicit or implicit division of a task into stages. It is a standard technique from algorithm design that has found its way to programming languages and environments. Examples are partial evaluation which refers to the global specialization of a program based on a division of the input into static (early) and dynamic (late) data, and run-time code generation which refers to the dynamic generation of optimized code based on run-time values of inputs.

At the heart of staged computation lies the goal to force evaluation of some part of the program before continuing with another part. For example, let us reconsider the result of `(funkyPlus 3)` which was `fun y -> 3 * 3 + y`. How could we force the evaluation of `3*3` so we actually produce a function `fun y -> 9 + y`. We will re-write the `funkyPlus` function as follows:

```
let funky x = let r = x * x in fun y -> r + y
```

Now the evaluation of `funky 3` will proceed as follows:

```
funky 3 = fun x -> let r = x * x in (fun y -> r + y) 3
⟹ let r = 3 * 3 in fun y -> r + y
⟹ let r = 9 in fun y -> r + y
⟹ fun y -> 9 + y
```

By introducing a let-expression before creating the second function which is waiting for the second input, we have pulled out the essential part of the code and will evaluate it before returning the function `fun y -> 9 + y` as a result.

In this example of `funkyPlus`, staging did not have a great impact on the execution and run-time behavior of our function. However, there are many examples where it can be more efficient to write staged code.

Consider we have first defined a horrible computation:

```
(* val horriblecomputation : int -> int *)
let rec horriblecomputation x =
  let rec ackermann m n = match (m , n) with
    | 0 , n -> n+1
    | m , 0 -> ackermann (m-1) 1
    | m , n -> ackermann (m-1) (ackermann m (n-1)) in
  let y = (abs x) mod 3 + 2 in
  let rec count n = match n with
    | 0 -> ackermann y 4
    | n -> count(n-1) + 0 * ackermann y 4 in
  let large = 1000 in
    (ackermann y 1) * (ackermann y 2) * (ackermann y 3) * count large
```

Executing the horrible computation will take a very long time (you can try!). Next, let's say we have two columns of test values and we are supposed to write a function `test` which takes the value `v1` from the first column and a value `v2` from the second column and computes

```
horribleComputation(v1) + v2
```

This is how the two columns look like.

Column 1	Column 2
10	5
2	2
	18
	22

So, we define a function `test` as follows:

```
(* val test : int * int -> int      *)
let test (x:int, y:int) =
  let z = horriblecomputation x  in
    z + y
```

If we execute `test` on a few instantiations, where the first instantiation for `x` does not change, then we have to execute this horrible computation each time.

```
let result11 = test(10, 5);
let result12 = test(10, 2);
let result13 = test(10, 18);
let result14 = test(10, 22);


let result21 = test(2, 5);
let result22 = test(2, 2);
let result23 = test(2, 18);
let result24 = test(2, 22);
```

What will happen? - Let's put this in a concrete perspective and assume computing `horribleComputation(10)` takes 5h. Then computing the results `result11`, `result12`, `result13`, and `result14` will take 20h! If we give it a list of 100 inputs, as follows

```
map (fun y -> test (10, y)) [1; 2; .... ; 100]
```

we will compute the `horribleComputation 10` exactly 100 times. This will take 500h. That's a long time.

Would it help to write a curried version?

```
(* val test' : int -> int -> int *)
let test' (x:int) (y:int) =
let z = horriblecomputation x in
  z + y
```

We can then write the following code

```
map (test' 10) [1; 2; 3; ... ; 100]
```

Isn't this better? – Well, let's see. The function `test'` is equivalent to the following :

```
let test' =
 fun x -> fun y -> let z = horriblecomputation x in z + y
```

Computing `test' 10` will simply replace `x` with `10` in the body of the function. According to our operational semantics, `test' 10` will evaluate to

```
test' 10 ⟹
fun y -> let z = horriblecomputation 10 in  z + y
```

We have achieved nothing, since the horrible computation is hidden within a function, and we never evaluate inside functions! Functions are values, hence evaluation will stop as soon as it has computed a function as a result. As a consequence we still will compute the horrible computation every time we call the function `test10`. But this is exactly what we wanted to avoid!

