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.

Plan

Semantics of a functional language

1. Semantics and interpretation
2. Types and safety

Assembly

3. The target: assembly code
4. Optimisations

Implementation of a functional language

5. Compilation
6. Automatic memory management

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

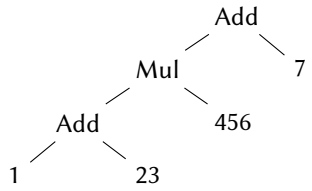
The arithmetic features of the FUN language enable a variety of numerical calculations. For instance:

```
> (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, appear in the concrete syntax for the correct association between operators and their operands, but 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)` ;
- 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`.

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

1.2 Inductive structures

Abstract syntax tree have an ***inductive*** structure, which can be described 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 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 a finite number of times. 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.

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 $f(e_i)$ for each $e_i \in E$.

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).

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

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

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

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 given any n element e_1, ..., e_n all $e_i \in E$ satisfy $P(e_i)$, then the property P is still valid for the combined element $c(e_1, \ldots, e_n)$. In other words, for any constructor and any elements, $P(e_{i_1}) \wedge \ldots \wedge P(e_{i_k}) \implies P(c(e_1, \ldots, e_n))$.

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 describes ***inductive cases*** (or *recursive* cases, one for each non-nullary constructor). In the second point hypotheses $P(e_{i_1})$ to $P(e_{i_n})$ that can be used for justifying $P(c(e_1, \dots, e_n))$ 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 $\mathsf{Add}(e_1, e_2) = e_1 \oplus e_2$.

— 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 environment providing the value of the variable `n` that `f` refers to". More generally, a function-value is a function together with an environment 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 evaluate the body `e` of this function under the new environment 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 $c = (x, e, \rho)$. 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` $f\ v\ \rho$.

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
  ...
  | Fix(f, Fun(x, e)) ->
     let rec v = VFix(Fun(x, e), f, v, env) in
     v
  | Fix _ -> failwith "unauthorized␣operation"
  | 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')

and force v = match v with
  | VFix(e, f, v, env) -> 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 of 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 is 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.

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

Remark in the equation concerning addition that the symbol + in $e_1 \oplus e_2$ 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 a 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.

$$\begin{aligned}
\mathsf{fv}(\mathsf{n}) &= \emptyset \\
\mathsf{fv}(\mathsf{x}) &= \{x\} \\
\mathsf{fv}(e_1 \oplus e_2) &= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\
\mathsf{fv}(e_1 \otimes e_2) &= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2) \\
\mathsf{fv}(\texttt{let x = } e_1 \texttt{ in } e_2) &= \mathsf{fv}(e_1) \cup (\mathsf{fv}(e_2) \setminus \{x\}) \\
\mathsf{fv}(\texttt{fun x -> } e) &= \mathsf{fv}(e) \setminus \{x\} \\
\mathsf{fv}(e_1\ e_2) &= \mathsf{fv}(e_1) \cup \mathsf{fv}(e_2)
\end{aligned}$$

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 define the set of values that our expression will produce: integer numbers, and functions.

$$\begin{aligned}
v \ &::= \ n \\
&\mid \ \texttt{fun } x \texttt{ -> } e
\end{aligned}$$

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.

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

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

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

  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.

- $\mathsf{eval}(\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2) = \mathsf{eval}(e_2[x := \mathsf{eval}(e_1)])$. 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.

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

- $\mathsf{eval}(\texttt{fun } x \texttt{ -> } e) = \texttt{fun } x \texttt{ -> } e$. 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.

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

- $\mathsf{eval}(e_1\ e_2) = \mathsf{eval}(e[x = \mathsf{eval}(e_2)])$ if $\mathsf{eval}(e_1) = \texttt{fun } x \texttt{ -> } e$. 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.

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

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

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

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

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

where the argument e_2 is substituted unevaluated.

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

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

This *call by name* semantics is *almost* equivalent to the *call by value* semantics: they mostly allow the derivation of the same judgments $e \Longrightarrow v$. 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

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

with the call by name semantics as follows.

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

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

However, these two semantics are *not fully equivalent*: there are some judgments $e \Longrightarrow v$ 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 $e \Longrightarrow v$. 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

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

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

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

- Case $n \implies n$: immediate, since n is a value, and $\mathrm{fv}(n) \subseteq \mathrm{fv}(n)$.
- Case `fun x -> e` $\implies$ `fun x -> e`: immediate as well.
- Case e_1 `+` $e_2 \implies n_1 + n_2$ with $e_1 \implies n_1$ and $e_2 \implies n_2$. By definition $n_1 + n_2$ is an integer value. Moreover $\mathrm{fv}(n_1 + n_2) = \emptyset \subseteq \mathrm{fv}(e_1$ `+` $e_2)$. *Note: induction hypotheses concerning e_1 and e_2 are not used here.*
- Case `let` x `=` e_1 `in` $e_2 \implies v$ with $e_2[x := e_1] \implies v$. The premise provides as induction hypothesis that $\mathrm{fv}(v) \subseteq \mathrm{fv}(e_2[x := e_1])$ (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.

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

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

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

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

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

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***, provides 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 perform 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)`.

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

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.

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

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.

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

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

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

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 $e \to e'$ describes elementary computation steps. We write:

— $e \to e'$ when 1 computation step leads from e to e', and
— $e \to^* e'$ 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 $e \to e'$. 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,

$$e \Longrightarrow v \qquad \text{if and only if} \qquad e \to^* v$$

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

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

— 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 that 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 a 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 integers. Following the same principles, the type of a function makes 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 justifying 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 a 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:

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

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

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

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

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

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:

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

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

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 $\Gamma \vdash \texttt{5 37} : \tau$, and the application of this rule requires justifying the two premises $\Gamma \vdash \texttt{5} : \tau' \to \tau$ and $\Gamma \vdash \texttt{37} : \tau'$ (the type τ' may be chosen freely, but it must be the same for both judgments). However, justifying a premise $\Gamma \vdash \texttt{5} : \tau' \to \tau$ is impossible: no rule allows giving a functional type to an integer constant (the only rule that could be applied for an integer constant would give the typing judgment $\Gamma \vdash \texttt{5} : \texttt{int}$).

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 $\Gamma \vdash \texttt{fun x -> x x} : \tau$ would necessarily have the shape

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

However, the premise $\Gamma, \texttt{x} : \tau_1 \vdash \texttt{x} : \tau_1 \to \tau_2$ 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 $\mathrm{fv}(e) \subseteq \mathrm{dom}(\Gamma)$, by induction on $\Gamma \vdash e : \tau$.

— Case $\Gamma \vdash n : \texttt{int}$. We have $\mathrm{fv}(n) = \emptyset$, and of course $\emptyset \subseteq \mathrm{dom}(\Gamma)$.

— Case $\Gamma \vdash x : \Gamma(x)$. We have $\mathrm{fv}(x) = \{x\}$, and the application of the rule indeed assumes that $\Gamma(x)$ is defined, which means $x \in \mathrm{dom}(\Gamma)$.

— Case $\Gamma \vdash e_1 + e_2 : \texttt{int}$, with premises $\Gamma \vdash e_1 : \texttt{int}$ and $\Gamma \vdash e_2 : \texttt{int}$. The premises give two induction hypotheses $\mathrm{fv}(e_1) \subseteq \mathrm{dom}(\Gamma)$ and $\mathrm{fv}(e_2) \subseteq \mathrm{dom}(\Gamma)$. By definition of free variables we have $\mathrm{fv}(e_1 + e_2) = \mathrm{fv}(e_1) \cup \mathrm{fv}(e_2)$. With the induction hypotheses we deduce that $\mathrm{fv}(e_1) \cup \mathrm{fv}(e_2) \subseteq \mathrm{dom}(\Gamma)$, and therefore $\mathrm{fv}(e_1 + e_2) \subseteq \mathrm{dom}(\Gamma)$.

— Case $\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau_2$, with premises $\Gamma \vdash e_1 : \tau_1$ and $\Gamma, x : \tau_1 \vdash e_2 : \tau_2$. The premises give two induction hypotheses $\mathrm{fv}(e_1) \subseteq \mathrm{dom}(\Gamma)$ and $\mathrm{fv}(e_2) \subseteq \mathrm{dom}(\Gamma) \cup \{x\}$ (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 $\mathrm{fv}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) = \mathrm{fv}(e_1) \cup (\mathrm{fv}(e_2) \setminus \{x\})$. The first induction hypothesis ensures that $\mathrm{fv}(e_1) \subseteq \mathrm{dom}(\Gamma)$. The second induction hypothesis ensures that $\mathrm{fv}(e_2) \subseteq \mathrm{dom}(\Gamma) \cup \{x\}$, from which we deduce $\mathrm{fv}(e_2) \setminus \{x\} \subseteq \mathrm{dom}(\Gamma)$. Therefore we have $\mathrm{fv}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) \subseteq \mathrm{dom}(\Gamma)$.
— 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.

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

We also introduce three additional typing rules for the missing constructs.

— An expression $e_1 < e_2$ is consistent whenever the expressions e_1 and e_2 are numbers. The result is of type `bool`.

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

— 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 or 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 stated 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.

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

— ***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.

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

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.

$$(e_1 : \tau) \ \to \ (e_2 : \tau) \ \to \ (e_3 : \tau) \ \to \ \ldots$$

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 $e \to e'$. Let us consider the simple types for the fragment of the FUN language defined by the following rules.

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

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

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

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 $\Gamma \vdash \texttt{fun} \ x \ \texttt{->} \ e : \tau_1 \to \tau_2$. Then $\texttt{fun} \ x \ \texttt{->} \ e$ is a value.

— Case $\Gamma \vdash e_1 \, e_2 : \tau_1$, with $\Gamma \vdash e_1 : \tau_2 \to \tau_1$ and $\Gamma \vdash e_2 : \tau_2$. Induction hypotheses give us the two following disjunctions.

1. e_1 is a value or $e_1 \to e_1'$,
2. e_2 is a value or $e_2 \to e_2'$.

We reason by case on these disjunctions.

— If $e_1 \to e_1'$, then $e_1 \, e_2 \to e_1' \, e_2$: goal completed.

— Otherwise, e_1 is a value v_1.

— If $e_2 \to e_2'$, then $v_1 \, e_2 \to v_1 \, e_2'$: goal completed.

— Otherwise, e_2 is a value v_2. Since we have as hypothesis the typing judgment $\Gamma \vdash v_1 : \tau_2 \to \tau_1$, we know that v_1 necessarily has the shape $\texttt{fun} \ x \ \texttt{->} \ e$ (***classification*** lemma detailed below). Then we have

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

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 $\tau = \tau_1 \to \tau_2$, 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 $e \to e'$. The property is "for all τ, if $\Gamma \vdash e : \tau$ then $\Gamma \vdash e' : \tau$".

- Case $n_1 \texttt{ + } n_2 \to n$ with $n = n_1 + n_2$. The hypothesis $\Gamma \vdash n_1 \texttt{ + } n_2 : \tau$ implies $\tau = \texttt{int}$ (***inversion*** lemma detailed below). Moreover $\Gamma \vdash n : \texttt{int}$: goal completed.
- Case $e_1 \texttt{ + } e_2 \to e_1' \texttt{ + } e_2$ 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 $\Gamma \vdash e_1 \texttt{ + } e_2 : \tau$ 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.

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

- Case $(\texttt{fun } x \texttt{ -> } e)\ v \to e[x := n]$. *Note: the corresponding rule has no premise, thus we have no induction hypothesis.*
  From the hypothesis $\Gamma \vdash (\texttt{fun } x \texttt{ -> } e)\ v : \tau$ we know there is τ' such that $\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau' \to \tau$ and $\Gamma \vdash v : \tau'$ (inversion lemma) and from $\Gamma \vdash \texttt{fun } x \texttt{ -> } e : \tau' \to \tau$ 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 $\Gamma \vdash e_1 \texttt{ + } e_2 : \tau$ 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.

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

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 detect 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 for 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.

```
  | 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"
```

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

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 does not have the expected type.

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

```
  | 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"
```

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

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 a 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).

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

The notion of free type variable is also defined similarly.

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

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 $\forall\alpha.\tau$.

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

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 $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ is defined by the equation

$$\mathrm{fv}(\{x_1 : \tau_1, \ldots, x_n : \tau_n\}) \quad = \quad \bigcup_{1 \leq i \leq n} \mathrm{fv}(\tau_i)$$

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*.

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

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.

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

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.

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

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).

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

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 its 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.

$$\begin{array}{rcl} \tau & ::= & \texttt{int} \\ & | & \tau \to \tau \\ & | & \alpha \\ \sigma & ::= & \forall \alpha_1 \dots \forall \alpha_n . \tau \end{array}$$

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

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:

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

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.

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

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

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

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

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-directed 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 $\Gamma_2 = \{f : \forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)\}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma_1 \vdash x : \alpha \to \alpha} \quad \dfrac{\dfrac{}{\Gamma_1 \vdash x : \alpha \to \alpha} \quad \dfrac{}{\Gamma_1 \vdash y : \alpha}}{\Gamma_1 \vdash x\,y : \alpha}}{x : \alpha \to \alpha, y : \alpha \vdash x\,(x\,y) : \alpha}}{x : \alpha \to \alpha \vdash \texttt{fun}\ y \texttt{->}\ x\,(x\,y) : \alpha \to \alpha}}{\vdash \texttt{fun}\ x \texttt{->}\ \texttt{fun}\ y \texttt{->}\ x\,(x\,y) : (\alpha \to \alpha) \to (\alpha \to \alpha)} \quad \dfrac{\dfrac{}{\Gamma_2 \vdash f : (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})} \quad \dfrac{\dfrac{}{\Gamma_2, z : \texttt{int} \vdash z : \texttt{int}} \quad \dfrac{}{\Gamma_2, z : \texttt{int} \vdash z\texttt{+}1 : \texttt{int}}}{\Gamma_2 \vdash \texttt{fun}\ z \texttt{->}\ z\texttt{+}1 : \texttt{int} \to \texttt{int}}}{f : \forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha) \vdash f(\texttt{fun}\ z \texttt{->}\ z\texttt{+}1) : \texttt{int} \to \texttt{int}}$$

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

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 represented 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 α_1, ..., α_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 $\tau_1 = \alpha \to \texttt{int}$ and $\tau_2 = (\texttt{int} \to \texttt{int}) \to \beta$, we can unify the types τ_1 and τ_2 using the instantiation $[\alpha \mapsto \texttt{int} \to \texttt{int}, \beta \mapsto \texttt{int}]$.

— If $\tau_1 = (\alpha \to \texttt{int}) \to (\alpha \to \texttt{int})$ and $\tau_2 = \beta \to \beta$, we can unify the types τ_1 and τ_2 using the instantiation $[\beta \mapsto \alpha \to \texttt{int}]$.

— The types $\alpha \to \texttt{int}$ and $\texttt{int}$ cannot be unified.

— The types $\alpha \to \texttt{int}$ 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` z->z+1).

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 $\eta =$ `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 $\zeta =$ `int` and $\theta =$ `int` $\to$ `int`.

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

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

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 of the types 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=b -> ()
  | 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
```

3 La cible : code assembleur

3.1 Architecture cible

La cible ultime de la compilation est la production d'un programme directement exécutable par un ordinateur physique. La nature d'un tel programme est intimement liée à l'architecture de l'ordinateur cible.

