Programming languages, semantics, compilers

Thibaut Balabonski @ Université Paris-Saclay. M1 MPRI, Fall 2024.

This course explores programming languages, focusing on two main topics:
- their ***semantics***, that is the formal description of the meaning of programs ;
- their ***compilation***, that is the decomposition of high-level source language programs into simpler instructions whose execution can be performed by a computer.

We will define a functional programming language with a rich type system, and build an optimizing compiler and an execution environment for this language.

Part I

Contents

1 Semantics and interpretation of a functional language

In this chapter, we define and interpret a minimal functional programming language, called FUN. Here is a sample FUN program.

```
let rec fact = fun n ->
  if n = 0 then
    1
  else
    n * fact (n-1)
in
fact 6
```

Programs in this language are made of expressions, combining basic arithmetic capabilities with richer elements, such as the definition and use of (possibly recursive) functions.

1.1 Concrete syntax and abstract syntax

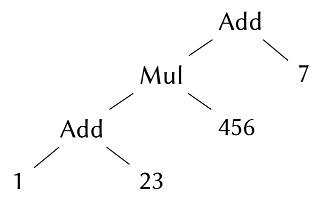
The arithmetic features of the FUN language enable a variety of numerical calculations. For instance:

```
> (1+23)*456+7
> (1 + 23) * 456 + 7
> ( 1+23 ) *456   +7
```

These three different writings all represent the same arithmetic expression $(1 + 23) \times 456 + 7$.

We consider two levels of syntax for any programming language. The ***concrete syntax*** corresponds to what is written by the programmer. It is a raw text, a character string, and represents the observable part of the language. Conversely, the ***abstract syntax*** gives a structured representation of a program, shaped as a tree. It is a central tool both for handling programs inside a compiler, and for reasoning on programs in a formal way. Moreover, the abstract syntax tends to focus on the heart of the language, abstracting away many writing artifacts.

Differences between the two levels of syntax. Unsignificant elements of the concrete syntax, such as spaces and comments, are not carried over to the abstract syntax. Parentheses, become in the concrete syntax for the correct association between operators and their operands, become useless in the intrinsically structured abstract syntax. Hence, the three strings of characters seen above correspond to the same arithmetic abstract syntax tree.



The concrete syntax of a programming language often provides simplified writings for common operations, named *syntactic sugar*. These simplified forms however do not entail any new structure in the abstract syntax: they are just combinations of core elements of the language. For instance:

- in python, the increment instruction `x += 1` is a shortcut for the assignment instruction `x = x + 1` and produces the very same abstract syntax tree;
- in C, the access instruction `t[i]` is nothing more than a small pointer arithmetic expression `*(t+i)` ;

This syntactic sugar as some funny consequences: it allows writing `2[t]` for the same effect as `t[2]`.

- in caml, a function definition `let f x = e` is actually an ordinary variable definition, whose value is provided by an anonymous function: `let f = fun x -> e`; similarly the definition of a two-parameter function `let f x y = e` is decomposed as `let f = fun x -> fun y -> e`.

1.2 Inductive structures

Abstract syntax tree have an ***inductive*** structure, which can be describe using a form of recursion: a program is built by combining program fragments, themselves build by combining smaller fragments, an so on. This structure allows defining *recursive functions* working on abstract syntax tree, and reasoning on programs using *structural induction.*

Inductive objects. We define a set of inductive objects with:

1. some ***base objects***,
2. a finite set of ***constructors***, that can combine already built objects to define new objects.

We then consider the set of all objects that can be built using the two previous points. The ***arity*** of a constructor is the number of elements it combines. Base objects can be seen as constructors with arity zero. The ***signature*** of a set of inductive objects is the set comprising its base objects and its constructors.

Example: lists. Linked lists can be seen as a set of inductive objects defined by:

— a unique base object: the empty list, written [],

— a unique constructor, which builds a new list $e : \ell$ by adding a new element e at the head of a list ℓ.

Example: arithmetic expressions. We define a minimal set of arithmetic expression $\mathbb{A}$ with:

— the integer constants as base objects,

$$0 \qquad 1 \qquad 2 \qquad 3 \qquad \cdots$$

— some ***binary*** constructors, each combining *two* already built expressions, such as addition or multiplication.

Add
e_1
e_2
Mul
e_1
e_2

On can lighten the handling of such expression using mathematical notations such as n, $e_1 \oplus e_2$ or $e_1 \otimes e_2$. For now, we keep symbols $\oplus$ and $\otimes$ that are different from the usual $+$ and $\times$, to unambiguously distinguish the language and its interpretation.

— The operator $+$ is a mathematical element, which applies to two numbers to define a third. It satisfies the equation $1 + 2 = 3$ and will serve for the interpretation of our language of expressions.

— The constructor $\oplus$ is a syntactic element, which applies to two expressions to build a third. It serves at defining the language itself, in which $1 \oplus 2 \neq 3$.

This light notation for the abstract syntax must always use enough parentheses, to remove any ambiguity in the structure of expressions. Thus we forbid $e_1 \oplus e_2 \oplus e_3 \oplus e_4$, and write one of the five possible structures instead, such as $(e_1 \oplus e_2) \oplus (e_3 \oplus e_4)$ or $e_1 \oplus ((e_2 \oplus e_3) \oplus e_4)$. Similarly, we do not let any of the usual mathematical priority conventions implicit: we explicitly write $1 \oplus (2 \otimes 3)$ when representing the usual mathematical expression $1 + 2 \times 3$.

Defining functions on a set of inductive objects. Each object of a set E of inductive objects can be built using only the base objects and the constructors. A function f applicable to the elements of E can thus be defined in a very succinct way:

— for each base element e, give $f(e)$,

— for each n-ary constructor c, describe $f(c(e_1, \dots, e_n))$ using the subelements e_i and their images $f(e_i)$.

This define a unique image for each element of E.

We provide below three sets of equations on our arithmetic expressions. These equations define functions nbCst, nbOp and eval, such that nbCst(e) gives the number of constants in the arithmetic expression e, nbOp(e) gives the number of operators in e, and eval(e) gives the numerical value obtained after performing the calculation described by e. Notive that writing such functions requires a clear distinction between arithmetic expressions themselves (the *syntax*) and the associated value (the *semantics*). In particular, the constructor $\oplus$ is a syntactic element (a constructor) representing an addition, that should not be confused with the operator + which is the mathematical interpretation of the addition (a function).

$$\begin{cases} \mathsf{nbCst}(\mathsf{n}) &= 1 \\ \mathsf{nbCst}(e_1 \oplus e_2) &= \mathsf{nbCst}(e_1) + \mathsf{nbCst}(e_2) \\ \mathsf{nbCst}(e_1 \otimes e_2) &= \mathsf{nbCst}(e_1) + \mathsf{nbCst}(e_2) \end{cases}$$

$$\begin{cases} \mathsf{nbOp}(\mathsf{n}) &= 0 \\ \mathsf{nbOp}(e_1 \oplus e_2) &= 1 + \mathsf{nbOp}(e_1) + \mathsf{nbOp}(e_2) \\ \mathsf{nbOp}(e_1 \otimes e_2) &= 1 + \mathsf{nbOp}(e_1) + \mathsf{nbOp}(e_2) \end{cases}$$

$$\begin{cases} \mathsf{eval}(\mathsf{n}) &= n \\ \mathsf{eval}(e_1 \oplus e_2) &= \mathsf{eval}(e_1) + \mathsf{eval}(e_2) \\ \mathsf{eval}(e_1 \otimes e_2) &= \mathsf{eval}(e_1) \times \mathsf{eval}(e_2) \end{cases}$$

Programming with inductive objects. Algebraic types in caml allow precisely the definition of sets of inductive objects, by providing a set of constructors and, for each constructor, the types of the elements it combines. Here a base object is just seen as a constructor with arity zero.

For instance, lists containing elements of type `'a` are defined by the base object `[]`, and a constructor `::` that applies to an element and a list.

```
type 'a list =
  | []
  | (::) of 'a * 'a list
```

Expressions are defined similarly, using three constructors.

```
type expr =
  | Cst of int
  | Add of expr * expr
  | Mul of expr * expr
```

Remark: we cannot define in caml an infinite amount of base objects. Thus we describe all integer constants using a unique constructor `Cst` that takes an integer as parameter. With such a definition, the abstract syntax expression $(1 \oplus 2) \oplus (3 \otimes 4)$ can be defined in caml by:

```
Add(Add(Cst 1, Cst 2), Mul(Cst 3, Cst 4))
```

With such algebraic types, the equations written above to define functions on arithmetic expressions translate straightforwardly into code, in the form of recursive functions.

```
let rec nb_cst = function
  | Cst n       -> 1
  | Add(e1, e2) -> nb_cst e1 + nb_cst e2
  | Mul(e1, e2) -> nb_cst e1 + nb_cst e2

let rec eval = function
  | Cst n       -> n
  | Add(e1, e2) -> eval e1 + eval e2
  | Mul(e1, e2) -> eval e1 * eval e2
```

Additional note on caml: parts of the expression that are not used in the computation can be ignored, and identical cases can be factored:

```
let rec nb_op = function
  | Cst _                     -> 0
  | Add(e1, e2) | Mul(e1, e2) -> nb_op e1 + nb_op e2
```

Reasoning by structural induction. Each object in a set E of inductive objects can be built using only the base objects and the constructors. Proving that a given property E is valid for all elements in E reduces to:

- proving that $P(e)$ is valid for each base element e,
- proving, for each n-ary constructor c, that if any n element e_1, ..., e_n all satisfy $P(e_i)$, then the property P is still valid for the combined element $c(e_1, \dots, e_n)$. In other words, for any constructor and any elements, $P(e_1) \wedge \dots \wedge P(e_n) \implies P(c(e_1, \dots, e_n))$.

Remark that this paragraph is very similar to the one concerning the definition of a function!

Thus, we ensure that it is not possible to build an element e that does not satisfy the target property P.

This proof technique is called proof by ***structural induction***. The first point describe ***base cases*** (one for each base element). The second point describe ***inductive cases*** (or *recursive* cases, one for each non-nullary constructor). In the second point hypotheses $P(e_1)$ to $P(e_n)$ that can be used for justifying $P(c(t_1, \dots, t_n))$ are called ***induction hypotheses***.

Here is how this principle can be instantiated for our two examples.

— Proving that a property P is valid for all lists reduces to:
  1. proving that it is valid for the empty list [],
  2. for any list ℓ and any element e, proving that if P is valid for ℓ (induction hypothesis), then it is still valid for $e : \ell$.

— Proving that a property P is valid for all arithmetic expressions reduces to:
  1. proving that it is valid for all integer constants,
  2. for any expressions e_1 and e_2, proving that if P is valid for e_1 and e_2 (induction hypotheses), then it is still valid for $e_1 \oplus e_2$,
  3. for any expressions e_1 and e_2, proving that if P is valid for e_1 and e_2 (induction hypotheses), then it is still valid for $e_1 \otimes e_2$.

Let us prove that for any arithmetic expression, the number of constants is exactly one more than the number of binary operators. Let us write $P(e)$ the property $\mathsf{nbCst}(e) = \mathsf{nbOp}(e) + 1$, and check the base and the inductive cases:

— Case of a constant (base case): for any constant n we have $\mathsf{nbCst}(\mathsf{n}) = 1$ and $\mathsf{nbOp}(\mathsf{n}) = 0$. Then the property P is satisfied by the term n.

— Case of an addition (inductive case): let e_1 and e_2 be two expressions satisfying the property P. Then

$$\begin{array}{lll} & \mathsf{nbCst}(e_1 \oplus e_2) & \\ = & \mathsf{nbCst}(e_1) + \mathsf{nbCst}(e_2) & \text{by definition of nbCst} \\ = & (\mathsf{nbOp}(e_1) + 1) + (\mathsf{nbOp}(e_2) + 1) & \text{by induction hypotheses} \\ = & (1 + \mathsf{nbOp}(e_1) + \mathsf{nbOp}(e_2)) + 1 & \text{(reorder)} \\ = & \mathsf{nbOp}(e_1 \oplus e_2) + 1 & \text{by definition of nbOp} \end{array}$$

Thus, the property P is still valid for the term $\mathrm{Add}(e_1, e_2) = e_1 \oplus e_2$.

— Case of a multiplication (inductive case): similar to the case of an addition.

Thus, using structural induction we proved that for any arithmetic expression e, we have $\mathsf{nbCst}(e) = \mathsf{nbOp}(e) + 1$.

1.3 An interpret for FUN

Now we will apply the principles seen above to the language FUN, which can be seen as the core of functional programming. This languages contains richer expressions, with in particular variables, conditional expressions, and definition and application of possibly recursive functions.

Abstract syntax. A FUN program is made of expressions only. Its abstract syntax is represented by a unique main type `expr`, with a constructor for each syntactic form. We factor all binary operations using a unique constructor `Bop`, whose first parameter gives the precise operation.

```
type bop = Add | Sub | Mul | Lt | Eq (* | ... *)
type expr =
  (* arithmetic *)
  | Int of int
  | Bop of bop * expr * expr
```

We add constructors `Var` for referring to the value of a variable, and `Let` for defining a local variable. Variables are identified by character strings.

```
  (* variables *)
  | Var of string
  | Let of string * expr * expr
```

Then, the expression `let x = 41 in x+1` of FUN is represented in caml by `Let("x", Int 41, Bop(Add, Var "x", Int 1))`.

We enrich the language with a ternary constructor `If` for conditional expressions, a constructor `Fun` for the creation of an anonymous function `fun x -> e`, and a constructor `App` for the application of a function to an argument.

```
    (* conditional *)
    | If  of expr * expr * expr
    (* functions *)
    | Fun of string * expr
    | App of expr * expr
    (* recursion *)
    | Fix of string * expr
```

Finally, the constructor `Fix` describes a recursive definition. The definition of a recursive function `let rec f x = e1 in e2` will be represented in caml by the abstract syntax tree `Let("f", Fix("f", Fun("x", e1)), e2)`. Note that the identifier `"f"` of the function appears twice here: once in the constructor `Let` for defining this name `f` in the expression `e2`, and once in the constructor `Fix` to enable the (recursive) use of `f` in the expression `Fun("x", e1)`.