How can we generate a function which will only compute the horrible computation once and capture this result so we can re-use it for different inputs of `y`? – The idea is that we need to factor out the horrible computation. When given an input `x`, we compute the horrible computation yielding `z`, and then create a function which awaits still the input `y` and adds `y` and `z`.

```
let testCorrect (x:int) =
  let  z = horriblecomputation x in (fun y -> z + y)

let r = map (testCorrect 10) [1;2;3; ... ; 100]
```

Note, we now compute the horrible computation once, when executing `testCorrect 10` and created a closure which stores its result. Generating `testCorrectly10` will take 5h. But, when we use `testCorrectly10`, we will be avoiding the recomputation of the horrible computation, and it will be able to compute the results very quickly.

To understand the impact of partial evaluation, it is key to understand how functions are evaluated. Try to figure out how often `horribleComputation(10)` gets executed in the following examples:

1. `map (curry test 10) [1; 2; ...; 100]`

2. `map (fun x -> test' 10 x) [1; 2;  .. ; 100]`

This idea of staging computation is based on the observation that a partial evaluator can convert a two-input program into a staged program that accepts one input and produces another program that accepts the other input and calculates the final answer. The point being that the first stage may be run ahead of time (e.g. at compile time) while the second specialized stage should run faster than the original program. Many current compilers for functional programming employ partial evaluation and staged computation techniques to generate efficient code. It is worth pointing out that to understand this optimization, we really need to understand the operational semantics of how programs are executed.

3.2.5 Example 5: Code generation

Let us consider again the following simple program to compute the exponent.

```
(* pow k n = n^k *)
let rec pow k n = if k = 0 then 1
  else n * pow (k-1) n
```

While we wrote this program in curried form, when we partially evaluate it with `k` = 2 we will not have generated a function "square". What we get back is the following:

```
fun n -> n * pow 1 n
```

The recursive call of pow is shielded by the function abstraction (closure). One interesting question is how we can write a version of this power function which will act as a generator. When given `2` it will produce the function "square", when given `3` it produce the function "cube", etc. So more generally, when given an integer `k` it will produce a function which computes $\underbrace{n * \ldots * n}_{k} * 1$. This result should in the end be completely independent of the original `pow` definition.

The simplest solution, is to factor out the recursive call before we build the closure.

```
let rec powGen k cont = if k = 0 then cont
  else powGen (k-1) (fun n -> n * (cont n))
```

This allows us to compute a power generation function. To illustrate, let us briefly consider what happens when we execute powG 2. Let us first start with powG 1

```
     powG 1
⇒    let c = powG 0 in (fun n -> n * c n)
⇒    let c = (fun n0 -> 1) in (fun n -> n * c n)
⇒    fun n -> n * (fun n0 -> 1) n
```

```
     powG 2
⇒    let c = powG 1 in (fun n2 -> n2 * c n2)
⇒    let c = (fun n1 -> n1 * (fun n0 -> 1) n1) in (fun n2 -> n2 * c n2)
⇒    (fun n2 -> n2 * (fun n1 -> n1 * (fun n0 -> 1) n1) n2)
```

Is this final result really the square function? – Yes, it is. Intuitively, this function is equivalent to

```
  fun n2 -> n2 * n2 * 1
```

Although OCaml will not evaluate any applications inside functions, conceptually, we can see that the result is equivalent to `fun n2 -> n2 * n2 * 1` by looking inside the function and reducing the applications.

```
               (fun n2 -> n2 * (fun n1 -> n1 * (fun n0 -> 1) n1) n2 )
reduces to     fun n2 -> n2 * n2 * (fun n0 -> 1) n2)
reduces to     fun n2 -> n2 * n2 * 1
```

So yes, we will have generated a version of the square function. Note that languages supporting templates or macros work in a very similar way and higher-order functions are one way of explaining such concepts.

Chapter 4

References

Previously, we were careful to emphasize that expressions in OCaml (or similar functional languages) always have a type and evaluate to a value. In this note, we will see that expressions can also have an *effect*. An *effect* is an action resulting from evaluation of an expression other than returning a value. In particular, we will consider language extension which supports allocation and mutation of storage during evaluation. During evaluation, we will see that storage may be allocated, and updated, and thereby effect future evaluation.