Architecture et boucle d'exécution. Dans ce cours on utilisera une architecture simple mais réelle appelée MIPS, qui est un précurseur de l'architecture moderne RISC-V. Les deux principaux composants de l'architecture sont :

- un *processeur*, comportant un petit nombre de registres qui permettent de stocker quelques données directement accessibles, ainsi que des unités de calcul qui opèrent sur les registres, et
- une grande quantité de *mémoire*, où sont stockées à la fois des données et le programme à exécuter lui-même.

On a deux unités de base pour la mémoire : l'*octet*, une unité universelle désignant un groupe de 8 bits, et le ***mot mémoire***, qui représente l'espace alloué à une donnée « ordinaire », comme un nombre entier. Dans notre architecture, le mot mémoire vaut 4 octets : les données manipulées sont donc généralement représentées par des séquences de 32 bits. En conséquence, un ***entier machine***, c'est-à-dire un nombre entier primitif manipulé par ce processeur est compris entre -2^{31} et $2^{31}-1$ (ou entre 0 et $2^{32}-1$ dans le cas d'entiers « non signés »).

La plupart des architectures majeures existent en versions 32 bits et 64 bits.

L'architecture MIPS comporte 32 registres, pouvant chacun stocker un mot mémoire. Ces données stockées dans les registres sont directement accessibles à l'unité de calcul.

La mémoire principale a une étendue de 2^{32} octets. Chaque octet de la mémoire est associé à une ***adresse***, qui est un entier entre 0 et $2^{32}-1$, et c'est exclusivement en utilisant ces adresses que l'on accède aux éléments stockés en mémoire. Les données étant organisées sur des mots mémoire de 4 octet, l'architecture impose en outre que les adresses des données ordinaires soient des multiples de 4 (il existe quelques exceptions). L'accès à la mémoire est relativement coûteux : une seule opération de lecture ou d'écriture en mémoire a un coût nettement supérieur à une opération arithmétique travaillant directement sur les registres.

Le programme à exécuter est une séquence d'instructions directement interprétables par le processeur. Chaque instruction est codée sur 4 octets, et l'ensemble est stocké de manière contiguë dans la mémoire.

Un registre spécial pc (qui ne fait pas partie des 32) contient l'adresse de l'instruction courante (*program counter*, ou *pointeur de code*). L'exécution d'un programme procède en répétant le cycle suivant :

1. lecture d'un mot mémoire à l'adresse pc (*fetch*),
2. interprétation des octets lus comme une instruction (*decode*),
3. exécution de l'instruction reconnue (*execute*),
4. mise à jour de pc pour passer à l'instruction suivante (par défaut : incrémenter de 4 octets pour passer au mot mémoire suivant).

Instructions. On distingue trois catégories principales d'instructions :

- des instructions arihtmétiques, appliquant des opérations élémentaires à des valeurs stockées dans des registres,
- des instructions d'accès à la mémoire, pour transférer des valeurs de la mémoire vers les registres ou inversement,
- des instructions de contrôle, pour gérer le pointeur de code pc.

Dans les 32 bits utilisés pour coder une instruction, les 6 premiers désignent l'opération à effectuer (***opcode***), et les suivants donnent les paramètres ou des précisions éventuelles sur l'opération. Ci-dessous, quelques exemples avant d'aborder à la section suivante la manière de programmer avec ces instructions.

Par exemple : l'instruction numéro 9 prend en paramètres un registre source r_s, un registre de destination r_t, et un nombre n non signé de 16 bits, et place dans r_t la valeur $r_s + n$. Il suffit de 5 bits pour désigner l'un des 32 registres. Si r_s est le registre numéro 5, r_t le registre numéro 17, et n la valeur 42, alors l'instruction est représentée par un mot mémoire décomposé ainsi :

op	r_s	r_t	n
001000	00101	10001	0000000000101010

Autre exemple : l'instruction numéro 15 prend en paramètres un registre de destination r_t et un nombre n de 16 bits, et place n dans les deux octets supérieurs de r_t. Si r_t est le registre numéro 17 et n la valeur 42, alors l'instruction est représentée par le mot mémoire suivant, où 5 bits ne sont pas utilisés.

op		r_t	n
001111	00000	10001	0000000000101010

Quizz : comment utiliser les instructions précédentes pour charger la valeur `0xeff1cace` *dans le registre numéro 2 ?*

La plupart des opérations arithmétiques binaires utilisent l'*opcode* 0, l'opération précise étant alors donnée par les 6 derniers bits de l'instruction (cette dernière partie est appelée la *fonction*). Par exemple, l'opération d'addition est identifiée par l'*opcode* 0 et la fonction 32. Ces instructions prennent invariablement trois paramètres : deux registres r_s et r_t pour les valeurs auxquelles appliquer l'opération, et un registre de destination r_d. Ci-dessous, opération d'addition avec pour r_s, r_t et r_d les registres de numéros respectifs 1, 2 et 3. Notez que 5 bits sont inutilisés.

op	r_s	r_t	r_d		f
000000	00001	00010	00011	00000	100000

Certaines opérations ignorent le paramètre r_s et fournissent à la place une donnée constante à l'aide des 5 bits qui étaient restés libres ci-dessus. Ainsi, l'opération de décalage vers la gauche prend trois paramètres : un registre r_t à qui appliquer un décalage, un registre r_d de destination, et une constante k sur 5 bits indiquant de combien de bits vers la gauche décaler r_t. C'est opération correspond toujours à l'*opcode* 0, mais cette fois avec la fonction 0. Ainsi, pour décaler de le registre numéro 4 de 5 bits vers la gauche et placer le résultat dans le registre numéro 6, on a :

op		r_t	r_d	k	f
000000	00000	00100	00110	00101	000000

Ce schéma général de découpage vaut encore pour les instructions d'accès à la mémoire. L'instruction de lecture d'un mot mémoire par exemple prend en paramètres deux registres r_s et r_t et un entier signé n sur 16 bits. Le registre r_s est supposé contenir une adresse, et l'entier n est appelé *décalage*. L'instruction de lecture calcule une adresse a en additionnant l'adresse de base r_s et le décalage n, et transfère vers le registre r_t le mot mémoire lu à l'adresse a. Le code de cette opération est 35. Ainsi, pour placer dans le registre numéro 3 la donnée trouvée à l'adresse obtenue en ajoutant 8 octets à l'adresse donnée par le registre numéro 5, on a la séquence :

op	r_s	r_t	n
100011	00101	00011	0000000000001000

Les instructions de contrôle utilisent encore une variante de ce même découpage. On a par exemple une instruction de saut conditionnel, qui prend en paramètres un registre r_s et un entier n signé sur 16 bits, et qui avance de n instructions dans le code du programme si la valeur de r_s est strictement positive. Le code de cette instruction est 7. Pour reculer de 5 instructions lorsque la valeur du registre 3 est strictement positive, on a donc l'instruction :

op	r_s	r_t	n
000111	00011	00000	1111111111111011

Enfin, l'instruction numéro 2 prend pour unique paramètre un entier sur 26 bits interprété comme l'adresse a d'une instruction i, et va faire se poursuivre l'exécution à partir de cette instruction i. Autrement dit, on écrase la valeur du pointeur `pc` pour la remplacer par a. Pour aller à l'instruction d'adresse `0x00efface` on aura donc l'instruction :

op	a
000010	00111011111111101011001110

Nous verrons à la section suivante qu'en pratique, on ne compose pas directement ces codes représentant chaque instruction. On utilise à la place une forme textuelle appelée le ***langage assembleur***, qui est ensuite automatiquement traduite en la version codée par des mots binaires.

Autres familles d'architectures. De nombreux aspects de l'architecture MIPS que nous venons de décrire sont partagés avec les autres architectures majeures. Pour commencer, la boucle d'exécution *fetch/decode/execute* avec un programme stocké en mémoire est une réalisation directe du modèle de von Neumann, à la base de tous les ordinateurs depuis l'origine de l'informatique.

D'une famille d'architectures à l'autre, on observe des variations par exemple sur la taille d'un mot mémoire, avec notamment les architectures 64 bits où le mot mémoire est de 8 octets. Le nombre de registres peut également varier, ainsi que l'ensemble des instructions disponibles ou les manières d'accéder à la mémoire. Les différences les plus significatives ne sont en revanche pas toujours les plus visibles pour le programmeur.

Dans l'architecture MIPS, les instructions sont codées suivant un schéma uniforme qui limite le nombre d'instructions qui peuvent être proposées. Cette approche est typique des architectures de la famille *RISC* (*Reduced Instruction Set Computer*), qui comprennent notamment les architectures ARM et RISC-V.

À l'inverse, les architectures de la famille *CISC* (*Complex Instruction Set Computer*) proposent un codage variable des instructions, par exemple allant de 1 à 8 octets. Ceci permet entre autres choses de proposer un nombre beaucoup plus importants d'instructions. Cette famille comprend notamment les architectures Intel.

Dans une représentation variable de type *CISC*, on dispose d'instructions riches permettant un code optimisé. En outre, le format variable permet de compresser le code, en donnant une représentation compacte aux instructions les plus fréquemment utilisées. Dans une représentation uniforme de type *RISC*, le décodage des instructions est plus simple, et l'ensemble du processeur est de même plus simple et utilise moins de transistors, ce qui permet une moindre consommation d'énergie. En conséquence les architectures ARM sont omniprésentes dans les *smartphones*, où la gestion de la batterie est critique. Les architectures Intel, réputées plus puissantes, ont à l'inverse longtemps été hégémoniques dans le domaine des ordinateurs grand public, mais cela pourrait basculer. En particulier, le format uniforme des architectures RISC facilite grandement le traitement de plusieurs instructions en parallèle, ce qui est une source très importante d'optimisation des architectures. La montée en puissance de ces optimisations rend maintenant les architectures RISC compétitives avec les architectures CISC en termes de puissance de calcul, et les puces ARM commencent à prendre une place dans le monde des ordinateurs grand public.

3.2 Langage assembleur MIPS

En pratique, on n'écrit pas directement les suites de bits du langage machine. On utilise à la place un langage d'assemblage, ou assembleur, qui est quasiment isomorphe au langage machine mais permet une écriture plus agréable. En langage assembleur on a en particulier :

- des écritures textuelles pour les instructions,
- la possibilité d'utiliser des étiquettes symboliques plutôt que des adresses explicites,
- une allocation statique des données globales,
- quelques *pseudo*-instructions, qui correspondent à des combinaisons simples d'instructions réelles.

Dans cette section, on présente le langage assembleur MIPS.

Pour tester nos programmes MIPS, on utilisera le simulateur MARS. Ce simulateur peut être appelé directement depuis la ligne de commande pour exécuter un programme assembleur donné, mais dispose aussi d'un mode graphique dans lequel on peut suivre pas à pas l'exécution d'un programme et l'évolution des registres et de la mémoire.

L'architecture RISC-V, bien que modernisée par rapport à MIPS, utilise en fait un langage assembleur quasiment identique. Du point de vue du programmeur, ce cours vaut donc *presque* pour les deux architectures.

Registres. Dans l'assembleur MIPS, les 32 registres sont désignés par leur numéro, de `$0` jusqu'à `$31`. Alternativement à son numéro, chaque registre possède en outre un nom reflétant sa fonction.

n°	*nom*
`$0`	`$zero`
`$1`	`$at`
`$2`	`$v0`
`$3`	`$v1`
`$4`	`$a0`
`$5`	`$a1`
`$6`	`$a2`
`$7`	`$a3`

n°	*nom*
`$8`	`$t0`
`$9`	`$t1`
`$10`	`$t2`
`$11`	`$t3`
`$12`	`$t4`
`$13`	`$t5`
`$14`	`$t6`
`$15`	`$t7`

n°	*nom*
`$16`	`$s0`
`$17`	`$s1`
`$18`	`$s2`
`$19`	`$s3`
`$20`	`$s4`
`$21`	`$s5`
`$22`	`$s6`
`$23`	`$s7`

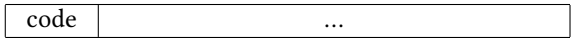
n°	*nom*
`$24`	`$t8`
`$25`	`$t9`
`$26`	`$k0`
`$27`	`$k1`
`$28`	`$gp`
`$29`	`$sp`
`$30`	`$fp`
`$31`	`$ra`

Les 24 registres `$v*`, `$a*`, `$t*` et `$s*` sont des registres ordinaires, que l'on peut librement utiliser pour des calculs (on verra les spécificités de ces quatre familles plus tard). Les 8 autres registres ont des rôles particuliers : `$gp`, `$sp`, `$fp` et `$ra` contiennent des adresses utiles, `$zero` contient toujours la valeur 0 et ne peut pas être modifié, et les 3 registres `$at` et `$k*` sont réservés respectivement pour l'assembleur et pour le système (il ne faut donc pas les utiliser).

Instructions et pseudo-instructions. Un programme en langage assembleur MIPS prend la forme d'un fichier texte, où chaque ligne contient une instruction ou une annotation. On introduit une liste d'instructions avec l'annotation

```
.text
```

Les instructions ainsi introduites sont placées dans une zone de la mémoire réservée au programme, tout en bas de la mémoire (c'est-à-dire aux adresses les plus basses).



Chaque instruction est construite avec un mot clé appelé ***mnémonique***, désignant l'opération à effectuer, et un certain nombre de paramètres séparés par des virgules. Voici par exemple une instruction qui initialise le registre `$t0` avec la valeur 42

```
li $t0, 42
```

et une instruction qui copie la valeur présente dans le registre `$t0` vers le registre `$t1`.

```
move $t1, $t0
```

Comme le montrent ces exemples, les paramètres donnés aux instructions peuvent être, selon les instructions, des registres désignés par leur nom et/ou des constantes entières écrites de manière traditionnelle.

Une mnémonique décrit l'opération à effectuer, en général à l'aide d'un mot ou d'un acronyme. On a ici `move` pour le « déplacement » d'une valeur ou `li` pour *Load Immediate*, c'est-à-dire pour le chargement d'une valeur constante (on appelle valeur *immédiate* une valeur constante directement fournie dans le code assembleur). Certaines de ces mnémoniques correspondent directement à des instructions machine, et peuvent être associées à un *opcode* et le cas échéant à une fonction. D'autres mnémoniques sont des *pseudo-instructions*, c'est-à-dire des raccourcis pour de courtes séquences d'instructions machine (généralement des séquences d'une ou deux instructions seulement).