Variables, values and environments. A variable denotes a value that has been computed by another part of the program. A function interpreting programs that may contain variables then requires two parameters: the expression that should be evaluated first, but also the values associated to the variables of this expression. This second part is called an ***environment***. It associates variables names (that is, character strings) with values.

To handle such an environment, we need to define the values that can be produced by the evaluation of an expression. Let us first focus on arithmetic and logic, for which we can distinguish numerical and boolean values.

```
  type value =
    | VInt  of int
    | VBool of bool
```

We then need a structure of ***association table***, which can be implemented by several data structures. In particular:
- balanced search trees (module `Map` in caml) can be used to implement an environment using an immutable structure, in a purely functional style,
- hashtables (module `Hashtbl` in caml) give an implementation based on a mutable data structure.

In the code of this chapter, we use the implementation based on balanced search trees, which can be setup with the following declarations.

```
  module Env = Map.Make(String)
  type env = value Env.t
```

After this declaration, the type `env` represents association tables with keys of type `string` and values of type `value`. The module `Env` provides a constant `Env.empty` for an empty table, and many functions, including `Env.find` for fetching the value associated to a given key, or `Env.add` for adding or updating an association.

Now we can define a function

```
  eval: expr -> env -> value
```

such that `eval` e ρ returns the value of the expression e evaluated in the environment ρ.

The treatment of basic arithmetic operators is similar to what we have already seen. The novelty is that each value has to be encapsulated using the appropriate constructor `VInt` or `VBool`, and that the kind of each operand also has to be checked.

```
  let rec eval e env = match e with
    | Int n -> VInt n
    | Bop(op, e1, e2) ->
       begin match op, eval e1 env, eval e2 env with
         | Add, VInt n1, VInt n2 -> VInt (n1 + n2)
         | Sub, VInt n1, VInt n2 -> VInt (n1 - n2)
         | Mul, VInt n1, VInt n2 -> VInt (n1 * n2)
         | Lt,  VInt n1, VInt n2 -> VBool (n1 < n2)
         | Eq,  v1,      v2      -> VBool (v1 = v2)
         | _ -> failwith "unauthorized operation"
```

```
      end
    | If(c, e1, e2) ->
      begin match eval c env with
        | VBool b -> if b then eval e1 env else eval e2 env
        | _ -> failwith "unauthorized operation"
      end
```

Evaluating a variable means fetching the associated value in the environment. The declaration of a new local variable with `let x = e1 in e2` defines an extended environment which associates `x` to the value of `e1`, and then evaluates the expression `e2` in this new environment.

```
    | Var x -> Env.find x env
    | Let(x, e1, e2) ->
      let v1 = eval e1 env in
      let env' = Env.add x v1 env in
      eval e2 env'
```

We deduce a function `eval_top` that evaluates an expression in the empty environment.

```
  let eval_top (e: expr): value =
    eval e Env.empty
```

Functions and functional closures. With functional programming, functions are seen as ordinary values, which can be passed to other functions as parameters, or returned as results. Their is a little subtlety though.

```
  let plus n =
    let f x = x + n in
    f
```

Here, a function `plus` defines *and returns* a local function `f`. The definition of `f` uses a variable `n` which is external to `f` (it is called a ***free variable***). Here, this variable `n` is a parameter of the function `plus`. Two calls `plus 2` and `plus 3` define two differents functions. Both correspond to the code `fun x -> x + n`, however we have `n = 2` in the former case, and `n = 3` in the latter.

We give a name to this function `f` for clarity, but the caml codes `let plus n = fun x -> x + n` or `let plus n x = x + n` would have produced the very same effect.

Thus, the saying that in functional programming a function is an ordinary value is slightly simplified: the *value* returned by our function `plus` is not only the function `f`, but rather "the function `f` together with an enrivonment providing the value of the variable `n` that `f` refers to". More generally, a function-value is a function together with an enrivonment providing at least the values of all the external variables used in the function (a simple version would be to keep the full environment in which the function has been defined, regardless of what is actually used). We call this function/environment pair a ***functional closure***.

We extend the type `value` of the possible results of the evaluation of an expression. In addition to numbers and booleans, it now contains functional closures, with the constructor `VClos`.

```
  type value = ...
    | VClos of string * expr * env
```

The value corresponding to the function `fun x ->` e defined in the environment ρ is represented by `VClos("x",` e, ρ`)`.

Now we can extend our evaluation function, with two cases for the definition and the application of a function. An anonymous function `fun x -> e` produces immediately a value: it is just paired with the current environment to form a closure. In the case of an application, we expect the value of the left member `e1` to be a functional closure. Then we evaluated the body `e` of this function under the new environement combining: the environment `env'` given by the closure (necessary for evaluating the external variables in the body of the function), and the value of the argument `e2` associated to the formal parameter `x` of the function.

```
  let rec eval e env = match e with
    ...
    | Fun(x, e) -> VClos(x, e, env)
    | App(e1, e2) ->
      let x, e, env' = match eval e1 env with
        | VClos(x, e, env') -> x, e, env'
        | _ -> failwith "unauthorized operation"
      in
      let v2 = eval e2 env in
      eval e (Env.add x v2 env')
```

Recursion. A recursive function *f*, like any ordinary function, evaluates to a functional closure $c = (f, \rho)$. Some special care is required however: for the function to be able to call itself recursively, we need the environment ρ of the closure *c* to contain *c* itself, besides other external elements. Ideally, we would like

```
  eval (Fix(f, Fun(x, e))) env
```

to produce a value `v` satisfying the recursive equation

```
  v = VClos(x, e, Env.add f v env)
```

However, although caml authorize in some circumstances the definition of recursive values, the definition

```
  let rec v = VClos(x, e, Env.add f v env)
```

would be rejected. Indeed, since here the value `v` would be, in the process of its own definition, passed as parameter to another function (namely `Env.add f`), the compiler cannot guarantee that the process is well defined.

For instance, what would happen if the function tried to explore the structure of `v` before it is sufficiently defined?

To circumvent this problem, we introduce a new shape of value `VFix`, used to tie the recursive knot.

```
  type value =
    ...
    | VFix of expr * string * value * env
```

Given an expression *e*, an identifier *f*, and an environment ρ, the recursive value v = `VFix(`e, f, v, ρ`)` is to be understood has the result of evaluating *e* in the environment `Env.add` f v ρ.

We complete our `eval` function with three elements:

— an evaluation rule for `Fix`, which produces a recursive value `VFix`, assuming the considered subexpression is a function,
— an auxiliary function `force: value -> value` which "opens" a recursive value,
— a use of `force` in the evaluation of a function application, to unpack a possibly recursive function.

Here are the parts that are added or modified:

```
  let rec eval e env = match e with
    ...
    | App(e1, e2) ->
       let x, e, env' = match force (eval e1 env) with
         | VClos(x, e, env') -> x, e, env'
         | _ -> failwith "unauthorized operation"
       in
       let v2 = eval e2 env in
       eval e (Env.add x v2 env')
    | Fix(f, Fun(x, e)) ->
       let rec v = VFix(Fun(x, e), f, v, env) in
       v
    | Fix _ -> failwith "unauthorized operation"

  and force v = match v with
    | VFix(e, f, v, env) -> force (eval e (Env.add f v env))
    | v -> v
```

1.4 Natural semantics

We provided a semantics for FUN through an interpreter written in caml. However, this semantics itself depends on the semantics of caml! Here, we take a more abstract view, with a direct mathematical description.

The ***semantics*** defines the meaning and the behaviour of programs. A programming language generally comes with a more or less informal description of the program behaviours that shall be expected. Here is an excerpt that appeared in the specification Java:

> *The Java programming language guarantees that the operands of operators appear to be evaluated in a specific order, namely, from left to right. It is recommended that code do not rely crucially on this specification.*

This kind of documentation often contains some quantity of imprecise description or ambiguities. However, we can also equip a language with a ***formal semantics***, a mathematical characterization of the computation described by a program. This more rigorous setting allows us to *reason* on the execution of programs.

Equational semantics. Earlier in this chapter, we have seen how to define an interpretation function for the expressions of a programming language, that an eval function which, given an expression e and an environment ρ associating values to the free variables of e, returns the expected result of evaluating e.

Here are some equations describing such a function, for a fragment of our FUN language.

$$\begin{array}{rcl}
\mathsf{eval}(n,\rho) & = & n \\
\mathsf{eval}(e_1\ \texttt{+}\ e_2,\rho) & = & \mathsf{eval}(e_1,\rho) + \mathsf{eval}(e_2,\rho) \\
\mathsf{eval}(x,\rho) & = & \rho(x) \\
\mathsf{eval}(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2,\rho) & = & \mathsf{eval}(e_2,\rho\cup\{x\mapsto\mathsf{eval}(e_1,\rho)\}) \\
\mathsf{eval}(\texttt{fun}\ x\ \texttt{->}\ e,\rho) & = & \mathsf{Clos}(x,e,\rho) \\
\mathsf{eval}(e_1\ e_2,\rho) & = & \mathsf{eval}(e,\rho'\cup\{x\mapsto\mathsf{eval}(e_2,\rho)\}) \\
 & & \text{if } \mathsf{eval}(e_1,\rho)=\mathsf{Clos}(x,e,\rho')
\end{array}$$

Remark in the equation concerning addition that the symbol + in $e_1\ \texttt{+}\ e_2$ is a syntactic element combining two FUN expressions, whereas the operator + in $\mathsf{eval}(e_1,\rho)+\mathsf{eval}(e_2,\rho)$ is the mathematical addition of the values v_1 and v_2 produces by the evaluation of the expressions e_1 and e_2. Also, the construct $\mathsf{Clos}(x,e,\rho)$ denotes a functional closure, that is a function together with its environment.

The environment ρ taken as parameter by this evaluation function was a device whose goal was to build an efficient interpreter, using caml data structures for associating variables and their values. For a purely mathematical specification of the value that should be produced by the evaluation of an expression in a purely functional setting, we can instead deal with expressions in which each variable is literally replaced by its value. By doing this replacement we totally bypass the notion of environment, as well as the associated necessity to deal with closures. We could have had a definition like

$$\begin{array}{rcl}
\mathsf{eval}(n) & = & n \\
\mathsf{eval}(e_1\ \texttt{+}\ e_2) & = & \mathsf{eval}(e_1) + \mathsf{eval}(e_2) \\
\mathsf{eval}(x) & = & \text{indéfini} \\
\mathsf{eval}(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2) & = & \mathsf{eval}(e_2[x:=\mathsf{eval}(e_1)]) \\
\mathsf{eval}(\texttt{fun}\ x\ \texttt{->}\ e) & = & \texttt{fun}\ x\ \texttt{->}\ e \\
\mathsf{eval}(e_1\ e_2) & = & \mathsf{eval}(e[x:=\mathsf{eval}(e_2)]) \\
 & & \text{if } \mathsf{eval}(e_1)=\texttt{fun}\ x\ \texttt{->}\ e
\end{array}$$

where $e[x:=e']$ denotes the replacement (called ***substitution***) of each occurrence of the variable x in the expression e by the other expression e', and is defined by its own set of equations.

$$\begin{array}{rcll}
n[x:=e'] & = & n & \\
(e_1\ \texttt{+}\ e_2)[x:=e'] & = & e_1[x:=e']\ \texttt{+}\ e_2[x:=e'] & \\
y[x:=e'] & = & \begin{cases} e' & \text{if } x=y \\ y & \text{otherwise}\end{cases} & \\
(\texttt{let}\ y\ \texttt{=}\ e_1\ \texttt{in}\ e_2)[x:=e'] & = & \texttt{let}\ y\ \texttt{=}\ e_1[x:=e']\ \texttt{in}\ e_2[x:=e'] & \text{if } x\neq y \text{ et } y\notin\mathsf{fv}(e') \\
(\texttt{fun}\ y\ \texttt{->}\ e)[x:=e'] & = & \texttt{fun}\ y\ \texttt{->}\ e[x:=e'] & \text{if } x\neq y \text{ et } y\notin\mathsf{fv}(e') \\
(e_1\ e_2)[x:=e'] & = & e_1[x:=e']\ e_2[x:=e'] &
\end{array}$$

Remark in the cases for `let` and `fun` a side condition concerning the set $\mathsf{fv}(e')$ of free variables of the substitution expression. This side condition is here to avoid having different variables whose name collide. Here, it will be automatically satisfied as soon as all the variables introduced in a program are given distinct names.

In the λ-calculus course, this part will require some extra care.

This set $\mathsf{fv}(e)$ of the ***free variables*** of an expression e is again defined by its own set of equations.

$$\begin{array}{rcl}
\mathsf{fv}(\mathtt{n}) & = & \varnothing \\
\mathsf{fv}(\mathtt{x}) & = & \{x\} \\
\mathsf{fv}(e_1 \oplus e_2) & = & \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\
\mathsf{fv}(e_1 \otimes e_2) & = & \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\
\mathsf{fv}(\mathtt{let\ x = } e_1 \mathtt{\ in\ } e_2) & = & \mathsf{fv}(e_1) \cup (\mathsf{fv}(e_2) \setminus \{x\}) \\
\mathsf{fv}(\mathtt{fun\ x\ ->\ } e) & = & \mathsf{fv}(e) \setminus \{x\} \\
\mathsf{fv}(e_1\ e_2) & = & \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2)
\end{array}$$

Natural semantics and call-by-value. The specification of the semantics of a program by a function is mostly adapted to the specification of deterministic programs whose execution goes well (where we indeed expect a value for every expression).

A more flexible approach consists in defining the semantics by a binary relation between expressions and the produced values. We would then write

$$e \implies v$$

for any pair of an expression e and a value v such that the expression e *may* evaluate to the value v.

This relation, called ***natural semantics***, or ***big step semantics***, is defined by inference rules and specifies possible evaluations of expressions. Any compiler is expected to comply with the semantics of its source language.

To derive our formalisation, let us first the set of values that our expression will produce: integer numbers, and functions.

$$\begin{array}{lcl}
v & ::= & n \\
 & | & \mathtt{fun}\ x\ \mathtt{->}\ e
\end{array}$$

Inference rules will then correspond to the equations that defined our eval function.