This is one of the main differences between pure functional languages such as Haskell (languages which do not support effectful computation directly) and impure functional language (languages which do support effectful computation directly).

4.1 Binding, Scope

So far, we have seen variables in let-expressions or functions. An important point about these local variables or *binders* were that 1) they only exist within a scope and 2) their names did not matter. Recall the following example:

```
let test () =
  let pi   = 3.14 in                          (* 1 *)
  let area = (fun r -> pi *. r *. r) in       (* 2 *)
  let a2   = area (2.0) in                    (* 3 *)
  let pi   = 6.0  in                          (* 4 *)
  let a3 = area 2.0 in
    print_string ("Area a2 = " ^ string_of_float a2 ^ "\n");
    print_string ("Area a3 = " ^ string_of_float a3 ^ "\n")
;
```

The binder or local variable `pi` in line `(* 1 *)` is bound to 3.14. If we are trying to establish a new binding for `pi` in line `(* 4 *)`, then this only overshadows the previos binding, but it is important to note that it does not effect or change other bindings such as area or a2. In fact, OCaml will give you a warning:

```
    let pi   = 6.0  in                                  (* 4 *)
        ^^
Warning Y: unused variable pi.
```

Since we do not use the second binding for pi in the subsequent code, the variable pi with the new value 6.0 is unused. The result of the expression above will be 12.56, and a2 will have value 12.56. The fact that we overshadowed the previous binding of pi in line (* 4 *) did not have any effect.

4.2 Reference Cells

To support mutable storage, we will extend our language with reference cells. A reference cell may be thought of as a container in which a data value of a specified type is stored. During execution of a program, the contents of a cell by be *read* or *replaced* by any other value of the appropriate type. Since reference cells are updated and read by issuing "commands", programming with cells is also called *imperative programming*.

Changing the contents of a reference cell introduces a temporal aspect. We often will speak of *previous* and *future* values of a reference cell when discussing the behavior of a program. This is in sharp contrast to the effect-free fragment we have encountered so far. A binding for a variable does not change when we evaluate within the scope of that variable. The content of a reference cell may well change when we evaluate another expression.

A reference cell is *created* or *allocated* by the constructor ref. ref 0 will create a reference cell with content 0, i.e. it is a reference cell in which we store integers. Hence the type of ref 0 will be int ref. Reference cells are like all other values. They may be bound to variable names, passed as arguments to functions, returned as results of functions, even be stored within other reference cells.

Here are some examples:

```
let r = ref 0
let s = ref 0
```

In the first line, we create a new reference cell and initializes it with content 0. The name of the reference cell is r. In the second line, we create a new reference cell with content 0 and bind it to the name s. It is worth pointing out that s and r are **not** referring to the same reference cell! In fact, we can compare s and r, by r == s which will evaluate to false.

In OCaml, there are two comparisons we can in fact make:

- `r == s` compares whether the cell `r`, i.e. `r`'s address in memory, is the same cell as `s`, i.e. `s`'s address in memory. `==` compares the actual location, i.e. addresses. In the above example, this returns false.

- `r = 1` compare the content of the cell `r` with the content of the cell `s`. In the above example, this will return true.

We can *read the content* of a reference cell using the construct `!`. `!r` yields the current content of the reference cell `r`, in this example it will return `0`.

We can also directly pattern match on the content of a cell writing the following pattern:

```
let {contents = x} = r;;
val x : int = 0
# let {contents = y} = s;;
val y : int = 0
```

Here `x` and `y` are bound to the values contained in cell `r` and `s` respectively.

To change the content of a reference cell, we use the *assignment* operator `:=` and it typically written as an infix operator. The first argument must be a reference cell of type τ `ref`, and the second argument must be an expression of type τ. The assignment `r := 5` will replace the content of the cell `r` with the value `5`. Note that the old content of the cell `r` has been destroyed and does not exist anymore!