En l'occurrence :

- `li` est une pseudo-instruction qui s'adapte à la constante à charger : si cette constante s'exprime sur 16 bits alors elle sera traduite par une unique instruction machine, mais dans le cas contraire on aura deux instructions pour charger indépendamment les deux octets hauts et les deux octets bas. Ainsi, cette pseudo-instruction permet au programmeur de s'affranchir de la limite de 16 bits pour les valeurs immédiates.
- `move` est une pseudo-instruction, traduite par une unique instruction d'addition : plutôt que d'inclure dans les circuits du processeur une instruction spécifique pour une affectation de la forme `$k` ← `$k'`, on réutilise l'instruction d'addition qui est de toute façon présente, en fixant le deuxième opérande à zéro : `$k` ← (`$k'` + `$zero`).

Arithmétique. Les opérations arithmétiques et logiques du langage assembleur MIPS suivent toutes le même format « trois adresses » : après la mnémonique identifiant l'opération viennent dans l'ordre le registre cible (où sera placé le résultat de l'opération) puis les deux opérandes.

```
<mnemo>  <dest>, <r1>, <r2>
```

On y trouve des opérations variées, dont la plupart vous seront familières.

- Opérations arithmétiques : `add`, `sub`, `mul`, `div`, `rem` (*remainder* : reste), ...
- Opérations logiques : `and`, `or`, `xor` (*exclusive or*), ...
- Comparaisons : `seq` (*equal* : égalité), `sne` (*not equal* : inégalité) `slt` (*less than* : $<$), `sle` (*less or equal* : $\leqslant$), `sgt` (*greater than* : $>$), `sge` (*greater or equal* : $\geqslant$), ...

À ces opérations binaires s'ajoutent quelques opérations unaires classiques également, avec un registre de destination et un seul opérande : `abs` (valeur absolue), `neg` (opposé), **`not`** (négation logique).

Les opérations arithmétiques admettent parfois des variantes pour l'arithmétique non signée, c'est-à-dire pour manipuler des entiers entre 0 et $2^{32} - 1$ plutôt que des entiers entre -2^{31} et $2^{31} - 1$. Ces opérations sont repérables par la présence d'un `u`. Par exemple : `addu`.

Un certain nombre des opérations précédentes admettent également une variante dans laquelle le deuxième opérande est une valeur immédiate plutôt qu'un registre. Ces opérations sont repérables par la présence d'un `i`. Par exemple : `addi`, ou encore `addiu`.

Enfin, les opérations de manipulation des séquences de bits par décalage ou rotation suivent le même format, mais en prenant par défaut une valeur immédiate pour le deuxième opérande (l'amplitude du décalage).

— Décalages et rotations : `sll` (*shift left logical*), `sra` (*shift right arithmetic*), `srl` (*shift right logical*), `rol` (*rotation left*), `ror` (*rotation right*), ...

Des variantes avec une amplitude de décalage variable existent, repérées par la présence d'un `v`. Par exemple : `sllv`. Dans ce cas, le deuxième opérande est un registre.

Voici par exemple une séquence d'instructions pour évaluer la comparaison `3*4 + 5 < 2*9`, ainsi qu'un tableau montrant l'évolution des différents registres à mesure que l'exécution progresse.

	$t0	$t1	$t2	$t3	$t4	$t5	$t6	$t7	$t8
`li $t0, 3`	3								
`li $t1, 4`	3	4							
`li $t2, 5`	3	4	5						
`li $t3, 2`	3	4	5	2					
`li $t4, 9`	3	4	5	2	9				
`mul $t5, $t0, $t1`	3	4	5	2	9	12			
`add $t6, $t5, $t2`	3	4	5	2	9	12	17		
`mul $t7, $t3, $t4`	3	4	5	2	9	12	17	18	
`slt $t8, $t6, $t7`	3	4	5	2	9	12	17	18	1

L'exemple précédent n'est cependant guère judicieux : il utilise un nouveau registre pour chaque opération, et n'utilise pas toujours les opérations les plus efficaces. Voici une meilleure version pour réaliser le même calcul.

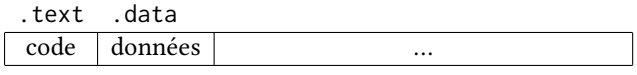
	$t0	$t1
`li $t0, 3`	3	
`li $t1, 4`	3	4
`mul $t0, $t0, $t1`	12	4
`addi $t0, $t0, 5`	17	4
`li $t1, 9`	17	9
`sll $t1, $t1, 1`	17	18
`slt $t0, $t0, $t1`	1	18

Données statiques. À côté des instructions, un programme assembleur peut également déclarer et initialiser un certain nombre de données, correspondant par exemple à des variables globales du programme.

On introduit une liste de déclarations de telles données avec l'annotation

```
.data
```

En mémoire, les données ainsi déclarées sont placées dans une zone située après la zone réservée aux instructions du programme.



La définition d'une donnée ou d'un groupe de données combine :

— la déclaration d'une étiquette symbolique, c'est-à-dire d'un nom qui pourra être utilisé pour accéder à la donnée,

— la fourniture d'une ou plusieurs valeurs initiales.

Un mot-clé permet de préciser la nature des données fournies : `.word` pour une valeur 32 bits (4 octets), `.byte` pour une valeur d'un unique octet (8 bits), `.asciiz` pour une chaîne de caractères au format ASCII (un caractère par octet, la chaîne étant terminée par l'octet zéro (`0x00`).

On déclare ainsi une donnée désignée par le nom `reponse` et valant 42 :

```
reponse: .word 42
```

Ainsi, un mot mémoire va être réservé dans la zone des données et initialisé avec la valeur 42. L'identifiant `reponse` désigne l'adresse de ce mot mémoire, et peut être utilisée notamment pour récupérer ou modifier la valeur qui y est stockée.

On peut de même introduire une séquence de données.

```
prems: .word 2 3 5 7 11 13 17 19
```

La première donnée est placée à l'adresse correspondant à l'étiquette `prems`, puis les suivantes sont l'une après l'autre, de 4 octets en 4 octets.

L'adresse réservée pour chaque donnée sera calculée au moment de la traduction du programme assembleur en langage machine. À cette occasion, chaque mention d'une étiquette sera remplacée en dur par un accès à l'adresse correspondante.

Accès à la mémoire. Le format standard des adresses mémoire en MIPS contient deux composantes :

— une adresse de base, donnée par la valeur d'un registre `$r`,
— un décalage, donné par une constante `d` (16 bits, signée).

On note `d($r)` une telle adresse, dont la valeur est donc `$r + d`.

Les instructions d'accès à la mémoire permettent alors de transférer une valeur depuis une adresse mémoire vers un registre (lecture, *load*) ou au contraire depuis un registre vers une adresse mémoire (écriture, *store*).

```
lw $t0, 8($a1)
sw $t0, 0($t1)
```

Les deux instructions précédentes enchaînent donc la lecture du mot mémoire à l'adresse obtenue en ajoutant 8 à la valeur du registre `$a1` (la valeur lue étant placée dans le registre `$t0`), puis l'écriture de la valeur du registre `$t0` à l'adresse donnée directement par la valeur du registre `$t1`.

Les mnémoniques `lw` et `sw` signifient respectivement *Load Word* et *Store Word*. Des variantes existent pour lire ou écrire seulement un octet (`lb`, `lbu`, `sb`), ou un demi-mot (`lh`, `lhu`, `sh`), ou encore un double mot (`ld`, qui transfère les 8 octets lus vers 2 registres : celui donné dans l'instruction et le suivant, et `sd` qui fonctionne symétriquement).

L'assembleur permet également quelques notations simplifiées des adresses, qui sont converties vers le format standard au moment de la traduction vers le langage machine. On peut par exemple omettre le décalage lorsqu'il vaut zéro

```
sw $t0, ($t1)
```

ou encore accéder directement à l'adresse donnée par une étiquette, avec un éventuel décalage positif ou négatif.

```
lw $t0, prems
lw $t1, prems+4
lw $t2, prems+12
```

Enfin, une instruction `la` (*Load Address*) permet de récupérer ou calculer une adresse, sans lire son contenu.

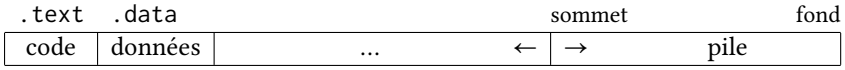
```
la $t0, prems
```

Pile. Le mécanisme des données statiques n'est pas suffisant dans un programme ordinaire[1]. On a généralement besoin de pouvoir stocker des données en mémoire, ne serait-ce que temporairement, sans avoir prédit précisément dès l'écriture du programme la quantité

[1] On s'astreint parfois à s'en contenter, pour limiter les possibilités de bugs dans des catégories particulières de programmes, notamment dans des domaines critiques comme l'avionique.

d'espace occupée. Ainsi, dans un programme ordinaire la plus grande partie de la mémoire est *dynamique*, c'est-à-dire que l'allocation de l'espace aux différentes données n'y est décidée que *pendant l'exécution* du programme, et qu'une adresse occupée par une donnée à un moment de l'exécution a vocation à être libérée lorsque la donnée n'est plus utile, pour être ensuite réaffectée à une nouvelle donnée.

La partie supérieure de cette grande région de mémoire dynamique est appelée la ***pile***, et fonctionne de manière identique à la structure de données de pile. Il s'agit donc d'une structure linéaire, dont une extrémité est appelée le ***fond*** et l'autre le ***sommet***, et qui est modifiée uniquement du côté de son sommet : on peut ajouter de nouveaux éléments au sommet de la pile, ou retirer les éléments du sommet. Cette organisation implique qu'un élément retiré de la pile est toujours l'élément de la pile le plus récemment ajouté.



Petite particularité notable ici : la pile est placée à l'extrémité de la mémoire, du côté des adresses les plus hautes. Il n'est donc pas possible de l'étendre « vers le haut ». Le fond de cette pile est donc calé sur l'adresse la plus grande de la mémoire, et le sommet à une adresse inférieure : la pile croît ainsi en s'étendant vers les adresses inférieures.

Le sommet de la pile est donc l'adresse mémoire la plus basse utilisée par la pile. Cette adresse est stockée dans le registre `$sp`. On réalise les opérations usuelles d'une structure de pile de la manière suivante.

— Pour consulter la valeur au sommet de la pile (peek), on lit un mot à l'adresse donnée par `$sp`.

```
    lw   $t0, 0($sp)
```

Pour aller consulter des éléments plus profonds dans la pile, on ajoute un décalage de 4 octets pour chaque élément à sauter. On consulte donc le quatrième élément avec un décalage de 12 octets.

```
    lw   $t0, 12($sp)
```

— Pour ajouter un élément au sommet de la pile (push), on décrémente `$sp` de 4 octets, puis on écrit la valeur souhaitée à la nouvelle adresse `$sp`.

```
    addi $sp, $sp, -4
    sw   $t0, 0($sp)
```

— Pour retirer l'élément au sommet de la pile, on incrémente `$sp` de 4 octets. Notez qu'il n'est pas nécessaire « d'effacer » la valeur présente à l'ancienne adresse `$sp` : le simple fait que cette adresse soit maintenant inférieure à `$sp` la désigne comme insignifiante.

```
    addi $sp, $sp, 4
```

L'opération usuelle pop, consistant à récupérer l'élément au sommet tout en le retirant de la pile, combine cet incrément avec une opération peek.

```
    lw   $t0, 0($sp)
    addi $sp, $sp, 4
```

Le code assembleur obtenu est d'une certaine manière symétrique à celui de push.

Tableaux. Un ***tableau*** est représenté par une séquence de mots consécutifs en mémoire, et identifié par l'*adresse* de son premier élément. Une définition C comme

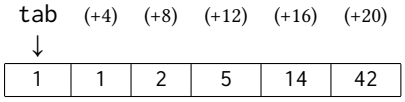
Dans quelle région précise de la mémoire, on le verra bientôt.

```
int tab[] = {1, 1, 2, 5, 14, 42};
```

ou son équivalent ImpScript

```
let tab = [1, 1, 2, 5, 14, 42];
```

pourra ainsi mener à un schéma comme le suivant, où chaque élément du tableau occupe quatre octets.



Ici, le premier élément du tableau est directement à l'adresse donnée par la variable `tab`, le dernier est à l'adresse `tab+20`, et de manière générale on accède à chaque élément en partant de l'adresse de base `tab`, et en ajoutant un décalage correspondant à l'indice de la case cherchée (par multiples de 4 octets).

C'est similaire à l'accès à un élément du tableau d'activation d'un appel de fonction.

Considérons d'abord la situation où le tableau `tab` précédent est un tableau global, défini à la racine du programme que l'on cherche à compiler. Comme les autres variables globales vues jusqu'ici, ce tableau peut être placé en mémoire dans les données statiques. En revanche, au lieu de lui allouer un unique mot mémoire on en donne six consécutifs : un pour chaque case du tableau. On initialise un tel tableau en MIPS avec le fragment suivant.

```
        .data
tab:    .word  1 1 2 5 14 42
```

Une caractéristique de cette version est qu'elle demande que les éléments du tableau soient connus « statiquement », c'est-à-dire dès la compilation du programme. Autrement dit, le texte même du programme doit les fournir explicitement et entièrement. On peut assouplir un petit peu cette contrainte, à condition de connaître statiquement au moins la taille du tableau. Ainsi, la déclaration d'un tableau non initialisé *de taille fixe*, écrite en C

```
int tab[6];
```

et en ImpScript

```
let tab = Array(6);
```

pourra être traduite par le code MIPS suivant où on a toujours bien une séquence de six valeurs (mais un choix arbitraire pour leur contenu).

```
        .data
tab:    .word  0 0 0 0 0 0
```

Dans l'une ou l'autre version, l'étiquette `tab` désigne l'adresse de base du tableau statique, et c'est d'elle que l'on se sert pour calculer l'adresse de chaque élément. On accède ainsi au quatrième élément à l'aide d'un décalage de 12 par rapport à l'adresse de base désignée par l'étiquette `tab` :

```
        la   $t0, tab
        lw   $t1, 12($t0)
```

Dans cette première version l'indice est connu statiquement : on peut calculer le décalage en amont, et l'écrire explicitement dans le code. Si en revanche l'indice auquel chercher est donné par un registre, il va falloir ajouter un peu de code MIPS pour calculer dynamiquement (c'est-à-dire : à l'exécution) le bon décalage. Pour accéder à une case dont l'indice est donné par le registre `$a0`, il faut donc multiplier la valeur de `$a0` par 4 pour obtenir le bon décalage (on le fait efficacement par un décalage de 2 bits vers la gauche, avec l'instruction `sll`), puis ajouter ce décalage à l'adresse de base.

