The Flow Caml System

version 1.00

Documentation and user's manual

VINCENT SIMONET

July, 2003

Acknowledgments I would like to thank François Pottier for his support and numerous comments in the design of the Flow Caml system and the writing of this document.

An electronic version of this document is available at the following address:
`http://cristal.inria.fr/~simonet/soft/flowcaml/manual/`

Abstract

Flow Caml is an extension of the Objective Caml language with a type system tracing information flow. Its purpose is basically to allow to write “real” programs and to automatically check that they obey some confidentiality or integrity policy. In Flow Caml, standard ML types are annotated with security levels chosen in a user-definable lattice. Each annotation gives an approximation of the information that the described expression may convey. Because it has full type inference, the system verifies, without requiring source code annotations, that every information flow caused by the analyzed program is legal with regard to the security policy specified by the programmer.

Contents

PART I

An introduction to Flow Caml

CHAPTER 1

Overview

1.1 Language-based Information Flow Analysis

A computer system generally handles considerable amount of data. It may be directly stored in memory (e.g. a physical drive) or transit through some network interface or interactive device. Thus, programs running on the system potentially have access to this information, as *inputs*—e.g. the program may read data stored in memory or listen to a network interface—but also as *outputs*—e.g. the program may write data to memory (appending new information to existing one or replacing it) or emit some message on a network interface. Then, they may violate the *privacy* or the *integrity* of data in the system by releasing secret information or corrupting sensitive one. That is the reason why it is mandatory in many situations to control manipulations performed by a program in order to ensure they fulfill some integrity or security policy.

A common solution is to provide an access control system. Roughly speaking, this consists in attaching to every fragment of data some access rights that specify who may read and/or write it; then, only authorized programs are allowed to read or write sensitive information. Such a mechanism is deployed by most operating systems, including all UNIX variants. However, this addresses only a part of the problem because it just controls accesses to information but does not trace the security or integrity laws through computation: for example, a program executed with privileged rights can read a secret location and copy its contents to a public place. Thus, access control mechanisms provide some protection but require the programs to which access is granted to be trusted without any restriction.

Information flow analysis consists in statically analyzing the source code of a program *before* its execution, in order to ensure that all the operations it performs respect the security policy of the system. In short, this requires to trace every information flow performed by the program and to check it is legal. Such an analysis may be formulated as a type system; this choice presents many advantages: types may serve as a formal specification language and offer automated verification of code—provided type inference is available. Moreover, because the analysis may be performed entirely at compile-time, it has no run-time cost.

Flow Caml is an extension of the Objective Caml language with a type system tracing information flow. Its purpose is basically to allow to write *real* programs and to automatically check that they obey some security policy. In Flow Caml, usual ML types are annotated with *security levels* chosen in a suitable lattice. Each annotation gives an approximation of the information which the expression that it describes may convey. Because it has full type inference, the system verifies, without requiring source code annotations, that every information flow performed by the analyzed program is legal w.r.t. the security policy specified by the programmer.

1.2 Relating Flow Caml to Objective Caml

Let us briefly discuss the relationship between Flow Caml and Objective Caml [LDG+02b]. First of all, one may mention that the Flow Caml system—including its type inference engine—is entirely written in Objective Caml. Although some part of Flow Caml's source code comes from that of Objective Caml, the system is distributed as a standalone program—not just a patch on Objective Caml—because its heart, the type inference engine, totally differs from the original one.

Putting aside these implementation issues which do not really concern the final user, the most important relationship between Flow Caml and Objective Caml lies in the fact that the former handles a (large) subset of the language of the latter. Roughly speaking, this means that a Flow Caml program may also be read as an Objective Caml one. However, this is not exactly true because Flow Caml type expressions include security annotations. Flow Caml handles all the core constructs of the Caml language, including imperative features (references, mutable values), exceptions (with the slight difference that exception names are no more first class values), datatypes and pattern matching. It also features most of the module layer of the language, including functors. However, Flow Caml does not support the object-oriented features of Objective Caml, nor polymorphic variants and labels. (In fact, the programming language of Flow Caml is approximately the same as that of the now defunct Caml Special Light.)

For the reason explained above, a Flow Caml program is generally not a valid input for the Objective Caml compiler. Nevertheless, the Flow Caml compiler outputs legal Objective Caml code from Flow Caml code. This allows to compile every program written in Flow Caml, using the byte-code or the native compiler, and running it as for every Objective Caml program. Moreover, it is possible to easily interface a program written in Flow Caml with Objective Caml code and hence to benefit from a large amount of existing libraries.

1.3 How to get the Flow Caml system ?

The Flow Caml system is freely available on the World Wide Web at the following address:

`http://cristal.inria.fr/~simonet/soft/flowcaml/`

The source distribution should compile on almost every UNIX machine where recent versions of GNU Make and Objective Caml are installed (including the Cygwin environment for Microsoft Windows). A binary distribution for Microsoft Windows operating systems is also provided.

1.4 Theoretical background and related work

The type system implemented in the Flow Caml system for tracing information flow has been developed by François Pottier and Vincent Simonet and is fully presented in [PS02, PS03]. These papers give a formal presentation of the type algebra and typing rules for the core of the language, that is Core ML (a λ-calculus with references, exceptions, primitives and *let*-polymorphism). They also provide a correctness proof of the type system. That means that the (non-interference) property that the system is supposed to enforce has been formally stated and verified.

The design of a type inference engine for a system providing both subtyping and polymorphism formed another part of the work. The form of subtyping present in Flow Caml is generally said to be *structural*. Dealing with subtyping constraints in an efficient way requires quite subtle algorithms; they are presented (and proved correct) in another article [Sim03]. In fact, the type inference engine of Flow Caml has been implemented independently of its final use and is also distributed as a separate library [Sim02]. Hence, we hope it will be suitable for a variety of applications.

The last step of the job consisted in integrating the information flow analysis in the Caml language itself. This required to extend in some way every programming construct provided by the language, including datatype definitions, the module system, the implementation/interface mechanism; in order to obtain a complete programming language.

To the best of our knowledge, the only other real size implementation of a language-based information flow analysis is the *Jif* system by Myers *et al.* [MNZZ01], based on the type system presented in Myers' thesis [Mye99]. This prototype handles a large subset of the Java language, which is roughly comparable to that of Flow Caml. Sketching a comparison, one of the main differences between Flow Caml and Jif is that, going up with the ML tradition, the former features polymorphism and has a full type inference algorithm, while the latter performs only local type reconstruction, in the Java style. In particular, in Jif programs, methods arguments must be annotated with their whole type, including the security annotations. On the other hand, Jif provides an interesting mechanism of *dynamic labels* which allows performing some checks at run-time. This has, for the time being, no counterpart in Flow Caml.

CHAPTER 2

A Tutorial

This chapter introduces various features of the Flow Caml language through examples, beginning with simple expressions, and finally touching upon more advanced points like the module system and the interaction with the outside world. Our goal is to provide an introduction to Flow Caml for someone who has some experience with programming in the Caml language (or possibly another functional language based on the ML type system, such as SML or Haskell). If the reader wishes to learn more about basic programming in Caml, we highly recommend the reading of the first chapter ("*The core language*") of Objective Caml's tutorial [LDG+02a].

Excepted in the last section, we use for this tutorial the interactive toplevel, which can be started by running the `flowcaml` command from the shell. Under the interactive toplevel, the user types Flow Caml phrases, terminated by `;;`, in response to the `#` prompt. The system type-checks them on the fly and prints the inferred type scheme.

We encourage the reader to run a Flow Caml toplevel while reading this tutorial in order to interactively discover the language and its type system, by testing the given examples and experimenting his/her own pieces of code. However, this tutorial is also self-contained, so it can be read off-line as well.

2.1 Security levels and data structures

In this section, we explain how ML (data)types are annotated in Flow Caml with *security levels* in order to describe information flow.

2.1.1 Simple types

Let us begin with this first definition:

```
let x = 1;;
x : 'a int
```

(In this tutorial, pieces of Flow Caml code are typeset in `roman typewriter`. They are generally followed by the output produced by the toplevel in *`slanted typewriter`*.) Our first example simply binds the identifier `x` to the integer constant `1`. The toplevel answers that this constant has type `'a int`. In Flow Caml, the type constructor `int` takes one argument, which is a *security level* belonging to an arbitrary lattice. These annotations allow the system to trace information flow. In the above example, the security level is a variable, `'a`; as every variable appearing free in a type, it is implicitly universally quantified. Basically, this means that outside of any context, the constant `1` may have any security level.

The security level of such a constant may be specified thanks to a simple type constraint. Assume we receive three integers from different sources named *Alice*, *Bob* and *Cecil* (such sources are often called *principals* in the literature):

```
let x1 : !alice int = 42;;
val x1 : !alice int
let x2 : !bob int = 53;;
val x2 : !bob int
let x3 : !charlie int = 11;;
val x3 : !charlie int
```

In Flow Caml, each data source may be symbolized by a constant security level such as `!alice`, `!bob` or `!charlie` (Any alphanumeric identifier preceded by a `!` is a suitable constant security level.) Initially, these security levels are incomparable points in the lattice: this means that the principals they represent cannot exchange any information. We will further on see how to allow some (see section 2.6).