- $\mathsf{eval}(n) = n$. An integer constant is its own value. The associated rule is an axiom.

$$\frac{}{n \implies n}$$

- $\mathsf{eval}(e_1\ \mathtt{+}\ e_2) = \mathsf{eval}(e_1) + \mathsf{eval}(e_2)$. The value of an addition expression is obtained by adding the values of the two subexpressions.

$$\frac{e_1 \implies n_1 \qquad e_2 \implies n_2}{e_1\ \mathtt{+}\ e_2 \implies n_1 + n_2}$$

  Note that this rule may apply only when the values n_1 and n_2 associated to e_1 and e_2 are indeed numbers.
- $\mathsf{eval}(\mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2) = \mathsf{eval}(e_2[x := \mathsf{eval}(e_1)])$. The value of an expression e_2 with a local variable x is obtained by evaluating e_2 after substituting every occurrence of x by the value of the association expression e_1.

$$\frac{e_1 \implies v_1 \qquad e_2[x := v_1] \implies v}{\mathtt{let}\ x\ \mathtt{=}\ e_1\ \mathtt{in}\ e_2 \implies v}$$

- $\mathsf{eval}(\mathtt{fun}\ x\ \mathtt{->}\ e) = \mathtt{fun}\ x\ \mathtt{->}\ e$. A function is its own value (since variables are now substituted, no closure is needed anymore). As for constants, the associated rule is an axiom.

$$\frac{}{\mathtt{fun}\ x\ \mathtt{->}\ e \implies \mathtt{fun}\ x\ \mathtt{->}\ e}$$

- $\mathsf{eval}(e_1\ e_2) = \mathsf{eval}(e[x = \mathsf{eval}(e_2)])$ if $\mathsf{eval}(e_1) = \mathtt{fun}\ x\ \mathtt{->}\ e$. For the value of an application to be defined, the value of its left member e_1 must be a function. Then the value of the application is obtained by substituting the formal parameter in the body of the function by the value of the argument e_2, then evaluating the obtained expression.

$$\frac{e_1 \implies \mathtt{fun}\ x\ \mathtt{->}\ e \qquad e_2 \implies v_2 \qquad e[x := v_2] \implies v}{e_1\, e_2 \implies v}$$

We obtain five rules for this fragment of FUN, and we can justify semantic judgment of the form $e \implies v$ using a derivation.

$$\dfrac{\dfrac{}{\texttt{fun}\,x\texttt{->}x\texttt{+}x \implies \texttt{fun}\,x\texttt{->}x\texttt{+}x} \qquad \dfrac{\dfrac{}{\texttt{fun}\,x\texttt{->}x\texttt{+}x \implies \texttt{fun}\,x\texttt{->}x\texttt{+}x} \quad \dfrac{\dfrac{}{20 \implies 20} \quad \dfrac{}{1 \implies 1}}{20\texttt{+}1 \implies 21} \quad \dfrac{\dfrac{}{21 \implies 21} \quad \dfrac{}{21 \implies 21}}{21\texttt{+}21 \implies 42}}{(\texttt{fun}\ x\ \texttt{->}\ x\ \texttt{+}\ x)\,(20\ \texttt{+}\ 1) \implies 42}}{\texttt{let}\ f\ \texttt{=}\ \texttt{fun}\ x\ \texttt{->}\ x\ \texttt{+}\ x\ \texttt{in}\ f(20\ \texttt{+}\ 1) \implies 42}$$

Remark that the rule given for the application of a function evaluates the argument e_2 before it is substituted in the body of the function. This behaviour is consistent with the interpretation function we introduced at the beginning, which implemented a *call by value* strategy.

Call by name semantics. We could also define a variant of the semantics, based on a *call by name* strategy. This variant essentially consists in replacing the application rule by the following simpler version

$$\dfrac{e_1 \implies \texttt{fun}\ x\ \texttt{->}\ e \qquad e[x := e_2] \implies v}{e_1\, e_2 \implies v}$$

where the argument e_2 is substituted unevaluated.

Optionnally, we can also use the following variant of the rule for local variables.

$$\dfrac{e_2[x := e_1] \implies v}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \implies v}$$

This *call by name* semantics is *almost* equivalent to the *call by value* semantics: they mostly allow the derivation of the same judgments $e \implies v$. The shapes of the derivations may be different, but the important point is whether a derivation *exists* or not. For instance, we can also derive

$$\texttt{let}\ f\ \texttt{=}\ \texttt{fun}\ x\ \texttt{->}\ x\ \texttt{+}\ x\ \texttt{in}\ f(20\ \texttt{+}\ 1) \implies 42$$

with the call by name semantics as follows.

Question: how does call by name affect the shape of the tree?

$$\dfrac{\dfrac{\dfrac{}{\texttt{fun}\,x\texttt{->}x\texttt{+}x \implies \texttt{fun}\,x\texttt{->}x\texttt{+}x} \qquad \dfrac{\dfrac{\dfrac{}{20 \implies 20} \quad \dfrac{}{1 \implies 1}}{20\texttt{+}1 \implies 21} \quad \dfrac{\dfrac{}{20 \implies 20} \quad \dfrac{}{1 \implies 1}}{20\texttt{+}1 \implies 21}}{(20\texttt{+}1)\texttt{+}(20\texttt{+}1) \implies 42}}{(\texttt{fun}\ x\ \texttt{->}\ x\ \texttt{+}\ x)\,(20\ \texttt{+}\ 1) \implies 42}}{\texttt{let}\ f\ \texttt{=}\ \texttt{fun}\ x\ \texttt{->}\ x\ \texttt{+}\ x\ \texttt{in}\ f(20\ \texttt{+}\ 1) \implies 42}$$

However, these two semantics are *not fully equivalent*: there are some judgments $e \implies v$ that can be derived in one but not in the other.

Question: can you find some?

Reasoning on the semantics. Since the natural semantics is defined by a system of inference rules, we can prove properties about programs and their semantics by reasoning by induction on the derivation of a judgment $e \implies v$. We get one case for each inference rule, and the premises of the rules provide induction hypotheses.

Let us consider the call by name semantics for FUN

$$\dfrac{}{n \implies n} \qquad \dfrac{e_1 \implies n_1 \qquad e_2 \implies n_2}{e_1\ \texttt{+}\ e_2 \implies n_1 + n_2} \qquad \dfrac{e_2[x := e_1] \implies v}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \implies v}$$

$$\dfrac{}{\texttt{fun}\ x\ \texttt{->}\ e \implies \texttt{fun}\ x\ \texttt{->}\ e} \qquad \dfrac{e_1 \implies \texttt{fun}\ x\ \texttt{->}\ e \qquad e[x := e_2] \implies v}{e_1\, e_2 \implies v}$$

and prove that if $e \implies v$, then v is value such that $\text{fv}(v) \subseteq \text{fv}(e)$, by induction on the derivation of $e \implies v$.

— Case $n \implies n$: immediate, since n is a value, and $\mathsf{fv}(n) \subseteq \mathsf{fv}(n)$.
— Case $\texttt{fun}\ x\ \texttt{->}\ e \implies \texttt{fun}\ x\ \texttt{->}\ e$: immediate as well.
— Case $e_1\ \texttt{+}\ e_2 \implies n_1 + n_2$ with $e_1 \implies n_1$ and $e_2 \implies n_2$. By definition $n_1 + n_2$ is an integer value. Moreover $\mathsf{fv}(n_1 + n_2) = \emptyset \subseteq \mathsf{fv}(e_1\ \texttt{+}\ e_2)$. *Note: induction hypotheses concerning* e_1 *and* e_2 *are not used here.*
— Case $\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \implies v$ with $e_2[x := e_1] \implies v$. The premise provides as induction hypothesis that $\mathsf{fv}(v) \subseteq \mathsf{fv}(e_2[x := e_1])$ (and v is a value).
We need here a lemma concerning the free variables of a term to which we apply a substitution. We use the following inclusion:

Exercise: prove this inclusion, by structural induction on the expression e.

$$\mathsf{fv}(e[x := e']) \subseteq (\mathsf{fv}(e) \setminus \{x\}) \cup \mathsf{fv}(e')$$

Using the lemma, we get $\mathsf{fv}(v) \subseteq (\mathsf{fv}(e_2) \setminus \{x\}) \cup \mathsf{fv}(e_1)$. Moreover, by definition we have

$$\mathsf{fv}(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2) = \mathsf{fv}(e_1) \cup (\mathsf{fv}(e_2) \setminus x)$$

Then $\mathsf{fv}(v) \subseteq \mathsf{fv}(\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2)$.
— Case $e_1\ e_2 \implies v$ with $e_1 \implies \texttt{fun}\ x\ \texttt{->}\ e$ and $e[x := e_2] \implies v$. The two premises give as induction hypotheses that v is a value, and that $\mathsf{fv}(\texttt{fun}\ x\ \texttt{->}\ e) \subseteq \mathsf{fv}(e_1)$ and $\mathsf{fv}(v) \subseteq \mathsf{fv}(e[x := e_2])$. Using the same lemma as in the last case, we get:

$$\begin{array}{rcl}
\mathsf{fv}(v) & \subseteq & \mathsf{fv}(e[x := e_2]) \\
 & = & (\mathsf{fv}(e) \setminus \{x\}) \cup \mathsf{fv}(e_2) \\
 & = & \mathsf{fv}(\texttt{fun}\ x\ \texttt{->}\ e) \cup \mathsf{fv}(e_2) \\
 & \subseteq & \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\
 & = & \mathsf{fv}(e_1\ e_2)
\end{array}$$

1.5 Small step operational semantics

Natural semantics associates expressions to the values their evaluation may produce. This semantics speaks only about computations that succeed. In particular, it does not give any information about expressions whose evaluation encounters a failure, such as `5(37)`, nor about expression whose evaluation never ends, such as `(fun x -> x x) (fun x -> x x)`. Actually, natural semantics is not even capable of distinguishing these two situations.

Small step semantics, or ***reduction semantics***, provide finer information by decomposing the evaluation $e \implies v$ in a sequence of computation steps $e \to e_1 \to e_2 \to \ldots \to v$. Then we may distinguish three main behaviours:
— a computation which, after some finite number of steps, reaches a result:

$$e \to e_1 \to e_2 \to \ldots \to v$$

where v is a value,
— a computation which, after some number of steps, stumbles on a failure state:

$$e \to e_1 \to e_2 \to \ldots \to e_n$$

where e_n is not a value, but cannot be evaluated further,
— a computation that never ends:

$$e \to e_1 \to e_2 \to \ldots$$

where computations steps go on infinitly.

Computation rules. A small step semantics is defined by a binary relation $e \to e'$ called ***reduction relation***, describing a single step of computation. This relation is itself defined by a set of inference rules. We provide on the one hand elementary computation rules, giving base cases, and on the second hand inference rules that allow the application of a computation rule in a subexpression.

Let us first give the axioms for our fragment of FUN. They correspond to the main computation rules, immediately applied at the root of an expression.

— Axiom for the application of a function (call by value). If a function `fun` x `->` e is applied to a value v, then we may substitute v for each occurrence of the formal parameter x in the body e of the function.

$$\frac{}{(\texttt{fun}\ x\ \texttt{->}\ e)\ v \to e[x := v]}$$

The application of this rule assumes that the argument of the application has been evaluated at a previous stage of the computation.

— Axiom for the replacement of a local variable by its value.

$$\frac{}{\texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ e \to e[x := v]}$$

As for the application of a function, this rule may be applied only if the value v associated to the variable x has already been computed.

— Axiom for the addition.

$$\frac{n_1 + n_2 = n}{n_1\ \texttt{+}\ n_2 \to n}$$

Be careful to the “pun” here: we start with an expression n_1 `+` n_2, where the symbol + is part of the syntax, and the result is n, the actual result of the mathematical addition of the two numbers n_1 and n_2. Again, this rule can be applied only if both operands have already been evaluated, and the obtained values are numbers.

Inference rules then describe how the base rules can be applied to subexpressions.

— Inference rules for the addition. In an expression of the form e_1 `+` e_2, we may performe a reduction step in one or the other of the subexpressions e_1 and e_2. This principle translates into two inference rules, one for each subexpression.

$$\frac{e_1 \to e_1'}{e_1\ \texttt{+}\ e_2 \to e_1'\ \texttt{+}\ e_2} \qquad \frac{e_2 \to e_2'}{e_1\ \texttt{+}\ e_2 \to e_1\ \texttt{+}\ e_2'}$$

Using these rules, we can derive the fact one step of computation may lead from the expression `((1+2)+3)+(4+5)` to the expression `(3+3)+(4+5)`.

$$\cfrac{\cfrac{\cfrac{1 + 2 = 3}{\texttt{1 + 2} \to \texttt{3}}}{\texttt{(1 + 2) + 3} \to \texttt{3 + 3}}}{\texttt{((1 + 2) + 3) + (4 + 5)} \to \texttt{(3 + 3) + (4 + 5)}}$$

Remark that these rules to not say anything about the order in which the two subexpressions e_1 and e_2 have to be evaluated. The rules even allow alternating between both subexpressions in arbitrary ways. Thus we can further derive all the steps of the following computation sequence.

$$\texttt{((1+2)+3)+(4+5)} \to \texttt{(3+3)+(4+5)} \to \texttt{(3+3)+9} \to \texttt{6+9} \to \texttt{15}$$

If we prefer forcing an evaluation order from left to right for the operands, we have to replace the last rule by the following variant, which authorize reducing the right operand of an addition only if the left operand is already a value.

$$\frac{e_2 \to e_2'}{v_1\ \texttt{+}\ e_2 \to v_1\ \texttt{+}\ e_2'}$$

— Inference rules for a local variables. The rule below allow a computation step to take place in the expression e_1 defining the value of a local variable x.

$$\frac{e_1 \to e_1'}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \to \texttt{let}\ x\ \texttt{=}\ e_1'\ \texttt{in}\ e_2}$$

— Inference rules for applications. The two rules below always allow a computation step to take place in the left member of an application, and restricts computation in the right member to the cases where the left member has already been evaluated.

$$\frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2} \qquad \frac{e_2 \rightarrow e_2'}{v_1\ e_2 \rightarrow v_1\ e_2'}$$

Remark that no rule allow a computation step to take place inside the body e of a function `fun` x `->` e. Indeed, such a function is already a value, and needs not be evaluated further. Computation will go on in the body of function only after the function receives an argument, whose value v is substited for x in e.