Le décalage donné dans l'instruction `lw` ne peut être qu'une constante entière explicite, on ne peut pas y mettre `$a0`.

```
        la   $t0, tab
        sll  $a0, 2
        add  $t0, $t0, $a0
        lw   $t1, 0($t0)
```

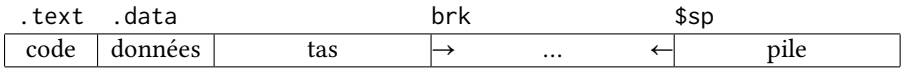
Lors de l'instruction `lw` on n'indique donc plus de décalage par rapport à l'adresse donnée par `$t0`, puisque que celui-ci contient déjà précisément l'adresse voulue.

Tableaux alloués dynamiquement. Imaginons cette fois le cas d'un tableau créé dynamiquement, avec une taille qui n'est pas connue explicitement à l'avance.

```
int f(int n) {
  int tab[n];
  ...
}
```

C'est beaucoup plus souple pour le programmeur, mais demande une nouvelle stratégie pour le compilateur. On peut imaginer que ce tableau soit représenté par n mots consécutifs en mémoire, rangés sur la pile avec les autres variables locales. C'est effectivement ce qui se passe en C, par défaut. Attention cependant : comme tout ce qui se trouve sur la pile, ce tableau a une durée de vie limitée et sera détruit à la fin de l'appel.

Nous allons nous concentrer ici sur une troisième stratégie, permettant d'allouer dynamiquement des structures de données ***persistantes***, qui vont survivre à l'appel de fonction qui les a créées. On utilise pour cela une nouvelle région de la mémoire : le ***tas***, qui se trouve entre les données statiques et la pile.



La plus petite adresse du tas est immédiatement au-dessus de la zone des données statiques. Un pointeur `brk` appelé *memory break* identifie la fin du tas, ou plus précisément la première adresse au-delà du tas. Ainsi le dernier mot du tas est à l'adresse `brk-4`. L'espace situé entre le tas et la pile, c'est-à-dire entre les pointeurs `brk` (inclus) et `$sp` (exclu), est vide. Lorsque l'on a besoin de plus de place sur le tas, on incrémente le pointeur `brk`, pour absorber une partie de l'espace libre entre le tas et la pile.

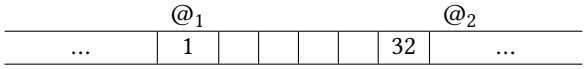
Le pointeur `brk` est géré par le système d'exploitation, aussi son déplacement est fait par un appel système `sbrk`, qui a le code 9 dans notre simulateur MIPS. Cet appel système prend comme argument, via `$a0`, le nombre d'octets dont `brk` doit être incrémenté. Il renvoie comme résultat, via `$v0`, l'ancienne valeur de `brk`, c'est-à-dire la première adresse de la zone qui vient d'être ajoutée au tas.

On garde un point commun avec la version précédente : tout tableau alloué ainsi par extension du tas est désigné par un pointeur vers sa première case. La différence est que ce pointeur ne correspond plus à une étiquette du code assembleur, mais transite par les registres comme une valeur ordinaire.

Voici un exemple d'utilisation de ce mécanisme, dans lequel on alloue sur le tas de l'espace pour un tableau de 6 cases (24 octets), avant d'écrire `1` dans la première case et `32` dans la sixième.

instruction	$a0	$v0	$t0	brk
				0x10040000
`li $a0, 24`	24			0x10040000
`li $v0, 9`	24	9		0x10040000
`syscall`	24	0x10040000		0x10040018
`li $t0, 1`	24	0x10040000	1	0x10040018
`sw $t0, 0($v0)`	24	0x10040000	1	0x10040018
`li $t0, 32`	24	0x10040000	32	0x10040018
`sw $t0, 20($v0)`	24	0x10040000	32	0x10040018

La forme du tas associée à cet exemple est la suivante, où l'adresse $@_1$ est la position d'origine de `brk` (0x10040000) et $@_2$ est la nouvelle position après appel à `sbrk` (0x10040018).



Sauts et branchements. Rappelons que les instructions d'un programme MIPS sont stockées en mémoire. Chaque instruction a donc une adresse. De même que les adresses des données, les adresses de certaines instructions peuvent être désignées par des étiquettes : une étiquette insérée dans une séquence d'instructions MIPS désigne l'adresse de l'instruction qui la suit immédiatement, et pourra être utilisée comme cible d'une instruction de saut.

MIPS propose une variété d'instructions (ou pseudo-instructions) de branchement conditionnel. En voici un exemple :

```
beq  $t0, $t1, lab
```

Cette instruction compare les valeurs des registres `$t0` et `$t1`. Si ces deux valeurs sont égales, alors l'exécution se poursuivra avec l'instruction d'étiquette `lab` (techniquement, l'instruction de branchement commande une modification du registre spécial `pc`). Sinon, l'exécution se poursuivra naturellement avec l'instruction suivante.

Des variantes existent pour d'autres modalités de comparaison

$=$	$\neq$	$<$	$\leqslant$	$>$	$\geqslant$
`beq`	`bne`	`blt`	`ble`	`bgt`	`bge`

et chacune admet à nouveau une variante pour comparer une unique valeur à zéro : `beqz`, `bnez`, `bltz`, ... Certaines admettent également pour deuxième paramètre une valeur immédiate plutôt qu'un registre. Enfin, une pseudo-instruction `b lab` provoque un branchement inconditionnel vers l'instruction d'étiquette `lab`.

Voici un exemple de fragment de code calculant la factorielle du nombre stocké dans le registre `$a0`, et plaçant le résultat dans le registre `$v0`.

```
1          move $t0, $a0
2          li   $t1, 1
3          b    test
     loop:
4          mul  $t1, $t1, $t0
5          addi $t0, $t0, -1
     test:
6          bgtz $t0, loop
7          move $v0, $t1
```

Les deux premières instructions initialisent les deux registres `$t0` et `$t1` qui seront utilisés pour le calcul. Les instructions 4 à 6 forment une boucle, et la dernière instruction transfère le résultat dans `$v0` comme demandé. Le corps de la boucle est constitué des instructions 4 et 5, la première étant associée à l'étiquette `loop`. Le test de la boucle est à l'instruction 6, associée à l'étiquette `test`. Lorsque ce test est positif, on déclenche un nouveau tour de boucle à l'aide d'un branchement vers `loop`. À l'inverse, lorsque le test est négatif on poursuit avec l'instruction suivante, c'est-à-dire l'instruction 7. Enfin, l'instruction 3 démarre la boucle en branchant vers le test. *Notez que l'on aurait pu se passer de* `$t0`*, de* `$t1` *et des instructions 1 et 7 en travaillant exclusivement avec* `$a0` *et* `$v0`*.*

Les instructions de branchement sont utilisées pour des déplacements « locaux » dans la séquence d'instructions, c'est-à-dire en restant dans la même unité de code (par exemple : la même fonction). Les instructions machine correspondantes font avancer ou reculer le pointeur de code d'un certain nombre d'instructions, ne dépassant pas 2^{15}.

À l'inverse, les instructions de saut définissent la prochaine instruction de manière absolue, en fournissant directement son adresse. Cela peut se faire à l'aide d'une étiquette, et dans ce cas la différence avec une instruction de branchement n'est pas flagrante,

```
  j    label
```

ou en fournissant directement une adresse calculée, stockée dans un registre.

```
  jr   $t0
```

Les instructions de saut servent notamment dans le cadre d'un saut non local, comme un appel de fonction. Pour cette situation, on dispose également de deux instructions qui, avant de sauter, sauvegardent une adresse de retour dans le registre `$ra`, qui permettra plus tard de revenir à l'exécution de la séquence en cours.

```
  jal  label
  jalr $t0
```

On reviendra sur ce mécanisme d'appel de fonction dans un chapitre ultérieur.

Appels système. Pour certaines fonctionnalités dépendant du système d'exploitation, un code assembleur doit faire appel aux bibliothèques système. Dans ce cours, on exécutera notre code MIPS dans un simulateur, qui prendra en charge ces différents services avec une instruction dédiée. On décrit donc ici des fonctionnalités du simulateur, qui miment des services dépendant normalement du système d'exploitation.

On déclenche un appel système à l'aide de l'instruction `syscall`, après avoir placé dans le registre `$v0` un code désignant le service demandé. Voici une petite sélection :

service	code	arg.	rés.
affichage	1 (entier) 4 (chaîne) 11 (ascii)	`$a0`	
lecture	5 (entier)		`$v0`
arrêt	10		
extension mémoire	9	`$a0`	`$v0`

On trouve encore des services pour la manipulation de fichiers, et d'autres variantes d'affichage ou de lecture.

Hello, world. Le programme le plus célèbre de la terre, en MIPS.

```
        .text
main:   li   $v0, 4
        la   $a0, hw
        syscall
        li   $v0, 10
        syscall

        .data
hw:     .asciiz "hello␣world\n"
```

Notes : 4 est le code de l'appel système d'affichage d'une chaîne de caractères, 10 celui de l'arrêt. L'annotation `.asciiz` permet de placer une chaîne de caractères dans les données statiques, dont l'adresse est ici associée à l'étiquette `hw`.

3.3 Fonctions et pile d'appels

Un point clé du développement de programmes réalistes est la possibilité de définir et d'appeler des fonctions. Du point de vue de la compilation, cela demande un peu de technologie supplémentaire. Fixons d'abord le vocabulaire, en observant l'***appel de fonction*** `f(6)` dans le code suivant.

```
1   function g() {
2     let y;
3     y = f(6);
4     print(y);
5   }

6   function f(x) {
7     return x*7;
8   }
```

L'expression `6` est un ***paramètre effectif*** (on dit aussi un *argument*) de l'appel de fonction `f(6)`. Le bloc de code allant des lignes 2 à 4 est le ***contexte appelant***, et la fonction `f` définie aux lignes 6 à 8 est la ***fonction appelée***. Du côté de la fonction appelée, la variable `x` est un ***paramètre formel*** de la fonction, et l'expression `x*7` calcule le résultat ***renvoyé*** par la fonction.

Certains utilisent *retourner* à la place de *renvoyer*, mais il s'agit d'un faux ami.

Articulation appelant/appelé. Un appel de fonction implique des transferts de différentes natures entre le contexte appelant et la fonction appelée. On a d'une part un transfert de données, avec

- un ou plusieurs paramètres effectifs transmis par le contexte appelant à la fonction appelée,
- un résultat, renvoyé au contexte appelant par la fonction appelée.

Pour cela, on peut utiliser la pile et/ou les registres, d'une manière à convenir. Outre ce transfert de données, on a un transert *temporaire* de contrôle :

- lors de l'appel, l'exécution *saute* au code de la fonction appelée,
- à la fin de l'appel, l'exécution *revient* au contexte appelant, en reprenant « juste après » le point où l'on a réalisé l'appel.

Ce transfert temporaire demande, au moment de l'appel, de mémoriser l'adresse de l'instruction à laquelle il faudra revenir après l'appel. En MIPS, on utilise pour cela le registre `$ra` (*Return Address*). Pour sauter au code de la fonction tout en mémorisant dans `$ra` l'adresse de l'instruction suivante du contexte appelant, on utilise l'instruction `jal` (*Jump And Link*) avec l'étiquette de la fonction appelée, ou sa variante `jalr` lorsque l'adresse de la fonction est donnée par un registre (qui est éventuellement le résultat d'un calcul).

```
  jal   f
  jalr  $t0
```

À la fin de l'appel de fonction, on rend alors la main à l'appelant avec un saut à l'adresse donnée par `$ra`.

```
  jr    $ra
```

Pour obtenir des adresse réalistes en MIPS, il faut ajouter à chacune la constante `0x400000`, qui est la première adresse de la région de la mémoire dédiée au programme.

Exemple. Voici une traduction possible du fragment de code précédent, où les nombres à gauche donnent l'adresse de chaque instruction. On décide ici que l'unique argument de `f` est placé dans le registre `$a0` et que son résultat est placé dans le registre `$v0`.

```
12  g:  li   $a0, 6          # définition du paramètre effectif
16      jal  f               # appel de fonction
20      move $a0, $v0        # affichage du résultat
24      li   $v0, 1
28      syscall

36  f:  li   $v0, 7          # calcul
40      mul  $v0, $a0, $v0
44      jr   $ra             # fin de la fonction
```

Voici une trace d'exécution de ce code assembleur. On y observe en particulier que l'exécution du saut `jal` rencontré à l'adresse `16` stocke dans `$ra` l'adresse suivante `20` (tout en sautant à l'adresse `36`).

instruction	$a0	$v0	$ra	pc	
				12	
`li $a0, 6`	6			16	
`jal f`	6		20	36	
`li $v0, 7`	6	7	20	40	
`mul $v0, $a0, $v0`	6	42	20	44	
`jr $ra`	6	42	20	20	
`move $a0, $v0`	42	42	20	24	
`li $v0, 1`	42	1	20	28	
`syscall`	42	1	20	32	affiche 42

Problème : appels imbriqués. Observons maintenant ce qui se passe si on ajoute un code principal réalisant un appel à notre fonction g. On complète le code comme suit.

```
00  main:  jal  g            # appel à g (sans argument)
04         li   $v0, 10      # fin du programme principal
08         syscall
12  g:     li   $a0, 6
16         jal  f
20         move $a0, $v0
24         li   $v0, 1
28         syscall
32         jr   $ra          # fin de g
36  f:     li   $v0, 7
40         mul  $v0, $a0, $v0
44         jr   $ra
```

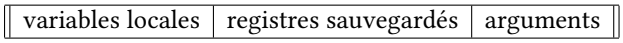
La trace d'exécution est maintenant la suivante.

instruction	$a0	$v0	$ra	pc	
				0	
`jal g`			4	12	
`li $a0, 6`	6		4	16	
`jal f`	6		20	36	
`li $v0, 7`	6	7	20	40	
`mul $v0, $a0, $v0`	6	42	20	44	
`jr $ra`	6	42	20	20	
`move $a0, $v0`	42	42	20	24	
`li $v0, 1`	42	1	20	28	
`syscall`	42	1	20	32	affiche 42
`jr $ra`	42	1	20	20	
`move $a0, $v0`	1	1	20	24	
`li $v0, 1`	1	1	20	28	
`syscall`	1	1	20	32	affiche 1
`jr $ra`	1	1	20	20	
`move $a0, $v0`	1	1	20	24	
…	…	…	…	…	

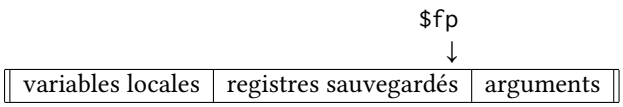
Nous voilà coincés dans une boucle ! Problème : lorsque l'instruction `jal f` d'appel à `f` stocke son adresse de retour dans `$ra`, elle écrase la valeur qui était déjà présente dans `$ra`, à savoir l'adresse de retour de l'appel à `g`. Ainsi, lorsque l'on arrive à l'instruction `jr $ra` qui achève le code de `g` à l'adresse 32, le registre `$ra` ne contient plus la valeur qui avait été sauvegardée au moment de l'appel à `g`, et notre saut se fait vers la mauvaise cible.