The above bindings are global in the toplevel, hence you can use them in the next expressions you enter:

```
x1 + x1;;
- : !alice int
x1 + x2;;
- : [> !alice, !bob] int
x1 * x2 * x3;;
- : [> !alice, !bob, !charlie]  int
```

The first expression contains information about only `x1`, so its security level is `!alice`. The sum `x1 + x2` is liable to leak information about `x1` and `x2`. Then, its security level must be greater than those of `x1` and `x2`: `[> !alice, !bob]` stands for any level which is greater than or equal to `!alice` and `!bob`. This can be read as the "*symbolic union*" of these two principals. Similarly, the security level of the last expression must be greater than or equal to `!alice`, `!bob` and `!charlie`.

However, some programming experience in Flow Caml shows that using such explicit level constants is most of the time unnecessary: thanks to ML polymorphism, universally quantified type schemes are generally expressive enough to describe a piece of code (such as a function) w.r.t. information flow. In fact, the fundamental use of level constants appears in the interaction with external channels (e.g. file i/o or networking) and will be discussed in section 2.6 of the current tutorial. For the time being, we will only use them in a somewhat artificial way, just for guiding your intuition.

We now define a function which computes the successor of an integer:

```
let succ = function x -> x + 1;;
val succ : 'a int -> 'a int
```

Once again, the system automatically computes the most general typing for this definition, which is `'a int -> 'a int`. This type means that the function `succ` takes as argument one integer of some level `'a` and returns another integer whose security level is exactly the same: indeed the result of this function carries information about its input. Because its type is polymorphic w.r.t. the security level of the integer argument, you can apply `succ` on arguments of different levels:

```
succ x1;;
- : !alice int
succ x2;;
- : !bob int
```

This example is sufficient to illustrate that polymorphism on security levels is a prominent feature for type systems tracing information flow: here, in the absence of polymorphism, one have to write a specialized version of the function `succ` for every level it is used with.

It is worth noting that information flow is traced in a somewhat conservative way: in the underlying security model, there is an information flow from an input to an output as soon as knowing the latter reveals some information, even incomplete, about the former. Let us for instance consider the following function which computes the euclidean division of an integer by 2 thanks to a logical shift.

```
let half = function x -> x lsr 1;;
val half : 'a int -> 'a int
```

The inferred scheme for `half` is exactly the same as that of `succ`: it reflects that the result produced by `half` reveals some information about its input. However, this leak is only partial, because it is, for instance, not possible to completely retrieve `x1` from the result of `half x1`. In some situations, this may yield typings which are surprising at first sight:

```
let return_zero = function x -> x * 0;;
val return_zero : 'a int -> 'a int
```

In this example, the system detects a dependency between the input and the output of the function, although there is none: `return_zero` always returns zero! Roughly speaking, this is because for the system, the result of a product always leaks information about the two factors, whatever they are. Obviously, if you rewrite the function as follows:

```
let return_zero' = function x -> 0;;
val return_zero' : 'a int -> 'b int
```

you obtain a more precise statement: in the inferred type, the security level of the result of the function (denoted by the variable `'b`) is not related to that of the input (`'a`), reflecting the absence of information flow from the latter to the former.

```
return_zero x1;;
- : !alice int
return_zero' x1;;
- : 'a int
```

In addition to integers, Flow Caml offers the usual basic datatypes: booleans, floating-point numbers and characters. Like `int`, the corresponding type constructors carry one security level:

```
let y0 = true;;
- : 'a bool
let z0 = 'a';;
val z0 : 'a char
```

These do not raise particular difficulties, since they behave exactly like integers w.r.t. information flow analysis. For instance, usual arithmetic functions for floating point numbers are available:

```
let pi = 3.14159265359;;
val pi : 'a float
let pi' = 4.0 *. atan 1.0;;
val pi' : 'a float
```

2.1.2 Strings

There are two flavors of character strings in Flow Caml: immutable strings (type `string`) mutable ones (type `charray`). One may wonder why we distinguish them, since in Objective Caml every string is mutable—even if it is not used as such—and everything works well. This design choice is motivated because, in type systems tracing information flow, mutable values require some particular care which increases the complexity of types (we will discuss this point in section 2.3) whereas, in many situations, strings are used without in place modification. Hence, providing a distinct type for such cases allows better typings.

Immutable string literals appear in source code between double quotes:

```
let s = "Flow Caml";;
val s : 'a string
```

As illustrated by the above example, the type constructor for immutable strings is `string` and it has one argument which is a security level. It naturally describes all information attached to the string. The module `String` provides functions for manipulating immutable strings.

2.1.3 Lists

The variables `'a`, `'b`, etc. we have encountered in type schemes up to now stand for *levels* of the security lattice. In Flow Caml, polymorphism applies naturally on whole types (as in Objective Caml) too. For instance, define the identity function:

```
let id x = x;;
val id : 'a -> 'a
```

In this scheme, the variable `'a` stands for a type. Indeed, in Flow Caml, a variable appearing in a typing can be of different kinds: it may stand either for a level or for a type (in section 2.4, we will see it also may denote a *row*). What is more, kinds are not given explicitly by the system (and the programmer does not have to give them when he enters a type expression) because they always can be deduced from the context. The type scheme inferred for `id` does not involve any security annotation: it simply says that the function takes an argument of some type `'a` and produces a result of the same type. For instance, if you specialize the identity function so that it applies only to integers, it will have the type `'b int -> 'b int`.

In Flow Caml, the `list` type constructor has two arguments (while in Objective Caml it has only one). Thus, in the type `('a, 'b) list`, `'a` is a *type* variable which gives the type of the elements of the list; `'b` is a *level* variable describing the information attached to the *structure* of the list. This corresponds for instance to the information leaked by testing whether the list is empty.

```
let l1 = [1; 2; 3; 4];;
val l1 : ('a int, 'b) list
let l2 = [x1; x2];;
val l2 : ([> !alice; !bob] int, 'b) list
```

As usual in ML, functions manipulating lists generally perform pattern-matching on their structure. Here is such a simple function testing whether a list is empty:

```
let is_empty = function
    [] -> true
  | _ :: _ -> false
;;
val is_empty: ('a, 'b) list -> 'b bool
```

In this type, the security annotation of the boolean produced by the function, 'b does not depend on the type of the list's elements, 'a, but is the same as the level of the input list, because the function reveals information only about its structure of the list. Functions manipulating lists are often recursive, but this does not raise any particular difficulty concerning typing:

```
let rec length = function
    [] -> 0
  | _ :: tl -> 1 + length tl
;;
val length: ('a, 'b) list -> 'b int
```

The type scheme obtained for length is similar to that of is_empty: the length of the list contains some information about its structure, but not about its elements. On the contrary, a function testing whether the integer 0 appears in a list reveals information about both the structure of the list and its elements, hence its type:

```
let rec mem0 = function
    [] -> false
  | hd :: tl -> hd = 0 || mem0 tl
;;
val mem0: ('a int, 'a) list -> 'a bool
```

The module List of the standard library provides usual functions operating on lists, including the following examples:

```
let rec rev_append l1 l2 =
  match l1 with
      [] -> l2
    | hd :: tl -> rev_append tl (hd :: l2)
;;
val rev_append: ('a, 'b) list -> ('a, 'b) list -> ('a, 'b) list
let rev l = rev_append l [];;
val rev: ('a, 'b) list -> ('a, 'b) list
```

2.1.4 Options

In ML, an option is a value which may be of two different forms: either None (the empty option) or Some v, where v is another value, the *content* of the option. The type option behaves similarly to that of lists. It has two arguments too: in ('a, 'b) option, 'a is the type of the content of the option while 'b is the security level attached to the option itself, describing the information attached to the knowledge of its form. This is illustrated by the following functions:

```
let is_none = function
    None -> true
  | Some _ -> false
;;
val is_none: ('a, 'b) option -> 'b bool
```

The function is_none tests whether an option is None, by a simple pattern matching. Thus, the security level of the obtained integer is exactly that of the option: the test is likely to leak information only about the form of the argument.

```
let default = function
    None -> 0
  | Some x -> x
;;
```

```
val default: ('a int, 'a) option -> 'a int
```

Similarly, `default` matches an integer option. If it is `None`, it returns the default value `0`, and otherwise the content of the option itself. Thus, the result produced by an application of `default` carries information about both the form of the option and its content.

2.1.5 Tuples

Tuples of arbitrary length are also available in Flow Caml: if `x1`, ..., `xn` are values whose respective types are `t1`, ..., `tn` then `(x1, ..., xn)` is a tuple of type `t1 * ... * tn`. For instance:

```
let pair0 = (0, true);;
val pair0 : 'a int * 'a bool
let triple0 = (0, 1, 'a');;
val triple0 = 'a int * 'a int * 'a char
```

Product types carry no particular security annotation because—with the slight exception of observations made by the physical equality operator, see ?—all the information carried by a tuple is in fact carried by its components. However, each component of a tuple has its own security annotation, which may differ from those of the others. For instance, one may define a pair of integers whose first integer has level `!alice` and the second `!bob` :

```
let pair1 = x1, x2;;
val pair0 : !alice int * !bob int
```

2.2 Constrained type schemes

We will now show that Flow Caml features a constraint-based type system with subtyping. ML's type system (which is the basis of SML, Objective Caml or Haskell) relies on unification; which means that the only expressible relationship between (type) variables is equality. Unfortunately, as will be demonstrated by our next examples, this is not expressive enough to faithfully trace information flow in many cases, and then type schemes must include *constraints* between security levels such as inequalities.