Summary of the inference rules defining a small step semantics for our fragment of FUN, in call by value, with left-to-right evaluation of operands of a binary operation.

$$\frac{e_1 \rightarrow e_1'}{e_1\ \texttt{+}\ e_2 \rightarrow e_1'\ \texttt{+}\ e_2} \qquad \frac{e_2 \rightarrow e_2'}{v_1\ \texttt{+}\ e_2 \rightarrow v_1\ \texttt{+}\ e_2'} \qquad \frac{n_1 + n_2 = n}{n_1\ \texttt{+}\ n_2 \rightarrow n}$$

$$\frac{e_1 \rightarrow e_1'}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \rightarrow \texttt{let}\ x\ \texttt{=}\ e_1'\ \texttt{in}\ e_2} \qquad \frac{}{\texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ e \rightarrow e[x := v]}$$

$$\frac{e_1 \rightarrow e_1'}{e_1\ e_2 \rightarrow e_1'\ e_2} \qquad \frac{e_2 \rightarrow e_2'}{v_1\ e_2 \rightarrow v_1\ e_2'} \qquad \frac{}{(\texttt{fun}\ x\ \texttt{->}\ e)\ v \rightarrow e[x := v]}$$

Exercise: define a small step semantics in call by name.

Note that at this fine-grained level of description of the computations, we could also define variants corresponding to alternative evaluation strategies.

Reduction sequences. The relation $e \rightarrow e'$ describes elementary computation steps. We write:

— $e \rightarrow e'$ when 1 computation step leads from e to e', and
— $e \rightarrow^* e'$ when a computation leads from e to e' using 0, 1 or more steps (this is called a computation ***sequence*** or a reduction sequence).

An ***irreducible*** expression is an expression e from which no reduction step can take place, that is such that there is no expression e' such that $e \rightarrow e'$. An irreducible expression might be one of two different things:

— a value, that is the expected result of a computation,
— a stuck expression, that is an expression describing a computation that is not over, but for which no rule allows taking a new step.

Example of a reduction reaching a value.

```
let f = fun x -> x + x in f (20 + 1)
→  (fun x -> x + x) (20 + 1)
→  (fun x -> x + x) 21
→  21 + 21
→  42
```

Example of of stuck reduction.

```
let f = fun x -> fun y -> x + y in 1 + f 2
→  1 + (fun x -> fun y -> x + y) 2
→  1 + (fun y -> 2 + y)
```

1.6 Equivalence between small step and big step

Big step and small step semantics give slightly different points of view. The former directly gives the results that can be expected from a program, whereas the latter gives a more fine-grained account of the operations performed. These two views are, however, *equivalent*, since they speficy the same evaluation relations. In other words,

$$e \implies v \qquad \text{of and only if} \qquad e \rightarrow^* v$$

Let us prove this for the two versions of the call by value semantics of our fragment of FUN. We consider the big step semantics given by the rules

$$\frac{}{n \implies n} \qquad \frac{}{\texttt{fun}\ x\ \texttt{->}\ e \implies \texttt{fun}\ x\ \texttt{->}\ e}$$

$$\frac{e_1 \implies n_1 \qquad e_2 \implies n_2}{e_1\ \texttt{+}\ e_2 \implies n_1 + n_2} \qquad \frac{e_1 \implies v_1 \qquad e_2[x := v_1] \implies v}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \implies v}$$

$$\frac{e_1 \implies \texttt{fun}\ x\ \texttt{->}\ e \qquad e_2 \implies v_2 \qquad e[x := v_2] \implies v}{e_1\, e_2 \implies v}$$

and the small step semantics given by the following rules.

$$\frac{e_1 \to e_1'}{e_1\ \texttt{+}\ e_2 \to e_1'\ \texttt{+}\ e_2} \qquad \frac{e_2 \to e_2'}{v_1\ \texttt{+}\ e_2 \to v_1\ \texttt{+}\ e_2'} \qquad \frac{n_1 + n_2 = n}{n_1\ \texttt{+}\ n_2 \to n}$$

$$\frac{e_1 \to e_1'}{\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \to \texttt{let}\ x\ \texttt{=}\ e_1'\ \texttt{in}\ e_2} \qquad \frac{}{\texttt{let}\ x\ \texttt{=}\ v\ \texttt{in}\ e \to e[x := v]}$$

$$\frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2} \qquad \frac{e_2 \to e_2'}{v_1\, e_2 \to v_1\, e_2'} \qquad \frac{}{(\texttt{fun}\ x\ \texttt{->}\ e)\, v \to e[x := v]}$$

There: $e \implies v$ **implies** $e \to^* v$. Let us show this by induction on the derivation of $e \implies v$.

- Case $n \implies n$. We have $n \to^* n$ with a sequence of 0 steps.
- Case $\texttt{fun}\ x\ \texttt{->}\ e \implies \texttt{fun}\ x\ \texttt{->}\ e$ immediate as well.
- Case $e_1\ \texttt{+}\ e_2 \implies n$ with $e_1 \implies n_1$, $e_2 \implies n_2$ and $n = n_1 + n_2$. The two premises give the induction hypotheses $e_1 \to^* n_1$ and $e_2 \to^* n_2$. From $e_1 \to^* n_1$ we deduce $e_1\ \texttt{+}\ e_2 \to^* n_1\ \texttt{+}\ e_2$ (since we consider a sequence rather than a single step, this would actually require a simple lemma proved by induction on the length of the sequence $e_1 \to^* n_1$). Similarly, from $e_2 \to^* n_2$ we deduce $n_1\ \texttt{+}\ e_2 \to^* n_1\ \texttt{+}\ n_2$. We then obtain the following sequence, by appending a final step using the base rule for addition.

$$\begin{array}{lll} e_1\ \texttt{+}\ e_2 & \to^* & n_1\ \texttt{+}\ e_2 \\ & \to^* & n_1\ \texttt{+}\ n_2 \\ & \to & n \end{array}$$

- Case $\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \implies v$ with $e_1 \implies v_1$ and $e_2[x := v_1] \implies v$. The two premises give the two induction hypotheses $e_1 \to^* v_1$ and $e_2[x := v_1] \to^* v$. We deduce the following reduction sequence.

$$\begin{array}{lll} \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 & \to^* & \texttt{let}\ x\ \texttt{=}\ v_1\ \texttt{in}\ e_2 \\ & \to & e_2[x := v_1] \\ & \to^* & v \end{array}$$

- Case $e_1\, e_2 \implies v$ with $e_1 \implies \texttt{fun}\ x\ \texttt{->}\ e$, $e_2 \implies v_2$ and $e[x := v_2] \implies v$. The three premises give the induction hypotheses $e_1 \to^* \texttt{fun}\ x\ \texttt{->}\ e$, $e_2 \to^* v_2$ and $e[x := v_2] \to^* v$. We deduce the following reduction sequence.

$$\begin{array}{lll} e_1\, e_2 & \to^* & (\texttt{fun}\ x\ \texttt{->}\ e)\, e_2 \\ & \to^* & (\texttt{fun}\ x\ \texttt{->}\ e)\, v_2 \\ & \to & e[x := v_2] \\ & \to^* & v \end{array}$$

Back: $e \to^* v$ **implies** $e \implies v$. Remark that in this statement, when writing $e \to^* v$ we assume v to be a value. Let us prove this by induction on the length of the reduction sequence $e \to^* v$.

- Case of a sequence of length 0. The expression e is thus already a value, and necessarily has the form either n or $\texttt{fun}\ x\ \texttt{->}\ e'$. We reach an immediate conclusion with one of the axioms

$$\frac{}{n \implies n} \qquad \frac{}{\texttt{fun}\ x\ \texttt{->}\ e' \implies \texttt{fun}\ x\ \texttt{->}\ e'}$$

- Case of sequence $e \to^* v$ of length $n+1$, assuming that for any reduction sequence $e' \to^* v$ of length n we have $e' \implies v$ (this is the induction hypothesis). Let us write

  $$e \to e' \to \ldots \to v$$

  our reduction sequence $e \to^* v$ in $n+1$ steps, with e' the expression obtained after the first step. We thus have $e' \to^* v$ in n steps, and by induction hypothesis $e' \implies v$.
  To conclude, we prove a lemma ensuring that, for any expressions, if $e \to e'$ and $e' \implies v$ then $e \implies v$.

Lemma: if $e \to e'$ and $e' \implies v$ then $e \implies v$. Proof by induction on the derivation of $e \to e'$.

- Case $n_1 + n_2 \to n$ with $n = n_1 + n_2$. Here we also have $n \implies v$, which is possible only if $v = n$. We conclude with the derivation

  $$\frac{\dfrac{}{n_1 \implies n_1} \qquad \dfrac{}{n_2 \implies n_2}}{n_1 + n_2 \implies n}$$

- Case $\texttt{let}\ x\ \texttt{=}\ w\ \texttt{in}\ e \to e[x := w]$, with w a value and where the hypothesis $e' \implies v$ can be written $e' = e[x := w] \implies v$. We conclude with the derivation

  $$\frac{w \implies w \qquad e[x := w] \implies v}{\texttt{let}\ x\ \texttt{=}\ w\ \texttt{in}\ e \implies v}$$

  Note that we did not use the induction hypothesis here.
- Case $e_1 + e_2 \to e_1' + e_2$ with $e_1 + e_1'$ and where the hypothesis $e' \implies v$ can be written $e_1' + e_2 \implies v$. The premise $e_1 + e_1'$ gives as induction hypothesis that "for any value v_1 such that $e_1' \implies v_1$ we have $e_1 \implies v_1$".
  Since the judgment $e_1' + e_2 \implies v$ is valid, we know that v is obtained as $n_1 + n_2$ with n_1 and n_2 such that $e_1' \implies n_1$ and $e_2 \implies n_2$ (the only inference rule whose conclusion is compatible with our case requires these premises). Using the induction hypothesis we then deduce $e_1 \implies n_1$, and we can use this judgment to complete the following derivation.

  $$\frac{e_1 \implies n_1 \qquad e_2 \implies n_2}{e_1 + e_2 \implies n}$$

- The other cases are similar.

Finally, both presentations of the semantics associate the same expressions to the same values. Small step semantics however give more information on computations that fail, which we will use in the next chapter.

1.7 Extensions

Things you can try to go further:

- formalize the semantics of `if` and `fix`, both in big step and in small step style;
- extend FUN with lazy operators such as `||` and `&&`, which do not evaluate their second operand when the first one already allows knowing the final result;
- extend FUN with algebraic data structures and pattern matching.

2 Types and safety

In this chapter we explore another aspect of the semantics of the FUN language, by classifying the various kinds of values a program may deal with, and by guaranteeing than programs handle them in a consistent way.

2.1 Types values and operations

Inside a computer, a piece of data is a sequence of bits. Here is 32-bits memory word.

```
1110 0000 0110 1100 0110 0111 0100 1000
```

For easier reading, we often use an hexadecimal representation. Here it would be

```
0x e0 6c 67 48
```

(the `0x` prefix introduces the hexadecimal representation, then each character represents a group of 4 bits).

What means this word? We can know it only with a *very* precise knowledge of the context:

— if these bits represent a memory address, then it is the address `3 765 200 712`,
— if these bits represent a 32-bit signed integer in 2's complement, this is the number −529 766 584,
— if these bits represent a simple precision floating point number following the IEEE754 standard, this is the number $15\ 492\ 936 \times 2^{42}$,
— if these bits represent a character string in Latin-1 encoding, this is the string ”`Holà`”.

If we forget about the context in which a sequence of bits means something, we may perform operations that make no sense.

For instance: applying integer addition to the representations of the two strings ”`5`” and ”`37`” may produce the new string ”`h7`”.

Inconsistent operations. All operations provided by a programming language are constrained. In caml for instance,

— the addition `5 + 37` of two integers is possible,
— but the operations ”`5`” `+ 37`, `5 + (fun x -> 37)` or `5(37)` are not.

We already observed this in the previous chapter, with the interpreter for the FUN language. The values that could be produced by an expression were split into several categories, including numbers, booleans, and functions

```
type value =
  | VInt  of int
  | VBool of bool
  | VClos of string * expr * env
```

and some operations behave differently depending on the kind of values given as operands. For instance, a binary arithmetic operation expected two numbers. It produced a result (of kind `VInt`) when its operands were both of kind `VInt`, and interrupted the program otherwise with the exception `Failure` ”`unauthorized operation`”.

```
let rec eval e env = match e with
  | Bop(op, e1, e2) ->
     begin match op, eval e1 env, eval e2 env with
       | Add, VInt n1, VInt n2 -> VInt (n1 + n2)
       ...
       | _ -> failwith ”unauthorized operation”
     end
```

Types: a classification of values. Programming languages usually distinguish numerous kinds of values, called ***types***. The precise classification may differ, but some kinds are seen in most languages. For instance:

— numbers: `int`, `double`,
— booleans: `bool`,
— characters: `char`,
— character strings: `string`.

Additionally, richer types can be built over these base types. For instance:

— arrays: `int[]`,

— functions: `int -> bool`,
— data structures: `struct point { int x; int y; };`,
— objects: `class Point { public final int x, y; ... }`.

Once this classification is set, each operation is defined to apply to elements of some given type.

In some cases, one operator may be applied to various types of elements, with different meanings depending on the type. This is called ***overloading***. For instance, in python or java the operator `+` may apply:

— to two integers, in which case it denotes an addition: `5 + 37 = 42`,
— to two strings, in which case it denotes a concatenation: `"5" + "37"= "537"`.

Some programming languages also allow ***casting***, that is converting a value of some type to another type. This may even be an implicit operation. For instance, the operation `"5" + 37` mixing a string and an integer may evaluate to:

— `42` in php, where the string `"5"` is converted into the number `5`,
— `"537"` in java, where the integer `37` is converted into the string `"37"`.

