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`.

```
1 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`.

```
1 (* Invariant: power:int >=0 *)
2
3 let rec power(n, k)=
4   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 `power`$(\overline{n}, \overline{k})$ evaluates in multiple steps to the value $\overline{n^k}$.

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

Proof. By induction on k.

Base Case $k = 0$

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

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

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

□

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

Proof. By complete induction on k.

Base Case $k = 0$

$\texttt{power}(\overline{n}, \overline{0})$

$\Rightarrow 1$ 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 `power`$(\overline{n}, \overline{k'}) \Downarrow \overline{n^{k'}}$ for any $k' < k$. We have to show that `power`$(\overline{n}, \overline{k}) \Downarrow \overline{n^k}$.

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

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 $k' < k$.

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

□

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.

```
1  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.

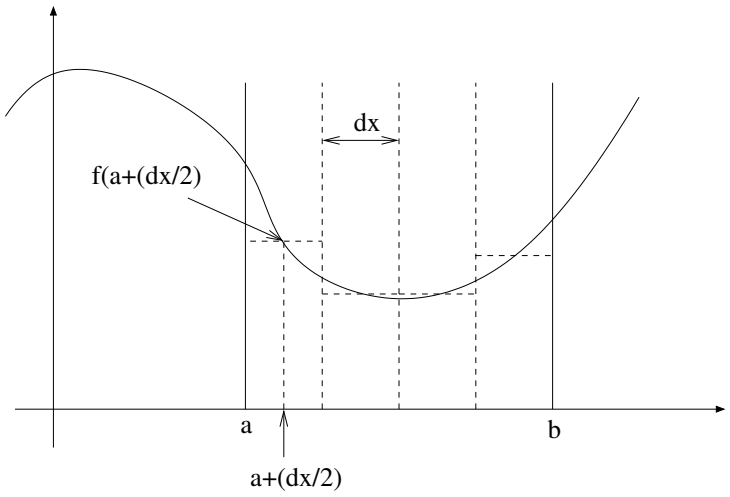


Figure 3.1: Integral between a and b

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`.

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

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`.

$$\texttt{funkyPlus 3 = (fun x -> fun y -> x * x + y) 3} \Longrightarrow \texttt{fun y -> 3 * 3 + y}$$

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:

```
1 let test () =
2   let pi   = 3.14 in                                (* 1 *)
3   let area = (fun r -> pi *. r *. r) in             (* 2 *)
4   let a2   = area (2.0) in                          (* 3 *)
5   let pi   = 6.0  in                                (* 4 *)
6   let a3 = area 2.0 in
7     print_string ("Area a2 = " ^ string_of_float a2 ^ "\n");
8     print_string ("Area a3 = " ^ string_of_float a3 ^ "\n")
9 ;
```

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:

```
1     let pi    = 6.0   in                                  (* 4 *)
2         ^^
3 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:

```
1 let r = ref 0
2 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
```

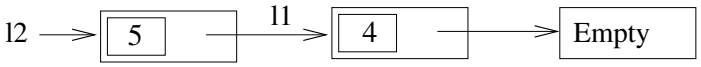
Then we can define a circular list as follows:

```
# 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;;
```

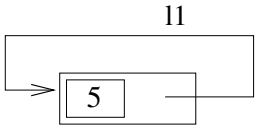
How does the list `l1` and the list `l2` look like?

Before the assignment

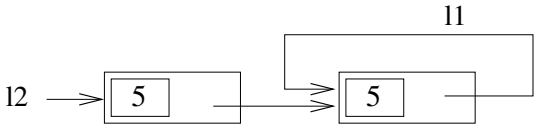


After the assignment

l1 := !l2



This is what l1 looks like...



This is what l2 looks 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`.

```
find l 55
⟶* try find ll 55 with NotFound -> find lr 55
⟶* try (try find Empty 55 with NotFound -> find Empty 55)
     with NotFound -> find lr 55
⟶* try (try raise NotFound with NotFound -> find Empty 55)
     with NotFound -> find lr 55
⟶* try find Empty 55
     with NotFound -> find lr 55
⟶* try (raise NotFound)
     with NotFound -> find lr 55
⟶* find lr 55
⟶* try find Empty 55 with NotFound -> find Empty 55
⟶* try (raise NotFound) with NotFound -> find Empty 55
⟶* find Empty 55
⟶* 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:

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

eventually steps to

```
1 ⟶* try (raise NotFound) with NotFound -> find r 55
2 ⟶* find r 55
3 ⟶* try  find (Node (Empty, (55, "55"), Empty)) 55
4      with NotFound -> find Empty 55
5 ⟶* try "55"
6      with NotFound -> find Empty 55
7 ⟶* "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 excpetion, 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.

Chapter 6

Modules

When it comes to controlling the complexity of developing and, more importantly, maintaining a large system, modularity is key.

Modules make large programs compilable by allowing to split them into pieces that can be separately compiled. They make large programs understandable by adding structure to them. More precisely, modules encourage, and sometimes force, the specification of the links (interfaces) between program components, hence they also make large programs maintainable and reusable. Additionally, by enforcing abstraction, modules usually make programs safer; it allows us to isolate bugs and correctness properties.

6.1 Basic idea behind modules and module types in OCaml

A primary motivation for modules is to package together related definitions (such as the definitions of a data type and associated operations over that type) and enforce a consistent naming scheme for these definitions. This avoids running out of names or accidentally confusing names (name space separation). Such a package is called a *structure* and is introduced by the `struct end` construct, which contains an arbitrary sequence of definitions. The structure is usually given a name with the module binding. In OCaml, we must name structures using `module`.

As an example, let us consider the definition of a stack.

```
1 module Stack  =
2   struct
3     type stack = int list
4     let empty () : stack = []
5     let push (i:int) (s:stack) = i::s
```



```
    let is_empty (s:stack) = match s with
       | [] -> true
       | _::_ -> false
    let pop (s:stack) = match s with
       | [] -> None
       | _::t -> Some t
    let top (s:stack) = match s with
       | [] -> None
       | h::_ -> Some h
    let stack2list(s:stack) = s
  end
```

To access functions defined in a module, we prefix the functions with the module name.

```
# Stack.empty ();;
- : Stack.stack = []
# let s = Stack.empty ();;
val s : Stack.stack = []
# let s0 = Stack.push 0 s;;
val s0 : int list = [0]
# let s1 = Stack.push 1 s0;;
val s1 : int list = [1; 0]
# Stack.top s1;;
- : int option = Some 1
# Stack.pop s1;;
- : int list option = Some [0]
#
```

So far, we have achieved a separation of name spaces. For example, we can now define a module for queues which also implements `empty`, `push`, etc. without naming conflicts. However, more importantly, we can ensure that a given module implements an expected functionality by declaring a *module type* (also called *signature*) and stating that the given module implements it.

```
module type STACK =
  sig
    type stack
    val empty : unit -> stack
    val push  : int -> stack -> stack
    val is_empty : stack -> bool
    val pop : stack -> stack option
    val top : stack -> int option
    val size : stack -> int
    val stack2list : stack -> int list
  end
```

Signatures list the names of abstract types together with the type of functions we expect. We may leave the concrete implementation of a type opaque. For example, we declare that any module with signature `STACK` must provide some type `stack`,

but we do not expose or restrict how stacks are in fact implemented. We can then subsequently declare that a given module implements the signature as follows.

```
module Stack : STACK  =
  struct
    type stack = int list
    let empty () : stack = []
    let push (i:int) (s:stack) = i::s
    let is_empty (s:stack) =  match s with
      | [] -> true
      | _::_ -> false

    ...

  end
```

When does a module (structure) implement a module type (signature) ? - Roughly speaking, the module must provide *all the components* and satisfy *all the type definitions* required by the module type. More generally, the following holds.

- *Minimize bureaucracy*: A module may provide more components than are strictly required by the module type. For example, we have defined a function `length` which we do not expose to the outside.
- *Enhance reuse*: A module may provide values with *more general* types than are required by the module type. For example, our signature demands a function `is_empty` of type `stack -> bool`. Our implementation of `is_empty` in fact has the more general type `'a list -> bool`.
- *Increase flexibility*: A module may consist of declarations presented in any sensible order, not just the order specified in the signature, provided the requirements of the specification are met.
- *Avoid over-specification*: A datatype may be provided where a type is required; similarly a value constructor may be provided where a value is required.

6.2 Keeping things abstract ...

In our module type STACK we fixed what elements we are storing in the stack although we were not specific how a stack is actually implemented. We might want to keep the element type abstract - how can we achieve this? - We simply introduce a type `type el` and generalize the functions `push`, `pop`, `pop`, and `stack2list`.


```
module type STACK =
  sig
    type stack                          (* keeping implementation of stack
    abstract *)
    type el                                (* keeping elements abstract *)
    val empty : unit -> stack
    val is_empty : stack -> bool
    val pop : stack -> stack option
    val push  : el -> stack -> stack
    val top : stack -> el option
    val size : stack -> int
    val stack2list : stack -> el list
  end
```

We can now still enforce that we are producing a stack containing integers by binding `el` to `int` when we declare that the module `IntStack` has module type `STACK with type el = int`.

```
module IntStack :  (STACK with type el = int) =
  struct
    type el = int
    type stack = int list


    ...

   end
```

6.3 Using abbreviations for module names

Calling functions declared in a module is done by prefixing the function name with the module name. This might however lead to long identifiers. It is hence often useful to introduce shorter module names using abbreviations.

```
module IS = IntStack
module FS = FloatStack
```

We can now write `IS.empty` and `IS.push` as well as `FS.empty` and `FS.push`.

6.4 Careful with `open`

Another to reduce clutter in the program, is to *open* a module to incorporate its bindings directly into the current environment.

```
open IntStack
```


This will incorporate and expose the body of the structure `IntStack` into the current environment so that we may simply write `empty`, `push`, etc. without qualification. Although this is surely convenient, using `open` can be dangerous and have unforeseen consequences. By using open, we are erasing any structure and name space separation. For example, when simultaneously opening two structures that have a component with the same name the functions declared in the second module will overshadow the functions from the first one. We typically do not allow dynamic dispatching on the type of functions to figure out which `push` you actually meant.

```
open IntStack
open FloatStack
```

Also note that by opening a module we are not only exposing the functions declared in the signature, i.e. its type; we are exposing also any helper functions which are now part of the current environment. This turns out to be a source of many bugs; it is best to use `open` sparingly.

6.5 Using modules to model currency and bank-client operations

To illustrate the power of abstraction and information hiding using modules, we consider here implementing a bank account in a given currency.

6.5.1 Modelling different currencies

We begin defining the `module type CURRENCY`. It defines a signature declaring an abstract type `t` describing our currency together with operations on them; we want to be able to add and multiply two values and convert them to string to display them back to the user.

```
module type CURRENCY =
sig
  type t
  val unit : t
  val plus : t -> t -> t
  val prod : float -> t -> t
  val toString : t -> string
end
```

We want to define currencis using floating point numbers. We therefore declare a module `Float` which declares `t` to be a `float` and provides the appropriate operations.

```
module Float =
struct
```



```
3   type t = float
4   let unit = 1.0
5   let plus = (+.)
6   let prod = ( *. )
7   let toString  x = string_of_float x
8 end;;
```

Note, we did not say directly when we declared the module `Float` that it matches the signature asked for by `CURRENCY`. The reason is that we want to be able to re-use the module `Float` for different currencies!

```
1 module Euro = (Float : CURRENCY);;
2 module USD  = (Float : CURRENCY);;
3 module CAD  = (Float : CURRENCY);;
```

We define three different module names here, `Euro`, `USD`, `CAD`; each of these modules provides the operations defined in `Float`. Declaring here `Float : CURRENCY` hides the fact that our currencies are implemented as floats. For the user of the module `Euro` how the currency is implemented is not visible nor is it accessible. Operations such as `plus` and `mult` become external operation; more importantly, in the module `Euro` these operations work on `Euro.t` only. Similarly, in the module `CAD` these operations work on `CAD.t` only! As a consequence, we cannot directly add Canadian dollars to Euros! There is no operation for it.

Abstraction here produces *two isomorphic but incompatible views of a same structure* `Float`. For instance, all currencies are represented by floats; however, all currencies are certainly not equivalent and should not be mixed. Currencies are isomorphic but disjoint structures, with respective incompatible units Euro and Dollar.

To illustrate:

```
1
2 # let euro x = Euro.prod x Euro.unit;;
3 val euro : float -> Euro.t = <fun>
4 # let x = Euro.plus (euro 10.0) (euro 20.0);;
5 val x : Euro.t = <abstr>
6 # Euro.toString x;;
7 - : string = "30."
8 # Euro.toString (Euro.plus (euro 10.0) (Dollar.unit));;
9 Characters 37-50:
10   Euro.toString (Euro.plus (euro 10.0) (Dollar.unit));;
11                                        ^^^^^^^^^^^^^
12 Error: This expression has type Dollar.t
13        but an expression was expected of type Euro.t
14 #
```

Note that keeping the currencies separate is a good thing which is enforced statically by the type system.

6.5.2 Modelling client and bank operations

Let us now define a bank account; we would like to enforce that while the clients can deposit and retrieve money from the account which is created in a given currency, only the bank has the ability to create the actual account.

```
module type CLIENT = (* client's view *)
sig
  type t
  type currency
  val deposit : t -> currency -> currency
  val retrieve : t -> currency -> currency
end;;

module type BANK = (* banker's view *)
sig
  include CLIENT
  val create : unit -> t
end;;
```

We use the keyword `include` to include all the declarations in the module type `CLIENT` in the module type `BANK`. We can say the signature `BANK` inherits all the declarations from `CLIENT`. In addition, `BANK` provides the functionality to create an account.

Note that in OCAML, there is no overloading - if you have a declaration with name `foo` in `CLIENT` and you are now declaring `foo` also explicitely in `BANK`, you are simply overshadowing the previous declaration given in `CLIENT`.

Let's now provide an implementation for `BANK` which is parameterized by the currency we want to use; this means we can obtain a branch of the bank using `Euro` and other branch using `CAD`.

```
module Old_Bank (M : CURRENCY) : (BANK with type currency = M.t) =
struct
  type currency = M.t
  type t = { mutable balance : currency }
  (* record type with one mutable field called balance which can be updated
    *)

  let zero = M.prod 0.0 M.unit
  and neg = M.prod (-1.0)

  let create() = { balance = zero }

  let deposit c x =
    if x > zero then
      c.balance <- M.plus c.balance x;
      c.balance

  let retrieve c x =
    if c.balance > x then
```



```
      deposit c (neg x)
    else
      c.balance
end;;
```

We can now define different banks, each with their own operations to retrieve and deposit money.

```
(* Illustrating inheritance and code reuse *)
module Post = Old_Bank (Euro);;

module Post_Client : (CLIENT with type currency = Post.currency
                            and type t = Post.t) = Post;;
```

We also defined a module `Post_Client`, which describes a client who as a Post account. Note, that we rely on information hiding again - while the module `Post` provides also a way to create an account the signature `CLIENT` which we say `Post_client` satisfies hides this information. As a consequence, the operation `Post_client.create` does not exist.

This proposed model for banks and clients is fragile because all information lies in the account itself. For instance, if the client loses his account, he loses his money as well, since the bank does not keep any record.

However, the example already illustrates some interesting benefits of modularity: the clients and the banker have different views of the bank account. As a result an account can be created by the bank and used for deposit by both the bank and the client, but the client cannot create new accounts.

```
# let my_account = Post.create ();;
# Post.deposit my_account (euro 100.0);;
# Client.deposit my_account (euro 100.0);;
```

Moreover, several accounts can be created in different currencies, with no possibility to mix one with another, such mistakes being detected by typechecking.

```
# module Citybank = Old_Bank (Dollar);;
# let my_dollar_account = Citybank.create();;

# Citybank.deposit my_account;;
# Citybank.deposit my_dollar_account (euro 100.0);;
```

6.5.3 A more realistic model for bank accounts

In our previous implementation, we identified a bank account with a name which described a reference in memory containing the current balance. A more robust and more realistic implementation of a bank would maintain a database where the client is only given an account number to access his or her account. Luckily, the implementation of the bank can be changed while preserving its signature.

```
module Bank (M : CURRENCY) : (BANK with type currency = M.t) =
struct
  let zero = M.prod 0.0 M.unit
  and neg = M.prod (-1.0)

  type t = int
  type currency = M.t
  type account = { number : int; mutable balance : currency } (* bank
    database *)

  let all_accounts = Hashtbl.create 10
  and last = ref 0

  let account n = Hashtbl.find all_accounts n

  let create() =
    let n = incr last; !last
    in
    Hashtbl.add all_accounts n {number = n; balance = zero};
    n

  let deposit n x =
    let c = account n in
    if x > zero then
      c.balance <- M.plus c.balance x;
      c.balance

  let retrieve n x =
    let c = account n in
    if c.balance > x then (c.balance <- M.plus c.balance x; x) else zero

end;;
```

Using functor application we can create several banks and they will have independent and private databases, as desired although their abatract types are the same.

```
module Central_Bank = Bank (CAD);;
module TD_Canada = Bank (CAD);;
module DeutscheBank = Bank (Euero)
```

Furthermore, since the two modules `Old_bank` and `Bank` have the same signature, one can be used instead of the other, so as to created banks running on different models.

```
module Old_post = Old_Bank(Euro)
module Post = Bank(Euro)
module Citybank = Bank(Dollar);;
```

All banks have the same signature, however they were built. In fact, it happens to be the case that the user cannot even observe the difference between either implementation; however, this might not be true in general. Indeed, such a property

cannot be enforced by the typechecker.