2.2.1 Subtyping

Subtyping between security levels Let us consider a first example of function whose type scheme comprises an inequality: `f1` takes one integer `x` as argument and returns a pair formed of its successor and its sum with the global constant `x1` defined above:

```
let f1 x = (x + 1, x + x1);;
val f1 : 'a int -> 'a int * 'b int
         with 'a < 'b
         and  !alice < 'b
```

The type scheme returned by the system involves two level variables, `'a` and `'b`. The first one, `'a`, is the security level of the function's argument. Naturally, it is also that of the first component of the pair returned by the function. The second integer returned by the function is labeled by the variable `'b`. This security level is related to `'a` by the first inequality appearing after the keyword **`with`**: `'a < 'b` tells us that `'b` must be greater than or equal to `'a` (note that the character `<` output by your terminal stands, in Flow Caml, for the mathematical symbol $\leq$). In what concerns information flow, this inequality reflects the fact that the integer labeled by `'b` depends on the one labeled `'a`; in other words that there is a flow from the latter to the former. The other constraint, `!alice < 'b` requires `'b` to be greater than or equal to the constant `!alice`. It says that there is a possible flow from data (namely x1) coming from the external source symbolized by the constant !alice (the principal *Alice*) to the second output of the function.

Now, we can apply this function to different integers:

```
f1 0;;
- : 'a int * !alice int
f1 x1;;
- : !alice int * !alice int
f1 x2;;
- : !bob int * [> !alice, !bob] int
```

From a type-theoretic point of view, the type scheme inferred for f1 means that every instance of 'a int -> 'a int * 'b int for some 'a and 'b which satisfy the inequalities 'a < 'b and !alice < 'b is a valid type for the function. This statement cannot be expressed as precisely in a unification-based type system. Indeed, in such a framework, every < must be read as =, i.e. the variables 'a and 'b must be unified with the constant !alice. Thus we would obtain the following judgment:

```
val f1 : !alice int -> !alice int * !alice int
```

which is much more restrictive than the previous one: here, applying f1 to the integer 0 would yield a result of type !alice int (instead of 'a int), while the expression f1 x2 would be ill-typed. The same observation can be made with the following function, f2, which takes three integer arguments and computes the sums of each pair of them:

```
let f2 x y z =
  (x + y, y + z, x + z)
;;
val f2 : 'a int -> 'b int -> 'c int -> 'd int * 'e int * 'f int
         with 'a < 'd, 'f
         and  'b < 'd, 'e
         and  'c < 'e, 'f
```

The obtained type scheme involves three constraints; each of them relates one argument of the function to two of its outputs. For instance, the constraint 'a < 'd, 'f (which is a shorthand for 'a < 'd **and** 'a < 'f) traces the information flow from the first argument, x to the first and third components of the result, x + y and x + z respectively. The next two constraints deal similarly with the second and third arguments of the function, respectively. Obviously, the system performs some arbitrary choice when it typesets a list of constraints. For instance, f2's scheme may equivalently be written:

```
val f2 : 'a int -> 'b int -> 'c int -> 'd int * 'e int * 'f int
         with 'a, 'b < 'd
         and  'b, 'c < 'e
         and  'a, 'c < 'f
```

When one applies the function to the three constants x1, x2 and x3, the constraints allow to compute the respective levels of the resulting integer:

```
f2 x1 x2 x3;;
- : [> !alice, !bob] int * [> !bob, !charlie] int
    * [> !alice, !charlie] int
```

Once again, if the system did not feature subtyping but only unification, f2 would have a much more restrictive typing

```
val f2 : 'a int -> 'a int -> 'a int -> 'a int * 'a int * 'a int
```

which tells only that each component of the returned tuple is likely to depend on all three arguments given to the function.

Subtyping between types In the previous examples, inequalities involve only security levels. However, types are also ordered by the partial order <, which is said to be a *subtyping* order. In general terms, subtyping consists of a partial order on types and a subsumption rule that allows every expression which has a given type to be used with any greater type, i.e. if an expression e has some type t and t is a subtype of t' ($t < t'$) then e also has type t'. In Flow Caml, subtyping is structural and defined by lifting the order between security levels throughout the structure of types: two comparable types must have the same "structure" and only their annotations may differ. For this purpose, every type constructor (such as `int`, `list` or `->`) has a *signature* which gives the variance (and the kind) of each of its argument. A variance is either `+` (*covariant*), `-` (*contravariant*) or `=` (*invariant*). The signature of a type constructor can be displayed in the toplevel thanks to the directive `#lookup_type`:

```
#lookup_type "int";;
type (#'a:level) int
```

This tells that the only argument (`'a`) of `int` is a level and is covariant. (The `#` symbol is a distinguished form of `+`, whose role will be explained in section 2.2.2. For the time being, you can simply read it as if it were `+`.) This defines the subtyping order on integer types: given two security levels `'a` and `'b`, `'a int < 'b int` holds if and only if `'a < 'b`. Similarly, the two arguments of `list` are also covariant:

```
#lookup_type "list";;
type (+'a:type, #'b:level) list = ...
```

Then, `('a1, 'b1) list < ('a2, 'b2) list` is equivalent to `'a1 < 'a2` **and** `'b1 < 'b2`. As a result, subtyping constraints involving two type structures can be decomposed recursively: for instance `('a1 int, 'b1) list < ('a2 int, 'b2) list` produces `'a1 int < 'a2 int` **and** `'b1 < 'b2` and then `'a1 < 'a2` **and** `'b1 < 'b2`.

The function `f3` takes three arguments and build three lists of two elements each:

```
let f3 x y z =
  ([x; y], [y; z], [x; z])
;;
val f3 : 'a ->  'b -> 'c -> ('d, 'e) list * ('f, 'g) list * ('h, 'i) list
         with 'a < 'd, 'h
         and  'b < 'd, 'f
         and  'c < 'f, 'h
```

The typing inferred by the system is similar to that of `f2`. Each constraint relates the type of one input to those of the result: thus, the type of the first argument, `'a` is "injected" in those of the elements of the first and third lists, reflecting the dependency. However, it is worth noting that here, the variables `'a`, `'b`, `'c`, `'d`, `'e` and `'f` are types, not levels.

The arrow type constructor `->` we have encountered in the previous examples has the following signature:

```
type (-'a:type) -> (+'b:type)
```

As usual in the presence of subtyping, the result type is a covariant parameter while the argument is a contravariant one. This means that the inequality `'a1 -> 'b1` $\leq$ `'a2 -> 'b2` holds if and only if `'b1` $\leq$ `'b2` and `'a2` $\leq$ `'a1`.

Simplification of type schemes Flow Caml automatically performs some simplifications before it outputs a scheme in order to make the printing as concise as possible. Indeed, because of the presence of subtyping, the same type scheme can be written in different but equivalent forms. To illustrate this, let us consider the integer sum operator, `+`. In the Flow Caml standard library, it is declared with the following scheme:

```
val ( + ) : 'a int -> 'a int -> 'a int
```

This type scheme apparently constrains its two arguments to have the same security level. However, it is still possible to compute the sum of two integers which have different security levels, as in the following example:

```
x1 + x2;;
- : [> !alice, !bob] int
```

The integer `x1` has the type `!alice int`. By subsumption, it can be freely used with any greater type, e.g. `[> !alice, !bob] int`. (The system is able to perform the coercion itself when needed, no explicit annotation is therefore required.) Similarly, `x2` has type `!bob int` but it can also be used as a value of type `[> !alice, !bob] int`. It follows that the expression `x1 + x2` is well-typed and produces a value of type `[> !alice, !bob] int`. Generalizing this process, one may naturally propose another type scheme for `( + )`, which explicitly includes the subsumption mechanism:

```
val ( + ) : 'a1 int -> 'a2 int -> 'a3 int
            with 'a1, 'a2 < 'a3
```

Nevertheless Flow Caml tries to output every type scheme in a form that is as concise as possible (for efficiency reasons, its simplification algorithm is however incomplete). In order to help you in reading types, each occurrence of a variable is printed with a color which indicates its polarity: negative occurrences appear in green while positive ones are in red. Then, when interpreting a type, any negative (resp. positive) occurrence of a variable `'a` can be replaced by a fresh variable `'b` with the constraint `'b < 'a` (resp. `'a < 'b`). Applying this principle to the scheme

```
val ( + ) : 'a int -> 'a int -> 'a int
```

one obtains

```
val ( + ) : 'a1 int -> 'a2 int -> 'a3 int
            with 'a1 < 'a
            and  'a2 < 'a
            and  'a < 'a3
```

which becomes, by transitivity of `<`,

```
val ( + ) : 'a1 int -> 'a2 int -> 'a3 int
            with 'a1 < 'a3
            and  'a2 < 'a3
```