Note that such a conversion may require an actual modification of the data! The number `5` is represented by the memory word `0x 00 00 00 05`, whereas the string `"5"` is represented by `0x 00 00 00 35`. Similarly, the number `37` is represented by `0x 00 00 00 25` and the string `"37"` by `0x 00 00 37 33`. In each case, casting from one type to the other requires computing the new representation.

Summary. The type of a value gives the key for interpreting the associated data, and may be required for selecting the appropriate operations. Moreover, inconsistent types are likely to reveal programming errors (and programs that should not be executed).

Static type analysis. Handling types at runtime is costly in several ways:

— some memory has to be used to pair each data with an identification of its type,
— runtime tests are necessary to select the operations to apply to the data,
— execution may be interrupted when a type error appears, ...

In ***dynamically typed*** languages such as python, theses costs are paid in full. Conversely, ***statically typed*** languages such as C, java or caml save us at least a part of this runtime cost, since types are handled at compile time.

Static type analysis, which means type analysis performed at compilation time, consists in associating with each expression in a program a type, which predicts the type of the value that will be obtained when evaluating the expression. This prediction is based on constraints given by each constructor of the abstract syntax. For instance, considering an addition expression `Bop(Add, e1, e2)` we can remark that:

— the expression will produce an integer,
— for the operation to be consistent, both subexpressions `e1` and `e2` must also produce integers.

Similarly, we associate to each variable a type, which gives the type of the value the variable refers to. Thus, in **`let`** `x = e` **`in`** `x + 1` the type of `x` is the type of the value produced by the expression `e`, and we expect it to be the type of intergers. Following the same principles, the type of a function make the expected types of all parameters explicit, as well as the type of the result.

This verification of type consistency before the execution of a program is associated to the idea, formulated by Robin Milner, that

Well-typed programs do not go wrong.

The aim of static type analysis is to reject absurd programs before they are ever executed (or released to clients...). However, we cannot identify with absolute precision all the buggy programs (questions of this kind are usually algorithmically undecidable). We are looking for decidable criteria which:

— give some ***safety***, by rejecting many absurd programs,
— and let to programmers enough ***expressiveness***, by not rejecting too many non-absurd programs.

This analysis may require some amount of annotations from the programmer in a source program. Here are a few possibilities.

1. Annotate each subexpression with its intended type.

```
fun (x : int) ->
  let (y : int) = ((x : int) + (1 : int) : int)
  in (y : int)
```

The programmer has to do all the work here, and the compiler just ***checks*** consistency. No language actually requires this amount of annotations.

2. Annotate only variables, and formal parameters of functions.

```
fun (x : int) -> let (y : int) = x+1 in y
```

Here, the compiler can deduce the type of each expression, using the given types of the variables. This is what is asked by C or java.

3. Annotate only function parameters.

```
fun (x : int) -> let y = x+1 in y
```

4. No annotation.

```
fun x -> let y = x+1 in y
```

In this last case, the compiler must ***infer*** the type of each variable and expression, with no help from the programmer. This is what happens with caml.

If type analysis is performed at compilation time, selecting the appropriate operation for overloaded operators is also done at compilation time, and costs nothing at execution time. Moreover, checking the consistency of types at compilation time allow early detection of many program inconsistencies, and consequently early correction of bugs.

In the remaining of this chapter, we formalize the notion of type and the associated constraints, we implement type checking and type inference for FUN, and we turn our vague notion of safety into a theorem about well-typed programs.

2.2 Typing judgment and inference rules

Well-typed programs are characterized by a set of rules that allow justififying that "in some context Γ, an expression e is consistent and has type τ". This sentence is called a ***typing judgment***, and is written

$$\Gamma \vdash e : \tau$$

The context Γ in a typing judgment maps a type to each variable of the expression e.

The typing judgment is not function that would give a type to every expression: it is instead a relation between three elements: context, expression, type. In particular, some expressions e have no type (because they are inconsistent), and in some situations several types are possible for a given expression and context.

Typing rules. Let us see how we can formalize the consistency and the type of an expression for a fragment of the FUN language:

$$\begin{array}{lcl} e & ::= & n \\ & | & e + e \\ & | & x \\ & | & \texttt{let } x \texttt{ = } e \texttt{ in } e \\ & | & \texttt{fun } x \texttt{ -> } e \\ & | & e\, e \end{array}$$

We need a base type for numbers, as well as function types.

$$\begin{array}{lcl} \tau & ::= & \texttt{int} \\ & | & \tau \to \tau \end{array}$$

A type of the form $\tau_1 \to \tau_2$ is the type of a function that expects a parameter of type τ_1 and returns a result of type τ_2.

We associate to each construction of the language a rule giving:

— the type such expression may have, and

— the constraints that have to be satisfied for the expression to be consistent.

We start with arithmetic, and state each rule under two formats: a natural language description, and its translation into an ***inference rule***.

— An integer constant n has the type `int`.

$$\frac{}{\Gamma \vdash n : \texttt{int}}$$

- If both expressions e_1 and e_2 are consistent and of type `int`, then the expression e_1 `+` e_2 is consistent, also with type `int`.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}$$

In the inference rule for addition, the judgments $\Gamma \vdash e_1 : \texttt{int}$ and $\Gamma \vdash e_2 : \texttt{int}$ are premises, and the judgment $\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}$ is the conclusion. In other words, if we can justify $\Gamma \vdash e_1 : \texttt{int}$ and $\Gamma \vdash e_2 : \texttt{int}$, then the rule allows to deduce that $\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}$. Conversely, the rule for the integer constant has no premise (it is called an axiom, or a base case), which means we do not need anything more than the rule to justified a judgment $\Gamma \vdash n : \texttt{int}$.

The rules concerning variables interact with the context Γ, also called ***environment***, since that is where the type a each variable is given.

- A variable has the type given by the environment.

$$\frac{}{\Gamma \vdash x : \Gamma(x)}$$

  Remark here that we consider Γ as a function: $\Gamma(x)$ is type that Γ associates to the variable x. Also, this rule can be applied only if $\Gamma(x)$ is actually defined, that is if x is in the domain of Γ.
- A local variable is associated to the type of the expression that defines it.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau_2}$$

  In such an expression, e_2 may refer to the local variable x. Thus e_2 is typed in an extended environment written $\Gamma, x : \tau_1$, which contains all the associations of Γ and also the association of type τ_1 to the variable x. On the other hand, x does not exist in e_1, and is not a free variable of the full expression: it does not appear in the typing environment for e_1, nor for `let` x `=` e_1 `in` e_2.

A function has a type of the form $\alpha \to \beta$, where α is the expected type of the parameter, and β the type of the result.

- A function has to be applied to an argument of the expected type.

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}$$

- In the body of a function, the formal parameter is seen as an ordinary variable, whose type corresponds to the expected type of the parameter.

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau_1 \to \tau_2}$$

Finally, the ***simple types*** for our fragment of the FUN language are fully defined by the six following inference rules.

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}$$

Typable expressions. A typing judgment is justified by a series of deductions obtained by applying the inference rules. For instance, given the context $\Gamma = \{ x : \texttt{int},\ f : \texttt{int} \to \texttt{int} \}$ we may reason as follow.

1. $\Gamma \vdash x : \texttt{int}$ is valid, by the rule on variables.

2. $\Gamma \vdash f : \mathtt{int} \to \mathtt{int}$ is valid, by the rule on variables.
3. $\Gamma \vdash \mathtt{1} : \mathtt{int}$ is valid, by the rule on constants.
4. $\Gamma \vdash f\ \mathtt{1} : \mathtt{int}$ is valid, by the rule on application, using the already justified points 2. and 3.
5. $\Gamma \vdash x + f\ \mathtt{1} : \mathtt{int}$ is valid, by the rule on addition, using 1. et 4.

This reasoning is called a ***derivation***, and can also be written as a ***derivation tree*** whose root is the conclusion we want to justify.

$$\dfrac{\dfrac{}{\Gamma \vdash x : \mathtt{int}} \qquad \dfrac{\dfrac{}{\Gamma \vdash f : \mathtt{int} \to \mathtt{int}} \quad \dfrac{}{\Gamma \vdash \mathtt{1} : \mathtt{int}}}{\Gamma \vdash f\ \mathtt{1} : \mathtt{int}}}{\Gamma \vdash x + f\ \mathtt{1} : \mathtt{int}}$$

In such a tree, each bar indicates the application of an inference rule, and each subtree justifies an auxiliary judgment (the premise of a rule).

In some situations, we may derive several judgments giving different types to the same expression in the same context. For instance:

$$\vdash \mathtt{fun\ x\ \text{->}\ x} : \mathtt{int} \to \mathtt{int}$$

$$\vdash \mathtt{fun\ x\ \text{->}\ x} : (\mathtt{int} \to \mathtt{int}) \to (\mathtt{int} \to \mathtt{int})$$

are both valid. Here, the absence of the context Γ means that we consider the empty context.

Untypable expressions. If an expression e is inconsistent, no judgment $\Gamma \vdash e : \tau$ can be justified using the typing rules. We can show that an expression is not typable by showing that any attempt at building a typing tree for the expression necessarily fails.

Consider the expression `5(37)`, which we can also write `5 37`. It is an application. Only one rule could possibly be used to justify $\Gamma \vdash \mathtt{5\ 37} : \tau$, and the application of this rule requires justifying the two premises $\Gamma \vdash \mathtt{5} : \tau' \to \tau$ and $\Gamma \vdash \mathtt{37} : \tau'$ (the type τ' may be chosen freely, but it must be the same for both judgments). However, justifying a premise $\Gamma \vdash \mathtt{5} : \tau' \to \tau$ is impossible: no rule allow giving a functional type to an integer constant (the only rule that could be applied for an integer constant would give the typing judgment $\Gamma \vdash \mathtt{5} : \mathtt{int}$).

Consider the other example `fun x -> x x`. Similarly, since only one rule can possibly be applied to each kind of expression, a derivation tree for a judgment $\Gamma \vdash \mathtt{fun\ x\ \text{->}\ x\ x} : \tau$ would necessarily have the shape

$$\dfrac{\dfrac{\Gamma, \mathtt{x} : \tau_1 \vdash \mathtt{x} : \tau_1 \to \tau_2 \qquad \Gamma, \mathtt{x} : \tau_1 \vdash \mathtt{x} : \tau_1}{\Gamma, \mathtt{x} : \tau_1 \vdash \mathtt{x\ x} : \tau_2}}{\Gamma \vdash \mathtt{fun\ x\ \text{->}\ x\ x} : \tau_2}$$

However, the premise $\Gamma, \mathtt{x} : \tau_1 \vdash \mathtt{x} : \tau_1 \to \tau_2$ cannot be justified: the inference rules allow only the type τ_1 for `x`, and there are no (finite) types τ_1 and τ_2 such that $\tau_1 = \tau_1 \to \tau_2$.

Reasoning on well-typed expressions. Let us prove that for any context Γ, any expression e and any type τ, if $\Gamma \vdash e : \tau$ is valid then all the free variables of e are in the domain of Γ.

Since valid typing judgments are defined by a system of inference rules, we can establish that some properties are true for all well-typed expressions by reasoning by induction on the structure of the typing derivation tree. We have one proof case for each inference rule, and each premise of the rule yields an induction hypothesis.

Let us prove that if $\Gamma \vdash e : \tau$ then $\mathsf{fv}(e) \subseteq \mathsf{dom}(\Gamma)$, by induction on $\Gamma \vdash e : \tau$.

- Case $\Gamma \vdash n : \mathtt{int}$. We have $\mathsf{fv}(n) = \emptyset$, and of course $\emptyset \subseteq \mathsf{dom}(\Gamma)$.
- Case $\Gamma \vdash x : \Gamma(x)$. We have $\mathsf{fv}(x) = \{x\}$, and the application of the rule indeed assumes that $\Gamma(x)$ is defined, which means $x \in \mathsf{dom}(\Gamma)$.
- Case $\Gamma \vdash e_1 + e_2 : \mathtt{int}$, with premises $\Gamma \vdash e_1 : \mathtt{int}$ and $\Gamma \vdash e_2 : \mathtt{int}$. The premises give two induction hypotheses $\mathsf{fv}(e_1) \subseteq \mathsf{dom}(\Gamma)$ and $\mathsf{fv}(e_2) \subseteq \mathsf{dom}(\Gamma)$. By definition of free variables we have $\mathsf{fv}(e_1 + e_2) = \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2)$. With the induction hypotheses we deduce that $\mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \subseteq \mathsf{dom}(\Gamma)$, and therefore $\mathsf{fv}(e_1 + e_2) \subseteq \mathsf{dom}(\Gamma)$.

— Case $\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau_2$, with premises $\Gamma \vdash e_1 : \tau_1$ and $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$. The premises give two induction hypotheses $\mathsf{fv}(e_1) \subseteq \mathsf{dom}(\Gamma)$ and $\mathsf{fv}(e_2) \subseteq \mathsf{dom}(\Gamma) \cup \{x\}$ (note that the premise related to the judgment $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$ mentions an environment extended with the variable x). By definition we have $\mathsf{fv}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) = \mathsf{fv}(e_1) \cup (\mathsf{fv}(e_2) \setminus \{x\})$. The first induction hypothesis ensures that $\mathsf{fv}(e_1) \subseteq \mathsf{dom}(\Gamma)$. The second induction hypothesis ensures that $\mathsf{fv}(e_2) \subseteq \mathsf{dom}(\Gamma) \cup \{x\}$, from which we deduce $\mathsf{fv}(e_2) \setminus \{x\} \subseteq \mathsf{dom}(\Gamma)$. Therefore we have $\mathsf{fv}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) \subseteq \mathsf{dom}(\Gamma)$.
— Both cases related to functions are similar to the cases above.