Chapter 7

Fun with functions: Continuations

Generally speaking, *a continuation is a representation of the execution state of a program (for example, a call stack) at a certain point in time*. Many languages have constructs that allow a programmer to *save the current execution state into an object*, and then *restore the state from this object* at a later point in time (thereby resuming its execution). One can distinguish between *first-class continuations* and *functions as continuations*. In these notes we only talk about the latter. Thus we do not refer to an extension of the language, but a particular programming technique based on higher-order functions.

7.1 A tail-recursive append function

Let us consider the function `append: 'a list -> 'a list -> 'a list` that appends two lists.

```
let rec append l k = match l with
  | [] -> k
  | h::t -> h::append t k
```

This function is not tail-recursive, since it applies the list constructor `::` to the result of the recursive call `append t k`. Nonetheless, this function is quite efficient and the definition above is fully satisfactory from a pragmatic point of view.

But one may still ask, if there is a way to write an append function in tail-recursive form. The answer is "yes". In fact, it is a deep property of ML that *every* function can be rewritten in tail-recursive form!

Suppose we have a function `f:'a -> 'b` and would like to rewrite it as a function `f'` in tail-recursive form. The basic idea is to give f' an additional argument called a *continuation*, which represents the computation that should be done on the result of `f`. In the base case, instead of returning a result, we call the continuation. This means that `f'` should have the type


```
f': 'a -> ('c -> 'b) -> 'b
```

In the recursive case, we add whatever computation should be done on the result to the continuation. So a continuation is like a functional accumulator! Or to put it differently, it will be a stack of functions.

When we use this function to compute `f` we give it the *initial continuation* which is often the identity function, indicating no further computation is done on the result.

Applying this basic idea, we implement a function `app_tr` of type `'a list -> 'a list -> ('a list -> 'b) -> 'b`. The corresponding program is below:

```
let rec app_tr l k =
  let rec app' l k c = match l with
    | []   -> c k
    | h::t -> app' t k (fun r -> c (h::r))
  in
    app' l k  (fun r -> r)
```

The first line of the function `app'` implements the idea that instead of returning the result `k` we apply the continuation `c` to `k`. The second line implements the idea that instead of constructing

```
h::append(l,t)
```

we call `app_tr` recursively on `l` and `k`, adding the task of prepending `x` to the argument `r` of the continuation `cont`. To illustrate how the functional accumulator is built during the recursive calls, consider the following sample computation.

```
      app' [1;2] [3;4] (fun r -> r)
⟹    app' [2] [3;4] (fun r1 -> (fun r -> r) (1::r1))
⟹    app' [] [3;4] (fun r2 -> (fun r1 -> (fun r -> r) (1::r1)) (2::r2))   (*)
⟹    (fun r2 -> (fun r1 -> (fun r -> r) (1::r1)) (2::r2)) [3;4]
⟹    (fun r1 -> (fun r -> r) (1::r1)) (2::[3;4]))
⟹    (fun r -> r) (1::2::[3;4])
⟹    (1::2::[3;4])
```

This example illustrate how we build up a stack of functions as an accumulator, which keeps track of the computation we still need to do. Although the function is now tail-recursive, we have not obtained a program with better performance. Both programs need to remember deferred operations. The difference is who is in control of the call stack and where is it stored. In the original program `append` saves the deferred operations on the run-time stack. This is done by the OCaml interpreter. The program `app_tr` saves the deferred operations in the closures of continuations. The programmer builds up the stack herself and is in control of it. So, while it is true that `app_tr` has in principal a smaller run-time stack, it will still need to store the closures of continuations.

The benefit of continuations is hence not always in gaining efficiency, but they provide us with a direct handle on future computation.

In the case of OCaml one can in fact observe the difference between the built-in append function which is not tail-recursive and the tail-recursive one we implemented.

The built-in function performs better on small to medium-sized lists. However, it may run out of memory when run on large lists. The tail-recursive version of append performs slower on small to medium-sized lists; however it will actually succeed, i.e. running at all, on long lists.

Let us first generate some large lists:

```
let rec genList n acc = if n > 0 then genList (n-1) (n::acc) else acc;;

let l1 = genList 800000 [];;
let l2 = genList 400000 [];;
```

Then executing the following expressions in OCaml demonstrates this:

```
# append l1 l2;;
Stack overflow during evaluation (looping recursion?).

app_tr l1 l2;;
- : int list =
[1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11; 12; 13; 14; 15; 16; 17; 18; 19; 20; 21;
 22; 23; 24; 25; 26; 27; 28; 29; 30; 31; 32; 33; 34; 35; 36; 37; 38; 39; 40;
 41; 42; 43; 44; 45; 46; 47; 48; 49; 50; 51; 52; 53; 54; 55; 56; 57; 58; 59;
 60; 61; 62; 63; 64; 65; 66; 67; 68; 69; 70; 71; 72; 73; 74; 75; 76; 77; 78;
 79; 80; 81; 82; 83; 84; 85; 86; 87; 88; 89; 90; 91; 92; 93; 94; 95; 96; 97;
 98; 99; 100; 101; 102; 103; 104; 105; 106; 107; 108; 109; 110; 111; 112;
 113; 114; 115; 116; 117; 118; 119; 120; 121; 122; 123; 124; 125; 126; 127;
 128; 129; 130; 131; 132; 133; 134; 135; 136; 137; 138; 139; 140; 141; 142;
 143; 144; 145; 146; 147; 148; 149; 150; 151; 152; 153; 154; 155; 156; 157;
 158; 159; 160; 161; 162; 163; 164; 165; 166; 167; 168; 169; 170; 171; 172;
 173; 174; 175; 176; 177; 178; 179; 180; 181; 182; 183; 184; 185; 186; 187;
 188; 189; 190; 191; 192; 193; 194; 195; 196; 197; 198; 199; 200; 201; 202;
 203; 204; 205; 206; 207; 208; 209; 210; 211; 212; 213; 214; 215; 216; 217;
 218; 219; 220; 221; 222; 223; 224; 225; 226; 227; 228; 229; 230; 231; 232;
 233; 234; 235; 236; 237; 238; 239; 240; 241; 242; 243; 244; 245; 246; 247;
 248; 249; 250; 251; 252; 253; 254; 255; 256; 257; 258; 259; 260; 261; 262;
 263; 264; 265; 266; 267; 268; 269; 270; 271; 272; 273; 274; 275; 276; 277;
 278; 279; 280; 281; 282; 283; 284; 285; 286; 287; 288; 289; 290; 291; 292;
 293; 294; 295; 296; 297; 298; 299; ...]
```

Hence, depending on your application you might choose to implement even functions provided from the basic library using tail-recursion.

7.2 Control with continuations

In the case of appending two lists continuations can be applied, but they do not necessarily help us in writing simpler or more efficient code. However, in functions with more complex control flow, they can often be used to our advantage. Here we will discuss two examples.

7.2.1 Finding element in a tree

The first example demonstrates a continuation-style implementation for finding an element `d` in a binary tree `T` which satisfies some property `p`. If such an element `d` exists, then we return `SOME(d)`, otherwise we return `NONE`. This problem can be solved straightforwardly as follows by implementing a function `find` of type `('a -> bool) -> 'a tree -> 'a option`.

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

let rec find p t = match t with
  | Empty -> None
  | Node (l, d, r) ->
   if  (p d) then Some d
   else (match find p l with
     | None -> find p r
     | Some d' -> Some d')
```

However, this solution is slightly unsatisfactory, since in every recursion we check, if we have found an element in the left tree (see recursive call `find(p,L)`). If we actually did find an element, we pass this information `Some d'` back and must propagate this information up. This seems unnecessary, because once we found an element `d` which satisfies the property `p`, we would like to simply return. Previously, we have seen how to rewrite this program using exceptions.

```
exception Fail
let rec find_exc p t = match t with
  | Empty -> raise Fail
  | Node (l,d,r) ->
    if (p d) then Some d
    else (try find_exc p l with Fail -> find_exc p r)

let find_ex p t =  (try find_exc p t with Fail -> None)
```

We will now rewrite this function using continuations. This will allow the programmer to be in control of the call stack and avoids using exceptions. We write

a function `find_tr:` which has type `('a -> bool) -> 'a tree -> (unit -> 'a option) -> 'a option`. It takes in three arguments: the function `p` of type `'a -> bool`, the tree `T` itself which we must traverse and the continuation `cont` of type `unit -> 'a option`. The continuation plays the role of a *failure continuation*, since it keeps track of what to do when we have not found an element `d` which satisfies `p`. The failure exception `Fail` played a similar role in the above program `find_exc`.

```
1 let rec find_tr p t cont = match t with
2   | Empty -> cont ()
3   | Node(l, d, r) ->
4     if (p d) then Some d                                    (* 1 *)
5     else find_tr p l (fun () -> find_tr p r cont)
6
7 let find' p t = find_tr p t (fun () -> None)                (* 2 *)
8
```

As mentioned, the continuation `cont` keeps track of the work we still need to do, i.e. it keeps track of the tree we still need to traverse. If we have reached a leaf (i.e. `t = Empty`), then we simply call the continuation. As a consequence, we will now consult our closure stack of continuations and continue with whatever is on top of the stack.

If we reach a node `Node (l, d, r)` and `d` does not satisfy the given property `p` (see line `(* 1 *)`), then we proceed to look for an element in the left sub-tree, but we must put the work which still needs to be done, i.e. `find_tr p r cont` on the continuation. If we found an element `d` which satisfies the property `p` (see line `(* 1 *)`) we simply return the result `Some d` and throw away the continuation.

We kick off the function `find_tr` with the initial continuation `fun () -> None`.

7.2.2 Regular expression matcher

Next, we discuss the implementation of a simple regular expression matcher. Regular expression matching is a very useful technique for describing commonly occurring patterns. For example, the unix shell provides a mechanism for describing a collection of files by patterns as `*.ml` or `hw[1-3].ml`. The emacs editor provides an even richer language for regular expressions.

Typically, the patterns can be

- Singleton : matching a specific character
- Alternation: choice between two patterns
- Concatenation: succession of patterns
- Iteration : indefinite repetition of patterns

Note that regular expressions provide no concept of nesting of one pattern inside another. For this we require a richer formalism, namely context-free language. We can describe regular expressions inductively:

$$\text{regular expression} r \quad ::= \quad a \mid r_1\ r_2 \mid 0 \mid r_1 + r_2 \mid 1 \mid r^*$$

where a represents a single character from an alphabet. We say a string s matches a regular expression r, iff s is in the set of terms described by r. So for example:

- $a(p^*)l(e+y)$ would match $apple$ or $apply$.
- $g(1+r)(e+a)y$ would match $grey$, $gray$ or gay.
- $g(1+o)^*(gle)$ would match $google$, $gogle$, $goooogle$ or $ggle$.
- $b(ob0+oba)$ would match $boba$ but would not succeed on bob.

In general, we can describe a simple algorithm for a regular expression matcher as follows:

- s never matches 0
- s matches 1 iff s is empty.
- s matches a iff $s = a$.
- s matches $r_1 + r_2$ iff either s matches r_1 or r_2
- s matches $r_1\ r_2$ iff $s = s_1\ s_2$ where s_1 matches r_1 and s_2 matches r_2.
- s matches r^* iff either s is empty or $s = s_1\ s_2$ where s_1 matches r and s_2 matches r^*.

To implement regular expression matcher, we first need to define our regular expressions. We can do this straightforwardly by the following datatype definition:

```
1  type regexp =
2  Char of char | Times of regexp * regexp | One | Zero |
3  Plus of regexp * regexp | Star of regexp
```

Next, we will write a function `acc` which takes a regular expression `r`, a character list `s`, and a *continuation* `k`, and yields a boolean value. Informally, the continuation determines how to proceed once an initial segment of the given character list has been determined to match the given regular expression. The remaining character list is passed to the continuation to compute the final result. Because the continuation

is called upon successfully matching an initial segment of characters, it is called a *success continuation*.

The type of the function `acc r s k` is

```
regexp -> char list -> (char list -> bool) -> bool
```

where `r` is a regular expression, `s` is a list of characters, and `k` is a success continuation which when given a list of characters continues with the next task on its stack. The final answer of the function `acc` is a boolean. It returns true, if a given list of characters `s` matches the given regular expression `r` and returns false otherwise.

```
let rec acc r clist k = match r , clist with
  | Char c         , []    -> false
  | Char c         , c1::s -> (c = c1) && (k s)
  | Times(r1, r2) , s     -> acc r1 s (fun s' -> acc r2 s' k)
  | One            , s     -> k s
  | Plus(r1, r2)  , s     -> acc r1 s k || acc r2 s k
  | Zero           , s     -> false
  | Star r         , s     ->
    (k s) || acc r s (fun s' -> not(s = s') && acc (Star r) s' k)
```

Note that the comparison `not (s = s')` inside the continuation in the last line is necessary, since when given `Star One` it would otherwise loop.

What is the initial continuation with which we should call `acc`? We succeed when the remaining character string is empty, i.e. we have exhausted our input string. But we must fail if we reach the end of our regular expression and our list of characters is not empty, i.e. there are some unmatched characters left. Hence, the initial continuation must test this, i.e. `(fun l -> l = [])`. The function `accept` has type `regexp * string -> bool`

```
let accept r s = acc r (string_explode s) (fun l -> l = []) ;
```

One final remark: out top-level matcher `accept` takes in as input a regular expression describing the pattern `r` and a string `s` for which we must decide whether it is accepted by the pattern `r`. To easily process the string sequentially, we "explode" the string `s` into a list of characters.

Chapter 8

Sometimes it's good to be lazy

Typically, languages such as OCaml (SML, Lisp or Scheme) evaluate expressions by a *call-by-value* discipline. All variables in OCaml are bound *by value*, which means that variables are bound to *fully evaluated* expressions. For example consider let-expressions: In OCaml, we would write `let x = e1 in e2`. How would this expression be evaluated? We first evaluate `e1` to some value `v1` and then bind the name of the bound variable `x` to the value `v1` before we continue evaluating `e2` in an environment where we have established the binding between `x` and `v1`.

A similar evaluation strategy is applied when evaluating function applications. We first evaluate the argument before it is passed to the function. For example, in the expression `(fun x -> e) e1` we first evaluate the expression `e1` to some value `v1` before we pass the value `v1` to the parameter `x` of the function effectively binding the name `x` to the value `v1`

According to a *call-by-value* strategy, expressions are evaluated and their values bound to variables, no matter if this variable is ever needed to complete execution. For example:

```
1 let  x = horribleComp(345) in  5
```

In this simple example, we will evaluate `horribleComp(345)` to some value `v1` and establish the binding between the variable `x` and `v1`, although it is clearly not needed. For this reason, languages whose evaluation is call-by-value are also called *eager*.

Alternatively, we could adapt a *call-by-name* strategy. This means variables can be bound to *unevaluated expressions*, i.e. *computation*. In the previous example, we would *suspend* the computation of `horribleComp(345)`, until the value for `x` is required by some operation. For example:

```
1 let x = horribleComp(345) in x + x
```

In this example, it is necessary to evaluate the binding for `x` in order to perform the addition `x + x`. Languages that adopt the *call-by-name* discipline are also called

lazy languages, because they delay the actual evaluation until it is required. We also often say evaluation for `horribleComp(345)` is *suspended*, and only *forced* when required.

One important aspect of lazy evaluation is *memoization*. Let us reconsider the previous example. Since the variable `x` occurs twice in the expression `x + x`, it is natural to ask if the expression `horribleComp(345)` is now evaluated twice. If this would be the case, then being lazy would have backfired! In reality, lazy evaluation adopts a refinement of the *call-by-name* principle, called *call-by-need* principle. According to the *call-by-need* principle, variables are bound to unevaluated expressions, and are evaluated only as often as the value of that variable's binding is required to complete the computation. So in the previous example, the binding of `x` is needed twice in order to evaluate the expression `x + x`. According to the *call-by-need* principle, we only evaluate once `horribleComp(345)` to a value `v1`, and then we *memoize* this value and associate the binding for `x` to the value `v1`. In other words, the by-need principle says the binding of a variable is evaluated *at most once*, not at all, if it is never needed, and exactly once, if it is ever needed. Once a computation is evaluated, its value is saved for the future. This is called memoization.

The main benefit of lazy evaluation is that it supports *demand-driven* computation. We only compute a value, if it is really demanded somewhere else. This is particularly useful when we have to deal with *online data structures*, i.e. data-structures that are only created insofar as we examine them.

- Infinite data structures: For example, if we would want to represent *all* prime numbers this cannot be done eagerly. We cannot ever finish creating them all, but we can create as many as we need for a given run of a program!

- Interactive data structures such as sequences of inputs. Users inputs are not predetermined at the start of the execution but rather created on demand in response to the progress of computation up to a given point.

Surprisingly we can model lazy programming with the tools we have seen so far, namely functions and records.

How do we prevent the eager evaluation of an expression? - The key idea is to *suspend computation using functions*. Since we never evaluate inside the body of a function, we can suspend the computation of `3+7` by wrapping it in a function, i.e. `fun () -> 3 + 7`. In general, we will use functions of type `unit -> 'a` to describe the suspended computation of type `'a`. To highlight the fact that a function `unit -> 'a` describes a suspended computation, we tag such suspended computations with the constructor `Susp`.