Tableaux d'activation. Lorsque des appels de fonction sont emboîtés, c'est-à-dire lorsque le code d'une fonction appelée réalise lui-même un appel de fonction et ainsi de suite, on se retrouve avec plusieurs appels actifs à un même moment. Plus précisément, l'appel le plus récent est en cours d'exécution et un certain nombre d'autres appels plus anciens sont en suspens.

Chaque appel de fonction actif possède son propre contexte, contenant notamment les arguments qui lui ont été passés, les valeurs de ses variables locales, ou encore l'adresse de retour stockée dans `$ra`. Toutes ces informations doivent être stockées de manière à survivre à des appels de fonction internes. La structure utilisée pour stocker les informations d'un appel donné est appelée ***tableau d'activation*** (en anglais *call frame*). On considère ici des tableaux d'activation avec la forme générale suivante, où la partie « registres sauvegardés » contient en particulier la valeur sauvegardée du registre `$ra`.



Ces tableaux sont stockés dans la mémoire. On utilise un registre dédié `$fp` (*Frame Pointer*) pour localiser le tableau actif, c'est-à-dire le tableau d'activation de l'appel actuellement en cours d'exécution. On peut ensuite, à partir de `$fp`, accéder aux différents éléments stockés dans le tableau. Le pointeur de base `$fp` désigne l'adresse du dernier mot dédié aux registres sauvegardés.

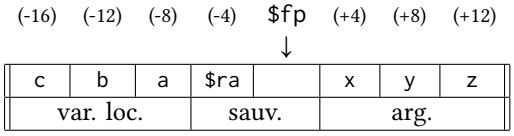


On accède donc aux arguments à partir de `$fp` avec un décalage positif, et aux variables locales avec un décalage négatif.

Exemple. En supposant une fonction avec trois paramètres `x`, `y` et `z` et trois variables locales `a`, `b` et `c`

```
function h(x, y, z) {
  let a, b, c;
  ...
```

on pourrait avoir un tableau d'activation de la forme suivante, où la valeur de `$ra` est sauvegardée à l'adresse `fp-4`, où les valeurs des paramètres effectifs sont stockées à partir de `fp+4` (vers les adresses supérieures), et les variables locales sont stockées à partir de `fp-8` (vers les adresses inférieures).



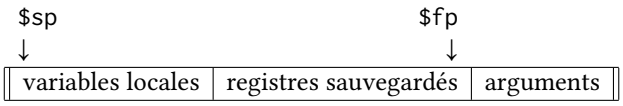
On laisse libre pour l'instant la case à l'adresse `fp`, qui servira à un autre registre sauvegardé.

Pile d'appels. La vie des tableaux d'activation se déroule ainsi :

- création d'un tableau d'activation au début d'un appel,
- destruction du tableau à la fin de l'appel.

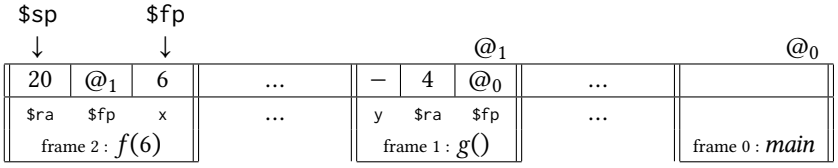
En outre, tout appel doit se terminer avant de rendre la main à son contexte appelant. Autrement dit, c'est toujours l'appel le plus récent qui terminera en premier, et toujours le tableau d'activation le plus récent qui sera détruit en premier. L'ensemble des tableaux d'activation a donc une structure de pile (*Last In First Out*), nommée ***pile d'appels*** (*call stack*). La pile d'appels est réalisée dans la pile MIPS elle-même, et en forme l'ossature principale.

Ainsi le premier mot du tableau d'activation correspond au sommet de la pile, désigné par le pointeur `$sp` (*Stack Pointer*).



En outre, pour faciliter l'identification du tableau d'activation précédent (qui est celui du contexte appelant), on peut ajouter dans tout tableau d'activation une sauvegarde de la valeur précédente de `$fp`, c'est-à-dire de l'adresse de base du tableau d'activation précédent. On obtient ainsi un chaînage des tableaux d'activation, allant du contexte courant vers les contextes plus anciens, jusqu'à l'appel de la fonction principale de notre programme.

Exemple. On reprend les fonctions `f` et `g` prises en introduction. Pendant l'appel `f(6)`, la pile contient trois tableaux d'activation : un pour le contexte principal (*main*) que l'on ne détaille pas, un pour l'appel `g()`, qui contient notamment un emplacement non initialisé pour la variable locale `y`, et un pour l'appel `f(6)`, qui contient notamment un emplacement pour la valeur du paramètre `x`. Le pointeur `$fp` courant pointe dans ce dernier tableau, qui est au sommet de la pile. Les tableaux de `f(6)` et `g()` contiennent une adresse `$ra` sauvegardée qui fait référence aux numéros de l'extrait de code assembleur p.125. On désigne dans ce tableau par $@_0$ et $@_1$ respectivement l'adresse de base du tableau d'activation principal et du tableau d'activation de `g()` (il s'agit des valeurs de `$fp` pour ces tableaux).



3.4 Convention d'appel

« Convention » : choix d'un cadre commun, qui ne préjuge pas de l'existence d'alternatives viables.

Pour assurer les bons transferts d'information et de contrôle, et la bonne gestion de la pile d'appels, le contexte appelant et la fonction appelée doivent respecter un protocole commun appelé ***convention d'appel***.

Protocole minimal. Un protocole d'appel se découpe en plusieurs étapes, à la charge de l'un ou l'autre des participants. On propose ici une version simple, que l'on utilisera pour compiler ImpScript.

1. **Appelant, avant l'appel :** évalue les arguments et place les valeurs obtenues sur la pile, puis déclenche l'appel avec `jal` ou `jalr` pour stocker l'adresse de retour dans `$ra` tout en passant la main à l'appelé.
2. **Appelé, au début de l'appel :** sauvegarde les valeurs de `$fp` et `$ra` sur la pile, réserve de l'espace supplémentaire pour ses variables locales, et donne à `$fp` sa nouvelle valeur (ceci complète la création du tableau d'activation).
3. **Appelé :** exécution du corps de la fonction appelée.
4. **Appelé, à la fin de l'appel :** place le résultat renvoyé dans le registre `$t0`, restaure `$fp` et `$ra`, libère l'espace de pile réservé à l'étape 2, et redonne la main à l'appelant avec `jr $ra`.
5. **Appelant, après l'appel :** retire les arguments placés sur la pile.

Le tableau d'activation est donc créé aux étapes 1 et 2 et détruit aux étapes 4 et 5 : il existe uniquement pendant l'exécution du corps de la fonction appelée. À la fin du protocole, la pile a retrouvé son état d'origine, et le résultat de l'appel est dans le registre `$t0`, prêt à être utilisé.

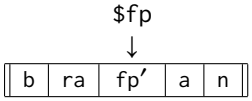
Exemple. On considère le programme ImpScript suivant, qui définit et utilise une fonction d'exponentiation rapide.

```
function power(a, n) {
  if (n == 0) { return 1; }
  else {
    let b = power(a*a, n>>1);
    if (n&1 == 0) { return b; }
    else { return a*b; }
```

```
  }
}

console.log(power(2, 9));
```

La fonction `power` a deux arguments `a` et `n`, et une variable locale `b`. Un tableau d'activation pour un appel de cette fonction a donc la forme suivante.



On accède aux arguments `a` et `n` respectivement aux adresses `fp+4` et `fp+8`, et à la variable locale `b` à l'adresse `fp-8`.

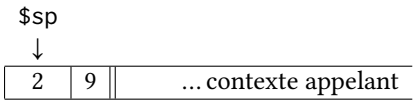
Protocole d'appel à `power`, ordre chronologique. Pour réaliser l'appel `power(2, 9)`, il faut d'abord placer les deux arguments au sommet de la pile,

```
li      $t0, 9              # chargement de 9
addi    $sp, $sp, -4        # passage sur la pile
sw      $t0, 0($sp)
li      $t0, 2              # chargement de 2
addi    $sp, $sp, -4        # passage sur la pile
sw      $t0, 0($sp)
```

puis appeler la fonction,

```
jal     power
```

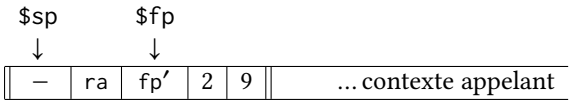
La fonction `power` prend alors le relais. La pile contient déjà les deux arguments



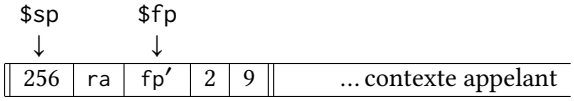
et il faut maintenant construire le reste du tableau d'activation. Pour cela, on déplace le pointeur `$sp`, on sauvegarde les valeurs des registres `$fp` et `$ra`, et on donne sa nouvelle valeur à `$fp`.

```
addi    $sp, $sp, -12       # allocation
sw      $fp, 8($sp)         # sauvegarde fp
sw      $ra, 4($sp)         # sauvegarde ra
addi    $fp, $sp, 8         # définition fp
```

Avec un déplacement de 12 octets (3 mots mémoire), on prévoit la place pour la variable `b` en plus des deux registres sauvegardés.



Le corps de la fonction peut alors s'exécuter. Après divers événements, dont des appels récursifs, nous arrivons à la fin de cette exécution avec l'instruction `return a*b;`, alors que la variable locale `b` a reçu la valeur 256 (résultat de `power(4, 4)`).



On peut alors calculer le résultat à renvoyer `a*b` (512) et le placer dans le registre `$t0`,

```
lw      $t0, 4($fp)         # t0 <- a
lw      $t1, -8($fp)        # t1 <- b
mul     $t0, $t0, $t1       # t0 <- a*b
```

restaurer les registres `$ra` et `$fp` et libérer l'espace correspondant sur la pile,

```
  lw      $ra, 4($sp)       # restauration ra
  lw      $fp, 8($sp)       # restauration fp
  addi    $sp, $sp, 12      # nettoyage de la pile
```

et enfin rendre la main à l'appelant.

```
  jr      $ra
```

Quand l'appelant reprend la main, la pile a donc retrouvé l'état qu'elle avait juste avant l'appel, avec les arguments encore en place. Il suffit de déplacer à nouveau `$sp` pour les retirer et conclure le protocole d'appel.

```
  addi    $sp, $sp, 8       # nettoyage de la pile
```

Enfin, l'appelant peut maintenant utiliser le résultat de l'appel, qui est disponible dans le registre `$t0`.

```
  move    $a0, $t0
  li      $v0, 1
  syscall                   # affichage
```

Code de l'exponentiation, dans l'ordre. Le protocole que nous venons de décrire permet de gérer un nombre arbitraire d'appels de fonction imbriqués. En particulier, il s'applique même aux appels récursifs de notre fonction `power`, dont on peut maintenant voir le code complet. Le code assembleur généré pour la fonction `power` commence à s'exécuter lorsque les arguments sont déjà placés sur la pile. Il commence par l'étape 2 du protocole d'appel (*appelé, au début de l'appel*), pour finir de mettre en place le tableau d'activation.

```
    power:
    # Construction du tableau d'activation (ÉTAPE 2)
052     addi    $sp, $sp, -12
056     sw      $fp, 8($sp)
060     sw      $ra, 4($sp)
064     addi    $fp, $sp, 8
```

Cette initialisation passée, on peut enchaîner avec le code correspondant au corps de la fonction (étape 3).

```
    # Exécution du corps de la fonction (ÉTAPE 3)
068     lw      $t0, 8($fp)     # test (n == 0)
072     bnez    $t0, power_rec
076     li      $t0, 1          # t0 <- résultat (return 1)
080     b       power_end
    power_rec:
```

On voit réapparaître l'étape 1 du protocole (*appelant, avant l'appel*) au moment où la fonction `power` s'apprête elle-même à réaliser un appel de fonction.

Ici, un appel récursif, mais cela ne change rien.

Note : dans ce fragment une instruction est inutile, car place dans un registre une valeur qui s'y trouve déjà. La voyez-vous ?

```
    # Déclenchement de l'appel récursif (ÉTAPE 1)
084     lw      $t0, 8($fp)     # push argument (n>>1)
088     sra     $t0, $t0, 1
092     addi    $sp, $sp, -4
096     sw      $t0, 0($sp)     # push argument (a*a)
100     lw      $t0, 4($fp)
104     mul     $t0, $t0, $t0
108     addi    $sp, $sp, -4
112     sw      $t0, 0($sp)
116     jal     power           # appel
```

Dans le code assembleur, l'étape 5 (*appelant, après l'appel*) apparaît immédiatement après l'instruction de saut, puisque c'est précisément là que l'exécution reviendra quand l'appelé rendra la main.

```
    # Après l'appel récursif (ÉTAPE 5)
120     addi    $sp, $sp, 8     # nettoyage arguments
```

Une fois le protocole d'appel terminé, la fonction (appelante) peut récupérer le résultat de l'appel et reprendre son travail.

```
    # Retour à l'exécution du corps de la fonction (ÉTAPE 3, suite)
124     sw      $t0, -8($fp)    # b <- résultat power(a*a, n>>1)
128     lw      $t0, 8($fp)     # test (n&1 == 0)
132     andi    $t0, $t0, 0x1
136     bnez    $t0, power_odd
140     lw      $t0, -8($fp)    # t0 <- résultat (return b)
144     b       power_end
    power_odd:
148     lw      $t0, 4($fp)
152     lw      $t1, -8($fp)
156     mul     $t0, $t0, $t1   # t0 <- résultat (return a*b)
```

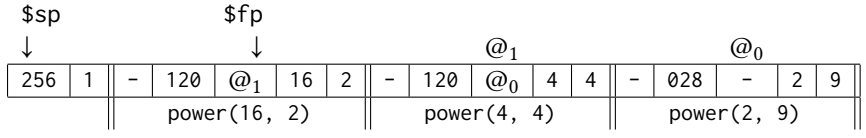
Enfin, le code généré termine par l'étape 4 du protocole (*appelé, à la fin de l'appel*), pour rendre la main au contexte appelant après avoir nettoyé la pile.

```
    power_end:
    # Destruction du tableau d'activation, fin de l'appel (ÉTAPE 4)
160     lw      $ra, 4($sp)
164     lw      $fp, 8($sp)
168     addi    $sp, $sp, 12
172     jr      $ra
```

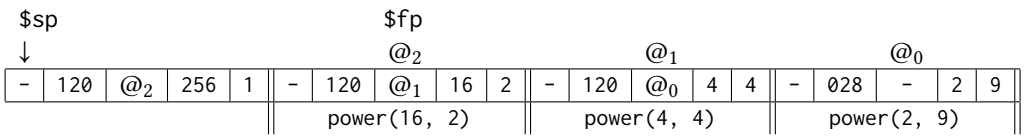
Suivi de la pile d'appels. Pour mémoire, voici également le code complet de l'instruction principale du programme (`console.log(power(2, 9))`).

```
    # Appel power(2, 9)
000     li      $t0, 9          # préparation des arguments
004     addi    $sp, $sp, -4
008     sw      $t0, 0($sp)
012     li      $t0, 2
016     addi    $sp, $sp, -4
020     sw      $t0, 0($sp)
024     jal     power           # appel
028     addi    $sp, $sp, 8     # nettoyage des arguments
    # Affichage du résultat
032     move    $a0, $t0
036     li      $v0, 1          # code 1 : affichage d'un entier
040     syscall
    # Fin du programme
044     li      $v0, 10         # code 10 : arrêt de l'exécution
048     syscall
```

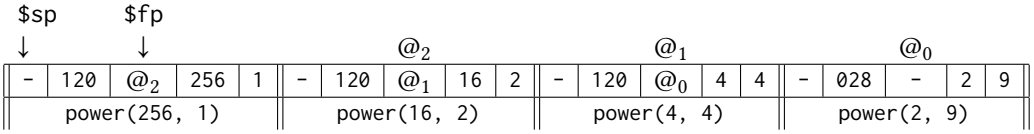
L'appel `power(2, 9)` réalisé ligne `024` va engendrer un appel récursif `power(4, 4)` (ligne `116`), qui engendre un appel `power(16, 2)` (ligne `116` encore), qui lui-même engendre un appel `power(256, 1)` (ligne `116` toujours). Plaçons-nous à la ligne `052`, au début de ce dernier appel. La pile d'appels a alors la forme suivante, avec les deux arguments `256` et `1` au sommet de la pile, et les tableaux d'activation des trois appels précédents. À ce stade, le pointeur de base `$fp` courant est celui de notre contexte appelant (l'appel `power(16, 2)`. On note $@_1$ et $@_0$ les adresses de base des autres tableaux d'activation présents. Dans chaque tableau, la case devant accueillir la variable locale `b` est présente, mais pas encore initialisée.