2.2.2 `level` constraints

Flow Caml provides the conditional construct **`if`** ... **`then`** ... **`else`** ..., which has the same semantics as that of Objective Caml, as well as polymorphic comparison primitives. As explained above, the type of boolean values carries one security level:

```
let y1 : !alice bool = false;;
val y1 : !alice bool
let y2 : !bob bool = false;;
val y2 : !bob bool
```

When it encounters a conditional construct, the execution of a program evaluates the condition, and depending on the result, continues with one of the two branches. Thus, the result produced by the whole expression is that of one of the two sub-expressions appearing after **`then`** and **`else`**. Hence, the type of the former must be a super-type of the latter. For instance, in the simple case where a conditional produces integers, this means that the security level of the whole expression must be the union of those of the two branches:

```
if y0 then x1 else x2;;
- : [> !alice, !bob] int
```

In this example, `x1` has type `!alice int`, `x2` has type `!bob int`, so the whole expression has type `[> !alice, !bob] int`. The value produced by a conditional also carries information about the result of the test. Hence, in order to take in account this possible information flow, the security level of the latter must *guard* the type of the former, this means that its security level(s) must be greater than or equal to that of the condition:

```
if y1 then 1 else 0;;
- : !alice int
```

Here, the condition, `y1`, has level `!alice`; hence the result of the whole expression must have this level too. Similarly, if a conditional evaluates to a tuple, the type of each of its components must be guarded by the level attached to the test:

```
if y1 then (x1, (true, 'a')) else (x2, (false, 'b'));;
- : [> !alice, !bob] int * (!alice bool * !alice char)
```

We now introduce some functions whose result depends on some test(s) performed on their argument(s). The function `int_of_bool` simply converts a boolean into an integer:

```
let int_of_bool x =
  if x then 1 else 0
;;
val int_of_bool : 'a bool -> 'a int
```

However, because of polymorphism, it is possible for the type produced by a conditional construct not to be known, for instance if it depends on that of some argument in an abstraction:

```
let choose y1 y0 x =
  if x then y1 else y0
;;
val choose : 'a -> 'a -> 'b bool -> 'a
             with 'b < level('a)
```

The result produced by `choose` clearly depends on the value of the boolean `x`, and hence must be guarded by its security level, `'b`. However, it has the type of the two other arguments of the function, `y0` and `y1`, but this may be arbitrary. That is the reason why this example involves a new form of constraint, `'b <` **`level`**`('a)`. (In [PS03], this is written `'b` ◁ `'a`.) Let us first remark that in this constraint `'b` and `'a` are variables of different kinds: `'b` stands for a security level while `'a` is a type. This constraint can be viewed as an inequality delayed until the structure of the type `'a` is known: roughly speaking, it means that the topmost security level(s) of the type `'a` must be greater than or equal to `'b`. For instance, if you instantiate `'a` by `'a1 int`, the constraint is simply decomposed as `'b < 'a1`, but, if you instantiate `'a` by `'a1 int * 'a2 bool`, it produces `'b < 'a1` **`and`** `'b < 'a2`. To illustrate how this decomposition mechanism works, one can consider some partial applications of the function:

```
let choose1 y = choose 1 0 y;;
val choose : 'a bool -> 'a int
let choose2 y = choose (1, 1) (0, 0) y;;
val choose : 'a bool -> 'a int * 'a int
```

In the first example, `'a` is instantiated by `'a1 int` and the constraint `'b <` **`level`**`('a1 int)` is decomposed as `'b < 'a1`. This yields the scheme

```
'b bool -> 'a1 int with 'b < 'a1
```

which is simplified into `'a bool -> 'a int`. In the second example, `'a` is instantiated by `'a1 int * 'a2 int`. The constraint becomes successively

```
'b < level('a1 int * 'a2 int)
'b < level('a1 int) and 'b < level('a2 int)
'b < 'a1 and 'b < 'a2
```

The obtained scheme can be simplified into `'a bool -> 'a int * 'a int`. Similarly, we can also consider lists:

```
let choose3 y = choose [] [1;2] y;;
val choose3 : 'a bool -> ('b, 'a) list
```

Here, the type variable `'a` is instantiated by `('a1, 'a2) list`, which yields the constraint `'b < level(('a1, 'a2) list)` that is decomposed into `'b < 'a2`.

A question naturally arises: how does the type-checker determine on which arguments of each type constructor the destructor **`level`** must be decomposed? This information is retrieved from signatures: the arguments on which **`level`** applies are those which are marked as "*guarded*" by a sharp symbol (**#**):

```
#lookup_type "int";;
type (#'a:level) int
#lookup_type "list";;
type (+'a:type, #'b:level) list = ...
```

This means for instance that **`level`** applies on the single argument of `int` while it considers only the second one of `list`. It is worth noting that **#** is a distinguished form of `+`, that means that guarded arguments are always covariant.

2.2.3 content constraints

Flow Caml supports the polymorphic comparison primitives of Objective Caml, such as `=` or `<=`. These operators can be used to compare data structures of any type, so, in Objective Caml, they have the following type

```
'a -> 'a -> bool
```

which means they expect two arguments of the same type and returns a boolean. In Flow Caml, the type of the boolean result must carry a security annotation, say the level variable `'b`. Moreover, because, the result produced by the operator is liable to carry information about the two compared values, `'b` must be related to the security levels which describe them, i.e. those that appear within the type `'a`. For instance, specialized versions of the equality for integers, pairs of integers, lists of integers and lists of pairs of integers should have the following type schemes:

```
val eq_int : 'a int -> 'a int -> 'b bool
             with 'a < 'b
val eq_int_pair : ('a1 int * 'a2 int) -> ('a1 int * 'a2 int) -> 'b bool
                  with 'a1, 'a2 < 'b
val eq_int_list : ('a1 int, 'a2) list -> 'b bool
                  with 'a1, 'a2 < 'b
val eq_int_pair_list : ('a1 int * 'a2 int, 'a3) list -> 'b bool
                       with 'a1, 'a2, 'a3 < 'b
```

Indeed, comparing two pairs can leak information about each member of each pair while comparing two lists gives some knowledge about the structure of the lists (e.g. their length) and/or their elements. Similarly, an equality operator which applies to integer references has the following type:

```
val eq_int_ref : ('a1 int, 'a2) ref -> ('a1 int, 'a2) ref -> 'b
                 with 'a1, 'a2 < 'b
```

Comparing two references reveal information about their respective addresses—hence `'a2 < 'b`—and their contents, hence `'a1 < 'b`.

We observe that in all cases, the security level `'b` that labels the boolean produced by the comparison must be greater than or equal to *every* security level that appears in the type of the arguments. This reflects how comparison applies recursively on data-structures. Thus, in order to give a principal type to these polymorphic operators, we need an additional form of constraint, `content('a) < 'b` where `'a` is a type and `'b` is a level. (In [PS03], this is written `'b ◀ 'a`.) This constraint requires every security annotation of the type `'a` to be less than or equal to the security level `'b`. For instance, `content('a1 int * 'a2 int) < 'b` is equivalent to `'a1 < 'b and 'a2 < 'b` while `content('a1 ref, 'a2) < 'b` stands for `'a1 < 'b and 'a2 < 'b`. This definition mimics the behavior of generic comparison operators which traverse data structures recursively. Then, in Flow Caml, `=` and `<=` have the following type:

```
val ( = ) : 'a -> 'a -> 'b bool
            when content('a) < 'b
val ( <= ) : 'a -> 'a -> 'b bool
             when content('a) < 'b
```

They can be used to implement a polymorphic function `mem` which searches whether an element is amember of a list:

```
let rec mem x = function
    [] -> false
  | hd :: tl -> (x = hd) || mem x tl
;;
val mem : 'a -> ('a, 'b) list -> 'b bool
          with content('a) < 'b
```

or an insertion sort on lists:

```
let rec insert x = function
    [] -> [x]
  | hd :: tl -> (min hd x) :: insert (max hd x) tl
;;
 val insert : 'a -> ('a, 'b) list -> ('c, 'b) list
              with 'a < 'c
              and  content('a) < level('c)
let rec sort = function
    [] -> []
  | hd :: tl -> insert hd (sort tl)
;;
val sort : ('a, 'b) list -> ('c, 'b) list
           with 'a < 'c
           and  content('a) < level('c)
```

In Objective Caml, it is even possible to apply polymorphic comparison primitives to *functional* values: for instance, if `f1` and `f2` are two functions, `(f1 = f2)` either returns `true` (if the two functions have the same memory address) or raises an exception in all other cases. Such an expression seems to have a very limited interest and is not really used because it largely depends on the implementation: for instance `let f = fun x -> x in (f = f)` returns `true` while `(fun x -> x) = (fun x -> x)` raises an exception. However, the Caml type system has no way to prevent such calls from arising. The SML [MTHM97] dialect of ML addresses these issues by introducing “*eq*” types, and hence refuses at compile time any application of a comparison primitive to values which (are likely to) contain closures. The same approach is followed in Flow Caml, where non-*eq* types are marked by the keywork **noneq** in their definition, and the constraint **content**('a) < 'b cannot be satisfied if 'a is a non-*eq* type. Hence, the following piece of code yields a type error:

```
(fun x -> x) = (fun x -> x);;
Magic generic primitives cannot be applied on expressions
of type ~a -> ~a
```

2.2.4 Same-skeleton constraints

In addition to inequalities, Flow Caml type schemes may involve another sort of constraints, which are referred to as *same-skeleton* constraints. Let us consider the following function:

```
let skel x y =
  if x = y then ();
  x
;;
val skel : 'a -> 'b -> 'a
           with 'a ~ 'b
```

skel x y tests whether x equals y, then returns x. In the case where the test succeeds, the function skel does nothing particular, but it should for instance be possible to replace () by an expression which performs side-effects, as we will do in section 2.3.2. However, the current function is sufficient to illustrate the need of *same-skeleton* constraints.