```
1 type 'a susp = Susp of (unit -> 'a)
```


We can then delay computation labelling it with the constructor `Susp` using the function `delay` and force the evaluation of suspended computation using the function `force`.

```
(* force:  'a susp → 'a *)
let force (Susp f) = f ()
```

We force a suspended computation by applying `f` which has type `unit -> 'a` to `unit`. For example, the following evaluates using call-by-name.

```
let  x = Susp(fun () -> horribleComp(345)) in  force x + force x
```

Only when we force `x` will be actually compute and evaluate `horribleComp(345)`.

8.1 Defining infinite objects via observations

Finite structures such as natural numbers or finite lists are modelled by (inductive) data types. For example, the data type `'a list` given below

```
type 'a list = Nil | Cons of 'a * 'a list
```

encodes the following inductive definition of finite lists:

- `Nil` is a list of type `'a list`.
- If `h` is of type `'a` and `t` is a list of type `'a list`, then `Cons (h,t)` is a list of type `'a list`.

The inductive definition defines how we can construct finite lists from "smaller" lists. In mathematical terminology, the given inductive definition defines a least fix point. We can manipulate and analyze lists (or other inductively defined data) via pattern matching. Key to writing terminating programs is that our recursive call is made on the "smaller" list.

How can we model infinite structures such as streams or processes? - Instead of saying how to construct such data, we define *infinite objects via the observations* we can make about them. While finite objects are intensional, i.e. two objects are the same if they have the same structure, infinite objects are extensional, i.e. they are the same if we can observe them to have the same behavior. We already encountered infinite objects with extensional behavior, namely functions. Given the two following functions:

```
let f x = (x + 5) * 2
let g x = 2*x + 10
```


we can observe that for any input `n` given to `g` the result will be same as `f n`. We cannot however pattern match on the body of the function `f` and `g` analyze their structure/definition. Similarly, when we defined operations on Church numerals we were unable to directly pattern match on the given definition of a Church numeral. Instead, we observed the behavior by applying them.

To summarize, we cannot directly pattern mach on a function; we can only apply the function to an argument, effectively performing an experiment, and observe the result or behavior of the function.

Similarly, we want to define a stream of natural numbers via the observations we can make about it. Given a stream `1 2 3 4 ...`, we may ask for the head of the stream and obtain `1` and we may ask for the tail of the stream obtaining `2 3 4 ...`. Programs which manipulate streams do not terminate - in fact talking about termination of infinite structures does not make sense. However, there is a notion when a program about streams is "good", i.e. we can continue to make observations about streams. We call such programs *productive*. They may run forever, but at every step we can make an observation. This is unlike looping programs which run forever but are not productive.

To define infinite structures via the observations we exploit the idea of delaying the evaluation of an expression. For example, we can define elements of type `'a str` (read as a stream of elements of type `'a`) via two observations: `hd` and `tl`. Asking for the head of a stream using the observation `hd` returns an element of type `'a`. Asking for the tail of a stream using the observation `tl` returns a suspended stream. It is key that the stream is suspended here, since we want to process streams on demand. We will only unroll the stream further, when we ask for it, i.e. we force it.

```
1 type 'a str = {hd: 'a  ; tl : ('a str) susp}
```

8.2 Create a stream of 1's

Now we are in the position to create an infinite stream of one's for example, i.e. `1 1 1 ...`.

```
1 let rec ones = {hd = 1 ; tl = Susp (fun () -> ones)}
```

The head of a stream of one's is simply `1`. The tail of a stream of one's is defined recursively referring back to its definition. We can see here that it is key that the recursive call `str_ones` is inside the function `fun () -> str_ones` thereby preventing the evaluation of `str_ones` eagerly which would result in a looping program.

Maybe even more interestingly, we can now create a stream for the natural numbers as follows. We first define a stream of natural numbers starting from `n`. Hence

the head of such a stream is simply `n`. The tail of such a stream is simply referring back to its definition incrementing `n`.

```
(* numsFrom : int -> int str  *)
let rec numsFrom n =
{hd = n ;
 tl = Susp (fun () -> numsFrom (n+1))}

let nats = numsFrom 0
```

8.3 Map function on streams

How can we write some of the functions we have seen for lists for streams? – Let's start with the map function.

```
(* smap: ('a -> 'b) -> 'a str -> 'b str *)
let rec smap f s =
{ hd = f (s.hd) ;
  tl = Susp (fun () -> smap f (force s.tl))
}
```

8.4 Writing recursive programs about infinite data

It is often useful to inspect a stream up to a certain number of elements. This can be done by the next function:

```
(* int -> 'a str -> 'a list *)
let rec take_str n s = match n with
  | 0 -> []
  | n -> s.hd :: take_str (n-1) (force s.tl)
```

Another example is dropping the first `n` elements of a stream.

```
(* stream_drop: int -> 'a str -> 'a str *)
let rec stream_drop n s = if n = 0 then  s
  else stream_drop (n-1) (force s.tl)
```

8.5 Adding two streams

Next, let us add two streams. This follows the same principles we have seen so far.

```
(* val addStreams : int str -> int str -> int str *)
let rec addStreams s1 s2 =
{hd = s1.hd + s2.hd ;
```



```
 tl = Susp (fun () -> addStreams (force s1.tl) (force s2.tl))
}
```

8.6 Filter

```
(* val filter_str : ('a -> bool) -> 'a str -> 'a str
   val find_hd : ('a -> bool) -> 'a str -> 'a * 'a str susp
*)
let rec filter_str p s =
  let h,t = find_hd p s in
  {hd = h;
   tl = Susp (fun () -> filter_str p (force t))
  }
and find_hd p s =
if  p (s.hd) then (s.hd, s.tl)
  else find_hd p (force s.tl)
```

Note that `find_hd` is not productive. In principle, we could test for a property `p` which is never fulfilled by any element in the stream `s`. We would then continue to call eagerly `find_hd p (force s.tl)`. In fact, productive programs should always suspend their recursive computation. Since `find_hd` is not productive, `filter` is also not necessarily productive.

Using `filter`, we can now define easily a stream of even and odd numbers.

```
let evens = filter_str (fun x -> (x mod 2) = 0) (force (nats.tl))

let odds = filter_str (fun x -> (x mod 2) <> 0) nats
```

8.7 Power and Integrate series

Additional Examples:

```
(* Sequent of 1 3 9 27 .. *)
let rec fseq n k =
  { hd = n ;
    tl = Susp (fun () ->  fseq (n*k) k) }

(* Sequent 1, 1/2, 1/4, 1/8, 1/16 ... *)
let rec geom_series x =
  { hd = (1.0 /. x) ;
    tl = Susp (fun () -> geom_series (x *. 2.0)) }
```

Next we show how to directly define the power series.

$$1,\ 2/3,\ 4/9,\ 8/27,\ \ldots = series(2^{x_i}/3^{x_i})$$


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

let rec power_series x =
{ hd = (float (pow x 2)) /. (float (pow x 3)) ;
  tl = Susp (fun () -> power_series (x+1)) }
```

We can now define how to integrate a series:

```
let integrate_series s =
  let rec aux s n =
    { hd = s.hd /. (float n) ;
      tl = Susp (fun () -> aux (force s.tl) (n + 1) )
    }
  in
  aux s 1
```

8.8 Fibonacci

Let us move on to generating some more interesting infinite sequences of numbers such as the Fibonacci number series. The Fibonacci Sequence is the series of numbers:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...$$

The next number is found by adding up the two numbers before it.

n	$\mathrm{Fib}(n)$	$+$	$\mathrm{Fib}(n+1)$	$\mathrm{Fib}(n+2)$
0	$0, 1, \ldots$		$1, \ldots$	$1, \ldots$
1	$1, 1, \ldots$		$1, \ldots$	$2, \ldots$
2	$1, 2, \ldots$		$2, \ldots$	$3, \ldots$

The stream in the right column, labelled $\mathrm{Fib}(n+1)$, is the tail of the stream described in the left column, labelled $\mathrm{Fib}(n)$. By adding up the two streams pointwise, we should be able to compute the next element in the stream!

$$\begin{array}{rcccccccccccc} \texttt{fibs} & = & 0 & & 1 & & 1 & & 2 & & 3 & & 5 & \ldots \\ & & + & \searrow & + & \searrow & + & \searrow & + & \searrow & + & \searrow & & \\ \texttt{fibs'} & = & 1 & & 1 & & 2 & & 3 & & 5 & & 8 & \ldots \end{array}$$

The key observation is that we need to know the n-th and (n+1)-th element to compute the next element. In terms of streams it means to define the next element in a stream we need to know already not only the head of the stream we are defining, but also the head of the tail of the stream!

We will define the fibonacci series hence mutually recursive as follows:

- The first element of the fibonacci series is `0`.
- The second element (i.e. the head of the tail or the head of `fibs'`) of the fibonacci series is `1`.
- The remaining series (i.e. the tail of the tail or the tail of `fibs'`) is obtained by adding the stream `fibs` and `fibs'`.

This can be implemented using records and suspended functions in OCaml directly.

```
let rec fibs =
{ hd = 0 ;
  tl = Susp (fun () -> fibs') }
and fibs' =
{hd = 1 ;
 tl = Susp (fun () -> addStreams fibs fibs')
}
```

8.9 Sieve of Eratosthenes

A great application of lazy programming and streams is computing a stream of prime numbers given a stream of natural numbers. The sieve of Eratosthenes is a simple, ancient algorithm for finding all prime numbers. It does so by iteratively marking or filtering out the multiples of each prime, starting with the multiples of 2.

The idea can be best described by an example. We begin with a stream s of natural numbers starting from 2, i.e. 2 3 4 5 6 ....

1. The first prime number we encounter is the head of the stream s, i.e. 2 when we have a stream 2 3 4 5 6 ....
2. Given the remaining stream 3 4 5 6 ... (i.e. the tail of s), we first remove all numbers which are dividable by 2 obtaining a stream 3 5 ..... We then start with 1. again.

To find all the prime numbers less than or equal to 30 (and beyond), proceed as follows. First generate a list of integers from 2 to 30 and beyond

2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 ...

The first number in the list is 2; this is the first prime number; cross out every 2nd number in the list after it (by counting up in increments of 2), i.e. all the multiples of 2:

$$3\ 5\ 7\ 9\ 11\ 13\ 15\ 17\ 19\ 21\ 23\ 25\ 27\ 29\ \ldots$$

Next prime number after 2 is 3; now cross out every number which can be divided by 3, i.e. all the multiples of 3:

$$5\ 7\ 11\ 13\ 17\ 19\ 23\ 25\ 29\ \ldots$$

Next prime number is 5; cross out all the multiples of 5:

$$7\ 11\ 13\ 17\ 19\ 23\ 29\ \ldots$$

Next prime number is 7; the next step would be to cross out all multiples of 7; but there are none. In fact, the numbers left not crossed out in the list at this point are all the prime numbers below 30.

Following the idea described, we now implement a function `sieve` which when given a stream `s` of natural numbers generates a stream of prime numbers. The head of the prime number stream is the head of the input stream. The tail of the prime number stream can be computed by filtering out all multiples of `s.hd` from the tail of the input stream.

```
let no_divider m n = not (n mod m = 0)

let rec sieve s =
{ hd = s.hd ;
  tl = Susp (fun () -> sieve (filter_str (no_divider s.hd) (force s.tl)))
}
```

8.10 Lazy finite lists

On demand evaluation is not only necessary when we deal with infinite structures, but it can also be useful when processing structures which can possibly be finite. We show here how to define (possibly) finite lists via observations to allow on demand evaluation. This is accomplished by nesting inductive and coinductive definitions. First, we define `'a lazy_list` coinductively via the observation `hd` and `tl`. The tail of a possibly finite list may be finite, i.e. we have reached the end of the list and the list is empty, or we can continue to make observations, i.e. the list is not empty.

```
type 'a lazy_list = {hd: 'a  ; tl : ('a fin_list) susp}
and 'a fin_list = Empty | NonEmpty of 'a lazy_list
```


We can now re-visit the standard functions for infinite lists and rewrite them for lazy (finite) lists. We show below the implementation of `map` for lazy lists. We split the function into two mutual recursive parts: `map` is a recursive function; if the given list `xs` is `Empty`, then we return `Empty`; if we have not reached the end, we continue to apply lazily `f` to `xs` tagging the result with `NonEmpty`. `map'` applies lazily `f` to a list `s`; this is done by defining what observations we can make about the resulting list given a function `f` and an input list `s`.

```
let rec map f s =
{ hd = f s.hd ;
  tl = Susp (fun () -> map' f (force s.tl))
}

and map' f xs = match xs with
  | Empty -> Empty
  | NonEmpty xs -> NonEmpty (map f xs)
```

8.11 Summary

We demonstrated in this Chapter how to write on-demand programs using observations. This is useful for processing infinite data such as streams but also finite data which may be too large to be processed and generated eagerly. The idea of modelling infinite data via observations (i.e. destructors) is the dual to modelling finite data via constructors. While this view has been explored in category theory and algebra, it has only recently been been picked and incorporated into actual programming practice up by A. Abel and B. Pientka together with their collaborators [**?**].

Chapter 9

Basic Principles of Programming Languages: syntax and semantics of a language

> "Good languages make it easier to establish, verify, and maintain the relationship between code and its properties."
>
> - R. W. Harper

So far we have seen key concepts and programming paradigms such as data-types, higher-order functions, continuations, lazy computation, state and other side-effects, structural induction, type inference, etc. We have studied these ideas using the programming language OCaml and often introduced these concepts by showing different examples. This gives us a good intuition of why these concepts are valuable and how we can use them, but often we need a more solid and more precise theoretical foundation to answer questions such as: What is the meaning of a program? How will it execute and what is its behavior? How can we reason about its execution? - We may also encounter that some expressions may be treated in an unexpected manner by a programming language; knowing about its theoretical foundation, will help explain such seemingly unexpected behavior.

When learning a new language, we often may have even more mundane question such as "What are the legal expressions I can write?" or "What concept of a variable do we have?".

Or we may ask more advanced questions: What are the expressions which will be well-typed? How does OCaml (or other typed languages) infer the type of a given expression? How is the program going to be executed?

Understanding these questions will help us to better understand the language we use and why some programs are not correct or are rejected as ill-typed. In other

words they will help us to debug the program. Understanding these theoretical concepts, will also help us to write well-structured and better code which is more likely to be correct. These notes introduce the theoretical concepts behind programming languages. This will allow us to understand the theory behind programming language design but also enable us to design our own little languages!

The goal is to show you a glimpse into the tools we use to describe programming languages and the reasoning about it. Designing a (programming) language is an art, but just as any creative artist we rely on a whole array of (mathematical) tools and techniques.

In order to talk rigorously about how programs behave and and reason about programs, we begin with a concise and formal specification of a programming language. Such a specification consists of three steps:

1. Grammar of the language (i.e. What are the syntactically legal expressions?)
2. Operational dynamic semantics (i.e. How is a given program executed?)
3. Static semantics (= type system) (i.e. When is a given program well-typed? What can we statically say about the execution of a program without actually executing it?)

We will begin by considering a tiny functional language called Nano-ML, and introduce its grammar and operational semantics. Later, we will also consider its static semantics. We would like to emphasize that this language Nano-ML is not identical to the programming language SML, although it share many of the main ideas. We will use this tiny fictional language to explain some of the underlying theoretical principles and ideas which underly language design. The techniques are general enough that the apply to many real languages such as SML, OCaml, or Haskell, and even object-oriented languages such as Java as well.

9.1 What are syntactically legal expressions?

First, we would like to describe the legal expressions our tiny functional language consists of. Our language will have numbers and some basic arithmetic operations, booleans and if-statements. We will define the syntactically well-formed expressions inductively.

Definition 9.1.1. *The set of expressions is defined inductively by the following clauses*

1. *A number* n *is an expression.*

2. *The booleans* true *and* false *are expressions.*
3. *If* e_1 *and* e_2 *are expressions, then* e_1 *op* e_2 *is an expression where* $op = \{+, =, -, *, <\}$.
4. *If* e, e_1 *and* e_2 *are expressions, then if* e *then* e_1 *else* e_2 *is an expression.*

This inductive definition is often considered too verbose, and a shorter way of describing what expressions are well-formed, is the Backus-Naur Form (BNF). The left side introduces the the symbol for operations op and expressions e, and the right side describes recursively the set of syntactically valid expressions. The vertical bar | indicates a choice.

$$\begin{array}{lll} \text{Operations op} & ::= & + \mid - \mid * \mid < \mid = \\ \text{Expressions } e & ::= & n \mid e_1 \text{ op } e_2 \mid \text{true} \mid \text{false} \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \end{array}$$

This grammar inductively specifies well-formed expressions. To illustrate, we consider some well-formed and some ill-formed expressions.