Full rules for FUN. Let us now complete our system to type the full FUN language. We need a new base type `bool` for boolean values.

$$\begin{array}{rcl} \tau & ::= & \texttt{int} \\ & | & \texttt{bool} \\ & | & \tau \rightarrow \tau \end{array}$$

We also introduce three additional typing rules for the missing constructs.
— An expression $e_1 \texttt{ < } e_2$ is consistent whenever the expressions e_1 and e_2 are numbers. The result is of type `bool`.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ < } e_2 : \texttt{bool}}$$

— The result of a conditional expression may come from one branch or the other. Thus the result type τ must be valid for both branches. If the expression c is consistent with type `bool`, and if both expressions e_1 et e_2 are consistent with a common type τ, then the expression $\texttt{if } c \texttt{ then } e_1 \texttt{ else } e_2$ is consistent and can be given the type τ.

$$\frac{\Gamma \vdash c : \texttt{bool} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{if } c \texttt{ then } e_1 \texttt{ else } e_2 : \tau}$$

— A recursive expression must have the same type τ that the recursive references it contains. If the expression e is consistent and of type τ, in an environment where the identifier x also has this type τ, then the expression $\texttt{fix } x = e$ is consistent, and of type τ.

$$\frac{\Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash \texttt{fix } x = e : \tau}$$

2.3 Type safety

The motto of typing is *well-typed programs do not go wrong*. In the context of our interpret for the language FUN, this means that evaluating a well-typed program never results in an "`unauthorized operation`" error. This is called a ***safety*** property of typed programs. In this section, we state and prove this property using the formal semantics of FUN.

Typing and big step semantics. Using the notion of natural semantics, we can prove the following statement relating the typing and the evaluation of an expression.

$$\text{If} \quad \Gamma \vdash e : \tau \quad \text{and} \quad e \Longrightarrow v \quad \text{then} \quad \Gamma \vdash v : \tau.$$

This means that the evaluation relation preserves the consistency and the types of expressions.

However, note that this statement takes as hypothesis that the evaluation is indeed possible and reaches a value. It does not prove that the evaluation of well-typed programs indeed produce a value, and says nothing about programs that break of loop. We need the small step semantics to get an actual safety property.

Type safety, small step version. Using a notion of reduction semantics, the safety property may be state as: the evaluation of a well-typed program never blocks on an inconsistent operation.

We formalize the property through two lemmas.

- ***Progress*** lemma: a well-typed expression is never blocked. In other words, if a well-typed expression e is not a value then we can perform at least one computation step from e.

  $$\textit{If}\quad \Gamma \vdash e : \tau \quad \textit{then } e \textit{ is a value or there is } e' \textit{ such that}\quad e \rightarrow e'.$$

- ***Type preservation*** lemma: reduction preserves types. If an expression e is consistent, then any expression e' obtained by reducing e is consistent, with the type as e.

  $$\textit{If}\quad \Gamma \vdash e : \tau \quad \textit{and}\quad e \rightarrow e' \quad \textit{then}\quad \Gamma \vdash e' : \tau.$$

  Historically, the type preservation lemma was called ***subject reduction*** (explanation: e is the "subject" of the predicate $\Gamma \vdash e : \tau$).

Theses two lemmas, applied together iteratively, imply the following behaviour of any evaluation of a well-typed expression e_1 with type τ: if e_1 is not already a value, then it reduces to e_2, which is still well-typed with type τ and thus, in case it is not a value, reduces in turn to e_3 well-typed of type τ, on so on.

$$(e_1 : \tau) \;\rightarrow\; (e_2 : \tau) \;\rightarrow\; (e_3 : \tau) \;\rightarrow\; \ldots$$

At the far right side of this sequence, there are two possible scenarios: either we reach a value v (which, by the way, is well-typed with type τ), or the reduction go on infinitely. The reduction cannot end with a blocked expression.

Progress. If $\Gamma \vdash e : \tau$, then e is a value, or there is e' such that $e \rightarrow e'$. Let us consider the simple types for the fragment of the FUN language defined by the following rules.

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}$$

We will prove the lemma by induction on the derivation of $\Gamma \vdash e : \tau$.

- Case $\Gamma \vdash n : \texttt{int}$. Then n is a value.
- Case $\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau_1 \rightarrow \tau_2$. Then $\texttt{fun } x \texttt{ -> } e$ is a value.
- Case $\Gamma \vdash e_1\, e_2 : \tau_1$, with $\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1$ and $\Gamma \vdash e_2 : \tau_2$. Induction hypotheses give us the two following disjunctions.
  1. e_1 is a value or $e_1 \rightarrow e_1'$,
  2. e_2 is a value or $e_2 \rightarrow e_2'$.

  We reason by case on these disjunctions.
  - If $e_1 \rightarrow e_1'$, then $e_1\, e_2 \rightarrow e_1'\, e_2$: goal completed.
  - Otherwise, e_1 is a value v_1.
    - If $e_2 \rightarrow e_2'$, then $v_1\, e_2 \rightarrow v_1\, e_2'$: goal completed.
    - Otherwise, e_2 is a value v_2. Since we have as hypothesis the typing judgment $\Gamma \vdash v_1 : \tau_2 \rightarrow \tau_1$, we know that v_1 necessarily has the shape $\texttt{fun } x \texttt{ -> } e$ (***classification*** lemma detailed below). Then we have

      $$e_1\, e_2 = (\texttt{fun } x \texttt{ -> } e)\, v_2 \rightarrow e[x := v_2]$$

      which completes our case.
- Other cases are similar.

Classification lemma for typed values. Let v be a value such that $\Gamma \vdash v : \tau$. Then:

- if $\tau = \texttt{int}$, then v has the shape n,
- if $\tau = \tau_1 \rightarrow \tau_2$, then v has the shape $\texttt{fun } x \texttt{ -> } e$.

Proof by case on the last rule applied in the derivation of $\Gamma \vdash v : \tau$*, knowing that the only two possible shapes for a value are:* n *or* $\texttt{fun } x \texttt{ -> } e$*.*

Type Preservation. If $\Gamma \vdash e : \tau$ and $e \to e'$ then $\Gamma \vdash e' : \tau$. Proof by induction on the derivation of $e \to e'$.

- Case $n_1 + n_2 \to n$ with $n = n_1 + n_2$. The hypothesis $\Gamma \vdash n_1 + n_2 : \tau$ implies $\tau = \texttt{int}$ (***inversion*** lemma detailed below). Moreover $\Gamma \vdash n : \texttt{int}$: goal completed.
- Case $e_1 + e_2 \to e_1' + e_2$ with $e_1 \to e_1'$. The premise gives as induction hypothesis "if $\Gamma \vdash e_1 : \tau'$, then $\Gamma \vdash e_1' : \tau'$".
  The hypothesis $\Gamma \vdash e_1 + e_2 : \tau$ implies $\tau = \texttt{int}$, $\Gamma \vdash e_1 : \texttt{int}$ and $\Gamma \vdash e_2 : \texttt{int}$ (inversion lemma). Thus by induction hypothesis $\Gamma \vdash e_1' : \texttt{int}$, from which we deduce the following typing derivation.

$$\frac{\Gamma \vdash e_1' : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

- Case $(\texttt{fun}\ x\ \texttt{in}\ e)\ v \to e[x := n]$. *Note: the corresponding rule has no premise, thus we have no induction hypothesis.*
  From the hypothesis $\Gamma \vdash (\texttt{fun}\ x\ \texttt{in}\ e)\ v : \tau$ we know there is τ' such that $\Gamma \vdash \texttt{fun}\ x\ \texttt{in}\ e : \tau' \to \tau$ and $\Gamma \vdash v : \tau'$ (inversion lemma) and from $\Gamma \vdash \texttt{fun}\ x\ \texttt{in}\ e : \tau' \to \tau$ we further deduce $\Gamma, x : \tau' \vdash e : \tau$ (inversion lemma again).
  We have on the one hand $\Gamma, x : \tau' \vdash e : \tau$ and on the other hand $\Gamma \vdash v : \tau'$, from which we deduce $\Gamma \vdash e[x := v] : \tau$ using a ***substitution*** lemma detailed below.
- The other cases are similar.

Inversion lemma.

- If $\Gamma \vdash e_1 + e_2 : \tau$ then $\tau = \texttt{int}$, $\Gamma \vdash e_1 : \texttt{int}$ and $\Gamma \vdash e_2 : \texttt{int}$.
- If $\Gamma \vdash e_1\ e_2 : \tau$ then there is τ' such that $\Gamma \vdash e_1 : \tau' \to \tau$ and $\Gamma \vdash e_2 : \tau'$.
- If $\Gamma \vdash \texttt{fun}\ x\ \texttt{->}\ e : \tau$ then there are τ_1 and τ_2 such that $\tau = \tau_1 \to \tau_2$ and $\Gamma, x : \tau_1 \vdash e : \tau_2$.

Proof by case on the last rule of the typing derivation.

Substitution lemma (replacing a typed variable by an identically typed expression preserves typing).

$$\text{If} \quad \Gamma, x : \tau' \vdash e : \tau \quad \text{and} \quad \Gamma \vdash e' : \tau' \quad \text{then} \quad \Gamma \vdash e[x := e'] : \tau.$$

Proof by induction on the derivation of $\Gamma, x : \tau' \vdash e : \tau$.

Type safety theorem. The following theorem combines the progress lemma and the type preservation lemme.

If $\Gamma \vdash e : \tau$ *and* $e \to^* e'$ *with* e' *not reducible, then* e' *is a value.*

The proof is by recurrence on the length of the reduction sequence $e \to^* e'$.

Summary: the safety property of typed expressions establishes a link between a static property (type consistency) and a dynamic property (evaluation without errors) of programs. It is still possible that a well-typed program fails to reach a value, in case the evaluation never ends. More generally, programming languages with a strict typing discipline are able to detects many errors early (at compilation time), which results in less errors at execution time.

2.4 Type verification for FUN

When source programs contain enough type information, it is rather easy to implement a ***type checker***, that is a(n other) program that checks whether a given source program is consistently typed. In this section we use caml to write a type checker from FUN programs, following the typing rules given in the previous sections. This program consists in a function `type_expr`, which takes as parameters an expression e and an environment Γ and which:

- returns the unique type that can be associated to e in the environment Γ if e is indeed consistent in this environment,
- fails otherwise.

We define a (caml) datatype to represent the types of the FUN language.

```
type typ =
  | TInt
  | TBool
  | TFun of typ * typ
```

We adapt the caml datatype representing abstract syntax trees of the FUN language to include some type annotations. Namely, we require type annotations for the argument of a function, and for recursive values. These annotations are the second argument of the constructors `Fun` and `Fix`.

```
  type bop = Add | Sub | Mul | Lt | Eq
  type expr =
    | Int of int
    | Bop of bop * expr * expr
    | Var of string
    | Let of string * expr * expr
    | If  of expr * expr * expr
    | App of expr * expr
    | Fun of string * typ * expr
    | Fix of string * typ * expr
```

Finally, we represent the environment as association tables relating variable identifiers (`string`) to FUN types (`typ`).

```
  module Env = Map.Make(String)
  type type_env = typ Env.t
```

The type checker is then a recursive function

```
type_expr: expr -> type_env -> typ
```

which reasons by case on the shape of the expression and applies the corresponding inference rule.

```
  let rec type_expr e env = match e with
```

An integer constant is always consistent, and its type is `int`.

```
    | Int _ -> TInt
```

$$\frac{}{\Gamma \vdash n : \texttt{int}}$$

A variable is seen as consistent when it exists in the environment.

```
    | Var(x) -> Env.find x env
```

$$\frac{}{\Gamma \vdash x : \Gamma(x)}$$

If it is not the case, the function `Env.find` will trigger an exception (namely: `Not_found`).
A binary operation requires each operand to be consistent, with the appropriate type.

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}$$

```
    | Bop(Add, e1, e2) ->
       let t1 = type_expr e1 env in
       let t2 = type_expr e2 env in
       if t1 = TypInt && t2 = TypInt then
         TypInt
       else
         failwith "type error"
```

Note that this code may fail at several distinct places: when type checking `e1` or `e2` if one or the other is not consistent, or explicitely with the last line if both `e1` and `e2` are consistent but one do not have the expected type.

A conditional expression requires the condition to be of boolean type, and both branch to have the same type.

$$\frac{\Gamma \vdash c : \texttt{bool} \qquad \begin{array}{c}\Gamma \vdash e_1 : \tau \\ \Gamma \vdash e_2 : \tau\end{array}}{\Gamma \vdash \texttt{if } c \texttt{ then } e_1 \texttt{ else } e_2 : \tau}$$

```
    | If(c, e1, e2) ->
       let tc = type_expr c env in
       let t1 = type_expr e1 env in
       let t2 = type_expr e2 env in
       if tc = TBool && t1 = t2 then
         t1
       else
         failwith "type error"
```

When a local variable x is introduced, we deduce its type from the expression e_1 that defines the value of the variable. The obtained type is then added to the environment used to typecheck the second expression e_2.

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau_2}$$

```
  | Let(x, e1, e2) ->
    let t1 = type_expr e1 env in
    type_expr e2 (Env.add x t1 env)
```

This case never fails by itself (although typechecking `e1` and `e2` may fail).

In the case of a function, we use the annotation to provide a type for the argument. We then check the body of the function, and use the returned type to build the full type of the function.

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau_1 \to \tau_2}$$

```
  | Fun(x, tx, e) ->
    let te = type_expr e (Env.add x tx env) in
    TFun(tx, te)
```

In the case of an application, we have to check that the left expression has the type of a function, and that the right expression has the type the function expects. These two points are two distincts reasons for which typechecking may fail.

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}$$

```
  | App(f, a) ->
    let tf = type_expr f env in
    let ta = type_expr a env in
    begin match tf with
      | TFun(tx, te) ->
        if tx = ta then
          te
        else
          failwith "type error"
      | _ -> failwith "type error"
    end
```

Finally, in the case of recursive value we typecheck the expression in an environment containing the type given by the annotation, and then check that the type of the expression also corresponds to the annotation.

$$\frac{\Gamma, f : \tau \vdash e : \tau}{\Gamma \vdash \texttt{fix f = e} : \tau}$$

```
  | Fix(f, t, e) ->
    let env' = Env.add f t env in
    let te = type_expr e env' in
    if te = t then
      t
    else
      failwith "type error"
```

2.5 Polymorphism

With the simply types seen in the beginning of the chapter, an expression such as

```
fun x -> x
```

may have several distinct types. However, it can have only one type at a time. In particular, in an expression such as

```
let f = fun x -> x in f f
```

we have to chose only one type for the `f`, and the expression cannot be typed. This is called a ***monomorphic type*** (literaly: *one shape*). *You may have noticed however that caml does not complain about the type of this expression.*

This section is about ***parametric polymorphism***, that is the possibility of using parametrized types, which cover many variants of a given shape of type. We extend the grammar of the types τ with two new elements:

- type ***variables***, or type ***parameters***, written α, β, ... denoting indeterminate types,
- a universal quantification $\forall\alpha.\tau$ denoting a ***polymorphic*** type, where the type variable α may, in τ, denote any type.