In Objective Caml, the two arguments of skel, x and y, will be required to be of the *same* type, in order to allow comparing them: skel's principal type scheme would be 'a -> 'a -> 'a. However, in Flow Caml, thanks to subtyping, it is no longer necessary to require them to have exactly the same type: indeed, they may have different security annotations, e.g. be two integers of different security levels. Formally, if x has type 'a and y has type 'b, it is sufficient to require the existence of a super-type 'c of 'a and 'b (i.e. such that 'a < 'c and 'b < 'c). This is what expresses the ~ constraint. Indeed, the above type scheme is equivalent to:

```
val skel : 'a -> 'b -> 'a
           with 'a < 'c
           and  'b < 'c
```

where 'c is an extra type variable. It is easy to check that such a 'c exists if and only if 'a and 'b are two types of the same *shape* or *skeleton* i.e. differ only by their non-invariant security annotations.

It is worth noting that the ~ predicate is transitive and associative (it is indeed the symmetric, transitive closure of <), so that same-skeleton constraints which involve a common variable can be merged, as in the following example:

```
let skel3 x y z =
  if x = y or y = z then ();
  x
;;
val skel3 : 'a -> 'b -> 'c -> 'a
            with 'a ~ 'b ~ 'c
```

2.2.5 Functions as values

Flow Caml is a functional language: functions are first class citizens and hence can be manipulated as regular values. Thus, one may define a function whose result itself is a function:

```
let pred x = x + 1;;
val pred : 'a int -> 'a int
let pred_or_succ y = if y then pred else succ;;
val pred_or_succ : 'a bool -> 'b int -{|| 'a}-> 'b int
```

(To help comprehension, the inferred type scheme may be parenthesized as follows:

```
val pred_or_succ : 'a bool -> ('b int -{|| 'a}-> 'b int)
```

However, this is naturally unnecessary because the arrow type constructor is right associative.) Knowing which function among `pred` or `succ` an application of `pred_or_succ` returns naturally leaks information about the boolean given as argument. In order to reflect this information flow, the type assigned to the function returned by `pred_or_succ` comprises an additional security level, `'a`, printed inside the arrow symbol: it intends to describe how much information is attached to the knowledge of the function. For instance, an application of `pred_or_succ` with a boolean of level `!alice` yields a function whose identity has level `!alice` too:

```
pred_or_succ y1;;
- : 'a int -{|| !alice}-> 'a int
```

The language naturally allows to observe the "identity" of a function by watching the result produced by some application of it. For instance, when one applies `(pred_or_succ y1)` to some integer, the result must be guarded by the level `!alice`, because it allows determining whether the function was `pred` or `succ` and hence the boolean `y1`.

```
(pred_or_succ y1) 0;;
- : !alice int
(pred_or_succ y1) x2;;
- : [> !alice, !bob] int
```

In fact, arrows in Flow Caml involve three security annotations; then, the general form of a function type is

```
'a -{'b | 'c | 'd}-> 'e
```

where `'a` and `'e` are the *types* of the argument expected by the function and the result it produces, respectively. Furthermore, `'b` and `'d` are *levels*. The former is a lower bound on the side effects performed by the function (it will be introduced in section 2.3) while the latter represents information about the function's identity, as explained above. Lastly, `'c` is a *row* describing the exceptions the function may raise (we will detail its usage in section 2.4). However, in order to improve readability, Flow Caml does not print annotations on arrows that carry no information, i.e. that are universally quantified and unconstrained type variables. For instance `'a -> 'b` is a shorthand for `'a -{'c | 'd | 'e}-> 'b` (where `'c`, `'d` and `'e` are fresh variables), while `'a -{|| 'b}-> 'c` stands for `'a -{'d | 'e | 'b}-> 'c` (where `'d` and `'e` are fresh).

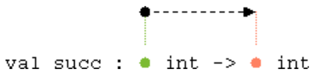
2.2.6 Interlude: the graphical output of type schemes

Flow Caml's toplevel is able to give a *graphical* representation of type schemes, in addition to the standard textual one. The graphical output is enabled when the toplevel is launched with the `-graph` option, or—at any time—by entering the following directive in the toplevel:

```
#open_graph;;
```

The graphical representation of a type scheme may be easier to interpret than its textual counterpart, because it gives a visual description of information flow. Let us explain how to read such representations on some examples.

```
let succ x = x + 1;;
val succ : 'a int -> 'a int
```

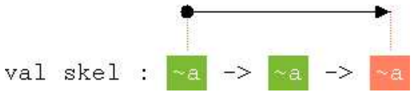


The graphical representation of a type scheme is made of two parts: at the bottom appears the *skeleton* of the scheme, which consists in the type expression where security annotations have been replaced with bullets •. Roughly speaking, ignoring these bullets, this can be read as a Caml type. A color code is adopted for drawing security annotations: contravariant annotations (which stand for *inputs*) appear in green while covariant ones (*outputs*) are drawn in red. Invariant annotations are in orange. Then subtyping constraints are represented in the top part of the drawing by arrows. In `succ`'s type scheme, the dashed arrow from the green bullet to the red one symbolizes an inequality whose left- (resp. right-) hand-side is the security annotation symbolized by the green (resp. red) bullet. Then, the drawing must be read as the following scheme

```
val succ : 'a int -> 'b int
           with 'a < 'b
```

which is equivalent to `'a int -> 'a int`. Let us now show how type variables are graphically represented.

```
let skel x y =
  if x = y then ();
  x
;;
val skel : 'a -> 'b -> 'a
           with 'a ~ 'b
```



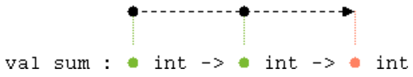
In the body of the graphical representation of `skel`'s type scheme, the boxes labeled `~a` stand for a *skeleton class*: each occurrence of `~a` must be read as a different type variable a_1, a_2, ..., a_n, with the constraint a_1 ~ a_2 ~ ...~ a_n. For instance, `~a -> ~a -> ~a` represents

```
'a₁ -> 'a₂ -> 'a₃
with 'a₁ ~ 'a₂ ~ 'a₃
```

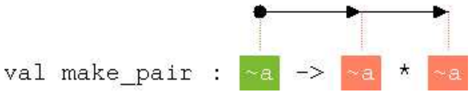
The same color code is followed for boxes as for bullets. Subtyping constraints between type variables are represented by black arrows: in skel's type scheme, the arrow from the first box to the third one stands for the constraint `'a1 < 'a3`.

Arrows heads and tails can be shared, as in the following examples:

```
let sum x y = x + y;;
val sum : 'a int -> 'a int -> 'a int
```



```
let make_pair x = (x, x);;
val make_pair : 'a -> 'a -> 'a
```

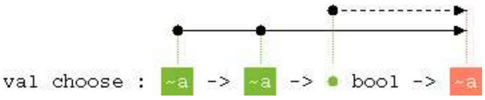


It remains to show how special forms of inequalities, i.e. constraints such as **content**`('a) < 'b`, `'a <` **level**`('b)` and **level**`('a) <` **content**`('b)`, are drawn. All of them are represented by a dashed arrow from (the box/bullet which stands for) `'a` to (the box/bullet which stands for) `'b`. No confusion can arise thanks to kinding as illustrated by the following table:

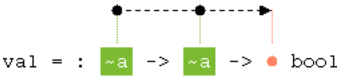
kind of 'a	kind of 'b	meaning of a dashed arrow from 'a to 'b
level	**level**	'a < 'b
level	**type**	'a < **level**('b)
type	**level**	**content**('a) < 'b
type	**type**	**content**('a) < **level**('b)

This is illustrated by the two following examples:

```
let choose y1 y0 x =
  if x then y1 else y0
;;
val choose : 'a -> 'a -> 'b bool -> 'a
             with 'b < level('a)
```



```
( = );;
val ( = ) : 'a -> 'a -> 'b bool
            with content('a) < 'b
```



In the type scheme of `choose`, the dashed arrow symbolizes the constraint `'b < **level**('a)`, while in that of `( = )` it stands for `**content**('a) < 'b`.

2.3 Imperative features

Though all the examples given so far in this tutorial are in a "purely functional" style, Flow Caml also provides full imperatives features. This includes mutable data structures such as references and arrays, as well as usual **while** and **for** loops.