Examples of well-formed expressions

- $3 + (2 + 4)$
- $\text{true} + (2 + 4)$
- $\text{if } (2 = 0) \text{ then } 5 + 3 \text{ else } 2$
- $\text{if true then } (\text{if false then } 5 \text{ else } 1 + 3) \text{ else } 2 = 5$
- $(\text{if } 0 \text{ then } 55 \text{ else } 77 - 23) = 0$

Examples of ill-formed expressions

- $\text{if true then } 2 \text{ else}$
- -4
- $+23$

Note that the grammar only tells us when expressions are syntactically well-formed. The grammar does not say anything whether a given expression is for example well-typed. In fact, you may notice that the syntactically well-formed expressions (3) and (4) are not well-typed. Similarly, you may argue that -4 is a perfectly sensible expression. This is of course correct, but our grammar does not accept it as a well-formed syntactically expression, since any operation requires two arguments.

9.2 How do we evaluate an expression?

Next, we would like to formally describe how a given expression is going to be executed. This is called *operational semantics*. The meaning (i.e. semantics) of an expression is defined by the value it will evaluate to. In this section, we will define a high-level operational semantics for the tiny language we have seen so far. We begin by defining the evaluation judgment:

$$e \Downarrow v \quad \text{Expression } e \text{ evaluates to a final value } v$$

The first question which comes to mind is: What are final values v? – In the tiny language we have introduced so far, we expect a final value to be either a number n or a boolean, true or false. More formally, we can write:

$$\text{Values } v \quad ::= \quad n \mid \mathsf{true} \mid \mathsf{false}$$

Now we can define recursively how to evaluate an expression by analyzing its structure. If we have already have a value (i.e. a number n or a boolean true or false), then there is nothing to evaluate and we will simply return this value. To evaluate the expression $e_1 + e_2$ we evaluate the sub-expression e_1 to some value v_1 and the sub-expression e_2 to some value v_2. The final value of $e_1 + e_2$ is then obtained by adding the two values v_1 and v_2. In other words, once we have values v_1 and v_2 we rely on our basic primitive operations of arithmetic and add the two values v_1 and v_2. This is written as $\overline{v_1 \text{ op } v_2}$. Similarly, to evaluate the expression $e_1 * e_2$, we evaluate the sub-expression e_1 to some value v_1 and the sub-expression e_2 to some value v_2. The final value of $e_1 * e_2$ is then obtained by multiplying the two values v_1 and v_2. More generally, we evaluate the expression e_1 op e_2 as follows:

- Evaluate sub-expression e_1 to some value v_1
- Evaluate sub-expression e_2 to some value v_2
- Compute the final value $\overline{v_1 \text{ op } v_2}$ by appealing to the corresponding primitive operation.

We will now turn the informal description given above into a formal one using inference rules. In general, an inference rule has the following shape:

$$\frac{\text{premise}_1 \quad \ldots \quad \text{premise}_n}{\text{conclusion}} \text{ name}$$

The part below the line is called conclusion while the parts above the line are called premises. To the right, we often write the name of the rule. We can read

an inference rule as follows: To achieve the conclusion, we must satisfy each of the premises. In other words, if the premises are satisfied, we can conclude the conclusion.

Let's look how we can use this formal notation to describe the evaluation of expressions.

We start by defining the evaluation rules for numbers and booleans. Numbers and booleans are already values, and hence there is nothing to evaluate and we simply return this number. This is described by the rule B-NUM, B-TRUE and B-FALSE. Note that these rules do not have any premises because any value evaluates to itself unconditionally.

$$\frac{}{n \Downarrow n}\ \text{B-NUM} \qquad \frac{}{\mathsf{true} \Downarrow \mathsf{true}}\ \text{B-TRUE} \qquad \frac{}{\mathsf{false} \Downarrow \mathsf{false}}\ \text{B-FALSE} \Downarrow$$

Next, let us reconsider the evaluation of expression e_1 op e_2 and cast the informal recipe given above in a more formal description.

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1\ \mathsf{op}\ e_2 \Downarrow \overline{v_1\ \mathsf{op}\ v_2}}\ \text{B-OP}$$

The rule B-OP says the following: If $e_1 \Downarrow v_1$ (i.e. expression e_1 evaluates to some value v_1") and $e_2 \Downarrow v_2$ (i.e. "expression e_2 evaluates to some value v_2), then e_1 op $e_2 \Downarrow \overline{v_1\ \mathsf{op}\ v_2}$ (i.e. "expression e_1 op e_2 evaluates to $\overline{v_1\ \mathsf{op}\ v_2}$).

Reading the rule B-OP from the bottom to the top, this rules gives a recipe on how to evaluate an expression e_1 op e_2, by recursively evaluating its sub-expressions. The first premise $e_1 \Downarrow v_1$ says "evaluate expression e_1 to some value v_1 and the second premise $e_2 \Downarrow v_2$ says "evaluate expression e_2 to some value v_2. Then compute the final value $\overline{v_1\ \mathsf{op}\ v_2}$ by appealing to the corresponding primitive operation.

Finally, let us consider the if-statement.

$$\frac{e \Downarrow \mathsf{true} \quad e_1 \Downarrow v}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v}\ \text{B-IFTRUE} \qquad \frac{e \Downarrow \mathsf{false} \quad e_2 \Downarrow v}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v}\ \text{B-IFFALSE}$$

There are two rules for evaluating if-expressions. If the guard e evaluates to true, we will evaluate the first branch (rule B-IFTRUE), and if the guard e evaluates to false, we will evaluate the second one (rule B-IFFALSE).

Remark Note that the evaluation rules do not impose an order in which premises need to be evaluated. For example, when evaluating e_1 op e_2, we can first evaluate e_1 and then e_2 or the other way round. The rule just specifies that both subexpressions need to be evaluated. Big-step evaluation provides a high-level description of the operational semantics, and abstracts over the order in which we evaluate subexpressions.

Evaluation How does evaluation work with the inference rules given? – We can evaluate an expression by applying the inference rules and constructing a derivation. Next, we give an example of an evaluation derivation to illustrate the use of the evaluation rules.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{4 \Downarrow 4}\,\text{B-NUM} \quad \dfrac{}{1 \Downarrow 1}\,\text{B-NUM}}{(4-1) \Downarrow 3}\,\text{B-OP} \quad \dfrac{}{6 \Downarrow 6}\,\text{B-NUM}}{((4-1) < 6) \Downarrow \mathsf{true}}\,\text{B-OP} \quad \dfrac{\dfrac{}{3 \Downarrow 3}\,\text{B-NUM} \quad \dfrac{}{2 \Downarrow 2}\,\text{B-NUM}}{3+2 \Downarrow 5}\,\text{B-OP}}{\mathsf{if}\ ((4-1) < 6)\ \mathsf{then}\ 3+2\ \mathsf{else}\ 4 \Downarrow 5}\,\text{B-IFTRUE}$$

Evaluating an expression essentially means to construct such a derivation. However, there are many expressions which do not have a value, for example $\mathsf{true} + 3$ or $\mathsf{if}\ 0\ \mathsf{then}\ 3\ \mathsf{else}\ 4$ do not have a value. So how do we know that some expressions do not yield a value? – The answer is that there exists no derivation tree for such expressions, since there are no rules for evaluating these expressions. For example the derivation for $\mathsf{if}\ 0\ \mathsf{then}\ 3\ \mathsf{else}\ 4$ is stuck because $0 \Downarrow 0$. But the rules for if-expressions require that the guard either evaluates to true or to false. Since there is no rule which specifies what to do when the guard evaluates to a number, evaluation fails. Failure is handled implicitly.

9.3 Implementing a simple evaluator

The declarative description in the previous section can be turned into an executable program in a straightforward way. We demonstrate how to do this in OCaml. Functional languages are particularly suited to prototyping and implementing languages, program transformations, program optimizations, compilers, etc. , because of the power of declaring your own types, pattern matching, and the fact that the functionality is implemented in one place.

We begin with turning our grammar for primitive operators and expressions into data type definitions.

```
type primop = Equals | LessThan | Plus | Minus | Times | Negate

type exp =
  | Bool of bool
  | Int of int
  | If of exp * exp * exp     (* if e then e1 else e2   *)
  | Primop of primop * exp list        (* e1 <op> e2  or  <op> e *)
```

We can now represent our expression

$$\text{if true then } 3 + 2 \text{ else } 1$$

as

```
If (Bool true, Primop (Plus, [Int 3 ; Int 2]), Int 1).
```

Note that our definition of primitive operators is flexible allowing for unary as well as binary operators.

We also need a way to evaluate the primitive operations. To achieve this, we implement a function `evalOp` which takes a primitive operator `op` and a list of values and then falls back to built-in OCaml operations for the primitive operations in our little languages. The function `evalOp` returns a an optional value. In particular, if we call `evalOp` with an operator and do not supply the correct number of arguments we will fail. We also fail, if we try to for example add a boolean to an integer.

A type checker would guarantee that these cases cannot happen during run-time. However, since we have not yet implemented a type checker, we must handle these cases in our evaluator and abort safely.

```
let rec evalOp op arg_list = match op , arg_list with
  | Equals  , [Int i; Int i'] -> Some (Bool (i= i'))
  | LessThan, [Int i; Int i'] -> Some (Bool (i < i'))
  | Plus,     [Int i; Int i'] -> Some (Int (i + i'))
  | Minus,    [Int i; Int i'] -> Some (Int (i - i'))
  | Times,    [Int i; Int i'] -> Some (Int (i * i'))
  | Negate,   [Int i]         -> Some (Int (-i))
  |  _                        -> None
```

Last, we implement the function `eval` which takes an expresion as input and returns an expression that is a value. It follows exactly the description of the operational semantics from the previous section. This is important – we do not invent here anything new. The formal description is our contract with the users of our little language and we adhere to it.

If evaluation of an expression is undefined in our formal description of the operational semantics, we raise an exception `Stuck` in our implementation.


```
  exception Stuck of string

(* eval : exp -> exp *)
 let rec eval e = match e with
    | Int n -> Int n
    | Bool b -> Bool b

  (* primitive arithmetic and boolean operations *)
    | Primop (po, args) ->
      (match evalOp po (Lst.map eval args) with
 | None -> raise (Stuck "Unexpected arguments to primitive operations")
 | Some v -> v)

  (* if expression *)
  | If(e,e1,e2) ->
    (match  eval e with
      | Bool true -> eval e1
      | Bool false -> eval e2
      | _ -> raise (Stuck "Expected boolean value"))

```

9.4 How do we know the definition of evaluation is sensible?

We want our definition of the operational semantics to be sensible - but what does this exactly mean? There are a few easy properties we should have: for example, we should give an evaluation rule for each expression; this property is referred to as coverage. We also might want to make sure that the result of an evaluation is indeed a value - not an arbitrary expression.

For example, for the language we defined, we know that any successful evaluation of an expression will indeed yield a value, i.e. either a number n or a boolean true or false. This property is called *value soundness*.

We also might want to establish that evaluation is deterministic and yields a unique value. This is usually expected in practice.

Coverage For all expressions e there exists an evaluation rule.

Value soundness If $e \Downarrow v_1$ then v_1 is a value.

Determinacy If $e \Downarrow v_1$ and $e \Downarrow v_2$ then $v_1 = v_2$.

Both these properties can be proven by structural induction, but this is beyond this course.

9.5 How can we describe well-typed expressions?

So far, we only have considered syntax (i.e. what expressions are syntactically well-formed?) and operational semantics (i.e. how are we executing an expression?). However, there are many expressions which cannot be evaluated and our interpreter would get stuck. Examples of such expressions include : if 0 then 45 else 33, $0 + \mathsf{true}$, (fn x => $x + 1$) true, etc.

How can we statically check whether an expressions would potentially lead to a runtime error? – We already encountered types in the language OCaml. Types approximate the runtime behavior and provide an effective light-weight tool for reasoning about programs. They allow us to detect errors statically and early on the development cycle. Type checkers give precise error messages pointing the programmer to the specific line in the code where the reasoning went wrong. Ultimately, this means programmers spend more time developing their programs but less time testing the programs; programs are also easier to maintain. More generally, a type system is a tractable syntactic method for proving the absence of certain program behaviors. This is done by classifying expressions according to the kinds of values they compute.

In this note, we will briefly describe how this can be done, and what issues arise. We will start by considering our Nano-ML language which consisted of numbers, booleans, if-expressions and primitive operations. Next, we will considering an extension where we have variables and let-expressions. Finally, we will include functions and function application.

We begin by considering Nano-ML which includes numbers, booleans, if-expressions and primitive operations. Recall our inductive definition of these expressions:

$$\begin{array}{lll} \text{Operations } \mathsf{op} & ::= & + \mid - \mid * \mid < \mid = \\ \text{Expressions } e & ::= & n \mid e_1 \ \mathsf{op} \ e_2 \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \end{array}$$

This grammar inductively specifies well-formed expressions. We will now concentrate on the questions: What expressions do we consider well-typed? How do we assign an expression a type?

As mentioned earlier, types classify expressions according to their value. So what are the values in Nano-ML? Values in Nano-ML are numbers and booleans. Therefore we will only have two basic types which suffice to approximate the run-time behavior of NanoML expressions. The type int characterizes number and the type bool describes the booleans true and false. In other words, the expressions true and false are elements of the type bool, and number n are elements of the type int. We also sometimes say true and false inhabit the type bool, or similarly number n inhabit the type int. We will write the capital letter T for types.

$$\text{Types } \mathsf{T} ::= \mathsf{int} \mid \mathsf{bool}$$

Next, we formally describe when an expression is well-typed using the following judgment:

$$e : \mathsf{T} \quad \text{expression } e \text{ has type } \mathsf{T}$$

We will define when an expression is well-typed inductively on the structure of the expression. A term is typable if there is some type T s.t. $e : \mathsf{T}$. We start by considering numbers and booleans.

$$\frac{}{n : \mathsf{int}}\ \text{T-NUM} \qquad \frac{}{\mathsf{true} : \mathsf{bool}}\ \text{T-TRUE} \qquad \frac{}{\mathsf{false} : \mathsf{bool}}\ \text{T-FALSE}$$

Next, let us consider the if-expression. When is $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ well-typed? And what should its type be? – Intuitively, we can assign $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ a type T, if

- expression e has type bool (i.e. expression e evaluates eventually to a boolean)
- expression e_1 has some type T
- expression e_2 has the same type T

This can be formalized using inference rules as follows:

$$\frac{e : \mathsf{bool} \quad e_1 : \mathsf{T} \quad e_2 : \mathsf{T}}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \mathsf{T}}\ \text{T-IF}$$

Finally, we consider the rules for primitive operations. We only show the rules for addition, multiplication, equality but the others are straightforward and follow similar principle.

$$\frac{e_1 : \mathsf{int} \quad e_2 : \mathsf{int}}{e_1 + e_2 : \mathsf{int}}\ \text{T-PLUS} \qquad \frac{e_1 : \mathsf{int} \quad e_2 : \mathsf{int}}{e_1 * e_2 : \mathsf{int}}\ \text{T-MULT} \qquad \frac{e_1 : \mathsf{T} \quad e_2 : \mathsf{T}}{e_1 = e_2 : \mathsf{bool}}\ \text{T-EQ}$$

9.6 Growing the language: variables, let-expressions and functions

In this section, we will extend the language we have seen so far with variables, let-expressions. We will hence obtain a language very close to the core of OCaml; there are only minor syntactic differences.

We first extend our inductive definition to allow variables, and let-expressions. Next, we will consider issues arising due to variables. In particular we will explain the notion of a bound and free variables, overshadowing of variables, and substitution. Finally, we are extending the operational semantics to evaluate let-expressions.

9.6.1 Syntax for variables and let-expressions

First, we will extend our inductive definition for expressions to allow for variables and let-expressions. We write ... to indicate we have all the previously defined expressions of numbers, booleans, primitive operations, and if-statements.

$$\text{Expressions } e ::= \ldots \mid x \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$$

We will write x, y, z for variables. To illustrate again these new expressions, here are some examples:

Examples of well-formed expressions

- $\mathsf{let}\ z = \mathsf{if}\ \mathsf{true}\ \mathsf{then}\ 2\ \mathsf{else}\ 43\ \mathsf{in}\ z + 123\ \mathsf{end}$
- $\mathsf{let}\ z = \mathsf{if}\ 7\ \mathsf{then}\ 2\ \mathsf{else}\ 43\ \mathsf{in}\ z + \mathsf{false}\ \mathsf{end}$
- $\mathsf{let}\ x = x + 3\ \mathsf{in}\ y + 123\ \mathsf{end}$
- $\mathsf{let}\ x = x + 3\ \mathsf{in}\ x + 123\ \mathsf{end}$

The last two examples are well-formed expressions, although we may not yet understand the meaning of it. In particular, we have not clarified the status of the variables x and y occurring in the expression.

Examples of ill-formed expressions

- $\mathsf{let}\ z = \mathsf{if}\ true\ \mathsf{then}\ 2\ \mathsf{else}\ 43\ \mathsf{in}\ -123\ \mathsf{end}$
- $\mathsf{let}\ x = 3\ \mathsf{in}\ x$
- $\mathsf{let}\ x = 3\ x + 2\mathsf{end}$

The first expression here is ill-formed because -123 is not syntactically allowed. The second expression is erroneous because the end is missing to indicate the end of the let-expression. The last one is rejected because the key word in is missing.