Consider the following piece of code:

```
let r = ref 0
let s = ref 0
let a = r=s
let a = r==s
let _ = r := 3          (* 1 *)
let x = !s + !r         (* 2 *)
let t = r               (* 3 *)
let w = r = s           (* 4 *)
let v = t = s           (* 5 *)
let b = s==t            (* 6 *)
let c = r==t            (* 7 *)
let _ = t := 5          (* 8 *)
let y = !s + !r         (* 9 *)
let z = !t + !r         (* 10 *)
```

In line `(* 2 *)`, the variable `x` will be bound to 3. Line `(* 3 *)` establishes a new binding. We can now use the name `t` to refer to the same reference cell as `r`. This is also called *aliasing*, i.e. two variables are bound to the same reference cell. Comparing the content of `r` with the content of `s` (see line `(* 4 *)`) will return `false`. Comparing the content of `t` and `s` (see line `(* 5 *)`)) also returns `false`. Comparing the address of cell `s` with the address of cell `t` will also return `false`. Obviously, the cell s and the cell t are not the same - they have different addresses in memory and contain different values.

However, comparing the address of cell r with the address of cell t will also return true! Updating the content of the reference cell t in line (* 8 *) will mean that also r will have the new value 5. Remember, that t and r do **not** stand for different reference cells, but they are just different names for the **same** reference cell!

Finally, y will evaluate to 5 (see line (* 9 *)), and z will evaluate to 10 (see line (* 10 *)).

Notice also the use of a wildcard in line (* 1 *). The value of the expressions r := 3 is discarded by the binding. Assignment r := 3 has the empty value () and is of type unit.

Often it is convenient to sequentially compose expressions. This can be done using the semicolon ;. The expression

```
 exp1 ; exp2
```

is shorthand for the expression

```
let _ = exp1 in  exp2
```

It essentially means we first evaluate exp1 for its effect, and then evaluate exp2. Rewriting the introductory example with references gives us the following:

```
let test_update () =
  let pi   = ref 3.14 in                                (* 1 *)
  let area = (fun r -> !pi *. r *. r) in                (* 2 *)
  let a2   = area (2.0) in                              (* 3 *)
  let _    = (pi := 6.0) in                             (* 4 *)
  let a3 = area 2.0 in
    print_string ("Area a2 = " ^ string_of_float a2 ^ "\n");
    print_string ("Area a3 = " ^ string_of_float a3 ^ "\n")
;
```

Note that now a2 will still bound to 12.56, while when we compute the value for a3 the result will be 24. Updating the reference cell pi will have an effect on previously defined function. Since an assignment destroys the previous values held in a reference cell, it is also sometimes referred to as *destructive update*.

4.3 Observation

It is worth pointing out that OCaml and other ML-languages distinguishe cleanly between the reference cell and the content of a cell. In more familiar languages, such as C all variables are implicitely bound to reference cells, and they are implicitly dereferenced whenever they are used so that a variable always stands for its current contents.

Reference cells make reasoning about programs a lot more difficult. In OCaml, we typically use references sparingly. This makes OCaml programs often simpler to reason about.

4.4 Programming well with references

Using references it is possible to mimic the style of programming used in imperative languages such as C. For example, we might define the factorial function in imitation of such languages as follows:

```
let imperative_fact n =
  let result = ref 1 in
  let i = ref 0 in
  let rec loop () =
    if !i = n then ()
    else (i := !i + 1; result := !result * !i; loop ())
  in
    loop (); !result
```

Notice that the function `loop` is essentially just a while loop. It repeatedly executes its body until the contents of the cell bound to `i` reaches `n`. *This is bad style!!!* The purpose of the function `imperative_fact` is to compute a simple function on natural numbers. There is no reason why state must be maintained. The original program is shorter, simpler, more efficient, and hence more suitable.

However, there are good uses of references and important uses of state. For example, we may need to generate fresh names. To implement a function which generates fresh names, we clearly need a global variable which keeps track of what variable names we have generated so far.

```
let counter = ref 0

(* newName () ===> a,  where a is a new name *)
(*
  Names are described by strings denoting natural numbers.
 *)
let newName () =
  (counter := !counter+1;
   "a"^ string_of_int (!counter))
```

Later on we will see how to in fact model objects using functions (=closures) and references.