For the main fragment of FUN, the set of types is then defined by the extended grammar

$$\begin{array}{rcl}\tau & ::= & \texttt{int} \\ & | & \tau \to \tau \\ & | & \alpha \\ & | & \forall\alpha.\tau\end{array}$$

Instantiation. Polymorphic expressions have the following property: if an expression e has a polymorphic type $\forall\alpha.\tau$, then for any type τ' we can consider e to also be of type $\tau[\alpha := \tau']$ (the type τ in which each occurrence of the type parameter α has been replaced by τ').

$$\frac{\Gamma \vdash e : \forall\alpha.\tau}{\Gamma \vdash e : \tau[\alpha := \tau']}$$

The notion of type substitution $\tau[\alpha := \tau']$ is defined by a set of equations that are similar to the ones defining the substitution of expressions (previous chapter).

$$\begin{array}{rcl}
\mathtt{int}[\alpha := \tau'] & = & \mathtt{int} \\
\beta[\alpha := \tau'] & = & \begin{cases} \tau' & \text{if } \alpha = \beta \\ \beta & \text{if } \alpha \neq \beta \end{cases} \\
(\tau_1 \to \tau_2)[\alpha := \tau'] & = & \tau_1[\alpha := \tau'] \to \tau_2[\alpha := \tau'] \\
(\forall\beta.\tau)[\alpha := \tau'] & = & \begin{cases} \forall\beta.\tau & \text{if } \alpha = \beta \\ \forall\beta.\tau[\alpha := \tau'] & \text{if } \alpha \neq \beta \text{ and } \beta \notin \mathsf{fv}(\tau') \end{cases}
\end{array}$$

The notion of free type variable is also defined similarly.

$$\begin{array}{rcl}
\mathsf{fv}(\mathtt{int}) & = & \emptyset \\
\mathsf{fv}(\alpha) & = & \{\alpha\} \\
\mathsf{fv}(\tau_1 \to \tau_2) & = & \mathsf{fv}(\tau_1) \cup \mathsf{fv}(\tau_2) \\
\mathsf{fv}(\forall\alpha.\tau) & = & \mathsf{fv}(\tau) \setminus \{\alpha\}
\end{array}$$

Generalisation. When an expression has a type τ containing a parameter α, and this parameter *is not constrained in any way* by the context Γ, then we can consider e as a polymorphic expression, with type $\forall\alpha.\tau$.

$$\frac{\Gamma \vdash e : \tau \qquad \alpha \notin \mathsf{fv}(\Gamma)}{\Gamma \vdash e : \forall\alpha.\tau}$$

In this rule, the condition "α is not constrained by Γ" is stated as "α does not appear in Γ".

Formally, the set of free type variables of an environment $\Gamma = \{x_1 : \tau_1, \dots, x_n : \tau_n\}$ is defined by the equation

$$\mathsf{fv}(\{x_1 : \tau_1, \dots, x_n : \tau_n\}) \quad = \quad \bigcup_{1 \le i \le n} \mathsf{fv}(\tau_i)$$

Note that this concerns only *type variables.* The variables x_i, which are variables of expressions, are out of scope.

Examples and counter-examples. We can now give to the identity function `fun x -> x` the polymorphic type $\forall\alpha.\alpha \to \alpha$, stating that this function takes an argument of *any type* and returns a result of *the same type.*

$$\dfrac{\dfrac{\dfrac{}{\mathtt{x} : \alpha \vdash \mathtt{x} : \alpha}}{\vdash \mathtt{fun\ x\ \text{->}\ x} : \alpha \to \alpha} \qquad \alpha \notin \mathsf{fv}(\emptyset)}{\vdash \mathtt{fun\ x\ \text{->}\ x} : \forall\alpha.\alpha \to \alpha}$$

The key here is that `fun x -> x` can have the type $\alpha \to \alpha$ in the empty context, and that the empty context, in particular, puts no constraint on α.

It is then possible to type the expression `let f = fun x -> x in f f`. Indeed, in the part of the derivation tree dealing with the expression `f f`, we have an environment $\Gamma = \{f : \forall\alpha.\alpha \to \alpha\}$ with which we can complete the derivation as follows.

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \mathtt{f} : \forall\alpha.\alpha \to \alpha}}{\Gamma \vdash \mathtt{f} : (\mathtt{int} \to \mathtt{int}) \to (\mathtt{int} \to \mathtt{int})} \qquad \dfrac{\dfrac{}{\Gamma \vdash \mathtt{f} : \forall\alpha.\alpha \to \alpha}}{\Gamma \vdash \mathtt{f} : \mathtt{int} \to \mathtt{int}}}{\Gamma \vdash \mathtt{f\ f} : \mathtt{int} \to \mathtt{int}}$$

Note that this is not the only solution: we could also have replaced the concrete type `int` by any type variable β, and even remark that the resulting type could be generalized, since β does not appear free in Γ.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \texttt{f} : \forall\alpha.\alpha\to\alpha}}{\Gamma \vdash \texttt{f} : (\beta\to\beta)\to(\beta\to\beta)} \qquad \dfrac{\dfrac{}{\Gamma \vdash \texttt{f} : \forall\alpha.\alpha\to\alpha}}{\Gamma \vdash \texttt{f} : \beta\to\beta}}{\Gamma \vdash \texttt{f f} : \beta\to\beta} \qquad \beta\notin \mathsf{fv}(\Gamma)}{\Gamma \vdash \texttt{f f} : \forall\beta.\beta\to\beta}$$

This system however *does not* allow the type $\alpha \to \forall\alpha.\alpha$ for the identity function **`fun`** `x -> x`. Indeed, this would require giving to x the type $\forall\alpha.\alpha$ in a context $\Gamma = \{\texttt{x} : \alpha\}$. Our axiom rule only allows the derivation of $\Gamma \vdash \texttt{x} : \alpha$, in which α cannot be generalized, since it appears in Γ. Thus we (fortunately) cannot use polymorphic types to allow the ill-formed expression `(`**`fun`**` x -> x) 5 37`.

Let us show that composition function **`fun`** `f ->` **`fun`** `g ->` **`fun`** `x -> g (f x)` has the polymorphic type $\forall\alpha\beta\gamma.(\alpha\to\beta)\to(\beta\to\gamma)\to(\alpha\to\gamma)$. Write Γ the environment $\{\texttt{f} : \alpha\to\beta, \texttt{g} : \beta\to\gamma, \texttt{x} : \alpha\}$. We can build the following derivation (where three consecutive uses of the generalization rule are merged, as well as three consecutive uses of the typing rule for functions).

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \texttt{g} : \beta\to\gamma} \qquad \dfrac{\dfrac{}{\Gamma \vdash \texttt{f} : \alpha\to\beta} \qquad \dfrac{}{\Gamma \vdash \texttt{x} : \alpha}}{\Gamma \vdash \texttt{f x} : \beta}}{\Gamma \vdash \texttt{g (f x)} : \gamma}}{\vdash \texttt{fun f -> fun g -> fun x -> g (f x)} : (\alpha\to\beta)\to(\beta\to\gamma)\to(\alpha\to\gamma)} \qquad \alpha,\beta,\gamma \notin \mathsf{fv}(\emptyset)}{\vdash \texttt{fun f -> fun g -> fun x -> g (f x)} : \forall\alpha\beta\gamma.(\alpha\to\beta)\to(\beta\to\gamma)\to(\alpha\to\gamma)}$$

Exercise: show that this composition function can also have the type $\forall\alpha\beta.(\alpha\to\beta)\to\forall\gamma.(\beta\to\gamma)\to(\alpha\to\gamma)$.

Hindley-Milner system. Without annotations from the programmer, the two following questions about polymorphic types in FUN are undecidable:

- type inference: an expression e being given, determine whether there is a type τ such that $\Gamma \vdash e : \tau$ (and provide the type),
- type verification: an expression e and a type τ being given, determine whether $\Gamma \vdash e : \tau$.

These undecidability results still hold for any language extending the FUN kernel.

If we wish to check the type consistency of a program, or infer the type of a program, we have to either require some amount of annotations from the programmer, or restrict the use of polymorphism. Each language sets is own balance between the amount of annotation and the expressiveness of the type system.

In caml, polymorphism is restricted by a simple fact: we cannot write any explicit quantifier in a type. Instead, every type variable that is globally free is implicitly considered to be universally quantified. Thus, the caml type for the first projection of a pair

```
fst: 'a * 'b -> 'a
```