Before we can define the operational semantics for evaluating let-expressions, we first clarify issues arising from variables. The first question we must answer is: When is variable bound and when is it considered free? Intuitively, a variable is considered free, if it is not bound. We will define these two concepts next.

9.6.2 Free variables and substitutions

"No one is free, even the birds are chained to the sky."
Bob Dylan

Fist, let us define inductively the set of free variables occurring in an expression e. We will define a function FV which takes as input an expression e and returns a set of the free variables occurring in e.

$$\begin{array}{lcl} \mathsf{FV}(x) & = & \{x\} \\ \mathsf{FV}(e_1 \text{ op } e_2) & = & \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \\ \mathsf{FV}(\text{if } e \text{ then } e_1 \text{ else } e_2) & = & \mathsf{FV}(e) \cup \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \\ \mathsf{FV}(\text{let } x = e_1 \text{ in } e_2 \text{ end}) & = & \mathsf{FV}e_1 \cup (\mathsf{FV}(e_2) / \{x\}) \end{array}$$

This definition also highlights the fact that a variable x may become *bound* in the let-expression. The let-expression let $x = e_1$ in e_2 end introduces a binder x which binds all the occurrences of x in e_2. An expression e which has no free variables, i.e. $\mathsf{FV}(e) = \emptyset$ is called *closed*. The following example clarifies the concept of bound and free variables.

$$\text{let } x = 5 \text{ in } \underbrace{(\text{let } y = \overbrace{x+3}^{\text{x is free}} \text{ in } \overbrace{y+x}^{\text{x, y are free}} \text{ end})}_{\text{x is free, y is bound}} \text{ end}$$

The subexpression $y + x$ contains the free occurrences of the variable x and y. The variable y then gets bound in let $y = x + 3$ in $y + x$ end. Hence only x remains a free. The scope of y is only in the body of the let expression. More generally, given a let-expression let $x = e_1$ in e_2 end, the binder x binds variables occurring in e_2 only.

To clarify let us consider the free and bound variables in the following expression:

$$\text{let } x = 5 \text{ in } (\text{let } x = \overbrace{x+3}^{\substack{\text{x is free} \\ \text{bound by x = 5}}} \text{ in } \overbrace{x+x}^{\substack{\text{x is free} \\ \text{bound by x = x+3}}} \text{ end}) \text{ end}$$

In other words, a free variable x gets bound by the first binder enclosing it. Another important property of bound variables is that their names do not matter.

Clearly, it should not matter, if we write

$$\mathsf{let}\ x = 5\ \mathsf{in}\ (\mathsf{let}\ y = x + 3\ \mathsf{in}\ y + y\ \mathsf{end})\ \mathsf{end}$$

or

$$\mathsf{let}\ x = 5\ \mathsf{in}\ (\mathsf{let}\ x = x + 3\ \mathsf{in}\ x + x\ \mathsf{end})\ \mathsf{end}$$

Both terms denote the same expression up to renaming of the bound variables. Before we describe the evaluation of a let-expression, we will define the important operation of substituting an expression for a free variable in another expression. We will write $[e'/x]e$ for replacing all *free* occurrences of the variable x in the expression e with the expression e'. We will define this operation inductively on the expression e. This is straightforward in most cases. Applying the substitution $[e'/x]$ to a variable x clearly should return the term e'. When applying the substitution $[e'/x]$ to the expression $e_1\ \mathsf{op}\ e_2$, we need to apply $[e'/x]$ to each of its sub-expressions. Similarly, when applying the substitution $[e'/x]$ to the expression $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$, we need to apply the substitution $[e'/x]$ to the sub-expression e, e_1, and e_2. This is defined as follows more formally:

$$\begin{array}{lcl} [e'/x](x) & = & e' \\ [e'/x](e_1\ \mathsf{op}\ e_2) & = & ([e'/x]e_1\ \mathsf{op}\ [e'/x]e_2) \\ [e'/x](\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2) & = & \mathsf{if}\ [e'/x]e\ \mathsf{then}\ [e'/x]e_1\ \mathsf{else}\ [e'/x]e_2 \end{array}$$

The most interesting case is what should happen when we apply the substitution $[e'/x]$ to the expression $\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$.

In the first attempt, we just apply the substitution $[e'/x]$ to the sub-expressions e_1 and e_2.

$$[e'/x](\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2\ \mathsf{end}) = \mathsf{let}\ y = [e'/x]e_1\ \mathsf{in}\ [e'/x]e_2\ \mathsf{end} \qquad \text{Attempt 1}$$

This seems to work fine for this example:

$$\begin{array}{l} [5/x](\mathsf{let}\ y = x + 3\ \mathsf{in}\ y + x\ \mathsf{end}) = \\ \mathsf{let}\ y = [5/x](x + 3)\ \mathsf{in}\ [5/x](y + x)\ \mathsf{end} = \\ \mathsf{let}\ y = 5 + 3\ \mathsf{in}\ y + 5\ \mathsf{end} \end{array}$$

However, what will happen when we try to replace the variable x with the term $y + 1$? Simple replacement would yield,

$$\begin{array}{ll} & [y + 1/x](\mathsf{let}\ y = x + 3\ \mathsf{in}\ y + x\ \mathsf{end}) \\ = & \mathsf{let}\ y = [y + 1/x](x + 3)\ \mathsf{in}\ [y + 1/x](y + x)\ \mathsf{end} \\ = & \mathsf{let}\ y = (y + 1) + 3\ \mathsf{in}\ y + (y + 1)\ \mathsf{end} \end{array}$$

But this seems wrong, since y occurred free in the term $y+1$, but becomes bound in the result! This phenomena is called *variable capture*, and clearly needs to be avoided. The key is the observation that the name of bound variables do not matter, and hence we can always rename the bound variable y in $(\mathsf{let}\ y = x+3\ \mathsf{in}\ y+x\ \mathsf{end})$ to z and obtain $(\mathsf{let}\ z = x+3\ \mathsf{in}\ z+x\ \mathsf{end})$ which is equivalent. A change of bound variables is often called *α-conversion*. To ensure substitution works correctly for let-expressions, i.e. $[e'/x](\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$, we need to impose the restriction that y does not occur free in the expression e'. This side condition can always be achieved by renaming bound variables. We can now define more formally the last case for let-expressions:

$$[e'/x](\mathsf{let}\ y = e_1\ \mathsf{in}\ e_2\ \mathsf{end}) = \mathsf{let}\ y = [e'/x]e_1\ \mathsf{in}\ [e'/x]e_2\ \mathsf{end}$$
$$\text{provided } x \neq y \text{ and } y \notin \mathsf{FV}(e')$$

The problem of variable capture occurs whenever we have bound variables. Once we extend the language to include functions for example we will encounter the same problem.

9.6.3 Evaluation of let-expressions

Finally, we are in a position to describe formally how a let-expression will be evaluated. Intuitively, when given an expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$, we first evaluate the expression e_1 to some value v_1. Next, we replace all free occurrences of x in the expression e_2 by the value v_1 and continue to evaluate $[v_1/x]e_2$ to some value v_2. This can be formally described by the following inference rule:

$$\frac{e_1 \Downarrow v_1 \quad [v_1/x]e_2 \Downarrow v}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end} \Downarrow v}\ \textsc{b-let}$$

Next, a sample derivation illustrating evaluation of a let-expression.

$$\dfrac{\dfrac{}{5 \Downarrow 5}\ \textsc{b-num} \qquad \dfrac{\dfrac{\dfrac{}{5 \Downarrow 5}\ \textsc{b-num} \quad \dfrac{}{3 \Downarrow 3}\ \textsc{b-num}}{5+3 \Downarrow 8}\ \textsc{b-op} \quad \dfrac{\dfrac{}{5 \Downarrow 5}\ \textsc{b-num} \quad \dfrac{}{8 \Downarrow 8}\ \textsc{b-num}}{5+8 \Downarrow 13}\ \textsc{b-op}}{(\mathsf{let}\ y = 5+3\ \mathsf{in}\ 5+y\ \mathsf{end}) \Downarrow 13}\ \textsc{b-let}}{\mathsf{let}\ x = 5\ \mathsf{in}\ (\mathsf{let}\ y = x+3\ \mathsf{in}\ x+y\ \mathsf{end})\ \mathsf{end} \Downarrow 13}\ \textsc{b-let}$$

9.6.4 Functions and Function application

Next we will add functions and function application. They follow the same principles we employed when handling let-expressions. We will first introduce nameless

functions as we have encountered them in OCaml, and recursive functions.

$$\text{Expressions } e ::= \ldots \mid \texttt{fn } x \texttt{ => } e \mid e_1\ e_2 \mid \texttt{rec f => } e$$

Before we consider evaluation, let us define the free variables occurring in a function.

$$\begin{array}{lcl} \mathsf{FV}(e_1\ e_2) & = & \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \\ \mathsf{FV}(\texttt{fn } x \texttt{ => } e) & = & \mathsf{FV}(e) / \{x\} \\ \mathsf{FV}(\texttt{rec f => } e) & = & \mathsf{FV}(e) / \{f\} \end{array}$$

The definition highlights that both fn x => e and rec f => e bind variables. In the first case of fn x => e the input variable x is bound, and in the second case of rec f => e the function name is bound.

Next, we extend the definition for substitution. Similar to let-expressions, we must be careful to avoid the problem of variable capture.

$$\begin{array}{lcll} [e'/x](e_1\ e_2) & = & [e'/x]e_1\ [e'/x]e_2 & \\ [e'/x](\texttt{fn } y \texttt{ => } e) & = & \texttt{fn } y \texttt{ => } [e'/x]e & \text{provided } x \neq y \text{ and } y \notin \mathsf{FV}(e') \\ [e'/x](\texttt{rec f => } e) & = & \texttt{rec f => } [e'/x]e & \text{provided } x \neq f \text{ and } f \notin \mathsf{FV}(e') \end{array}$$

Functions are considered first-class values, and hence will evaluate to themselves (see rule B-FN). Evaluation of function application $(e_1\ e_2)$ is done in three steps. First, we evaluate e_1 which will (hopefully!) yield a function fn x => e. In addition, we evaluate e_2 to obtain some value v_2. Finally, we need to evaluate the function body e where we have replaced any occurrence of x with v_2.

$$\frac{}{\texttt{fn } x \texttt{ => } e \Downarrow \texttt{fn } x \texttt{ => } e}\ \text{B-FN} \qquad \frac{e_1 \Downarrow \texttt{fn } x \texttt{ => } e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1\ e_2 \Downarrow v}\ \text{B-APP}$$

This allows us to evaluate functions and function application. However, sometimes we would also like to write recursive functions. This is handled by the construct rec f => e. The intention is that we separate the name of the recursive function and the name of the arguments passed to it. Intuitively, the correspondence between the function written in OCaml and in our fictional language is as follows. Recall that in OCaml we write

```
1 let rec sum x = if x = 0 then 1 else x + sum (x-1)
```

Using our syntax, we can write this function as :

$$\texttt{rec } sum \texttt{ => fn } x \texttt{ => if } x = 0 \texttt{ then } 1 \texttt{ else } x + sum(x-1)$$

Our language disentangles therefore recursion and function abstraction. How do we evaluate recursive functions? – The simplest way to model recursive evaluation is again by substitution. To evaluate rec f => e we will simply replace any occurrence of f in e with the actual function definition rec f => e. This means that when we need call the recursive function we will have the actual code present. This is a very simple model which gives a very high level description and is ideally suited for reasoning about evaluation. However, we would like to point out that this is typically not how execution of recursive functions is implemented for real.

$$\frac{[\text{rec f => } e/\text{f}]e \Downarrow v}{\text{rec f => } e \Downarrow v}\ \text{B-REC}$$

To illustrate recursive evaluation, consider the following program:

$$\text{sum} = \text{rec f => fn x => if } x = 0 \text{ then } 0 \text{ else } x + f(x-1)$$

We will write sum' for the first unfolding of the recursion:

$$\text{sum'} = \text{fn x => if } x = 0 \text{ then } 0 \text{ else } x + \text{sum}(x-1)$$

How does evaluation then proceed? – Let's look at an example.

$$\dfrac{\dfrac{\dfrac{}{\text{sum' } \Downarrow \text{ sum'}}\ \text{B-FN}}{\text{sum } \Downarrow \text{ sum'}}\ \text{B-REC} \quad \dfrac{}{2 \Downarrow 2}\ \text{B-NUM} \quad \dfrac{\vdots}{\text{if } 2 = 0 \text{ then } 0 \text{ else } 2 + \text{sum}(2-1) \Downarrow 3}\ \text{B-IFFALSE}}{\text{sum } 2 \Downarrow 3}\ \text{B-APP}$$

$$\dfrac{\dfrac{\dfrac{}{2 \Downarrow 2}\ \text{B-NUM} \quad \dfrac{}{0 \Downarrow 0}\ \text{B-NUM}}{2 = 0 \Downarrow \text{true}}\ \text{B-OP} \quad \dfrac{\dfrac{}{2 \Downarrow 2}\ \text{B-NUM} \quad \dfrac{\vdots}{\text{sum}(2-1) \Downarrow 1}\ \text{B-APP}}{2 + \text{sum}(2-1) \Downarrow 3}\ \text{B-OP}}{\text{if } 2 = 0 \text{ then } 0 \text{ else } 2 + \text{sum}(2-1) \Downarrow 3}\ \text{B-IFFALSE}$$

$$\dfrac{\dfrac{\dfrac{}{\text{sum' } \Downarrow \text{ sum'}}\ \text{B-FN}}{\text{sum } \Downarrow \text{ sum'}}\ \text{B-REC} \quad \dfrac{\dfrac{}{2 \Downarrow 2}\ \text{B-NUM} \quad \dfrac{}{1 \Downarrow 1}\ \text{B-NUM}}{2 - 1 \Downarrow 1}\ \text{B-OP} \quad \dfrac{\vdots}{\text{if } 1 = 0 \text{ then } 0 \text{ else } 1 + \text{sum}(1-1) \Downarrow 1}\ \text{B-IFFALSE}}{\text{sum}(2-1) \Downarrow 1}\ \text{B-APP}$$

The essential idea is to copy the code for the recursive function at the appropriate places during recursion so we have it available when we execute the recursive call.

Chapter 10

Types

So far, we only have considered syntax (i.e. what expressions are syntactically well-formed?) and operational semantics (i.e. how are we executing an expression?). However, there are many expressions which cannot be evaluated and our interpreter would get stuck. Examples of such expressions include : `if 0 then 45 else 33`, $0 + \mathsf{true}$, `(fn x => x + 1) true`, etc.

How can we statically check whether an expressions would potentially lead to a runtime error? – We already encountered types in the language SML. Types approximate the runtime behaviour and provide an effective light-weight tool for reasoning about programs. They allow us to detect errors statically and early on the development cycle. Type checkers give precise error messages pointing the programmer to the specific line in the code where the reasoning went wrong. Ultimately, this means programmers spend more time developing their programs but less time testing the programs; programs are also easier to maintain. More generally, a type system is a tractable syntactic method for proving the absence of certain program behaviours. This is done by classifying expressions according to the kinds of values they compute.

In this note, we will briefly describe how this can be done, and what issues arise. We will start by considering our Nano-ML language which consisted of numbers, booleans, if-expressions and primitive operations. Next, we will considering an extension where we have variables and let-expressions. Finally, we will include functions and function application.

10.1 Basic Types

We begin by considering Nano-ML which includes numbers, booleans, if-expressions and primitive operations. Recall our inductive definition of these expressions:

$$\begin{array}{lll} \text{Operations } \mathsf{op} & ::= & + \mid - \mid * \mid < \mid = \\ \text{Expressions } e & ::= & n \mid e_1 \ \mathsf{op} \ e_2 \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \end{array}$$

This grammar inductively specifies well-formed expressions. We will now concentrate on the questions: What expressions do we consider well-typed? How do we assign an expression a type?