4.4.1 Mutable data structures

So far we have only considered immutable data structures such as lists or trees, i.e. data structures that it is impossible to change the structure of the list without building a modified copy of that structure. Immutable data structures are persistent, i.e. operations performed on them does not destroy the original structure. This often makes our implementations easier to understand and reason about. However, sometimes we do not want to rebuild our data structure. A classic example is maintaining a dictionary. It is clearly wasteful if we would need to carry around a large dictionary and when we want to update it, we need to make a copy of it. This is what we would like in this case is an ”in place update” operation. For this we must have *ephemeral* (opposite of persistent) datastructures. We can achieve this by using references in SML.

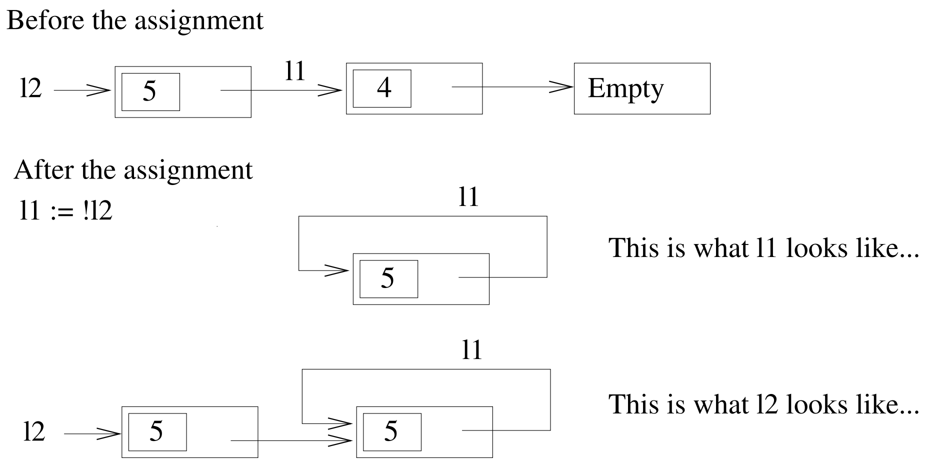
```
(* Datatype for Reference Lists. *)
type 'a rlist = Empty | RCons of 'a * 'a rlist ref
```

Then we can define a circular list as follows:

```
# let l1 = ref (RCons(4, ref Empty))
  let l2 = ref (RCons(5, l1));;
val l1 : int rlist ref = {contents = RCons (4, {contents = Empty})}
val l2 : int rlist ref =
  {contents = RCons (5, {contents = RCons (4, {contents = Empty})})}


# l1 := !l2;;
```

How does the list l1 and the list l2 look like?



You can also ask OCaml to show you the result of `l1`. It is quite patient and will print the circular list repeating `RCons(5, ...)` until you simply see

```
RCons (5,{contents = RCons (...)})
```

We can also observe its output behavior using the following function `observe:'a rlist * int -> ()` . Given a reference list l and a bound n, this function will print the first n elements. If we would not have this bound n, then observe would print repeatedly the elements in a circular list and would loop for circular lists.

```
let rec observe l n = match l with
  | Empty ->  print_string "0"
  | RCons(x, l) ->
     if n = 0 then print_string "STOP\n"
     else (print_string (string_of_int x ^ " ");
     observe !l (n-1))
```

So for example here is what happens if we want to observe how `l1` looks like.

```
# observe !l1 5;;
5 5 5 5 5 STOP
- : unit = ()
#
```

4.4.2 Destructive append and reverse

Next, consider appending two reference lists. Intuitively, the following should happen: The first list is a pointer to a reference list. So we just walk to the end of this list, and re-assign it to the second list! What should the type of the function `rapp` be? Is there anything returned? No. We are destructively modifying the first list. So we will write the following function:

```
type 'a refList = ('a rlist) ref

(* rapp: ' a refList * 'a refList -> unit*)
let rec rapp r1 r2 = match r1 with
  | {contents = Empty} ->  r1 := !r2
  | {contents = RCons (h,t)} -> rapp t r2
```

Note that as a side effect the reference list `r1` is updated. This change can only be observed by reading `r1` after we have appended another list. So for example:

```

# let r1 = ref (RCons (1, ref Empty));;
val r1 : int rlist ref = {contents = RCons (1, {contents = Empty})}
# let r2 = ref (RCons (4, ref (RCons (6, ref Empty))));;
val r2 : int rlist ref =
  {contents = RCons (4, {contents = RCons (6, {contents = Empty})})}
# rapp r1 r2;;
- : unit = ()
# r1 ;;
- : int rlist ref =
{contents =
  RCons (1,
   {contents = RCons (4, {contents = RCons (6, {contents = Empty})})})}
# r2 ;;
- : int rlist ref =
{contents = RCons (4, {contents = RCons (6, {contents = Empty})})}

```

Note how executing the function `rapp` has modified the list `r1` is pointing to.