2.3.1 Direct and indirect information flows

Unfortunately, in a programming language equipped with side effects, it is possible to leak information in *indirect* ways. Let us consider the following pieces of code:

```
r := not y
r := if y then false else true
if y then r := false else r := true
r := true; if y then r := false
```

All of them are semantically equivalent: they update the content of the reference `r`, storing in it the negation of the boolean `y`. Hence, this produces some information flow from `y` to `r`. However, depending on the cases, it is of a different nature. In the two first examples, the flow is said to be *direct*: a value depending from `y` is computed and then stored in `r`; this is very similar to what we have encountered up to now. On the contrary, in the last two expressions, the value in every right-hand-side of the `:=` operator does not involve `y`: it is even given explicitly in the source code. However, the reference's update is performed in a branch of the program whose execution is conditioned by the value of `y`. In this situation, we say there is an *indirect* flow form `y` to `r`. The last example calls for an additional comment: in the case where the boolean `y` is `false`, the reference `r` is never updated in a context conditioned by `y`. However, the information flow from the latter to the former still exists: it is indeed possible to leak information through the *absence* of a certain effect. (This last example shows that it would be very difficult to detect information flow at run time.)

2.3.2 References

In Flow Caml, the type constructor for references, `ref`, has two arguments:

```
#lookup_type "ref";;
type (='a:type, +'b:level) ref = ...
```

The first one is the type of the value stored in the reference. Because the content of a reference is accessible in reading and writing, it must be at the same time covariant and invariant, i.e. it is *invariant*. The second argument of `ref` is a security level, which is guarded and covariant. It describes how much information is attached to the *identity* of the reference, in other words its memory address.

Let us now illustrate how information flow with mutable structures is traced by some examples of Flow Caml code. We first define two references `r1` and `r2` whose contents are declared to be booleans of levels `!alice` and `!bob`, respectively.

```
let r1 : (!alice bool, 'a) ref = ref true;;
val r1 : (!alice bool, 'a) ref
let r2 : (!bob bool, 'a) ref = ref true;;
val r2 : (!bob bool, 'a) ref
```

In the above example, the content of reference `r1` has type `!alice bool`. This means it may receive any boolean whose security level is less than or equal to `!alice`, i.e. a boolean *Alice* is allowed to read. The boolean `y1` (defined in section 2.1) has level `!alice`. Hence it can legally be stored in `r1`:

```
r1 := y1;;
- : unit
```

This expression only produces a side-effect, so it has type `unit`. Because there is only one value of this type, the constant `()`, the value of a `unit` expression yields no information. At a result, the `unit` type constructor does not carry any security annotation. On the contrary, the boolean `y2` has been declared with the level `!bob`. Because information flow from `!bob` to `!alice` is not allowed (see section 2.6), assigning it to `r1` raises a typing error:

```
r1 := y2;;
This expression generates the following information flow:
  from !bob to !alice
which is not legal.
```

Similarly, the reference `r1` can be updated in a context whose execution depends on `y1` but not `y2`:

```
if y1 then r1 := false else r1 := true;;
- : unit
if y2 then r1 := false else r1 := true;;
This expression generates the following information flow:
  from !bob to !alice
which is not legal.
```

Lastly, reading the content of `r1` naturally yields a boolean of level `!alice`:

```
!r1;;
- : !alice bool
```

Let us explain in a few words how the type system is able to trace indirect information flow in the above examples. Flow Caml associates to every context of an expression (i.e. every point of the program) a security level telling how much information the given sub-expression gains when it is executed. (In the literature, this security level is generally written *pc*, in reference to *program counter*.) Basically, each time a conditional construct is traversed, this level is augmented with the annotation of the condition, as illustrated in this example:

```
if y1 (* y1 has type !alice bool *) then
  ... (* this branch is typechecked at level !alice *)
else
  if y2 (* y2 has type !bob bool *) then
    ... (* this branch is typechecked at level [> !alice, !bob] *)
  else
    ... (* this branch is typechecked at level [> !alice, !bob] *)
```

Moreover, when some data is written in a reference, the system constrains the level of its content to be greater than or equal to the security level attached to the context, reflecting the fact that, because of the update, the content of the reference is liable to carry information about all the tests traversed to reach this point of the program.

The purpose of the second argument of the type constructor `ref` appears when a reference is used as first class value, e.g. if it is the result of some function. For instance, let us define a version of the function `choose` specialized for references by a type constraint:

```
let choose_ref y r1 r0 : (_, _) ref =
  if y then r1 else r0
;;
val choose_ref : 'a bool ->
                 ('b, 'a) ref -> ('b, 'a) ref -> ('b, 'a) ref
```

The security level of the reference returned by this function must be greater than or equal to that of the boolean given as argument. Indeed, revealing which reference is returned by the function may leak information about the value of the condition, `y`. Such an observation can be performed, for instance, by updating its content.

Some additional difficulty arises when one defines a function performing side-effects: because the body of a function is executed at the point of the program where it is applied —and not the one where it is defined— it must be type-checked at the level of the former rather than the latter. This is the purpose of the first security annotation appearing in function types (see section 2.2.5). For instance, consider a function which sets the content of `r1` to `false`:

```
let reset_r1 () =
  r1 := false
;;
val reset_r1 : unit -{!alice ||}-> unit
```

This function can only be executed in a context whose level is less than or equal to `!alice`. This is reflected by the annotation `!alice` printed "inside" the arrow symbol of the above type: this security level is a lower bound on the effects performed by the function and an upper bound on the contexts where it can be applied. In many cases, it is a variable related to (parts of) the type of the function's argument:

```
let reset r =
  r := false
;;
val reset : ('a bool, 'a) ref -{'a ||}-> unit
```

The function reset takes a reference as argument and sets its content to false. The type system constrains the level of the content of the reference to be equal to or greater than (1) the level attached to the reference's identity and (2) the level attached to the context where the function is applied.
The identity of the reference returned by this function carries information about those of the references given as argument, but also about the boolean y. This is reflected in the inferred scheme by the fact that all of them are annotated by the same security level, 'a.

We now re-implement the function calculating the length of a list, length, in imperative style:

```
let length' list =
  let counter = ref 0 in
  let rec loop = function
      [] -> ()
    | _ :: tl ->
      incr counter;
      loop tl
  in
  loop list;
  !counter
;;
val length' : ('a, 'b) list -{'b ||}-> 'b int
```

The obtained scheme appears more restrictive than length's type:

```
val length: ('a, 'b) list -> 'b int
```

Indeed, with length', the result's security level must be greater than or equal to the function's *pc* parameter. However, the difference is only superficial; it can be checked that both types in fact have the same expressive power.

2.3.3 Arrays, strings and loops

We conclude this section by a few words about arrays and (mutable) strings. In Flow Caml, the type constructor for arrays (array) carries two arguments:

```
[|0; 1; 2|];;
- : ('a int, 'b) array
```

The respective roles of these arguments are similar to those of ref: the former is the type of the content of the cells of the array, and the latter is a security level, related to the array identity. A slight novelty is that this comprises information attached to the length of the array. Indeed, the function returning the length of an array has the following type:

```
Array.length;;
- : ('a, 'b) array -> 'b int
```

(which is similar to that of the function calculating the length of a list.)

In the Caml language, the only difference between a mutable string and an array of characters concerns its representation in the runtime system. Hence, the type of mutable strings is isomorphic to that of an array of characters:

```
[|'a'; 'b'; 'c'|];;
- : ('a char, 'b) array
"abc";;
- : ('a, 'b) charray
```

Indeed, the type constructor charray expects two security levels as arguments. The first one describes information attached to the characters stored in the string while the second one is related to the identity of the string (including its length).

2.4 Dealing with exceptions

In this section, we explain how Flow Caml deals with exceptions. For the programmer, exceptions are a powerful mechanism for signaling and handling exceptional conditions. As in Objective Caml, *exceptions names* are declared with the **exception** construct and signaled with the **raise** operator:

```
exception X;;
exception X
exception Y;;
exception Y
raise X;;
- : 'a
```

However, the exception machinery provided by Flow Caml is slightly restricted in comparison with that of Objective Caml, mostly because exceptions are not first class values. Basically, an exception name (such as `X` in the above example) is not a value, and hence cannot be bound to a variable or passed as argument to a function (while in Objective Caml, it is a legal value of type `exn`). Similarly, in Objective Caml, **raise** is a regular function which accepts an arbitrary argument (of type `exn`), but, in Flow Caml, it is a built-in construct which requires the name of the raised exception to be statically specified. For instance, the following Objective Caml piece of code cannot be written in Flow Caml:

```
let f x =
  raise (if x then X else Y)
;;
```

but, in this particular case, it may be rewritten into:

```
let f x =
  if x then raise X else raise Y
;;
```

Although it should theoretically be possible to deal with exceptions as first class citizens in Flow Caml [PS02], we believe our design choice to be a good balance between expressiveness and simplicity: having first class exceptions would generate complex typings (which involve *conditional* constraints), whereas, according to our experience, the use of exceptions as values in *real* programs seems to be rather limited. To mitigate the loss in expressiveness and provide alternatives for the most common usages of exceptions as first class values made in Caml programs, Flow Caml provides two additional constructs for handling exceptions: **try** ... **finally** and **try** ... **propagate** (see section 2.4.3).