As mentioned earlier, types classify expressions according to their value. So what are the values in Nano-ML? Values in Nano-ML are numbers and booleans. Therefore we will only have two basic types which suffice to approximate the run-time behaviour of NanoML expressions. The type int characterizes number and the type bool describes the booleans true and false. In other words, the expressions true and false are elements of the type bool, and number n are elements of the type int. We also sometimes say true and false inhabit the type bool, or similarly number n inhabit the type int. We will write the capital letter T for types.

$$\text{Types } \mathsf{T} \ ::= \ \mathsf{int} \mid \mathsf{bool}$$

Next, we formally describe when an expression is well-typed using the following judgment:

$$e : \mathsf{T} \quad \text{expression } e \text{ has type } \mathsf{T}$$

We will define when an expression is well-typed inductively on the structure of the expression. A term is typable if there is some type T s.t. $e : \mathsf{T}$. We start by considering numbers and booleans.

$$\frac{}{n : \mathsf{int}}\ \text{T-NUM} \qquad \frac{}{\mathsf{true} : \mathsf{bool}}\ \text{T-TRUE} \qquad \frac{}{\mathsf{false} : \mathsf{bool}}\ \text{T-FALSE}$$

Next, let us consider the if-expression. When is $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ well-typed? And what should its type be? – Intuitively, we can assign $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ a type T, if

- expression e has type bool (i.e. expression e evaluates eventually to a boolean)
- expression e_1 has some type T
- expression e_2 has the same type T

This can be formalized using inference rules as follows:

$$\frac{e : \mathsf{bool} \quad e_1 : \mathsf{T} \quad e_2 : \mathsf{T}}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \mathsf{T}}\ \text{T-IF}$$

Finally, we consider the rules for primitive operations. We only show the rules for addition, multiplication, equality but the others are straightforward and follow similar principle.

$$\frac{e_1 : \mathsf{int} \quad e_2 : \mathsf{int}}{e_1 + e_2 : \mathsf{int}}\ \text{T-PLUS} \qquad \frac{e_1 : \mathsf{int} \quad e_2 : \mathsf{int}}{e_1 * e_2 : \mathsf{int}}\ \text{T-MULT} \qquad \frac{e_1 : \mathsf{T} \quad e_2 : \mathsf{T}}{e_1 = e_2 : \mathsf{bool}}\ \text{T-EQ}$$

10.2 Typing for tuples and projections

In a similar fashion, we can introduce types for tuples and projections.

$$\text{Expressions}\ \ e \ ::= \ \ \ldots \mid (e_1, e_2) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e$$

We can create tuples via $(e_1,\ e_2)$ and we can take them apart via $\mathsf{fst}\ e$ which extracts the first component of a tupel e, and $\mathsf{snd}\ e$ will extract the second component of a tupel e. Considering the evaluation rules for tuples, it is clear we have now not only numbers and booleans as values, but a tuple is also a possible value.

The type classifying tuples is called *product* and written as $\mathsf{T}_1 \times \mathsf{T}_2$. The rules for tuples and projections are then in fact straightforward.

$$\frac{e_1 : \mathsf{T}_1 \quad e_2 : \mathsf{T}_2}{(e_1,\ e_2) : \mathsf{T}_1 \times \mathsf{T}_2}\ \text{T-PAIR} \qquad \frac{e : \mathsf{T}_1 \times \mathsf{T}_2}{\mathsf{fst}\ e : \mathsf{T}_1}\ \text{T-FST} \qquad \frac{e : \mathsf{T}_1 \times \mathsf{T}_2}{\mathsf{snd}\ e : \mathsf{T}_2}\ \text{T-SND}$$

10.3 Typing for variables and let-expressions

In this section, we would like to extend our typing rules to handle variables and let-expressions. The first observation is that we need to be able to reason with assumptions about variables. For example, consider the following let-expressions: $\mathsf{let}\ x = 5\ \mathsf{in}\ x + 3\ \mathsf{end}$. We would like to argue as follows:

- 5 is an integer. Therefore the variable x will be bound at runtime to a value of type int.
- Assuming that x has type int, $x+3$ has type int, because each subexpression has type int.

More generally, the expression $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$ has type T, if

- e_1 has type T_1

- assuming x has type T_1, the expression e_2 has type T.

To handle bound variables and be able to reason about them, we introduce a context Γ, which keeps track of our assumptions. The assumption "variable x has type T" is written $x{:}T$. We can define a context Γ inductively as follows:

$$\text{Context } \Gamma \;:=\; \cdot \mid \Gamma, x{:}T$$

$\cdot$ describes the empty context, i.e. there are no typing assumptions. If we have a context Γ then $\Gamma, x{:}T$ is also a context. Sometimes we also call $x{:}T$ a typing declaration. Every typing declaration $x{:}T$ occurs uniquely, i.e. for any two declarations $x{:}T$ and $x'{:}T'$, we have $x \neq x'$.

Next, we will generalize our typing judgment to take into account the context Γ (the set of assumptions available).

$$\Gamma \vdash e : T \quad \text{Expression } e \text{ has type } T \text{ using the typing declarations in } \Gamma$$

In the previous rules, the set of assumptions does not change, and hence we simply carry it as an extra parameter.

$$\frac{}{\Gamma \vdash n : \mathsf{int}}\ \text{T-NUM} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}}\ \text{T-TRUE} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}\ \text{T-FALSE}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}\ \text{T-PLUS} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 * e_2 : \mathsf{int}}\ \text{T-MULT}$$

$$\frac{\Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 = e_2 : \mathsf{bool}}\ \text{T-EQ} \qquad \frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : T}\ \text{T-IF}$$

The more interesting question is how to formally deal with variables and describe the recipe for assigning a type to a let-expression. If we have an assumption $x{:}T$ in Γ, then we can conclude that a variable x has type T. The recipe for the let-expression is directly translated into an inference rule.

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \text{T-VAR} \qquad \frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x{:}T_1 \vdash e_2 : T}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end} : T}\ \text{T-LET}\ \ x \text{ must be new}$$

We note that x must be new to ensure that the declaration $x{:}T_1$ does not clash with any other declaration in Γ. This can always be achieved by appropriately renaming the bound variable x in the expression e_2.

To check whether an expression e has a type T, we construct a proof for $\cdot \vdash e : T$. Here is an example, how we can check whether $\mathsf{let}\ x = 5\ \mathsf{in}\ x + 2\ \mathsf{end}$ has type int.

$$\dfrac{\dfrac{}{\cdot \vdash 5 : \mathsf{int}}\ \text{T-NUM} \qquad \dfrac{\dfrac{\dfrac{}{x{:}\mathsf{int} \vdash x : \mathsf{int}}\ \text{T-VAR} \quad \dfrac{}{x{:}\mathsf{int} \vdash 2 : \mathsf{int}}\ \text{T-NUM}}{x{:}\mathsf{int} \vdash x + 2 : \mathsf{int}}\ \text{T-PLUS} \quad \dfrac{\dfrac{}{x{:}\mathsf{int}, y{:}\mathsf{int} \vdash x : \mathsf{int}}\ \text{T-VAR} \quad \dfrac{}{x{:}\mathsf{int}, y{:}\mathsf{int} \vdash y : \mathsf{int}}\ \text{T-V}}{x{:}\mathsf{int}, y{:}\mathsf{int} \vdash x * y : \mathsf{int}}\ \text{T-M}}{x{:}\mathsf{int} \vdash \mathsf{let}\ y = x + 2\ \mathsf{in}\ x * y\ \mathsf{end} : \mathsf{int}}\ \text{T-LET}}{\cdot \vdash \mathsf{let}\ x = 5\ \mathsf{in}\ (\mathsf{let}\ y = x + 2\ \mathsf{in}\ x + y\ \mathsf{end})\ \mathsf{end} : \mathsf{int}}\ \text{T-LET}$$

Note that we can have unused assumptions in some branches. There are some additional structural properties we have for contexts: 1) The order of assumptions does not matter and in fact we can exchange assumptions. In this theoretical descriptions, our context Γ is essentially a set of assumptions. 2) We can re-use assumptions from the context Γ as often as we need to. 3) We do not have to use assumptions, i.e. we can have unused assumptions. For example, if we type check the expressions $\mathsf{let}\ x = 5\ \mathsf{in}\ 3\ \mathsf{end}$ then we check that 3 has type int under the assumption that x has type int; but the assumption about x is never needed.

Type-checking vs Type inference So far we mainly have talked about assigning a type to an expression. In practice however we often distinguish between type checking and type inference. In type checking, we give an expression e and a type T and we check whether $e : T$, i.e. expression e indeed has type T. However, we may not always have both the expression and the type. For example, in the rule T-LET, we cannot simply check whether expression e_1 has the type T_1, because we don't know what it is. We must infer a type for expression e_1.

In type inference, we give the expression e and infer a type T s.t. the expression has type T. The next question, we can ask is, whether we can always infer a unique type corresponding to an expression. In the language we have encountered so far, this is the case. This is a property about our type system and can be stated more formally as follows:

Uniqueness If $\Gamma \vdash e : T$ and $\Gamma \vdash e : T'$ then $T = T'$.

In fact it is quite easy to see that the typing rules are deterministic. At any given point, there is exactly one rule applicable for an expression.

10.4 Typing for functions and function application

Next we extend our type system to include functions and function application. We have seen that functions are first-class values, and since types classify values we will add a new type to our NanoML types, the function type T_1 -> T_2.

$$\text{Types } T ::= \text{int} \mid \text{bool} \mid T_1 \text{ -> } T_2$$

A function fn x => e has type T_1 -> T_2 if assuming x has type T_1, we can show that the body e_2 has type T_2. An application e_1 e_2 has type T if expression e_1 has type T_2 -> T and e_2 has type T_2. In other words, expression e_2 is a suitable input to the function computed by e_1. More formally,

$$\frac{\Gamma, x{:}T_1 \vdash e : T_2}{\Gamma \vdash \text{fn } x \text{ => } e : T_1 \text{ -> } T_2}\ \text{T-FN} \qquad \frac{\Gamma \vdash e_1 : T_2 \text{ -> } T \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ e_2 : T}\ \text{T-APP}$$

In interesting question to ask whether it is still true that every function has a unique type. This is no longer true. For example,

fn x => x has type int -> int
fn x => x has type bool -> bool
fn x => x has type (int -> int) -> (int -> int)
...

In fact, the identity function has infinitely many types! How can we recover that every expression has a unique type? – There are two solutions to this problem. The easiest is to allow type annotations to resolve ambiguity. As a consequence, we would write fn x:T => e and annotate the input variable x with its type. So to use the identity function as a function for booleans we would need to write fn x : bool => x and when we would like to use it for integers we would need to write fn x : int => x. This is unfortunate, because the function is generic and can be executed with integers and with booleans. A remedy to this problem is to allow type variables α. Instead of requiring that every expression has a unique type, we allow that an expression may have multiple types, but it must have a *principal type*, a type which is more general from which all others can be derived. We will come back to how to infer that a given expression has a principal type T later. So our types include now:

$$\text{Types } T ::= \text{int} \mid \text{bool} \mid T_1 \text{ -> } T_2 \mid \alpha$$

10.5 References

In this section, we briefly highlight how to extend our syntax and types to include references.

$$\begin{array}{lll} \text{expression } e & ::= & \ldots \mid e_1 := e_2 \mid !e \mid \text{ref } e \mid () \\ \text{types } T & ::= & \ldots \mid T \text{ ref} \mid \text{unit} \end{array}$$

Just to consider the typing for this extension is in fact fairly straightforward. However, we do not consider extending the operational semantics with references which would require us to model formally the heap.

$$\frac{\Gamma \vdash e_1 : \mathsf{T}\ \mathsf{ref} \quad \Gamma \vdash e_2 : \mathsf{T}}{\Gamma \vdash e_1 := e_2 : \mathsf{unit}} \qquad \frac{\Gamma \vdash e : \mathsf{T}\ \mathsf{ref}}{\Gamma \vdash\ !e : \mathsf{T}} \qquad \frac{\Gamma \vdash e : \mathsf{T}}{\Gamma \vdash \mathsf{ref}\ e : \mathsf{T}\mathsf{ref}} \qquad \frac{}{\Gamma \vdash () : \mathsf{unit}}$$

10.6 Properties of type systems

As we mentioned earlier, type systems approximate the run-time behaviour and we designed the typing rules in such a way that certain expressions which could potentially cause problems during run-time are considered as ill-typed. For example, $\mathsf{true} + 5$ would lead to an error during runtime and the type systems identifies this expression as ill-typed. The intention is that if an expression e has a type T, then evaluation of e will not get stuck. Moreover, if expression e has type T and e evaluates to some value v then v will have the same type T. In other words types are preserved during evaluation. Both of these two properties together are called *type safety*:

Safety If expression e has type T, then either e is already a value or we can evaluate it in one step to another expression e' and e' has type T.

Safety incorporates really two properties which we merged together in the previous statement. The first one, called *progress*, says if an expression e is well-typed, then either it is a value or we can evaluate it to some other expression e'. The second property, called *preservation*, says that if an expression e has type T and e evaluates some expression e' then e' has type T.

The operational semantics we have introduced previously is a big-step semantics, and evaluates expressions to values in one big step. Hence we do not have the ability to reason about intermediate expressions which may occur during evaluation. To prove progress, we need a more fine grained model of evaluation, namely a small-step semantics. However, on property we can even prove about our big-step semantics is *type preservation*. If an expression e has type T and e evaluates to some final value v, then v will have the same type T. More formally this can be stated as follows:

Preservation If $e \Downarrow v$ and $e : \mathsf{T}$ then $v : \mathsf{T}$.

These meta-theoretic properties ensure that in fact the typing rules do what we intended them to do. Or in other words, if a program type checks, then it will not go wrong during evaluation. Remarkably this property even holds in the presence of references, exceptions and other features we find in real programming languages. In the presence of references for example we can guarantee statically that we will never try to access a memory location which wasn't created appropriately earlier. This in essence guarantees that the program when executed will not core dump.

10.7 Polymorphism and type inference

So far we have mainly been concerned with assigning a type to a given piece of code. To be able to infer a type from a given expression, we required typing annotations in function. In the expression `fun x:T -> e`, we ensure we always know the type of the input this function requires. In these notes we will explain how to infer a type for a given expression without requiring any typing annotations. We will not only be able to infer some type for a given expression, but the principal type, i.e. the most general type.

Before we discuss type inference more deeply, we will first concentrate on the role of polymorphic types and type variables. For example, we can check that the function

$$\mathsf{double} = \mathsf{fn}\ f\ \Rightarrow\ \mathsf{fn}\ x\ \Rightarrow\ f(f(x))$$

has the type : $(\alpha \to \alpha) \to \alpha \to \alpha$. Intuitively this means we can use this function in multiple ways for example, when we use it in

$$\mathsf{double}\ (\mathsf{fn}\ x\ \Rightarrow\ x + 2)\ 3$$

the type variable α will get instantiated to int. When we use it in

$$\mathsf{double}\ (\mathsf{fn}\ x\ \Rightarrow\ x)\ \mathsf{false}$$

the type variable α will be instantiated to bool. We would like to emphasize that the same code will get executed in both expression.

10.8 Type variables

We will begin by clarifying the use of type-variables. First, let us extend the definition for types with type variables:

$$\text{Types } T ::= \mathsf{int} \mid \mathsf{bool} \mid T_1 \to T_2 \mid T_1 \times T_2 \mid \alpha$$

What do we mean by "instantiating a type variable α with a concrete type bool in the type $\alpha \to \alpha$?" – Essentially we mean we apply a substitution to the type $\alpha \to \alpha$ which replaces all free occurrences of the type variable α with the type bool. Defining substitution is in fact straightforward:

$$\begin{array}{lcl} [T/\alpha](\alpha) & = & T \\ [T/\alpha](\beta) & = & \beta \\ [T/\alpha](\mathsf{int}) & = & \mathsf{int} \\ [T/\alpha](\mathsf{bool}) & = & \mathsf{bool} \\ [T/\alpha](T_1 \times T_2) & = & [T/\alpha]T_1 \times [T/\alpha]T_2 \\ [T/\alpha](T_1 \to T_2) & = & [T/\alpha]T_1 \to [T/\alpha]T_2 \end{array}$$

We will write σ for the simultaneous substitution $[T_1/\alpha_1, \ldots T_n/\alpha_n]$.

A crucial property of type substitutions is that they preserve the validity of typing statements: If an expression e has type T and σ is a type substitution, then expression e will also have type $[\sigma]T$.

Theorem 10.8.1. *If* $\Gamma \vdash e : T$ *and* σ *is a type substitution then* $[\sigma]\Gamma \vdash e : [\sigma]T$.

10.8.1 Two views of type variables

Suppose we have an expression e which has type T in a context Γ, i.e. $\Gamma \vdash e : T$, where T and Γ may possibly contain type variables. Then we can ask two different questions:

1. Are *all* substitution instances of e well-typed? That is for every type substitution σ, we have $[\sigma]\Gamma \vdash e : [\sigma]T$.

2. Is *some* substitution instance of e well-typed? That is we can find a type substitution σ, such that $[\sigma]\Gamma \vdash e : [\sigma]T$.

The first answer implies that type variables are held abstract during type checking. They are merely place holders which can be instantiated with any concrete type. For example:

$$\mathsf{fn\ f => fn\ x => f(f(x))} \text{ has type } (\alpha \to \alpha) \to \alpha \to \alpha$$

By replacing α with bool, we still maintain that the function is well typed, and in fact has exactly the same typing derivation! Holding type variables abstract in such

a way leads to *parametric polymorphism*. Type variables are used to encode the fact that a term can be used in many concrete contexts with different concrete types.

In the second view, the original term e may not even be well typed. What we want to know is whether we can instantiated the type variables such that it will be well-typed. For example, we could ask whether there exists an instantiation for the type variables β_1 and β_2 s.t. the term

$$\texttt{fn f => fn x => f(f(x))} \text{ has type } \beta_1 \to \beta_2$$

well-typed.

The answer to this question would be yes, if we instantiate β_1 with $(\alpha \to \alpha)$ and β_2 with $\alpha \to \alpha$. Recall that function types are parenthesized to the right.

To illustrate let us consider a few more examples:

1. Does there exist and instantiation for the type variables β s.t. we have

   $$\texttt{fn x => x + 1} \quad \text{has type} \quad \beta\ ?$$

   Yes, if we instantiate β to $\texttt{int} \to \texttt{int}$.