Les lignes `052` à `060` déplacent le sommet `$sp` de la pile et enregistrent les valeurs courantes de `$fp` et `$ra` dans cet espace ajouté à la pile. On y note également $@_2$ l'adresse de base du contexte appelant qui, pour l'instant, est encore donnée par `$fp`.



Puis la ligne `064` complète l'initialisation du tableau d'activation en définissant la nouvelle valeur du pointeur de base `$fp`. À nouveau, la case devant accueillir la variable locale `b` est déjà présente mais pas encore initialisée.



À la fin de l'appel, les lignes `160` à `168` effectuent les opérations inverses, et nous ramènent à la situation du premier schéma.

Convention d'appel avec registres. La convention d'appel standard de MIPS utilise les registres d'une manière plus fine que ce que nous avons fait dans la convention simplifiée. Ceci concerne en premier lieu le passage des arguments : dans la convention standard de MIPS, les quatre premiers arguments sont passés par les registres `$a0` à `$a3`. Seuls les suivants éventuels, à partir du cinquième, sont passés par la pile. Le résultat, lui, est renvoyé via le registre `$v0`.

Notez que cela est cohérent avec les appels système. Dans l'appel système d'affichage d'un entier par exemple (ligne `040` dans le code p.130), la valeur à afficher est placée dans le registre `$a0`. De même, les appels système qui produisent un résultat le placent dans `$v0`.

Sauvegarde des registres. Dans le code assembleur que nous avons vu jusqu'ici, on utilisait beaucoup la pile mais très peu les registres : uniquement les registres spéciaux `$sp`, `$ra`, `$fp` pour gérer la pile et les appels de fonction, plus `$t0` et `$t1` de manière temporaire pour les calculs. On y a vu que `$sp` devait être maintenu à jour par chaque fonction, et que `$ra` et `$fp` devaient être sauvegardés chaque fois que leur valeur risquait d'être écrasée, et restaurés ensuite.

Lorsqu'un code assembleur tire réellement parti de l'ensemble des registres, pour stocker plus de valeurs intermédiaires des calculs, ou pour encore pour stocker des variables locales, la convention d'appel doit également régler quand et comment la valeur de chacun doit être sauvegardée/restaurée. Pour cela, les registres sont séparés en deux paquets.

- Les registres *callee-saved* doivent être préservés par l'appelé. Si l'appelé les utilise, il doit donc au préalable sauvegarder leurs valeurs, puis les restaurer avant de rendre la main. En MIPS, il s'agit des registres `$s*` (*Saved*) et `$fp`.
- Les registres *caller-saved* peuvent être écrasés par l'appelé. C'est à l'appelant de les sauvegarder avant l'appel et de les restaurer ensuite s'il a encore besoin de leur valeur après l'appel. En MIPS, il s'agit des registres `$a*` (*Argument*), `$t*` (*Temporary*) et `$ra` (*Return Address*).

Notez que l'on a déjà mentionné que le `$ra` d'un appel de fonction devait être sauvegardé dans son tableau d'activation avant d'éventuels appels imbriqués, et que le `$fp` du contexte appelant devait être sauvegardé par la fonction appelée avant que celle-ci n'en modifie la valeur pour désigner son propre tableau d'activation.

Autrement dit, dans l'étape 1 du protocole d'appel, l'appelant doit également sauvegarder les valeurs des registres `$a*` ou `$t*` dont il aurait encore besoin après l'appel (et il peut ensuite les restaurer à l'étape 5). Inversement, l'appelé n'a besoin à l'étape 2 de sauvegarder que les valeurs des registres `$s*`, et même plus précisément que les valeurs des registres qu'il modifiera.

En conséquence, on utilise les registres `$t*` en priorité pour des valeurs intermédiaires à la durée de vie courte, pour ne pas avoir besoin de les sauvegarder avant de réaliser un appel. À l'inverse, les éléments que l'on souhaite préserver sur une durée plus longue seront avantageusement stockés dans les registres `$s*` : ils n'auront alors à être sauvegardés que lorsqu'une fonction appelée aura effectivement besoin d'utiliser ces mêmes registres.

C'est exactement ce qu'on a fait dans la convention simplifiée avec `$t0` et `$t1`, dont la durée de vie était si courte qu'ils n'ont jamais eu besoin d'être sauvegardés avant un appel.

Exponentiation rapide, avec la convention MIPS. On peut écrire une nouvelle version assembleur de notre fonction `power`, utilisant cette nouvelle convention. On aura donc ici le premier argument `a` transmis par le registre `$a0`, le deuxième argument `n` transmis par le registre `$a1`, et on va choisir en plus de stocker la variable locale `b` dans un registre temporaire `$t2` (on ne s'intéresse à la valeur de cette variable que pendant un petit intervalle de temps, qui ne contient pas d'appel).

Avec ces nouveaux choix, le code principal économise la manipulation de la pile.

```
    # Appel power(2, 9)
000'    li      $a1, 9          # préparation des arguments
012'    li      $a0, 2
024     jal     power           # appel
    # Affichage
032'    move    $a0, $v0
036     li      $v0, 1
040     syscall
    # Fin du programme
044     li      $v0, 10
048     syscall
```

Le tableau d'activation va maintenant contenir 4 cases, pour stocker `$ra` et `$fp` comme avant, mais aussi pour préserver les arguments `$a0` et `$a1` en cas d'appel récursif. En revanche, il n'y a plus de nécessité de prévoir une place pour `b`, dont le registre suffira.

```
    power:
    # Construction du tableau d'activation
052     addi    $sp, $sp, -16   # 4 mots réservés au lieu de 3
056     sw      $fp, 12($sp)
060     sw      $ra, 8($sp)
064     addi    $fp, $sp, 12
    # Exécution du corps de la fonction
072'    bnez    $a1, power_rec  # test (n == 0)
076     li      $v0, 1          # v0 <- résultat (return 1)
080     b       power_end
```

Avant et après l'appel récursif, on insère de nouvelles instructions pour sauvegarder puis restaurer les registres contenant les arguments. En revanche, la préparation des arguments de l'appel est simplifiée.

```
    power_rec:
    # Appel récursif
084'    sw      $a0, -8($fp)    # sauvegarde a0 et a1
084''   sw      $a1, -12($fp)
088'    sra     $a1, $a1, 1     # prép. argument (n>>1)
104'    mul     $a0, $a0, $a0   # prép. argument (a*a)
116     jal     power           # appel
120'    lw      $a0, -8($fp)    # restauration a0 et a1
120''   lw      $a1, -12($fp)
```

Le calcul du résultat est lui aussi simplifié, et le nettoyage final du tableau d'activation reste identique à la version précédente.

```
    # Retour à l'exécution du corps de la fonction
124     move    $t2, $v0        # b <- résultat power(a*a, n>>1)
132'    andi    $t0, $a1, 0x1   # test (n&1 == 0)
136     bnez    $t0, power_odd
140     move    $v0, $t2        # v0 <- résultat (return b)
144     b       power_end
    power_odd:
156'    mul     $v0, $t2, $a0   # v0 <- résultat (return a*b)
    power_end:
    # Destruction du tableau d'activation
160     lw      $ra, 8($sp)
164     lw      $fp, 12($sp)
168     addi    $sp, $sp, 16
172     jr      $ra
```

Simplifications. En observant un peu le dernier code obtenu, on peut remarquer quelques simplifications possibles. D'une part, on a deux transfers directs entre le registre temporaire `$t2` choisi pour stocker la valeur de la variable `b` et le registre `$v0` dans lequel doit se trouver le résultat de tout appel de fonction :

— ligne `124`, où le résultat de l'appel récursif obtenu dans `$v0` est stocké dans `b`,
— ligne `140`, où cette même valeur de `b` revient dans `$v0` comme résultat de l'appel courant.

On pourrait économiser un registre et ces deux transferts en utilisant directement `$v0` pour stocker la valeur de `b`. La fin du corps de la fonction deviendrait donc

```
    # Retour à l'exécution du corps de la fonction
132'    andi    $t0, $a1, 0x1   # test (n&1 == 0)
136     bnez    $t0, power_odd
144     b       power_end
    power_odd:
156'    mul     $v0, $v0, $a0   # v0 <- résultat (return a*b)
    power_end:
```

où les lignes `124` et `140` ont disparu, et où la ligne `156'` a été adaptée. Ensuite, on peut voir que la disparition de la ligne `140` a achevé de vider la branche positive du test `(n&1 == 0)`, qui ne fait plus rien d'autre que sauter à la séquence de fin. On peut alors réorganiser ces branches pour une version finale plus directe.

```
    # Retour à l'exécution du corps de la fonction
132'    andi    $t0, $a1, 0x1   # test (n&1 == 0)
136'    beqz    $t0, power_end
156'    mul     $v0, $v0, $a0   # v0 <- résultat (return a*b)
    power_end:
```

3.5 Optimisation des appels terminaux

On pourrait imaginer aller plus loin dans les simplifications. Si on peut réorganiser le code de `power` pour que l'un ou l'autre des arguments `a` et `n` ne soit plus utile après l'appel récursif, la sauvegarde de cet argument avant chaque appel ne serait plus utile. C'est ce qui se serait passé par exemple pour `n` si nous avions pris comme code d'origine la version suivante.

```
function power(a, n) {
  if (n == 0) { return 1; }
  else { if (n&1 == 0) { return power(a*a, n>>1); }
         else          { return a*power(a*a, n>>1); } }
}
```

En approfondissant cette idée, on peut aboutir à une simplification bien plus spectaculaire, dans laquelle ni les arguments, ni `$ra`, ni `$fp` n'ont besoin d'être sauvegardés. Pour cela, il nous faut une version de `power` dans laquelle tous les appels sont ***terminaux*** :

```
1   function power(a, n) {
2     return power_aux(a, n, 1);
3   }

4   function power_aux(a, n, acc) {
5     if (n == 0) { return acc; }
6     else {
7       if (n&1 == 0) { return power_aux(a*a, n>>1, acc); }
8       else          { return power_aux(a*a, n>>1, a*acc); }
9     }
10  }

11  console.log(power(2, 9));
```

Dans ce code, l'appel à `power_aux` ligne `2` est quasiment la dernière action de la fonction `power` : après cet appel, il ne reste à `power` qu'à nettoyer son propre tableau d'activation et sauter au contexte appelant, en transférant directement le résultat qui a été donné par `power_aux`.

Remarquez que ce dernier point est particulièrement simple : le résultat que `power` doit placer dans `$v0` a justement été laissé là par `power_aux` ! On pourrait alors imaginer que la fonction `power`, plutôt que de demander un résultat `power_aux` et transmettre ce résultat tel quel au contexte appelant, demande à `power_aux` de transmettre directement son résultat au contexte appelant. Pour cela, on réalise l'appel à `power_aux` par un saut simple

```
  j   power_aux
```

au lieu de l'habituel saut `jal`. Ainsi, l'appel à `power_aux` aura comme adresse de retour la valeur courante de `$ra`, c'est-à-dire ici justement l'adresse à laquelle il fallait reprendre après l'appel à `power`.

Il reste cependant à régler la question du nettoyage du tableau d'activation de power. Pour cela, on peut partir d'une deuxième remarque : plus aucun élément du tableau d'activation de l'appel à power ne sera utile après l'appel à power_aux, exceptée la valeur sauvegardée du pointeur $fp du contexte appelant qui doit être restaurée. On pourrait donc détruire le tableau d'activation de power (et restaurer le $fp appelant) *avant* de réaliser l'appel à power_aux. Ou encore, pour limiter les sauvegardes/restaurations redondantes, réutiliser le tableau d'activation de power pour l'appel à power_aux sans modifier les valeurs sauvegardées de $ra et $fp. Ou enfin ici, ne même jamais construire ce tableau d'activation, puisque dans notre cas il n'aura jamais besoin de sauvegarder quoi que ce soit.

Remarquez en outre que les deux appels récursifs à power_aux, aux lignes 7 et 8, sont encore terminaux. La simplification que nous venons de décrire pour power peut donc encore être appliquée à power_aux. Voici le code que l'on obtiendrait, en passant les arguments a et n par les registres $a0 et $a1 (pour power comme pour power_aux), en passant l'argument acc de power_aux par le registre $a2, en réalisant chaque appel terminal par un saut simple et en se passant des tableaux d'activation superflus.

On commence avec le même code que précédemment pour l'instruction principale réalisant l'appel power(2, 9) et affichant le résultat. Celui notamment est bien toujours réalisé avec jal.

```
    # Appel power(2, 9)
00      li      $a0, 2
04      li      $a1, 9
08      jal     power
    # Affichage
12      move    $a0, $v0
16      li      $v0, 1
20      syscall
    # Fin du programme
24      li      $v0, 10
28      syscall
```

Après, les choses se simplifient drastiquement. La fonction power réalise un unique appel terminal à power_aux. Il n'y a pas de tableau d'activation à créer, les deux premiers arguments sont déjà dans $a0 et $a1, il suffit de placer le troisième argument dans $a2 puis sauter à la fonction auxiliaire.

```
    power:
32      li      $a2, 1          # initialisation de l'accumulateur
36      j       power_aux       # appel power_aux(a, n, 1)
```

Notez que l'on utilise l'instruction de saut simple j et pas l'instruction d'appel jal, puisque $ra a déjà la bonne valeur.