is actually the generalized type $\forall\alpha\beta.\alpha\times\beta\to\alpha$. Similarly, the caml type for the left iterator of a list

```
List.fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

has to be understood as $\forall\alpha\beta.(\alpha\to\beta\to\alpha)\to\alpha\to\beta\,\texttt{list}\to\alpha$.

This restricted polymorphism is common to all languages of the ML family, and called the ***Hindley-Milner system***. It only allows "prenex" quantification, and distinguishes the notion of ***type*** τ without quantification, and the notion ***type scheme*** σ which is a type extended with an arbitrary number of global quantifiers.

For our fragment of FUN, this can be described by the following grammar.

$$
\begin{array}{rcl}
\tau & ::= & \texttt{int} \\
 & \mid & \tau \rightarrow \tau \\
 & \mid & \alpha \\
 & & \\
\sigma & ::= & \forall\alpha_1 \ldots \forall\alpha_n.\tau
\end{array}
$$

In this system, we can work with type schemes such as $\forall\alpha.\alpha \rightarrow \alpha$ and $\forall\alpha\beta\gamma.(\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \gamma)$, but we cannot express a type with the shape $(\forall\alpha.\alpha \rightarrow \alpha) \rightarrow (\forall\alpha.\alpha \rightarrow \alpha)$.

In the Hindley-Milner system, we adapt contexts and typing judgments to allow the association of a type scheme to a variable or an expression:

$$x_1 : \sigma_1, \ldots, x_n : \sigma_n \vdash e : \sigma$$

Note that a type scheme with zero quantifier is just a type: this new shape may also be used to deal with simple types.

Typing rules are also adapted in a way such that type schemes are authorized only at some specific places.

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash e_1 : \sigma_1 \qquad \Gamma, x : \sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \sigma_2}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau_1 \rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}$$

$$\frac{\Gamma \vdash e : \forall\alpha.\sigma}{\Gamma \vdash e : \sigma[\alpha := \tau]} \qquad \frac{\Gamma \vdash e : \sigma \qquad \alpha \notin \mathsf{fv}(\Gamma)}{\Gamma \vdash e : \forall\alpha.\sigma}$$

The generalization of the type of an expression is only allowed at two places:
— at the root of the program,
— for the argument of a `let` definition.

Indeed, the typing rule for `let` contains type schemes, while the type of an application requires both the type of the function and the type of its arguments to be simple types.

The Hindley-Milner type system has two notable properties:
— type checking *and* type inference are decidable (see next section),
— the system ensures type safety: the evaluation of the well-typed program cannot be stopped by an inconsistent operation (the proof extends the one already given for simple types).

2.6 Type inference

Writing a type checker for simple types in FUN was relatively easy, thanks to two properties of that simple system:
— typing rules were *syntax-directed*, which means that for any shape of expression there was only one inference rule that could possibly apply,
— some type annotations were required at the few places where we did not have a simple way of guessing the right type (namely, the parameter of a function and a recursive value).

Conversely, the Hindley-Milner system does not satisfy the first property: the two rules for instantiation and generalization may be applied to any expression. Moreover, we aim at ***full inference***, that is we do not want any annotation.

Syntax-dorected Hindley-Milner system. As first step, let us define a syntax-directed variant of the Hindley-Milner system, by restricting the places where generalization and instantiation may be applied.
— We allow the instantiation of a type variable only when recovering from the environment the type scheme associated to a variable. We obtain this by merging the instantiation rule and the axiom $\Gamma \vdash x : \Gamma(x)$, keeping only one rule combining two effects:

    1. fetch the type scheme σ associated to x in Γ,
    2. instiantiate *all* universal variables of σ (thus obtaining a simple type).
- Symmetrically, we allow generalization only for `let` definitions. We obtain this by merging the generalization rule and the `let` rule, keeping only one rule combining two effects:
    1. type the expression e_1 in the environment Γ, and call τ_1 the obtained type,
    2. generalize *all* the free variables of τ_1 that can possibly be genezalized, to obtain a type schema σ_1,
    3. type e_2 in the extended environment where x is associated to σ_1.

Remark that type schemes are not allowed at any place other than the context anymore.

Here is a type derivation in this system, with $\Gamma_1 = \{x : \alpha \to \alpha, y : \alpha\}$ and $\Gamma_2 = \{f : \forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash x : \alpha \to \alpha} \quad \dfrac{\dfrac{}{\Gamma_1 \vdash x : \alpha \to \alpha} \quad \dfrac{}{\Gamma_1 \vdash y : \alpha}}{\Gamma_1 \vdash x\,y : \alpha}}{x : \alpha \to \alpha, y : \alpha \vdash x\,(x\,y) : \alpha}}{x : \alpha \to \alpha \vdash \mathtt{fun}\ y\ \mathtt{\text{->}}\ x\,(x\,y) : \alpha \to \alpha}}{\vdash \mathtt{fun}\ x\ \mathtt{\text{->}}\ \mathtt{fun}\ y\ \mathtt{\text{->}}\ x\,(x\,y) : (\alpha \to \alpha) \to (\alpha \to \alpha)} \quad \dfrac{\dfrac{}{\Gamma_2 \vdash f : (\mathtt{int} \to \mathtt{int}) \to (\mathtt{int} \to \mathtt{int})} \quad \dfrac{\dfrac{\dfrac{}{\Gamma_2, z : \mathtt{int} \vdash z : \mathtt{int}} \quad \dfrac{}{\Gamma_2, z : \mathtt{ir}}}{\Gamma_2, z : \mathtt{int} \vdash z\mathtt{+1} : \mathtt{ir}}}{\Gamma_2 \vdash \mathtt{fun}\ z\ \mathtt{\text{->}}\ z\mathtt{+1} : \mathtt{int}}}{f : \forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha) \vdash f\,(\mathtt{fun}\ z\ \mathtt{\text{->}}\ z\mathtt{+1}) : \mathtt{int} \to \mathtt{int}}}{\vdash \mathtt{let}\ f = \mathtt{fun}\ x\ \mathtt{\text{->}}\ \mathtt{fun}\ y\ \mathtt{\text{->}}\ x\,(x\,y)\ \mathtt{in}\ f\,(\mathtt{fun}\ z\ \mathtt{\text{->}}\ z\mathtt{+1}) : \mathtt{int} \to \mathtt{int}}$$

It can be proven that this syntax-directed variant of the Hindley-Milner system is equivalent to the original version, since it allows the derivation of essentially the same typing judgments.

Constraint generation and unification. The second step consists in performing inference without any annotation. The algorithm W implements for this the following ideas.
- Each time we need to introduce a new type which cannot be computed in a straightforward way, we introduce instead a new type variable. This concerns the type of the parameter of a function, and also the types used for instantiating the universal variables of a type scheme $\Gamma(x)$.
- The actual types represneted by these type variables are computed *later*, when checking/solving the constraints related to the typing rules (for instance, for application or addition).

When a typing rule requires an identity between two types τ_1 and τ_2 containing type variables $\alpha_1, ..., \alpha_n$, we try to ***unify*** these two types, that is we look for an instantiation f of the type variables α_i such that $f(\tau_1) = f(\tau_2)$.
Examples of unification:
- If $\tau_1 = \alpha \to \mathtt{int}$ and $\tau_2 = (\mathtt{int} \to \mathtt{int}) \to \beta$, we can unify the types τ_1 and τ_2 using the instantiation $[\alpha \mapsto \mathtt{int} \to \mathtt{int}, \beta \mapsto \mathtt{int}]$.
- If $\tau_1 = (\alpha \to \mathtt{int}) \to (\alpha \to \mathtt{int})$ and $\tau_2 = \beta \to \beta$, we can unify the types τ_1 and τ_2 using the instantiation $[\beta \mapsto \alpha \to \mathtt{int}]$.
- The types $\alpha \to \mathtt{int}$ and $\mathtt{int}$ cannot be unified.
- The types $\alpha \to \mathtt{int}$ and α cannot be unified.

Unification criteria:
- τ is always unified with itself,
- unification of $\tau_1 \to \tau_1'$ with $\tau_2 \to \tau_2'$ requires unifying τ_1 with τ_2 and τ_1' with τ_2',
- unification of τ with a variable α, when α does not appears in τ, is done by instantiating α by τ (if α appears in τ, unification is not possible),
- in any other case, unification is not possible.

Algorithm W, example. Let us infer a type for the expression `let` f = `fun` x `->` `fun` y `->` $x(xy)$ `in` f `(fun`
We first focus on `fun` x `->` `fun` y `->` $x\,(x\,y)$, proceeding as follows.
- The variable x is given the type α, where α is a new type variable.
- Similarly, the variable y is given the type β with β a new type variable.
- Then we type the expression $x\,(x\,y)$.
    - The application $x\ y$ requires the type α of x to be a functional type, whose parameter corresponds to the type β of y. Thus we unify α with $\beta \to \gamma$, for γ some new type variable, and define a first element of instantiation: $\alpha = \beta \to \gamma$.
    - Therefore, the application $x\,y$ has the type γ.
    - The application $x\,(x\,y)$ requires the type $\alpha = \beta \to \gamma$ of x to be a functional type, whose parameter corresponds to the type γ of $x\ y$. Thus we unify $\beta \to \gamma$ with

$\gamma \to \delta$, for δ a new type variable. Then we get new instantiation information: $\gamma = \delta = \beta$.

We also deduce that the application $x\,(x\;y)$ has the type β.

- Finally, `fun` x `->` `fun` y `->` $x\,(x\;y)$ get the type $\alpha \to (\beta \to \beta)$, which is $(\beta \to \beta) \to (\beta \to \beta)$, and in the empty typing context this can be generalized as $\forall\beta.(\beta \to \beta) \to (\beta \to \beta)$.

Let us now focus on the expression f (`fun` z `->` z+1), in a context where f has the generalized type $\forall\beta.(\beta \to \beta) \to (\beta \to \beta)$. This expression is an application: we first type both subexpressions, and then solve the constraints.

- We type f by fetching the type scheme $\forall\beta.(\beta \to \beta) \to (\beta \to \beta)$ from the context, and instantiating the universal variable β with a new type variable ζ. We get for f the type $(\zeta \to \zeta) \to (\zeta \to \zeta)$.
- Typing of `fun` z `->` z+1.
  - The variable z is given the type η, where η is a new type variable.
  - Then we type the addition z+1.
    - z has the type η, that has to be unified with `int`. Then we define η = `int`.
    - 1 has the type `int`, that has to be unified with `int`: done already.

    Thus z+1 has the type `int`.

  Thus `fun` z `->` z+1 has the type $\eta \to$ `int`, which is `int` $\to$ `int`.
- To type the application itself, we have to unify the type $(\zeta \to \zeta) \to (\zeta \to \zeta)$ of f with the type (`int` $\to$ `int`) $\to \theta$ of a function that takes a parameter of type `int` $\to$ `int` (the type of the argument `fun` z `->` z+1), with θ a new type variable. Thus we complete the instantiation with ζ = `int` and θ = `int` $\to$ `int`.

Finally, the expression f (`fun` z `->` z+1) has the type θ = `int` $\to$ `int`, and we conclude that

$\vdash$ `let` f = `fun` x `->` `fun` y `->` $x\,(x\;y)$ `in` f (`fun` z `->` z+1) : `int` $\to$ `int`

Algorithm W, in caml. We take the raw abstract syntax of FUN, without type annotations.

```
type bop = Add | Sub | Mul | Lt | Eq
type expr =
  | Int of int
  | Bop of bop * expr * expr
  | Var of string
  | Let of string * expr * expr
  | If  of expr * expr * expr
  | App of expr * expr
  | Fun of string * expr
  | Fix of string * expr
```

We extend simple types with a notion of type variable, and define a type scheme as a pair of a simply type `typ` and a set `vars` of universally quantified type variables.

```
type typ =
  | TInt
  | TBool
  | TFun of typ * typ
  | TVar of string

module VSet = Set.Make(String)
type schema = { vars: VSet.t; typ: typ }
```

A typing environment associate a type scheme to each variable of the program.

```
module SMap = Map.Make(String)
type env = schema SMap.t
```

We will build a function `type_inference: expr -> typ` that computes a type for the expression given as parameter, trying to get a type that is as general as possible (which means: that does not instantiate type variables more than what is necessary). This function uses an auxiliary function `new_var: unit -> string` for creating new type variables.

```
let type_inference t =
  let new_var =
    let cpt = ref 0 in
    fun () -> incr cpt; Printf.sprintf "tvar_%i" !cpt
  in
```

These type variables will be associated to concrete types (or at least more precise types) when new constraints are discovered and analyzed. These associations are recorded in a hash table `subst`, that grows as the inference proceeds.

```
  let subst = Hashtbl.create 32 in
```

Thus, the types used during inference will contain type variables, some of which will have a definition in `subst`. To read such a type, we use auxiliary unfolding functions `unfold` and `unfold_full`, which take a type τ a replace its type variables by their definition in `subst` (for those that have one). The function `unfold` is a “shallow” replacement: it replaces only what is necessary to see the superficial structure of the type and in particular to distinguish between the cases `TInt`, `TBool`, `TFun` or `TVar`. The function `unfold_full` performs a complete replacement, in order to know the full type (this one is only used to decode the final result of the inference).

```
  let rec unfold t = match t with
    | TInt | TBool | TFun _ -> t
    | TVar a ->
       if Hashtbl.mem subst a then
         unfold (Hashtbl.find subst a)
       else
         t
  in

  let rec unfold_full t = match unfold t with
    | TFun(t1, t2) -> TFun(unfold_full t1, unfold_full t2)
    | t -> t
  in
```

Example of the use of unfolding: to check whether a type variable α appears in a type τ, we reason by case on the shape of the type τ. We insert a call to the (shallow) unfolding function before the match to reveal the actual structure of the type if it is already known.

```
  let rec occur a t = match unfold t with
    | TInt | TBool -> false
    | TVar b -> a=b
    | TFun(t1, t2) -> occur a t1 || occur a t2
  in
```

Algorithm W itself reasons by case on the shape of the analyzed expression. In the case of an integer constant, we just return the base type `TInt`. In the case of a binary operation we infer a type for each operand, and then check that the obtained types `t1` and `t2` are consistent with the expected type (for instance: `TInt`). This consistency check is performed by an auxiliary function `unify`, which records on the fly the new associations between type variables and concrete types.

```
  let rec w e env = match e with
    | Int _ ->
       TInt

    | Bop((Add | Sub | Mul), e1, e2) ->
       let t1 = w e1 env in
       let t2 = w e2 env in
       unify t1 TInt; unify t2 TInt;
       TInt

    | Bop(Lt, e1, e2) ->
       let t1 = w e1 env in
       let t2 = w e2 env in
       unify t1 TInt; unify t2 TInt;
       TBool

    | Bop(Eq, e1, e2) ->
       let t1 = w e1 env in
       let t2 = w e2 env in
       unify t1 t2;
       TBool
    | If(c, e1, e2) ->
      let tc = w c env in
      let t1 = w e1 env in
      let t2 = w e2 env in
      unify tc TBool;
      unify t1 t2;
      t1
```

In the case of a variable, we instantiate the type scheme obtained in the environment using a dedicated auxiliary function `instantiate`, which replace each universal variable by a fresh type variable.

```
    | Var x -> instantiate (SMap.find x env)
```

Conversely, in the case of a `let` we generalize the type infered for the expression `e1` using a dedicated auxiliary function `generalize`, which returns a type scheme in which type variables are generalized whenever this is possible.

```
    | Let(x, e1, e2) ->
      let t1 = w e1 env in
      let st1 = generalize t1 env in
      let env' = SMap.add x st1 env in
      w e2 env'
```

When typing a function, we introduce a new type variable for the type of the parameter. Since the type of a parameter cannot be generalized, we fix an empty set of universal variables, and then we type the body of the function in this extended environment.

```
    | Fun(x, e) ->
      let v = new_var() in
      let env = SMap.add x {vars = VSet.empty; typ = TVar v} env in
      let t = w e env in
      TFun(TVar v, t)
```

When typing an application we first infer types for each subexpression, and then try to solve the constraints of the application rule: the type `t1` of the left member `e1` must be a functional type, and the type `t2` of the right member must be the expected parameter type of the function.

```
    | App(e1, e2) ->
      let t1 = w e1 env in
      let t2 = w e2 env in
      let v = TVar (new_var()) in
      unify t1 (TFun(t2, v));
      v
```

Finally, a recursive value `Fix`(f, e) is typed using a new type variable that is used twice: it is first associated to f in the typing environment when analyzing e, and then it is unified with the type obtained for e.

```
    | Fix(f, e) ->
      let v = new_var() in
      let env = SMap.add f {vars = VSet.empty; typ = TVar v} env in
      let t = w e env in
      unify t (TVar v);
      t

  in
  unfold_full (w t SMap.empty)
```

We now give the definitions of the auxiliary functions enumerated above. The type constraints are solved by a unification algorithm, which takes two type parameters τ_1 and τ_2 and try to instiantiate the type variables of τ_1 and τ_2 to make both types identical. In cases where both types have the same shape, it is enough to propagate unification on the immediate subexpressions.

```
let rec unify t1 t2 = match unfold t1, unfold t2 with
  | TInt, TInt   -> ()
  | TBool, TBool -> ()
  | TFun(t1, t1'), TFun(t2, t2') -> unify t1 t2; unify t1' t2'
```

When one the type is a variable, there are several possible situations.

— If τ_1 and τ_2 are the same variable, there is nothing to be done: the types are equal already.
— If one type is a variable α, and the other one, written τ, is another variable or a different shape of type, then we add the association $[\alpha \mapsto \tau]$ in the instantiation table `subst`. Note there is an exception to this main idea: if the variable α appears in τ, then unification fails, since α cannot be a part of its own definition.

```
  | TVar a, TVar b when a=a -> ()
  | TVar a, t | t, TVar a ->
     if occur a t then
       failwith "unification error"
     else
       Hashtbl.add subst a t
```

Unification fails in any other case.

```
  | _, _ -> failwith "unification error"
in
```

The instantiation auxiliary function creates a new type variable for each universal variable of the type scheme given as argument, and replace the quantified variables.

```
let instantiate s =
  let renaming = VSet.fold
      (fun v r -> SMap.add v (TVar(new_var())) r)
      s.vars
      SMap.empty
  in
  let rec rename t = match unfold t with
    | TVar a as t -> (try SMap.find a renaming with Not_found -> t)
    | TInt -> TInt
    | TBool -> TBool
    | TFun(t1, t2) -> TFun(rename t1, rename t2)
  in
  rename s.typ
in
```

The generalization auxiliary function takes as parameters a type τ and an environment Γ, and computes the set of free type variables in τ which do not appear in the environment Γ. These variables are then declared "universal".

```
let rec fvars t = match unfold t with
  | TInt | TBool -> VSet.empty
  | TFun(t1, t2) -> VSet.union (fvars t1) (fvars t2)
  | TVar x -> VSet.singleton x
in
let rec schema_fvars s =
  VSet.diff (fvars s.typ) s.vars
in
let generalize t env =
  let fvt = fvars t in
  let fvenv = SMap.fold
      (fun _ s vs -> VSet.union (schema_fvars s) vs)
      env
      VSet.empty
  in
  {vars = VSet.diff fvt fvenv; typ=t}
in
```