2. Does there exist and instantiation for the type variables β s.t. we have

   $$\texttt{fn x => x} \quad \text{has type} \quad \beta\ ?$$

   Yes, if we instantiate β to `int -> int` or we choose β to be `bool -> bool` or we choose β to be $\alpha \to \alpha$.

3. Does there exist and instantiation for the type variables β s.t. we have

   $$\texttt{fn x => x + 1} \quad \text{has type} \quad \beta \texttt{ -> bool}\ ?$$

   The answer is no.

The answer to whether there exists an instantiation for the type variables in T such that an expression e has type T is not necessarily unique as the examples show. There may be many possible instantiations. A principal solution is an instantiation σ for the free type variables in $\Gamma \vdash e : T$ s.t. any other solution can be obtained from it.

10.9 Type inference

When inferring the type for an expression e, we essentially ask whether there exists an instantiation for the type variable β s.t. e has type β. Informally, the reasoning process can be split in two phases: *inferring a type* for an expression and *checking that certain constraints are satisfied*. For example, to infer the type for $\mathsf{if}\ 3 = 1\ \mathsf{then}\ 55\ \mathsf{else}\ 44$ we *infer the type* T for $3 = 1$, the type T_1 for 55 and the type T_2 for 44. In the second phase we *check* that certain conditions are satisfied, namely $T = \mathsf{bool}$ and $T_1 = T_2$.

An expression e has a type T, if we infer some type T for e and certain constraints are satisfied. We will concentrate first on constraint generation, and then discuss how to solve constraints.

10.9.1 Constraint generation

Constraints can be viewed as a set of equality constraints on types (written as $T_1 = T_2$). To satisfy the set of constraint $C = \{C_1, \ldots, C_n\}$ all the individual constraints C_i must be satisfied.

How do constraints arise? Let us consider as an example, how we infer a type T for an if-expression $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ we will argue as follows:

1. Infer the type T' for expression e.
2. Infer the type T_1 for expression e_1
3. Infer the type T_2 for expression e_2

Then $\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ will have T_1 if $\{T' = \mathsf{bool},\ T_1 = T_2\}$, i.e. the listed constraints can be satisfied.

$$\Gamma \vdash e \Rightarrow T/C \quad \text{Infer type } T \text{ for expression } e \text{ in the typing environment } \Gamma \text{ modulo the constraints } C.$$

Ultimately, we want that expression e will have type T if the constraints C are satisfied. For now however, we will just generate constraints. We will consider the typing rules individually.

Numbers and booleans and variables First the rules for number, booleans, and variables. For these expressions there are no constraints which need to be satisfied.

$$\frac{}{\Gamma \vdash n \Rightarrow \mathsf{int}/\emptyset}\ \text{B-NUM} \qquad \frac{x{:}T \in \Gamma}{\Gamma \vdash x \Rightarrow T/\emptyset}\ \text{B-VAR}$$

$$\frac{}{\Gamma \vdash \mathsf{true} \Rightarrow \mathsf{bool}/\emptyset}\ \text{B-TRUE} \qquad \frac{}{\Gamma \vdash \mathsf{false} \Rightarrow \mathsf{bool}/\emptyset}\ \text{B-FALSE}$$

If-expressions and primitive operations Next, the rule for if-expressions. It follows our recipe from above.

$$\frac{\Gamma \vdash e \Rightarrow T/C \qquad \Gamma \vdash e_1 \Rightarrow T_1/C_1 \quad \Gamma \vdash e_2 \Rightarrow T_2/C_2}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Rightarrow T_1/C \cup C_1 \cup C_2 \cup \{T = \mathsf{bool},\ T_1 = T_2\}}\ \text{B-IF}$$

Similarly, we can define rules for our arithmetic operations.

$$\frac{\Gamma \vdash e_1 \Rightarrow T_1/C_1 \quad \Gamma \vdash e_2 \Rightarrow T_2/C_2}{\Gamma \vdash e_1 + e_2 \Rightarrow \mathsf{int}/C_1 \cup C_2 \cup \{T_1 = \mathsf{int},\ T_2 = \mathsf{int}\}}\ \text{B-PLUS}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow T_1/C_1 \quad \Gamma \vdash e_2 \Rightarrow T_2/C_2}{\Gamma \vdash e_1 = e_2 \Rightarrow \mathsf{bool}/C_1 \cup C_2 \cup \{T_1 = T_2\}}\ \text{B-EQ}$$

Let-expressions The rule for let-expressions is straightforward.

$$\frac{\Gamma \vdash e_1 \Rightarrow T_1/C_1 \quad \Gamma, x{:}T_1 \vdash e_2 \Rightarrow T/C_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end} \Rightarrow T/C_1 \cup C_2}\ \text{B-LET}$$

Functions The most interesting case arises for functions, since we lack the information necessary to determine the type of the input argument. This is resolved by generating a new type variable α.

$$\frac{\Gamma, x{:}\alpha \vdash e \Rightarrow T/C}{\Gamma \vdash \mathsf{fn}\ x\ \texttt{=>}\ e \Rightarrow \alpha\ \texttt{->}\ T/C}\ \text{B-FN}$$

Before we continue, let us consider how we can infer a type for the function $\mathsf{fn}\ x\ \texttt{=>}\ x + 1$. We simplify a little bit along the way and writing instead of $C \cup T$ just C.

$$\frac{\dfrac{\dfrac{}{x{:}\alpha \vdash x \Rightarrow \alpha/\emptyset}\ \text{B-VAR} \quad \dfrac{}{x{:}\alpha \vdash 1 \Rightarrow \mathsf{int}/\emptyset}\ \text{B-NUM}}{x{:}\alpha \vdash x + 1 \Rightarrow \mathsf{int}/\{\alpha = \mathsf{int},\ \mathsf{int} = \mathsf{int}\}}\ \text{B-PLUS}}{\cdot \vdash \mathsf{fn}\ x\ \texttt{=>}\ x + 1 \Rightarrow \alpha\ \texttt{->}\ \mathsf{int}/\{\alpha = \mathsf{int},\ \mathsf{int} = \mathsf{int}\}}\ \text{B-FN}$$

Simplifying a little bit the constraints, this essentially means fn x => x + 1 will have type α -> int if we can satisfy α = int. Note, that our typing derivation will now always succeed! It may also happen that our constraints cannot be satisfied. For example, fn x => x + true will lead to the constraints α = int $\cup$ int = bool. Obviously these constraints cannot be satisfied. We give a general constraint solving algorithm in the next section. For now, let us continue with the typing rules.

Application Let us consider application next. We recursively infer the type for e_1 and e_2, and must impose certain constraints on their type respectively. We introduce a type variable α to extract the return type of the function described by T_1.

$$\frac{\Gamma \vdash e_1 \Rightarrow T_1/C_1 \quad \Gamma \vdash e_2 \Rightarrow T_2/C_2}{\Gamma \vdash e_1\ e_2 \Rightarrow \alpha/C_1 \cup C_2 \cup \{T_1 = (T_2 \text{ -> } \alpha)\}}\ \text{B-APP}$$

Let us briefly summarize. We have so far seen how to collect constraints during type checking. This process will always succeed. However an expression is only well-typed, if we can collect some constraints C (a process which always succeeds) and we can solve the constraints C (a process which may fail).

10.9.2 Solving typing constraints

The general question we are answering in this section is whether there exists an instantiation for the type variables in the constraint C s.t. all constraints occurring in C are satisfied. For example,

- $\{\alpha = \text{int},\ \alpha \text{ -> } \beta = \text{int -> bool}\}$ can be satisfied by instantiating α with int and β with bool.
- $\{\alpha_1 \text{ -> } \alpha_2 = \text{int -> } \beta,\ \beta = \text{bool}\}$ can be satisfied by instantiating α_1 with int and β with bool, and α_2 with bool.
- $\{\alpha_1 \text{ -> } \alpha_2 = \text{int -> } \beta,\ \beta = \alpha_2 \text{ -> } \alpha_2\}$ cannot be satisfied! The first constraint suggests that $\alpha_2 = \beta$, but the second suggests $\beta = \alpha_2$ -> α_2! We can never find an instantiation for β s.t. $\beta = \beta$ -> β!!

Solving constraints of this form is done via *unification*. In general, we say two types T_1 and T_2 are *unifiable* if there exists an instantiation σ for the free type variables in T_1 and T_2 s.t. $[\sigma]T_1 = [\sigma]T_2$, i.e $[\sigma]T_1$ is syntactically equal to $[\sigma]T_2$.

Unification is a general algorithm to determine whether two objects can be made syntactically equal. We will present here a version which will only test whether a set

of constraints is unifiable. We often write $\{C,\ T_1 = T_2\}$ where C denotes a possibly empty sequence of constraints $C_1, \ldots, C_n$.

$$
\begin{array}{lcll}
\{C,\ \mathsf{int} = \mathsf{int}\} & \Longrightarrow & C & \\
\{C,\ \mathsf{bool} = \mathsf{bool}\} & \Longrightarrow & C & \\
\{C,\ (T_1 \texttt{ -> } T_2) = (S_1 \texttt{ -> } S_2)\} & \Longrightarrow & \{C,\ T_1 = S_1,\ T_2 = S_2\} & \\
\{C,\ \alpha = T\} & \Longrightarrow & \{[T/\alpha]C\} & \text{provided that } \alpha \notin \mathsf{FV}(T) \\
\{C,\ T = \alpha\} & \Longrightarrow & \{[T/\alpha]C\} & \text{provided that } \alpha \notin \mathsf{FV}(T)
\end{array}
$$

We can solve the constraints C by transforming C using the rules above. We terminate, when no rules is applicable anymore and we have derived the empty set, i.e. $C \Longrightarrow^* \emptyset$, otherwise we fail. We also not that solving a constraints C using the unification rules always terminates, either showing that C is unifiable or it is not unifiable.

10.9.3 Type inference in practice

In practice, type inference algorithms do not strictly separate the two phases of generating constraints and checking constraints, but rather check the constraints eagerly. This however requires that we propagate instantiations for type-variables.

10.10 Polymorphism and Let-expressions

The type inference algorithm described above can be easily generalized to provide ML-style polymorphism, also known as let-polymorphism. Let us consider again the example from the introduction, where we defined the function double as follows:

$$\mathsf{double} = \mathsf{fn}\ f \texttt{ => } \mathsf{fn}\ x \texttt{ => } f(f(x))$$

Since the type we infer for this function is polymorphic, we should be able to use this function in two ways:

(a) $\mathsf{double}\ (\mathsf{fn}\ x \texttt{ => } x + 2)\ 3$

(b) $\mathsf{double}\ (\mathsf{fn}\ x \texttt{ => } x)\ \mathsf{false}$

However, what will happen when we try to use the same double function with both booleans and numbers? In other words, will we be able to type-check the following piece of code?

$$\begin{array}{ll}\mathsf{let} & \mathsf{d} = \mathsf{fn}\ \mathsf{f} \Rightarrow \mathsf{fn}\ \mathsf{x} \Rightarrow \mathsf{f}(\mathsf{f}(\mathsf{x})) \\ & \mathsf{x} = \mathsf{d}\ (\mathsf{fn}\ \mathsf{x} \Rightarrow \mathsf{x} + 2)\ 3 \\ & \mathsf{y} = \mathsf{d}\ (\mathsf{fn}\ \mathsf{x} \Rightarrow \mathsf{x})\ \mathsf{false} \\ \mathsf{in} & (\mathsf{x}, \mathsf{y})\ \ \mathsf{end}\end{array}$$

The answer is in fact no. This is best explained at a simpler example. What will happen when we try to infer a type for the following expression

$$\mathsf{let}\ \mathsf{f} = \mathsf{fn}\ \mathsf{x} \Rightarrow \mathsf{x}\ \mathsf{in}\ (\mathsf{f}\ 3,\ \mathsf{f}\ \mathsf{true})\ \mathsf{end}?$$

We will first infer the type of $\mathsf{fn}\ \mathsf{f} \Rightarrow \mathsf{x}$ as $\alpha \to \alpha$. Next, we need to infer the type for $(\mathsf{f}\ 3,\ \mathsf{f}\ \mathsf{true})$ in the typing environment where we assume that f has type $\alpha \to \alpha$. For the expression $\mathsf{f}\ 3$ we will infer a type β_0 together with the constraints $\alpha \to \alpha = \mathsf{int} \to \beta_0$, as demonstrated by the following derivation:

$$\frac{\overline{\mathsf{f}{:}\alpha \to \alpha \vdash \mathsf{f} \Rightarrow \alpha \to \alpha / \mathsf{tt}} \qquad \overline{\mathsf{f}{:}\alpha \to \alpha \vdash 3 \Rightarrow \mathsf{int}/\mathsf{tt}}}{\mathsf{f}{:}\alpha \to \alpha \vdash (\mathsf{f}\ 3) \Rightarrow \beta_0 / \alpha \to \alpha = \mathsf{int} \to \beta_0}$$

Next, let us consider the expression $(\mathsf{f}\ \mathsf{true})$. For this expression we will infer a type β_1 together with the constraints $\alpha \to \alpha = \mathsf{bool} \to \beta_1$, as demonstrated by the following derivation:

$$\frac{\overline{\mathsf{f}{:}\alpha \to \alpha \vdash \mathsf{f} \Rightarrow \alpha \to \alpha / \mathsf{tt}} \qquad \overline{\mathsf{f}{:}\alpha \to \alpha \vdash \mathsf{true} \Rightarrow \mathsf{bool}/\mathsf{tt}}}{\mathsf{f}{:}\alpha \to \alpha \vdash (\mathsf{f}\ \mathsf{true}) \Rightarrow \beta_1 / \alpha \to \alpha = \mathsf{bool} \to \beta_1}$$

In order for the tuple $(\mathsf{f}\ 3,\ \mathsf{f}\ \mathsf{true})$ to have some type $\beta_0 \times \beta_1$ we must satisfy the constraints:

$$\{\alpha \to \alpha = \mathsf{int} \to \beta_0,\ \alpha \to \alpha = \mathsf{bool} \to \beta_1\}$$

But this is impossible since α is supposed to be int and bool at the same time! What went wrong? The problem is that we use the same type variable α for both uses of the function f. By doing so we impose a constraint which is too strong. It requires us to use f in the same way, not in different ways. What we'd like is to break this connection, i.e. we would like to associate different types with the variable f. In other words, every time we use f we can instantiate the type variable α as we need.

How can this problem be fixed? – The simplest solution is to change the rule for let-expressions. Recall the rule for let.

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, \mathsf{x}{:}T_1 \vdash e_2 : T}{\Gamma \vdash \mathsf{let}\ \mathsf{x} = e_1\ \mathsf{in}\ e_2\ \mathsf{end} : T}$$

Instead of calculating the type of e_1 and then using the type in inferring the type of e_2, we will just re-calculate the type of e_1 every time we need it. We will just substitute for any occurrence of x in the expression e_2 the expression e_1

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash [e_1/x]e_2 : T}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end} : T}$$

You may wonder whether it is necessary to type-check e_1 in the first premise of this typing rule. The answer is yes. If x does not occur in e_2 we will actually never type check e_1. This is considered bad, since we would like to ensure that every expression is type-checked without considering special cases of whether a bound variable is used or not.

There is however another more serious objection to this proposal. If x is used many times within the body of e_2, then we will re-type-check e_1 multiple times. Since the right hand side itself can contain let-bindings, this typing rule can cause the type-checker to perform an amount of work that is exponential in the size of the original term! To avoid re-type-checking, practical implementations actually use a more clever though equivalent reformulation of the typing rules. To type-check the term $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end}$ we infer the type T_1 of e_1. This can be for example done by using the constraint typing rules to calculate the type S_1 and a set of constraints C_1, and then solving the constraints C_1 to obtain an instantiation σ for the free type variables. We can then obtain the principal type T_1 for e_1 by applying σ to S_1. The type T_1 may still contain free type variables. Next, we *generalize* over the free type variables in T_1. If $\alpha_1, \ldots \alpha_n$ are the free type variables in T_1, then we write $\forall\alpha_1 \ldots \forall\alpha_n.T_1$ for the *principal type scheme* of the expression e_1. Finally, we continue to type check e_2 in the extended context $\Gamma, x{:}\forall\alpha_1 \ldots \forall\alpha_n.T_1$.

$$\frac{\Gamma \vdash e_1 : T_1 \quad \mathsf{generalize}(T_1) = \forall\alpha_1 \ldots \forall\alpha_n.T_1 \quad \Gamma, x{:}\forall\alpha_1 \ldots \forall\alpha_n.T_1 \vdash e_2 : T}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\ \mathsf{end} : T}$$

When using an assumption such as $x{:}\forall\alpha_1 \ldots \forall\alpha_n.T_1$ we can re-instantiate all the type variables α_i appropriately as often as we need to.

The algorithm is much more efficient than the simplistic approach of substituting e_1 into the body of e_2. OCaml's type inference algorithm essentially builds on these described ideas. In practice, type inference is linear in the size of the input program. However, the worst-case complexity of type inference is still exponential as shown by Mairson and independently by Kfoury, Tiuryn and Urzyczyn in 1990. The example they constructed involves using deeply nested sequences of lets in the right-hand sides, which cause the types to grow exponentially.