Vient ensuite la fonction auxiliaire power_aux. Celle-ci n'a pas non plus besoin de tableau d'activation, puisqu'elle n'a pas de variables locales et n'opère que des appels terminaux à elle-même. On réalise donc directement le test du cas d'arrêt. S'il est positif, on saute à la branche « then », ici l'étiquette power_end qui terminera la fonction. Petite simplification possible au passage : le saut ligne 36 est superflu, puisque sa cible power_aux est justement l'instruction suivante.

```
    power_aux:
40      beqz    $a1, power_end  # cas d'arrêt
```

Sinon, power_aux va ensuite préparer les arguments a′, n′ et acc′ pour le prochain appel récursif, puis relancer power_aux, toujours avec un saut simple. Les arguments a et n sont mis à jour systématiquement aux lignes 56 et 60. En revanche, le test ligne 48 est susceptible de court-circuiter la mise à jour de acc ligne 52. On obtient donc bien l'un des deux appels récursifs (ligne 7 ou ligne 8 du code source) selon la parité de n.

```
44      andi    $t0, $a1, 0x1   # test (n == 0)
48      beqz    $t0, power_even
52      mul     $a2, $a0, $a2   # acc' <- a*acc
    power_even:
56      mul     $a0, $a0, $a0   # a' <- a*a
60      sra     $a1, $a1, 1     # n' <- n>>1
64      j       power_aux
```

Dans le cas d'arrêt, on place le résultat dans `$v0`, et on revient au contexte appelant avec l'instruction habituelle `jr`, en ciblant le registre `$ra` dont la valeur n'a jamais été écrasée au cours de l'exécution !

```
    power_end:
68      move    $v0, $a2        # return acc
72      jr      $ra
```

On a donc finalement un code qui travaille uniquement avec les registres, et plus du tout avec la pile. Le résultat est efficace, et évite tout risque de d'erreur « `StackOverflow` ». En réalité, ce code assembleur est essentiellement équivalent à celui que nous aurions obtenu en compilant le code suivant, qui utilise une boucle plutôt qu'une fonction auxiliaire récursive.

```
function power(a, n) {
  let acc = 1;
  while (n != 0) {
    if (n&1 != 0) { acc = a*acc; }
    a = a*a;
    n = n>>1;
  }
  return acc;
}
```

L'optimisation des appels terminaux décrite ici est particulièrement utile pour des langages utilisant massivement des fonctions récursives (comme caml), mais reste pertinente y compris en dehors. Elle est par exemple également réalisée par GCC, lorsque l'on demande à ce compilateur un niveau d'optimisation élevé. Elle est cependant assez rarement réalisée, certains langages préférant maintenir une pile d'appels complète à des fins de débogage (python, java).

4 Analyses de programmes et allocation de registres

On a vu que les données manipulées par un programme peuvent être stockées soit dans les registres, soit dans la mémoire. Inconvénient des données stockées en mémoire : elles doivent être transférées dans des registres avant d'être utilisées, et ces transferts sont coûteux. À l'inverse, les données stockées dans des registres sont directement utilisables. Cependant, les registres sont eux-même une ressource *très* limitée (typiquement, on en a 32) et il est donc impensable de tout y stocker.

Pour obtenir un code assembleur efficace, il faut donc choisir quoi stocker dans les registres et quoi stocker en mémoire, de manière à utiliser au mieux les premiers, et à limiter les transfert avec la seconde. On appelle cette opération l'***allocation de registres***. Dans ce chapitre, on va voir comment un compilateur peut analyser un programme source pour déterminer une allocation de registres raisonnable.

4.1 Les contours du problème

On se place dans le contexte du noyau de langage impératif ImpScript, utilisé au chapitre précédent. Pour garder un cadre raisonnable, on va travailler à l'échelle de la fonction, c'est-à-dire analyser et optimiser chaque fonction indépendamment des autres. On considère deux sortes de données pour lesquelles il faut déterminer une allocation :

— les variables locales déclarées par la fonction,
— les valeurs intermédiaires intervenant dans le calcul des différentes expressions.

Nous allons considérer ces deux aspects à tour de rôle.

Utilisation des registres pour les variables locales. Le chapitre précédent suggère une solution minimaliste à la question de l'allocation de registres pour les variables locales : tout stocker dans le tableau d'activation, sans utiliser les registres. Une amélioration évidente consisterait à réserver un certain nombre R_v de registres pour les R_v premières variables, et ne stocker que les suivantes dans le tableau d'activation. Dans ce modèle, une variable x associée à un registre r est « propriétaire » de ce registre r sur toute la durée de l'exécution de la fonction, et est la seule à pouvoir l'utiliser.

Cependant, une variable donnée n'a pas nécessairement l'utilité de son registre sur l'ensemble de l'exécution de la fonction. Par exemple, une variable n'a besoin de rien avant d'être initialisée, ni après la dernière fois où elle est utilisée.

```
EXEMPLE DE PROGRAMME AVEC DURÉES DE VIE MANIFESTEMENT DISJOINTES
```

On pourrait ainsi imaginer que plusieurs variables « partagent » un même registre, pour peu qu'elles n'en aient pas besoin en même temps. Le problème d'allocation de registres pour les variables locales va donc se transformer en une analyse en deux temps :

1. déterminer quelles variables peuvent ou non partager un même registre,
2. résoudre le problème d'optimisation visant à regrouper du mieux possible les variables compatibles.

On espère ainsi augmenter le nombre de variables qui pourront être stockées dans un registre et diminuer celles qui devront être placées dans le tableau d'activation.

Utilisation des registres pour les valeurs intermédiaires. Concernant l'allocation de registres pour les valeurs intermédiaires des calculs des expressions, on a à nouveau une solution minimaliste : tout stocker sur la pile sans utiliser les registres.

On a également encore une amélioration simple : réserver un certain nombre R_i de registres pour les R_i valeurs intermédiaires les plus récentes, et ne sauvegarder des valeurs sur la pile que lorsqu'elles sont trop anciennes. Cette solution a l'avantage d'effectivement réutiliser certains registres pour plusieurs valeurs qui n'ont pas besoin d'être sauvegardées en même temps. Un défaut en revanche de cette approche est d'imposer une séparation stricte entre les registres dédiés aux valeurs intermédiaires et ceux dédiés aux variables locales.

Pour éviter cette séparation artificielle, on propose de faire de chaque valeur intermédiaire une nouvelle variable locale, que l'on intègre à l'algorithme d'allocation précédent. Cela revient à transformer le programme en réduisant chaque expression à une séquence d'opérations élémentaires.

```
EXEMPLE DE CODE AVEC DES EXPRESSIONS ET UN APPEL DE FONCTION
```

Ainsi, finalement, il suffira de se concentrer sur l'allocation de registres pour les variables locales.

4.2 Interférences et allocation par coloration, première version

On s'intéresse donc aux variables locales d'une fonction ImpScript. Supposons pour l'instant avoir pu déterminer quelles variables peuvent ou non partager un même registre. On peut résumer cet ensemble d'information par un ***graphe d'interférence***, c'est-à-dire un graphe non orienté dans lequel :

- chaque sommet représente une variable,
- il y a une arête entre deux sommets x et y lorsque ces deux variables *ne peuvent pas* partager un même registre (on dit qu'elle *interfèrent*).

L'affectation à chaque variable d'un registre, d'une manière qui respecte les interférences, se ramène à un problème de coloration de graphe où les « couleurs » sont les registres disponibles : on veut associer à chaque sommet (chaque variable) une couleur (un registre) de sorte que deux sommets adjacents (deux variables qui interfèrent) n'aient pas la même couleur (le même registre).

Coloration gloutonne. La coloration de graphe est un problème NP-complet. Le résoudre de manière optimale a donc un coût prohibitif, et nous allons devoir trouver une approche plus simple donnant néanmoins des résultats satisfaisants.

On part donc d'une structure d'algorithme simple : considérer les sommets un par un, et pour chacun choisir une couleur parmi celles qui n'ont pas encore été utilisées chez ses voisins. Il s'agit d'un algorithme glouton : on considère chaque sommet une fois, et on prend une décision définitive pour chaque sommet sur la base d'informations locales. Le résultat obtenu va fortement dépendre de l'ordre dans lequel on considère les sommets, et c'est dans le choix de cet ordre que va se trouver l'intelligence de l'algorithme.

EXEMPLE DE COLORATION AVEC DEUX ORDRES DONNANT DES RESULTATS DIFFERENTS

Choix de l'ordre des sommets. L'astuce va consister à déterminer d'abord les sommets qui seront considérés *en dernier*. On part pour cela de la remarque suivante : si on cherche à colorier le graphe avec R couleurs, et qu'un sommet x a un degré d strictement inférieur à R, alors on sait que ce sommet pourra toujours être colorié, quelles que soient les couleurs choisies pour ses d voisins. On peut donc décider que ce sommet x sera colorié en dernier, et l'ignorer dans la suite du processus. On passe ensuite à la sélection du sommet à colorier en avant-dernier (ou plus généralement : en dernier parmi ceux qui n'ont pas encore été sélectionnés). À noter : à mesure que l'on progresse ainsi les degrés des sommets restants diminuent, puisque l'on ne considère plus les sommets déjà sélectionnés.

EXEMPLE ENCHAÎNANT TROIS SÉLECTIONS

Si, en revanche, aucun sommet n'a un degré strictement inférieur à R, alors il n'y a aucun sommet pour lequel on a la certitude qu'il pourra bien être colorié si on le traite en dernier. On sélectionne alors un sommet x (une variable x) arbitraire. Au moment de la coloration il y aura alors deux scénarios possibles :

- soit les voisins de x n'utilisent pas toutes les couleurs (tous les registres), et on peut finalement lui en donner une (un) malgré l'incertitude que l'on avait au moment de la sélection ;
- soit les voisins de x utilisent déjà toutes les couleurs (tous les registres), et x ne peut pas être colorié (devra être stocké dans le tableau d'activation).

EXEMPLES DES DEUX SITUATIONS

Au moins deux critères peuvent guider le choix de la variable x à sélectionner lors d'une telle étape :

- du fait que x risque de ne pas avoir de registre, et donc de générer des accès mémoire, on gagne à sélectionner une variable qui semble peu utilisée ;
- du fait que le sommet x est retiré du graphe, entraînant une diminution du degré de ses sommets voisins, on gagne à sélectionner une variable qui a un fort degré, pour contribuer à obtenir plus de sommets de degré $d < R$ dans les étapes suivantes de sélection.

Dans tous les cas la sélection sera heuristique et peut se faire par exemple en donnant à chaque variable un score tenant compte, entre autres, de ces deux aspects.

Algorithme. On obtient une formulation particulièrement compacte avec un algorithme récursif. Partant d'un graphe G que l'on cherche à colorier avec R couleurs :

- s'il existe un sommet x de degré $d < R$, colorier le graphe G' obtenu en retirant x de G, puis donner à x une couleur qui n'a été donnée à aucun de ses voisins ;
- sinon, procéder de même avec un sommet x arbitraire.

4.3 Calcul du graphe d'interférence

Informellement, le critère caractérisant deux variables en interférence est simple : deux variables x et y ne peuvent pas partager un même registre dès lors qu'il existe un « moment » dans l'exécution de la fonction où les valeurs v_x et v_y de ces deux variables doivent toutes deux être conservées en mémoire.

Vivacité. Ce critère lui-même peut être exprimé en fonction de l'*utilité future* de chaque valeur : à un instant donné, la valeur courante v_x d'une variable x doit être conservée en mémoire si elle est destinée à être utilisée à un moment ultérieur. On a alors deux principes très simples :

- la valeur v_x d'une variable x que l'on s'apprête à lire est utile ;

  DESSIN y = x+1

- la valeur v_x d'une variable x que l'on s'apprête à écraser n'est pas utile.

  DESSIN x = y+1

En combinant ces deux principes, on peut faire le lien entre l'utilité de la valeur v_x d'une variable x et la vie de cette variable : à un moment donné, la valeur courante v_x d'une variable x est *utile* si la suite de l'exécution contient une lecture de x avant toute écriture de x. Une variable dont la valeur courante est utile est dite ***vivante***.

Quid de `x = x+1;` ?

Dans le détail les choses ne sont cependant pas si simples, notamment du fait qu'une fonction donnée n'a généralement pas une seule manière de s'exécuter. En fonction des boucles et des branchements, on peut voir différents scénarios d'exécution, qui ne rendent pas vivantes les mêmes variables et donc ne génèrent pas nécessairement les mêmes interférences. Comme le code assembleur produit doit être correct en toutes circonstances, on considère donc que deux variables interfèrent dès qu'elle interfèrent dans *au moins un* scénario d'exécution possible. Autrement dit, une variable x est vivante à un moment donné s'il existe au moins un scénario d'exécution futur dans lequel sa valeur courante v_x sera consultée.

Hélas, le critère tel qu'énoncé est indécidable : il ne peut pas exister d'algorithme qui, pour toute fonction ImpScript et toutes deux variables x et y de cette fonction, détermine à coup sûr et en temps fini si ces deux variables interfèrent ou non. Ceci vient notamment du fait que l'on ne peut pas calculer en général « l'ensemble des scénarios d'exécution possibles » d'un programme.

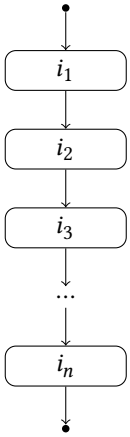
Cette indécidabilité est la norme pour les questions non triviales portant sur l'ensemble des exécutions possibles d'un programme.

Nous allons donc devoir nous contenter d'une approximation de la relation d'interférence. Pour déterminer quelles approximations sont acceptables, rappelons notre cahier des charges : le code assembleur produit doit s'exécuter correctement dans tous les scénarios possibles. En particulier, nous ne pouvons pas ignorer un scénario. Notre algorithme déterminera donc les interférence en fonction de l'ensemble des scénarios réels, mais également en fonction de certains « faux positifs » (des scénarios impossibles en pratique, mais que l'analyse a priori ne sait pas écarter automatiquement). En conséquence, on aura potentiellement également des faux positifs dans le graphe d'interférence. Le tout sera d'avoir néanmoins une analyse suffisamment précise pour que ces faux positifs n'empêchent pas tout partage des registres.