Next, let us consider how reverse could be implemented. Notice again that we accomplish this destructively, and the function `rev` returns unit as a result.

```
(* rev : ’a refList -> unit *)
let rec rev l0 =
  let r = ref Empty in
    let rec rev’ l = match l with
    | {contents = Empty} -> l0 := !r
    | {contents = RCons(h,t)} ->
      (r := RCons(h, ref (!r));
       rev’ t)
  in
    rev’ l0
```

The idea is that we first create a tempory reference to an empty list. While we recursively traverse the input list `l`, we will pop the elements from `l` and push them on to the tempory reference list `r`. When we are done, we let reassign `l`. Note that `l` is a pointer to a list, and by `l := !r` we let this pointer now point to the reversed list. In our program, we use pattern matching on reference cells to retrieve the value stored in a location. Instead of pattern matching, we could also have used the `!`-operator for simply reading from a location the stored value.

```
(* rev : ’a refList -> unit *)
let rev l0 =
  let r = ref Empty in
  let rec rev’ l = match !l with
    | Empty -> l0 := !r
    | RCons (h,t) ->
      (r := RCons(h, ref (!r));
       rev’ t)
  in
    rev’ l0
```

Again we can observe the behavior.

```
# rev r1;;
- : unit = ()
# r1;;
- : int rlist ref =
{contents =
  RCons (6,
   {contents = RCons (4, {contents = RCons (1, {contents = Empty})})})}
```

```
#
```

Often in imperative languages, we explicitly return a value. This amounts to modifying the program and returning `l0`.

```
let rev l0 =
  let r = ref Empty in
  let rec rev' l = match !l with
    | Empty -> l0 := !r
    | RCons (h,t) ->
      (r := RCons(h, ref (!r));
       rev' t)
  in
    (rev' l0; l0)
```

It is not strictly necessary to actually return `l0`, as whatever location we passed as input to the function `rev` was updatd destructively and hence it will have changed.

4.5 Closures, References and Objects

We can also write a counter which is incremented by the function `tick` and reset by the function `reset`.

```
let counter = ref 0 in
let tick () = counter := !counter + 1; !counter in
let reset () = (counter := 0) in
  (tick , reset)
```

This declaration introduces two functions `tick:unit -> int` and `reset:unit -> unit`. Their definitions share a private variable `counter` that is bound to a reference cell containing the current value of a shared counter. The `tick` operation increments the counter and returns its new value, and the `reset` operation resets its value to zero. The types already suggest that implicit state is involved. We then package the two functions together with a tuple.

Suppose we wish to have several different instances of a counter and different pairs of functions tick and reset. We can achieve this by defining a counter generator. We first declare a record `counter_object` which contains two functions (i.e. methods). You can think of this as the interface or signature of the object. We then define the methods in the function `newCounter` which when called will give us a new object with methods `tick` and `reset`.

```
type counter_object = {tick : unit -> int ; reset: unit -> unit}

let newCounter () =
  let counter = ref 0 in
    {tick = (fun () -> counter := !counter + 1; !counter) ;
     reset = fun () -> counter := 0}
```

We've packaged the two operations into a record containing two functions that share private state. There is an obvious analogy with class-based object-oriented programming. The function `newCounter` may be thought of as a *constructor* for a class of counter *objects*. Each object has a private instance variable `counter` that is shared between the methods `tick` and `reset`.