2.4.1 Rows

The exceptions that an expression is likely to raise are traced in Flow Caml's type system using a *row*. A row is a mapping from exception names to security levels: for every exception name, it tells how much information is leaked if the related expression effectively raises an exception of this name. Because the set of exception names is open (in the sense that the programmer can incrementally define an arbitrary number of them), rows must range over all potentially definable exceptions names; hence they are infinite objects. So, in order to allow denoting them in a finite concrete syntax, Flow Caml uses *row variables* and adopts Rémy's row syntax. For instance, the (row) expression `X: 'a; Y: 'b; 'c` stands for the row which maps the exception name `X` to `'a`, `Y` to `'b` and whose other entries are given by `'c`. Here, `'a` and `'b` are levels while `'c` is a row variable of *domain* `{X, Y}`: it stands for a row ranging over all exception names except `X` and `Y`. The order in which fields appear is not significant: the above row is equal to `Y: 'b; X: 'a; 'c`. Row variables can appear in constraints: the subtyping order is extended point-wise to rows. Indeed, if `'c1` and `'c2` are two row variables of the same co-domain, the constraint `'c1 < 'c2` means that every entry of `'c1` must be less than or equal to the corresponding one in `'c2`. Hence, constraints involving expanded row terms may be decomposed: `X: 'a1; Y: 'b1; 'c1 < X: 'a2; Y: 'b2; 'c2` is equivalent to `'a1 < 'a2 and 'b1 < 'b2 and 'c1 < 'c2`. Lastly, for the sake of conciseness, when it prints a type scheme, Flow Caml omits unconstrained universally quantified row variables: for instance, `A: 'a; Y: 'b` stands for `A: 'a; Y: 'b; 'c` where `'c` is a fresh row variable.

Because exceptions constitute an observable form of result for functions, they must be taken in account in their types. Let us for instance define a simple function which raises the exception `X`:

```
let raise_X () =
  raise X
;;
val raise_X : unit -{'a | X: 'a |}-> 'b
```

The second annotation appearing on the arrow is a row describing the exceptions that the function is likely to raise when it is called. Here, `X: 'a`, tells that the given function may raise an exception of name `X`: catching this exception leaks information about the context where the function is called, so the security level associated to `X` is constrained to be at least that of the context where the function is applied (which appears as usual in first place in the arrow). In the following example,

```
let raise_X' y =
  if y then raise X
;;
val raise_X' : 'a bool -{'a | X: 'a |}-> unit
```

catching the exception `X` gives information about both the context where `raise_X'` has been applied *and* the boolean argument given to the function. Thus, the annotation associated to the entry `X` in the row of this function must be greater than or equal to the security levels of both.

When a function is likely to raise exceptions of different names, its row comprises one entry for each of them:

```
let raise_X_or_Y x y =
  if x then raise X;
  if y then raise Y
;;
val raise_X_or_y : 'a bool -> 'b bool -{'a | X: 'a; Y: 'b |}-> unit
                   with 'a < 'b
```

The type scheme inferred by the system distinguishes one security level for each exception name: handling `X` yields information only about the first argument, `x`; while handling `Y` about both.

Let us now define a function which takes an integer as argument, raises `X` if it is zero and returns `false` otherwise:

```
let test_zero x =
  if x = 0 then raise X;
  false
;;
val test_zero: 'a int -> {'a | X: 'a |}-> 'b bool
```

The inferred type schemes states that the boolean returned by the function does not depend on its argument. Indeed, if the function effectively produces a value, it is invariably `false`. However, this function can reveal information about its argument through its effect. This is reflected by the security level associated to the exception `X` in its type: it must be greater than or equal to the levels of the context where the function is applied and the integer argument.

Exceptions can be trapped with the **`try ... with`** construct.

```
try
  test_zero x1
with
  X -> true
;;
- : !alice bool
```

In this example, `test_zero x1` is liable to raise an exception `X` with the level `!alice`, which will be catched by the handler **try** ... **with** X ->. Thus, the value produced by the whole construct must be guarded by the level of the handled exception, i.e. `!alice`. Let us embed this piece of code in a function:

```
let f5 x =
  try
    test_zero x
  with
    X -> true
;;
- : 'a int -{'a ||}-> 'a bool
```

The type scheme output by the system reflects that the output of `f` carries information about its argument, but also about the context where the function is called although it does not. However, we witness the same phenomenon as for side effects: once again this is only a superficial difference with the typing obtained for the function written in a direct style:

```
let f6 x =
  if x = 0 then true else false
;;
val f6 : 'a int -> 'a bool
```

The **with** part is actually a kind of pattern-matching on exception names (however, this is not a regular pattern matching since exceptions are not values). In particular, one **try** .. **with** construct can catch several exceptions names (or even all of them using the _ pattern), as illustrated by the following example:

```
let f7 x y =
  try
    raise_X_or_Y x y;
    0
  with
    X -> 1
  | Y -> 2
;;
val f7 : 'a bool -> 'a bool -{'a ||}-> 'a int
```

Many functions in the standard library raise an exception when they cannot complete normally. For instance, the integer division yields `Division_by_zero` when its second argument is zero, as reflected by its type:

```
val ( / ) : 'a int -> 'b int -{'c | Division_by_zero: 'c |}-> 'a int
            with 'b < 'c, 'a
```

It is noticeable that there is no relationship between the level of the first argument and that of the exception `Division_by_zero`. This reflects that this operator does not need to match its first argument before raising the exception.

For the purpose of obtaining a precise information flow analysis, the evaluation order of expressions must be specified. As a result, the right-to-left evaluation order of the current implementation of the Objective Caml language is made part of the specification of the Flow Caml core language. For instance, Flow Caml type system takes into account that the arguments passed to a function are evaluated from left to right:

```
let f8 x y =
  (if x = 0 then raise X else x) + (if y = 0 then raise Y else y)
;;
val f8 : 'a int -> 'b int -{'c | X: 'd; Y: 'e |}-> 'f int
         with 'b < 'd, 'e, 'f
         and  'c < 'd, 'e
         and  'a < 'd, 'f
```

The inferred type scheme reflects that the condition `y = 0` is considered before `x = 0`, so the exception `X` carries information about `x` and `y` while `Y` only about `y`. Assuming a left-to-right evaluation order, one would obtain the following scheme, where the roles of the variables `'a` and `'b` are exchanged:

```
'a int -> 'b int -{'c | X: 'd; Y: 'e |}-> 'f int
with 'b < 'd, 'f
and  'c < 'd, 'e
and  'a < 'd, 'e, 'f
```

Lastly, if the evaluation order were not specified, the type system would have to consider every possible strategy. In this case, the type scheme for `f8` would be much less informative about the function:

```
'a int -> 'a int -{'b | X: 'b; Y: 'b |}-> 'a int
with 'a < 'b
```

Indeed, the security levels associated to the two exceptions are no longer differenced and must both be greater than or equal to those of the two integer arguments.

2.4.2 Exceptions and side-effects

When, in a toplevel phrase, an exception is raised outside of any handler, it will not be trapped and hence the program must terminate. As a consequence, if they gain control, the next phrases of the program observe that the exception was not raised; and this must be taken in account in the type system.

```
test_zero x1;;
- : 'a bool
Current evaluation context has level !alice
```

In this example, if `x1` is zero, evaluating this toplevel phrase causes the program to terminate. Hence, if this does not happen and execution continues, the remaining expressions receive some information about `x1` when they are evaluated. Therefore, they must be type-checked in a context augmented with the security level of the information carried by `x1`, i.e. `!alice`. This point is expressed by the second line output by the system. Thus, all side-effects performed afterward by the program must affect data of levels greater than or equal to `!alice`. For instance, the reference `r1` (whose content has level `!alice`) can be updated while `r2` (whose content has level `!bob`) cannot:

```
r1 := false;;
- : unit
r2 := false;;
```

```
This expression is executed in a context of level !alice
but has an effect at level !bob.
This yields the following information flow:
  from !alice to !bob
which is not legal.
```

Similarly, if one feeds `test_zero` with `x2`, the level of the toplevel context is increased by `!bob`:

```
test_zero x2;;
- : 'a bool
Current evaluation context has level !bob, !alice
```

and now, the reference `r1` can no longer be updated, because information flow from `!alice` to `!bob` is not allowed (i.e. `!alice` is not inferior to `!bob`).

In a regular program, any increase of the security level associated with the topmost evaluation context is naturally irremediable. However, in the interactive toplevel, for the convenience of the user, it is possible to reset it to its initial value—as if a new program was started—by the simple directive:

```
#reset_context;;
Level of evaluation context reset
```

The previous examples illustrate how the security level attached to the evaluation context increases through a sequence of toplevel phrases. Sequencing of statements is also possible inside expressions, thanks in particular to the `;` operator. Let us for instance consider the following function:

```
let f9 x r =
  if x = 0 then raise X;
  r := false
;;
val f9 : 'a int -> ('b bool, 'b) ref -{'a | X: 'a |}-> unit
        with 'a < 'b
```

`f9` takes two arguments: an integer `x` and a boolean reference `r`. If the integer is `0` then the exception `X` is raised, and the following statement is not performed. Otherwise, execution continues and the reference `r` is set to `false`. We now explain the typing inferred by the system, which reflects the two possible observable effects of the function. First of all, it may raise the exception `X`. The security level attached to this effect, `'a`, must be greater than or equal to that attached to the context where the function is applied and that of the integer argument, because the exception raising is conditioned by a test on `x`. The second effect that the function is liable to have is the update of the reference `r`. Observing it gives information naturally about the context where `f` has been called, but also reveals whether the exception `X` has been raised, and, as a consequence, about the integer `x`. That is the reason why, the security level of the content of the reference, `'b`, must be greater than the levels of the context where the expression is applied and the first argument, as reflected by the constraint `'a < 'b`.