In a full-blown programming language with let-polymorphism, we need to be a bit careful when we try to combine polymorphism with side-effects. Consider the following OCaml code:

```
let r = ref (fun x -> x) in
  r := (fun x -> x + 1) ; (!r) true
```

Using the algorithm sketched above we would compute the type of `fun x -> x` to be α `->` α. Hence the type of `r` is (α `->` α) `ref`. Since we have type variables, we will generalize and continue to type-check the body with the assumption that `r` is $\forall\alpha.$(`'a -> 'a`) `ref`. Then the type-checker will consider the body of the let-expression. Clearly, the update of storing the function `fn x -> x+1` in the reference cell `r` is ok. Unfortunately, also the expression `(!r) true` is ok, since we only know that `r` is a cell which contains polymorphic functions of type α `->` α. By having generalized over the type variables, we now can use this reference cell with different instantiations for α! But this is clearly WRONG! We have updated the function which is stored in the reference cell `r`, and when we read from `r`, we do not get a polymorphic function, but a function of type `int -> int`, which cannot be applied to `true`!

The problem is that the typing rules have gotten out of sync with the evaluation rules, and do not properly approximate anymore the runtime behaviour! The typing rules see two uses of the reference cell and analyze them under different assumptions. But at run time only one reference cell is actually allocated. The fix used in many programming languages is to restrict the generalization of type variables in the let-expression let $x = e_1$ in e_2 end. Only when the expression e_1 is a syntactic value, we are allowed to generalize. This is often called the *value restriction*.

The value restriction solves our problem at some cost in expressiveness. We can no longer write programs in which the right-hand side of let-expressions can both perform some interesting computation and be assigned a polymorphic type. Surprisingly, this makes hardly any difference in practice. Moreover, OCaml generalizes this value restriction even further; it allows some expressions which are not syntactic values, to be used polymorphic.

Chapter 11

Subtyping - An introduction

We will briefly describe the basics about subtyping. Up to now, we have shown how to enrich a language by adding in new language constructs together with their type. As a very important property we maintained that every expression had a unique type. In contrast, we will now consider subtyping, a feature which cross-cuts across all other language construct and affects our basic principles. In particular, we will not be able to maintain that an expression will have a unique type – an expression can have more than one type.

The goal of subtyping is to refine the existing typing rules so we can accept more programs. Consider the following simple program:

```
1 let area r = 3.14 *. r *. r
```

The type of `area` will be `float -> float`. Hence, we would raise a type error, if we apply this function to an integer. This seems quite annoying, since we should be able to provide an `integer` where a `float` is required.

So how can we enrich our type system to handle subtyping? What effect will this have? Will type checking still be decidable?

Remark: This will be a purely theoretical discussion. OCaml does not offer subtyping.

To extend a type system with subtyping, we first add an inference rule, called *subsumption*:

$$\frac{\Gamma \vdash e : S \quad S \leq T}{\Gamma \vdash e : T}\ T - \mathsf{Sub}$$

The rule says that an expression e is of type T if the expression e has type S and S is a subtype of T. So for example, 4 is of type `float` because 4 has type `int` and `int`

is a subtype of the type `float`. The intuition of subtyping can be made precise with the subtyping principle.

> **Basic subtyping principle:** $S \leq T$
>
> S is a subtype of T if we can provide a value of type S whenever a value of type T is required.

or

> Every value described by S is also described by T.

We will now try to formalize this subtyping relation between S and T. We consider each form of type (base types, tuples, functions, references, etc.) separately, and for each one we will define formally when it is safe to allow elements of one type of this form to be used where another is expected.

First, some relations between base types. Let us assume that in addition to `float` and `int`, we also have a base type `pos`, describing positive integers. We can then define the basic subtyping relations between the base types as follows.

$$\frac{}{\texttt{int} \leq \texttt{float}} \qquad \frac{}{\texttt{pos} \leq \texttt{int}}$$

Clearly, we should be able to provide a positive integer whenever a `float` is required, but our inference rules do not yet allow us to conclude this. To reason about subtyping relations, we will stipulate that subtyping should be *reflexive* and *transitive*.

$$\frac{}{T \leq T}\ \mathsf{S{-}ref} \qquad \frac{S \leq T' \quad T' \leq T}{S \leq T}\ \mathsf{S{-}trans}$$

Product

First, we will consider pairs and cross-products. Let's consider some examples.

`int * int`	$\leq$	`float * float`	yes
`int * int`	$\leq$	`int * float`	yes
`int * int`	$\leq$	`float * int`	yes
`int * float`	$\leq$	`int * float`	yes
`int * float`	$\leq$	`float * int`	NO!

From these examples, we can conclude that the following subtyping relation between products seems sensible.

$$\frac{S_1 \leq T_1 \quad S_2 \leq T_2}{S_1 * S_2 \leq T_1 * T_2}\ \mathsf{S{-}Prod}$$

The subtyping rule for products is *co-variant*.

Records

We have seen records, which essentially were n-ary labeled tuples. Hence we can generalize the rule for products to records as follows:

$$\frac{S_1 \leq T_1 \quad \ldots \quad S_n \leq T_n}{\{x_1 : S_1, \ldots, x_n : S_n\} \leq \{x_1 : T_1, \ldots x_n : T\}} \; \mathsf{S{-}RDepth}$$

Since we compare each element of the record, this is also called *depth subtyping*.

However, we often would like to have a more powerful rule for records. Let us consider an example again, where we apply a function `fun r:{x:int} -> r.x` which extracts the x-component of a record `r` is applied to a record which contains an x-component and a y-component.

```
1  (fun r:{x:int} -> r.x) {x=0; y=1}
```

Clearly, this should be safe, since the function just requires that its argument is a record with a field `x` and its body never even access the y-component! In other words, it should always be safe to pass an argument `{x=0; y=1}` whenever a value of type `{x:int}` is required. Hence, it is always safe to forget some fields This seems to indicate the following subtyping rule for records.

$$\frac{n < k}{\{x_1 : T_1, \ldots, x_k : T_k\} \leq \{x_1 : T_1, \ldots, x_n : T_n\}} \; \mathsf{S{-}Width}$$

It may be surprising that the subtype (the type considered "smaller") is actually the one with more fields. The easiest way to understand this is to view the record type `{x:int}` as describing the "set of all records with at least a field `x` of type `int`". Therefore, `{x=0}` is an element of this type, and so is for example `{x=0, y=1}` and `{x=0, y=true}`

A longer record constitutes a more informative and more demanding specification, and hence describes a smaller set of values.

This rule is also called *width subtyping*.

Finally, we do not care about the order of elements in a record, since every element has a unique label. Hence it should not matter whether we write `{x=0, y=1}` or `{y=1, x=0}`. This leads us to the third subtyping rule for records.

$$\frac{\text{where } \phi \text{ is a permutation}}{\{x_1 : T_1, \ldots, x_k : T_k\} \leq \{x_{\phi(1)} : T_{\phi(1)}, \ldots, x_{\phi(n)} : T_{\phi(n)}\}} \; \mathsf{S{-}Perm}$$

Functions

Since we are working with functions, and functions can be passed as arguments and returned as results, it is natural to ask when a function of type $T \to S$ can be used in

place of a function of type $T' \to S'$. Let's consider again some examples.

```
  (* areaSqr: float -> float *)
let areaSqr (r:float) = r *. r in
  (* fakeArea: float -> int *)
let fakeArea (r:float) = 3 in
  areaSqr 2.2 + 4.2
```

The expression `areaSqr 2.2` has type `float` and `areaSqr` has type `float -> float`. Clearly, we should be able to replace the function `areaSqr` with the function `fakeArea:float -> int`, since whenever a `float` is required it suffices to provide an `int`. This seems to suggest the following rule for functions:

$$\frac{S_2 \leq T_2}{S \to S_2 \leq S \to T_2}\ \text{S—Fun—Try1}$$

Let's modify the program from above a little bit.

```
  (* areaSq: int -> int *)
let areaSq (r:int) = r *. r in
  (* fakeArea: float -> int *)
let fakeArea (r:float) = 3 in
  areaSq 2 + 2
```

The expression `areaSq 2` has type `int` and `areaSq` has type `int -> int`.

What happens if we replace `areaSq` with the function `fakeArea`? Clearly this is safe, because it is always safe to pass an integer to a function of type `float -> int`. This seems to suggest that we can provide a function of type `float -> int` whenever a function of type `int -> int` is required. This seems to suggest the following rule for functions.

$$\frac{T_1 \leq S_1}{S_1 \to S \leq T_1 \to S}\ \text{S—Fun—Try2}$$

Notice that the sense of subtyping is reversed (contra-variant) for the argument type!

Combining the two rules into one, we derive at our subtyping rule for functions.

$$\frac{T_1 \leq S_1 \quad S_2 \leq T_2}{S_1 \to S_2 \leq T_1 \to T_2}\ \text{S—Fun}$$

Subtyping for functions is *contra-variant* in the input argument and *co-variant* in the result.

References

References are slightly tricky, since we can use a value of type `ref T` for reading and writing. Let us repeate the subtyping principle:

> Suppose the program context expects a value of type T, then we can provide a value of type S where $S \leq T$.

Example 1 If we have a variable `x` of type `ref float`, then we know that whenever we read from `x` we will have `!x:float`.

```
let x = ref 2.0 in
let y = ref 2           in (!x) * 3.14
```

But intuitively, it should be safe to replace `!x` with `(!y):int`, since whenever a value of type `float` is required it suffices to provide a value of type `int`. This seems to suggest that whenever we need a value of type `float ref` it suffices to provide a value of type `int ref`.

$$\frac{S \leq T}{S\ \texttt{ref} \leq T\ \texttt{ref}}\ \text{S—Ref—Try1}$$

On the other hand, we can not only use references for reading but also for writing.

```
let x = ref 2.0 in
let y = ref 2     in  y := 4
```

Intuitively, it should also be safe to replace the assignment `y := 4` with the assignment `x := 4`, since `x` is a reference cell containing values of type `float` it should be safe to store a value of type `int`. But this suggests that when we require a value of type `int ref`, it suffices to provide a value of type `float ref`. So in other words this suggest the following rule:

$$\frac{T \leq S}{S\ \texttt{ref} \leq T\ \texttt{ref}}\ \text{S—Ref—Try2}$$

However, now we have a problem. The two suggested rules are (almost) contradictory! To achieve a sound rule for subtyping in the presence of references, we cannot allow any subtyping to happen.

$$\frac{T \leq S \quad S \leq T}{S\ \texttt{ref} \leq T\ \texttt{ref}}\ \text{S—Ref}$$

This means references are co-variant *and* contra-variant, in other words they are *invariant*. S and T must be the same type.

The importance of being coherent In OCaml, it would be convenient if int $\leq$ string and float $\leq$ string: then, instead of constantly writing `Int.toString` and `Float.toString`, as in `"i = " ^ Int.toString i ^ "; x = " ^ Float.toString x`, we could write

```
    "i = " ^ i ^ "; x = " ^ x
```


This gives a chain of subtypings:

$$\mathsf{int} \leq \mathsf{float} \leq \mathsf{string}$$

where at each link in the chain, hidden coercions convert values of the subtype into values of the supertype.

But it's not quite that simple. The first coercion from int to float is `float` , and the second coercion from float to string is `string_of_float`. So, inserting coercions in the obvious way into the expression above gives

```
1    "i = " ^ (string_of_float (float i)) ^ "; x = " ^ (string_of_float x)
```

which doesn't give the same result as the boring OCaml expression: instead of `string_of_float` we have `string_of_float (float i))`. So, if the original OCaml expression evaluated to `"i = 77; x = 2.5"`, the coerced expression would evaluate to `"i = 77.0; x = 2.5"`.

We could solve this particular problem by adding *another* coercion `string_of_int` directly from int to string. But we would have to make sure the compiler follows a convention that it use the shortest sequence of coercions, or special-case a preference for `string_of_int` over `string_of_float o float`. Now the programmer might have to reason nontrivially about which coercions are applied. Whenever there is more than one sequence of coercions to get from a subtype to a supertype, we say that subtyping is *incoherent*.

A nastier example of incoherence is a well-intentioned attempt to make programming more convenient by having, besides int $\leq$ float, the converse float $\leq$ int. Yes, above we treated this as being false or nonsensical, but why not just have a coercion `Float.toInt IEEEFloat.TO_NEAREST`? Well, now we could, logically, coerce float to int and then back to float, silently turning 2.3 into 2.0. So again we need to invoke a shortest-sequence convention, or a "no cycles" convention.

A few words on subtyping in Java Subclassing, a central feature of object-oriented languages, is a form of subtyping. In principle, this is perfectly reasonable, sound, and efficient. In practice, many object-oriented languages handle subtyping very imperfectly. The root of the problem is the combination of mutable state—arrays and objects with mutable fields (or *members*)—with subtyping.

When we add subtyping to an ML-like language, we resign ourselves to having no useful subtyping on references: S ref $\leq$ T ref if and only if S $\leq$ T and T $\leq$ S, that is, if S and T are equivalent types with the same set of values. This is the *only* statically sound formulation of the rule![1] This is highly restrictive, but the restriction

[1] Besides the even more restrictive rule S ref $\leq$ S ref.

is tolerable: While ML is an "impure" functional language—it *does* have mutable references, uncontrolled I/O, and other "imperative" features—it is still a functional language. There is often no need whatsoever to use those impure features, and when they are used, they often don't call for subtyping. (Think of using a reference as a counter, or a flag to control debugging.)

By contrast, in Java mutable state is used everywhere, so using a statically sound rule isn't practical. Instead, Java resorts to run-time checks to keep objects of the wrong type from being used.

An array is a collection of individually mutable elements. Logically, it is equivalent to a collection of mutable references. If Direct_Itinerary is a subtype (subclass) of Itinerary, then:

- code that expects an array of Itinerary can always *get* elements from a Direct_Itinerary array, because Direct_Itinerary is a subtype of Itinerary, but
- the same code cannot necessarily *put* elements into the Direct_Itinerary array, because it might try to write an Itinerary that is not a Direct_Itinerary. It can only reliably put an element that is a subtype of Direct_Itinerary.

The statically sound subtyping rule for arrays is, therefore, the same as for references:

$$\frac{S \leq T \qquad T \leq S}{(S \text{ array}) \leq (T \text{ array})}$$

However, this would prevent us from, for example, passing an array of Direct_Itinerary to a function that prints an Itinerary array, even if that function only reads from the array. So instead of the statically sound covariant-contravariant (or "bivariant" or "invariant") rule above, Java uses a covariant rule:

$$\frac{S \leq T}{(S \text{ array}) \leq (T \text{ array})}$$

This rule is statically unsound, but Java is supposed to be (unlike, say, C++) a *safe* language: Java programs should never crash in a completely uncontrolled way. So, whenever a Java program writes to an array of objects, it must compare the "tag" of the object being put in the array to a tag on the array itself, to make sure they are compatible. If we write an Itinerary to an array of Direct_Itinerary, the check that the class represented by the tag Itinerary is a subclass of the Direct_Itinerary will fail, and Java will raise an exception (which is bad, but better than segfaulting).

Unfortunate fact: In Java we omit the first premise during type checking. This is generally considered a flaw. Java fixes this flaw by inserting a runtime check for every assignment to an array thereby achieving type safety. This leads to a performance penalty.

Final remarks

Subtyping is characteristically found in *object-oriented* languages and is often considered essential. Generally, we find *nominal* subtyping in object-oriented systems, which is in contrast to *structural* subtyping, the subtyping we have considered here. In *nominal* subtyping, the user specifies what type (class) extends another type (class).

In Java, we often abuse subtyping. For example, elements in lists belong to the maximal type object in order to provide operations on for all lists, no matter what their elements are. If we add an element `x:int` to a list, we promote it to type object (upcast). If we take it out again, we unfortunately also only know it is an object. To do reasonable things with it, we must downcast it to the appropriate type. This however is an abuse of subtyping and often considered a "poor man's polymorphism". Newer versions of Java include some form of polymorphism, named generics, to avoid this kind of situation.

Up and down casts

Earlier we saw the subsumption rule.

$$\frac{\Gamma \vdash e : S \quad S \leq T}{\Gamma \vdash e : T}\ T-\mathsf{Sub}$$

This rule is completely unguided, and will always be applicable if we are trying to check or infer a type for an expression e. Moreover, it is not obvious that the subtyping relation itself is decidable. One easy way to achieve decidability, is to annotate the subsumption rule with a type. Typically, we have upcasts and downcasts.

$$\frac{\Gamma \vdash e : S \quad S \leq T}{\Gamma \vdash (T)e : T}\ T\text{—}\mathsf{Up} \qquad \frac{\Gamma \vdash e : S}{\Gamma \vdash (T)e : T}\ T\text{—}\mathsf{Down}$$

In an upcast, we ascribe a supertype of the type the type-reconstruction algorithm would find. This is always safe. It is useful if we want to hide some information in a record for example. In a downcast, we have no demand on the relationship between S and T. For example, we can write the following code:


```
1  fun (x:float) -> (int)x + 1
```

This is in general dangerous, because not all values of type `float` can be safely cast to an integer. In effect, the programmer says, "I know why this can't happen – trust me!". To remain safe, a compiler must verify the typing constraint during run-time.

Bibliography