Here is how to use counters.

```

# let c1 = newCounter();;
val c1 : counter_object = {tick = <fun>; reset = <fun>}
# let c2 = newCounter();;
val c2 : counter_object = {tick = <fun>; reset = <fun>}
# c1.tick;;
- : unit -> int = <fun>
# c1.tick ();;
- : int = 1
# c1.tick ();;
- : int = 2
# c2.tick ();;
- : int = 1
# c1.reset ();;
- : unit = ()
# c2.tick ();;
- : int = 2
# c2.tick ();;
- : int = 3
# c1.tick ();;
- : int = 1
#
```

Notice, that `c1` and `c2` are distinct counters!

4.6 Other Common Mutable Data-structures

We already have seen records as an example of a mutable data-structure in OCaml[1]. Records are useful to for example create a data entry. Here we create a data entry for a student by defining a record type `student` with fields `name`, `id`, `birthdate`, and `city`. We also include a field `balance` which describes the amount the student is owing to the university. We define the field `balance` as `mutable` which ensures that we can update it. For example, we want to set it to 0 once the outstanding balance has been paid.

```
type student =
{name : string ;
```

[1]Records are simply a labelled n-ary tuple where we can use the label to retrieve a given component. In OCaml records are a a labelled n-arry tuple where each entry itself is a location in memory. In general, records do not have to be mutable.

```
 id: int;
 birthdate: int * int * int;
 city : string;
 mutable balance: float }
```

We can now create the first student data entry.

```
# let s1 = {name = "James McGill"; id = 206234444; birthdate = (6, 10, 1744);
    city = "Montreal"; balance = 1650.0 };;
val s1 : student =
  {name = "James McGill"; id = 206234444; birthdate = (6, 10, 1744); city = "
    Montreal"; balance = 1650.}
#
```

To access and update the student's balance we simply write:

```
# s1.balance <- 0.0;;
- : unit = ()
# s1;;
- : student =
{name = "James McGill"; id = 206234444; birthdate = (6, 10, 1744); city = "
    Montreal"; balance = 0.}
#
```

We can only update a field in a record, if it is declared to be mutable. For example, we cannot update the city to Toronto, in case James McGill moved.

```
# s1.city <- "Toronto";;
Characters 0-20:
  s1.city <- "Toronto";;
  ^^^^^^^^^^^^^^^^^^^^
Error: The record field city is not mutable
#
```

Last, we mention that OCaml has both static and dynamic arrays that resize themselves when elements are added or removed, which could be used to create a database of student data entries. For more information on arrays, please see the documentation:

https://caml.inria.fr/pub/docs/manual-ocaml/libref/Array.html

Chapter 5

Exceptions

"Ever tried. Ever failed. No Matter.
Try again. Fail again. Fail better."

Samuel Beckett

OCaml as any other flavor of the ML-language family is a safe language. This is ensured by static type checking and by dynamic checks that rule out violations that cannot be detected statically. Examples of run time exceptions are:

```
# 3 / 0;;
Exception: Division_by_zero.
```

Here the expression `3 / 0` will type check, but evaluation will incur a runtime fault that is signalled by rasing the exception `Division_by_zero`. This is an example, where the expression `3 / 0` has a type, namely `int`, it has not value, but it does have an effect, namely it raises the exception `Division_by_zero`.

Another example of a runtime exception is `Match_failure`.

```
# let head (x::t) = x ;;
Characters 9-19:
  let head (x::t) = x ;;
           ^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val head : 'a list -> 'a = <fun>
# head [];;
Exception: Match_failure ("//toplevel//", 37, -60).
#
```

The function definition of `head` and the call `head []` type check. However, `head []` does not have a value, but has an effect.

So far, we have considered built-in exceptions. Since they are pre-defined, they have a built-in meaning. However, we can also introduce and define our own exceptions to signal a specific program error.

```
exception Domain

let fact n =
  let rec f n =
    if n = 0 then 1
    else n * f (n-1)
  in
    if n < 0 then raise Domain
    else f(n)


let runFact n =
  try
    let r = fact n in
      print_string ("Factorial of " ^ string_of_int n ^ " is " ^
    string_of_int r ^ "\n")
  with Domain -> print_string "Error: Invariant violated -- trying to call
    factorial on inputs < 0 \n"
```

5.1 It's all about control

One of the most interesting uses of exceptions is for backtracking or more generally controlling the execution behaviour. Backtracking can be implemented by looking greedily for a solution, and if we get stuck, we undo the most recent greedy decision and try again to find a solution from that point. There are many examples where solutions can be effectively found using backtracking. A classic example is searching for an element in a tree.