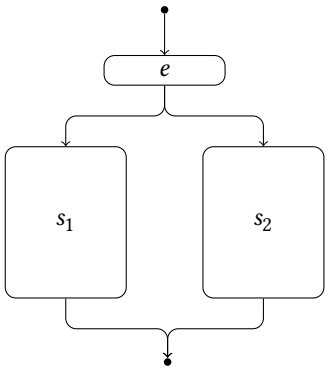
Graphe de flot de contrôle. Un ***graphe de flot de contrôle*** est un graphe orienté dont les sommets sont des instructions, et les arêtes représentent les successions possibles entre deux instructions. Ainsi, on a un arc d'une instruction i vers une instruction i' lorsqu'il est possible que l'exécution de i' succède directement à celle de i. Chaque exécution concrète d'une fonction correspond à un chemin dans le graphe de flot de contrôle correspondant.

Partant d'un langage impératif comme ImpScript, les graphes de flot de contrôle sont obtenus en composant les éléments suivants (on représente pour chacun un point d'entrée et un point de sortie, au niveau desquels se font les compositions).

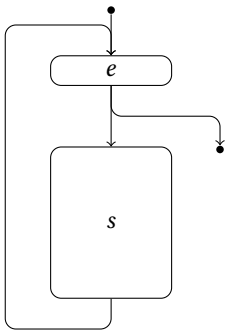
— Séquences. Une séquence d'instructions $i_1\,;\ i_2\,;\ \ldots\,;\ i_n\,;$ est représentée par une chaîne dont chaque sommet correspond à une instruction i_k.



— Branchements. Un branchement conditionnel `if (`e`) { `s_1` } else { `s_2` }` est représenté par un sommet de branchement correspondant au test de la condition e, lié à deux graphes correspondant aux séquences s_1 et s_2. Les arêtes sortantes de s_1 et s_2 se rejoignent alors vers le prochain sommet.



— Boucles. Une boucle `while (`e`) { `s` }` est représentée par un sommet de branchement correspondant au test de la condition e, lié d'une part au graphe représentant s, et d'autre part au point de sortie. On a également un arc revenant de la sortie de s à l'entrée de e, formant la « boucle » à proprement parler.



Par exemple, quels graphes pour `goto`, `switch` ou `break` ?

De nouveaux schémas pourraient apparaître si l'on étendait le langage avec de nouvelles constructions.

Chaque scénario d'exécution possible d'une fonction donnée correspond à un chemin dans son graphe de flot de contrôle. À l'inverse, certains chemins ne correspondent pas à des exécutions réelles. Ainsi, l'ensemble des chemins du graphe décrit un sur-ensemble des exécutions possibles, qui correspond bien au type d'approximation que l'on cherchait. C'est sur la base de cette approximation que nous allons déterminer la vivacité de chaque variable aux différents « moments » de l'exécution.

Et cette fois, le calcul sera exact !

4.4 Analyse de flot de données

Notre objectif est d'établir la liste des variables vivantes à chaque « moment » de notre programme. Ces « moments » sont définis par les instructions du code, c'est-à-dire en l'occurrence par les sommets du graphe de flot de contrôle. Notez cependant qu'une instruction ne désigne pas tant un point dans le temps qu'une transition entre un état d'origine et un état d'arrivée, et que l'ensemble des variables vivantes peut être différent dans ces deux états. Pour chaque instruction, nous considérerons donc deux « moments » : l'instant qui *précède* l'exécution de l'instruction, et l'instant qui la *suit*.

Techniquement, on cherchera donc à caractériser pour chaque instruction i :

— l'ensemble $IN[i]$ des variables vivantes *en entrée* de i, c'est-à-dire avant l'exécution de i,
— l'ensemble $OUT[i]$ des variables vivantes *en sortie* de i, c'est-à-dire après l'exécution de i.

Une variable x étant vivante dès lors que sa valeur courante v_x est susceptible d'être utilisée dans le futur, les informations de vivacité vont circuler dans le graphe de flot de contrôle à rebours de la direction d'exécution. En particulier, l'ensemble $IN[i]$ des variables vivantes en entrée d'une instruction i est calculé en fonction de l'action de i, et de l'ensemble $OUT[i]$ des variables vivantes en sortie de cette instruction.

L'action d'une instruction i sur l'ensemble des variables vivantes se résume à deux critères :

— les variables dont la valeur est écrasée par i ne sont pas vivantes en entrée de i,
— les variables consultées par i sont vivantes en entrée de i (et cette deuxième condition est prioritaire sur la première).

Ainsi, en notant :

— $USE[i]$ l'ensemble des variables consultées par l'instruction i, et
— $DEF[i]$ l'ensemble des variables écrasées par l'instruction i,

on obtient la première équation

$$IN[i] = (OUT[i] \setminus DEF[i]) \cup USE[i]$$

En outre, l'ensemble $OUT[i]$ des variables vivantes en sortie de l'instruction i est lui-même déduit des variables vivantes en entrée de la ou des instructions suivant immédiatement i. Comme il suffit qu'une variable soit utilisée dans l'un des chemins futurs pour être vivante à un point donné, on procède à une union des variables vivantes en entrées des différentes instructions suivantes possibles. Ainsi, en notant $SUCC[i]$ l'ensemble des instructions succédant à i dans le graphe de flot de contrôle, on obtient la deuxième équation

$$OUT[i] = \bigcup_{s \in SUCC[i]} IN[s]$$

Ces équations sont appelée ***équations de flot de données***. Nous en avons deux pour chaque sommet du graphe de flot de contrôle, qui forment un système caractérisant l'ensemble des variables vivantes aux différents points du programme. Nous n'avons plus qu'à résoudre ce système.

Il est aisé de résoudre un tel système tant que le programme d'origine ne contient pas de boucle : il suffit de partir de la dernière instruction, puis remonter progressivement jusqu'à la première.

EXEMPLE AVEC PROGRAMME SÉQUENCE + BRANCHEMENT

Avec une boucle les choses deviennent plus délicate : un cycle apparaît dans le graphe, et les équations deviennent récursives. Ainsi, dans un boucle comme la suivante, l'ensemble $IN[e]$ dépend de $OUT[e]$, lui-même dépendant de $IN[s]$, et donc de $OUT[s]$, et donc de $IN[e]$ lui-même.

DESSIN

Ainsi, on n'a plus d'ordre de dépendance clair permettant de calculer chaque ensemble à son tour. Le caractère « fini » de notre problème va cependant permettre une résolution par raffinements successifs.

Résolution par accumulation. On doit calculer, pour chaque instruction i d'une fonction, les ensembles $IN[i]$ et $OUT[i]$ des variables vivantes en entrée et en sortie de cette instruction i. La technique consiste à d'abord considérer ces ensembles comme vides, puis les faire grandir progressivement en déduisant des équations de flot de données des variables q'ils contiennent nécessairement.

EXEMPLE AVEC BOUCLE SIMPLE

Quand on ne découvre plus de nouvelles variables à ajouter, à aucun ensemble, alors on a bien des ensembles IN et OUT qui vérifient toutes les équations. Un théorème de point fixe nous assure que cette procédure aboutit bien toujours à un état stable.

Théorème du point fixe de Kleene.

Algorithme de Kildall.

4.5 Allocation de registres optimisée

Observons la réalisation d'un appel de fonction

```
x = f(e1, ..., eN);
```

avec l'allocation de registres telle que décrite jusqu'ici. Au préalable, l'expression est décomposée en une séquence d'instructions où la valeur de chaque argument est associée à une nouvelle variable locale.

```
x1 = e1;
x2 = e2;
...
xN = eN;
x  = f(x1, ..., xN);
```

L'appel sera alors réalisé ainsi :

1. les valeurs des arguments `e1` à `eN` sont calculées, et stockées dans les registres (ou emplacements de pile) alloués aux variables `x1` à `xN` ;
2. ces valeurs sont transférées dans les registres dédiés `$a` (ou sur la pile) ;
3. sauvegarde des registres *caller-saved* ;
4. appel ;
5. restauration des registres *caller-saved*, et transfert du résultat de `$v` au registre (ou emplacement de pile) alloué à la variable `x`.

Cette procédure est insatisfaisante en au moins deux points.

— On a une séparation stricte entre des registres « de calcul » utilisés pour l'allocation (`$s` et `$t`), et des registres dédiés aux appels de fonction (`$a`, `$v`, `$fp`, `$ra`). Cette séparation empêche les registres réservés aux fonctions d'être utilisés pour les calculs, même quand ils sont inutilisés. En particulier, les registres `$a` ne peuvent même pas être utilisés pour le calcul des futurs arguments de fonctions, ces derniers devant d'abord être calculés dans un registre `$s` ou `$t` avant d'être transférés.

— Avant et après chaque appel de fonction on sauvegarde puis on restaure tous les registres *caller-saved*, même lorsque les valeurs qu'ils contiennent ne sont plus utiles à la suite du calcul. Cette inefficacité est particulièrement frustrante lorsque l'on songe que l'on vient justement de réaliser une analyse de vivacité, qui pourrait permettre de détecter ce genre de sauvegardes inutiles. Mais notre analyse ne portait jusque là que sur les variables locales.

Explicitation des registres et des conventions. Nous allons améliorer notre procédure en intégrant tous les registres aux procédures d'analyse de vivacité et d'allocation. Pour cela, on considère chaque registre comme une variable ordinaire : aux variables locales de la fonction analysée on ajoute des variables $\$a_i$ représentant les registres $\texttt{\$a}_i$, et de même pour les `$s`, `$t`, `$v`, etc. Et évidemment, lors de l'allocation chacune de ces variables sera pré-affectée au registre correspondant !

Conjointement à cette introduction de nouvelles variables, on insère dans notre programme des instructions explicites de copie pour tous les transferts demandés par la convention d'appel. En particulier :

— avant un appel de fonction, les valeurs des arguments sont transférées dans les variables $\$a_i$;
— après l'appel, le résultat est récupéré dans la variable $\$v_0$.

À noter alors que l'appel de fonction est réduit à sa plus simple expression, un simple transfert de contrôle. Ainsi, l'appel `x = f(x1, x2);` devient

```
$a0 = x1;
$a1 = x2;
call f;
x = $v0;
```

Pour les sauvegardes des registres *caller-saved* et *callee-saved*, on introduit également des copies vers de nouvelles variables dédiées à leur sauvegarde.

Oui, cela commence à faire beaucoup de variables.

— Pour chaque appel de fonction, on introduit une nouvelle variable pour la sauvegarde de chaque registre *caller-saved*. On ajoute une copie pour la sauvegarde avant l'appel, et une pour la restauration après l'appel.

```
EXEMPLE POUR QUELQUES $T
```

— On introduit en outre une variable pour la sauvegarde de chaque registre *callee-saved*, avec une instruction de sauvegarde au début du code de la fonction, et une copie de restauration à la fin de l'exécution de la fonction.

Avec ceci on ouvre la voie à la résolution de nos deux problèmes :

— il devient possible d'utiliser les registres `$a` pour le calcul des arguments (mais rien ne le force) ;
— l'analyse de vivacité nous dit quels registres `$t` contiennent des valeurs qui seront encore utiles après un appel de fonction (mais tout est copié dans de nouvelles variables quoi qu'il arrive).

La dernière étape consiste à enrichir notre problème d'allocation en y intégrant la consigne suivante :

— lorsqu'une variable y est une copie d'une variables x, essayer d'allouer à x et y le même registre.

Ainsi par exemple, un argument de fonction destiné à être transféré dans le registre `$a0` serait préférentiellement directement calculé dans ce registre s'il est bien déjà disponible. De même, lorsqu'un registre $\$r$ est sauvegardé dans une variable z, cette variables z serait préférentiellement réalisée par $\$r$ lui-même si la sauvegarde n'est pas réellement utile.

Relation de préférence et algorithme de Georges-Appel. Pour matérialiser cette consigne additionnelle on introduit une nouvelle notion de graphe d'interférence comportant deux sortes d'arêtes distinctes. À partir de maintenant, un ***graphe d'interférence*** est un graphe non orienté dans lequel :

— chaque sommet représente une variable ou un registre réel,
— il y a une ***arête d'interférence*** entre deux sommets x et y lorsque ces deux variables ne peuvent pas partager un même registre,
— il y a une ***arête de préférence*** entre deux sommets x et y lorsque l'on souhaite que ces deux variables partagent, si possible, un même registre.

À noter : on distingue dans ce graphe les sommets représentant des variables ordinaires et celles représentant [une variable représentant] un registre réel. En effet, les registres réels ont une couleur imposée !

On va enrichir également l'algorithme de coloration afin que les sommets liés par des arêtes de préférence soient fusionnés (et donc nécessairement associés à un même registre) lorsque cela peut être fait sans obérer la possibilité de colorier le graphe. Le ***critère de George*** énonce deux conditions suffisantes pour la possibilité de fusionner deux sommets x et y tout en préservant la colorabilité d'un graphe (l'une des deux conditions suffit).

— Si x représente une variable ordinaire et si
  — tout voisin z de y représentant un registre réel est également un voisin de x, et
  — tout voisin z de y de degré $\geq R$ est également un voisin de x,

  alors on peut fusionner x et y.
— Si x représente un registre réel et si
  — tout voisin z de y représentant une variable ordinaire est également un voisin de x, et
  — tout voisin z de y de degré $\geq R$ est également un voisin de x,

  alors on peut fusionner x et y.

L'algorithme de George-Appel intègre ce critère à l'algorithme de coloration déjà vu. On peut l'exprimer simplement à l'aide de cinq fonctions mutuellement récursives.

`simplify` Fonction principale. Tente de sélectionner un sommet coloriable à coup sûr (sans arêtes des préférence).

`coalesce` Tente de fusionner deux sommets liés par une arête de préférence.

`freeze` Tente d'abandonner les arêtes de préférence d'un sommet.

`spill` Écarte un sommet qui ne pourra peut-être pas être colorié.

`select` Tente de colorier le sommet sélectionné (après avoir colorié le reste du graphe).

Le pseudo-code ci-dessous combine ces cinq fonctions.

```
let rec simplify g =
  if il existe un sommet x sans arête de préférence,
     de degré minimal et < R
  then
    select g x
  else
    coalesce g

and coalesce g =
  if il existe une arête de préférence x-y satisfaisant
     le critère de George
  then
    g <- fusionner g x y
    c <- simplify g
    c[x] <- c[y]
    renvoyer c
  else
    freeze

and freeze g =
  if il existe un sommet x de degré minimal et < R
  then
    g <- oublier les arêtes de préférence de x
    simplify g
  else
    spill g

and spill g =
  if g est vide
  then
    renvoyer le coloriage vide
  else
    choisir un sommet x de coût minimal
    select g x

and select g x =
  c <- simplify (g privé de x)
  if il existe une couleur r possible pour x
  then
    c[x] <- r
  else
    c[x] <- spill
  renvoyer c
```

OBJECTIF : ÉVITER LES INSTRUCTIONS MOVE INUTILES

SAUVEGARDE CALLER-SAVED AU MOYEN DE VARIABLES ADDITIONNELLES, OUVRE LA VOIE À L'USAGE DES CALLEE-SAVED PLUTÔT QUE RECOURS SYSTÉMATIQUE À LA PILE

DEUX PARTIES INDÉPENDANTES : INCLURE LES

Contents