Specifically, we want to implement a function `find t k` where `t` is a binary tree and `k` is a key. The function `find` has the type `('a * 'b) tree -> 'a -> 'b`, i.e. we store pairs of keys and data in the tree. Given the key `k` (the first component) we return the corresponding data entry `d`, if the pair `(k,d)` exists in the tree. We make no assumptions about the tree being a binary search tree. Hence, when we try to find a data entry, we may need to traverse the whole tree. We proceed as follows.

- If the tree is `Empty`, then we did not find a data corresponding to the key `k`. Hence we fail and raise the exception `NotFound`.
-

```
exception NotFound

type key = int

type 'a tree =
  | Empty
  | Node of 'a tree * (key * 'a) * 'a tree

let rec find  t k = match t with
  | Empty -> raise NotFound
  | Node (l, (k',d), r) ->
      if k = k' then d
      else
  try find l k with NotFound -> find r k
```

To understand what happens, let's see how we evaluate `find t 55` in a tree `t` which is defined as follows:

```
let ll = Node (Empty, (3, "3"), Empty)
let lr = Node (Empty, (44, "44"), Empty)
let l = Node ( ll, (7, "7"), lr)
let r = Node ( Node (Empty, (55, "55"), Empty)  , (7, "7"), Empty)
let t = Node ( l, (1, "1"), r)
```

To evaluate `find t 55` we proceed as follows:

```
find t 55
⟶* try find l 55 with NotFound -> find r 55
```

Note that we install on the call stack an exception handler to which we return in the event of `find l 55` returning the exception `NotFound`; in this case, we then proceed to compute `find r 55`.

However first, we must evaluate `find l 55`.

```
1  find l 55
2  ⟶* try find ll 55 with NotFound -> find lr 55
3  ⟶* try (try find Empty 55 with NotFound -> find Empty 55)
4        with NotFound -> find lr 55
5  ⟶* try (try raise NotFound with NotFound -> find Empty 55)
6        with NotFound -> find lr 55
7  ⟶* try find Empty 55
8        with NotFound -> find lr 55
9  ⟶* try (raise NotFound)
10       with NotFound -> find lr 55
11 ⟶* find lr 55
12 ⟶* try find Empty 55 with NotFound -> find Empty 55
13 ⟶* try (raise NotFound) with NotFound -> find Empty 55
14 ⟶* find Empty 55
15 ⟶* raise NotFound
```

Hence another exception handler is installed in line 2. If the evaluation of `find ll 55` returns an exception `NotFound`, we proceed with computing `find lr 55`. Line 3 shows the evaluation of `find lr 55`. Note that another excpetion handler is installed. Now, `find Empty 55` raises the exception `NotFound` (line4). It is handled by the innermost handler (see line 6) and we proceed to evaluate `find Empty 55`. Note here `Empty` denotes the right subtree of `ll`. This again triggers the exception `NotFound` and we proceed to `find lr 55`. This will eventually also raise the exception `NotFound`.

As a consequence, we return to the first exception handler that was installed when we were starting the evaluation:

```
find t 55
⟶* try find l 55 with NotFound -> find r 55
```

eventually steps to

```
⟶* try (raise NotFound) with NotFound -> find r 55
⟶* find r 55
⟶* try  find (Node (Empty, (55, "55"), Empty)) 55
    with NotFound -> find Empty 55
⟶* try "55"
    with NotFound -> find Empty 55
⟶* "55"
```

At this point we have found our answer `''55''` and we simply return it.

As tracing through the evaluation shows in each recursive call, we install another exception handler. If a recursive call raises an exception, then it is handled by the innermost handler and propagated up.

Exception handling provides a way of transferring control and information from some point in the execution of a program to a handler associated with a point previously passed by the execution (in other words, exception handling transfers control up the call stack).

We want to emphasize that using exceptions to transfer controll is not unique to OCaml nor is it unique to functional langauages. It exists in many languages such as C++, Java, JavaScript, etc. and is always a way to transfer controll.