Similarly, every side-effect performed in an exception handler must have a level greater than or equal to that of the trapped exception. If one rewrites `f9` into the following, the inferred type scheme is similar:

```
let f10 x r =
  try
    if x <> 0 then raise X;
    ()
  with
    X -> r := false
;;
```

```
val f10 : 'a int -> ('a bool, 'a) ref -{'a ||}-> unit
```

Indeed the reference update, `r := false` is increased by the security level associated to `X` in the expression between **`try`** and **`with`**.

2.4.3 The special constructs: `try ... finally` and `try ... propagate`

In addition to the traditional **`try ... with`**, Flow Caml features two other ways of handling expressions. Two reasons have motivated their introduction: firstly, they partially counterbalance the loss of expressiveness resulting of our decision to make exception names second-class citizens; secondly, they allow a more precise typing (w.r.t. information flow) of common idioms.

Each clause appearing in the **`with`** part of a **`try ... with`** may be terminated by the keyword **`propagate`**. In this case, the exception trapped by the handler is re-raised at the end of its execution. For instance:

```
try
  e
with
  X | Y -> e'; propagate
```

evaluates `e`. If it raises `X` or `Y` then `e'` is executed and, then, the trapped exception, `X` or `Y`, is raised again. In Objective Caml, this can be implemented by binding the exception to an identifier:

```
try
  e
with
  X | Y as x -> e'; raise x
```

In addition to the fact that this is not possible with the second-class exceptions of Flow Caml, providing a dedicated idiom for this idiom allows a fine-grained typing: the exception catched by the handler is propagated with exactly the same level than it was in `e`. In particular, two different levels may be associated to `X` and `Y`.

The **`try ... finally`** construct of Flow Caml is a translation of the Java's construct for the Caml language. Indeed

```
try
  e1
finally
  e2
```

first, evaluates `e1`, which yields either a regular value or an exception. In both cases, `e2` is executed, and the result produced by `e1` is returned. Once again, this can be encoded in regular Caml (without even using exception values):

```
try
  let r = e1 in e2; r
with
  exn -> e2; raise exn
```

(assuming `e2` does not raise any exception). However, using the dedicated construct **`try ... finally`** allows better typings (w.r.t. information flow): this makes explicit that the expression `e2` is *always* executed (whether `e1` raises an exception or not). Thus, the type system is able to take this in account and type-checks `e2` in a context whose level is not altered by those of the exceptions possibly raised by `e1`, whereas, in the proposed encoding, it is. For instance the following piece of code is accepted by the type system:

```
try
  if y1 then raise X
finally
  r2 := false
;;
```

while its expansion is rejected

```
try
  if y1 then raise X;
  r2 := false;
with
  _ -> r2 := false; propagate
;;
```

2.4.4 Parameterized exception names

In Objective Caml, it is possible to declare an exceptions name which takes some argument when it is raised. The type of the argument is given when the exception is defined, this may be for instance an integer corresponding to some error code:

```
exception Error of int;;
```

In Flow Caml, such a definition is also possible. However, the type of integers is parameterized by a security level which must in consequence appear in the declaration of the exception. For instance, one may introduce the exception `ErrorAlice` which is parameterized by an integer of level `!alice`:

```
exception ErrorAlice of !alice int;;
```

and then this exception can be raised with different integers of level less than or equal to `!alice` as argument:

```
raise (ErrorAlice 0);;
raise (ErrorAlice x1);;
```

However, this has two limitations: first, it is not possible to raise the exception `ErrorAlice` with an argument whose level is, for instance, `!bob`. Obviously, a workaround may consist in defining another exception name:

```
raise (ErrorAlice x2);;
This expression generates the following information flow:
  from !bob to !alice
which is not legal.
exception ErrorBob of !bob int
;;
raise (ErrorBob x2);;
```

However, this does not seem practical. That is the reason why, in Flow Caml, the type of exceptions arguments may be parameterized by some security level:

```
exception Error : 'a of 'a int;;
```

To explain how the security level of the argument given to an exception is traced, let us define the following function:

```
let error code =
  raise (Error code)
;;
```

```
val error: 'a int -{'a | Error: 'a |}-> 'b
```

The security level associated to the exception `Error` in the row of this function combines two pieces of information: first, the security level of the context where the exception is raised and, that of the integer argument. Merging these two annotations into a single one is relatively *ad hoc*; however, this allows keeping concise typings, and works well with most common usage of exceptions with arguments. It should be possible to provide a more flexible mechanism for parameterizing types of exceptions arguments, for instance by allowing several security levels as arguments, which will also appear in rows. However, this would increase the complexity of the system, as well as the verbosity of function types.

To conclude this section about exceptions, let us mention that some of the built-in exceptions of Objective Caml are not defined in the Flow Caml library. This is the case for instance of `Out_of_memory` and `Stack_overflow` which are respectively raised by the garbage collector when there is insufficient memory to complete the computation and the bytecode interpreter when the evaluation stack reaches its maximal size. Indeed, analyzing them with Flow Caml would be of little sense, because, in absence of sophisticated memory and stack analyzes, one must assume them to be possibly raised at almost every point of the program. That is the reason why they are not provided in Flow Caml library: thus, they cannot be catched by programs and become fatal errors.

Before reading further this tutorial, please reset the security level of the evaluation context in your toplevel by entering:

```
#reset_context;;
```

2.5 Defining new types

In Flow Caml, the programmer can introduce types thanks to the **`type`** declaration. First and foremost, this allows to define new data structures using records and variants. The mechanism used to define types in Flow Caml is similar to that of Objective Caml. However, type declarations involve additional information, in order to deal with the extra features of the type system related to the security analysis.

2.5.1 Variants

We begin this introduction with some examples of variant datatypes. Let us first define a type whose values are the four cardinal points:

```
type 'a cardinal =
    North
  | West
  | South
  | East
  # 'a
;;
type (#'a:level) cardinal = North | West | South | East # 'a
```

In the Flow Caml type system, information carried by a value of type `cardinal` (which is one of the four symbolic constants listed in the declaration) is described by one security level, similarly to the built-in enumerated types, such as integers or characters. Indeed, the type constructor `cardinal` has one argument which is a security level. In the above definition, this argument, `'a`, is declared to be the information level related to the sum by the clause `# 'a`.

The answer produced by the system when a type is defined gives its signature, which is automatically inferred. This gives the kind and the variance of every argument: `#'a:level` means that `'a` is a parameter of kind `level`, is covariant and must be guarded.

```
let p0 = North;;
val p0 : 'a cardinal
let p1 : !alice cardinal = North;;
val p1 : !alice cardinal
let p2 = if y2 then North else South;;
val p2 : !bob cardinal
```

We now define the function `rotate`, which takes as argument a cardinal point and returns its successor in the clockwise order:

```
let rotate = function
    North -> East
  | West -> North
  | South -> West
  | East -> South
;;
val rotate: 'a cardinal -> 'a cardinal
```

As reflected by the inferred type scheme, this function takes a cardinal point of any level as argument and returns another cardinal point of the same level as a result.

In sections 2.1.3 and 2.1.4, we have encountered two examples of variant datatypes, which are predefined in the system. Obviously, there is nothing magic with them and, now, we can give their regular definition:

```
type ('a, 'b) option =
    None
  | Some of 'a
  # 'b
;;
type (+'a:type, #'b:level) option = None | Some of 'a # 'b
```

The definition lists all the possible forms of a value of type `('a, 'b) option`: it is either the constant `None` or the constructor `Some` with some argument of type `'a`. The fourth line of the declaration, `# 'b` tells that `'b` is the security level attached to the knowledge of the form of the option, i.e. whether it is `None` or `Some`. (Let us recall that, in the second case, information carried by `Some`'s argument is reflected by the security levels appearing in the type `'a` itself.)

The output produced by the system after the definition of this type gives the signature of the type `option`: `+'a:`**`type`** means that the first argument is covariant and is a type; while **`#`**`'b:`**`level`** means that the second argument is a level, and is covariant and guarded.

The definition of the type `list` is naturally recursive; but this has no particular consequence and the declaration is therefore similar to the previous one:

```
type ('a, 'b) list =
    []
  | :: of 'a * ('a, 'b) list
  # 'b
;;
type (+'a:type, #'b:level) list = [] | (::) of 'a * ('a, 'b) list # 'b
```

Following the same principle, one can also define binary trees as follows:

```
type ('a, 'b) tree =
    Leaf
  | Node of ('a, 'b) tree * 'a * ('a, 'b) tree
  # 'b
;;
```

```
type (+'a:type, #'b:level) tree =
    Leaf
  | Node of ('a, 'b) tree * 'a * ('a, 'b) tree
  # 'b
```

and define the function which computes the height of a tree:

```
let rec height = function
    Leaf -> 0
  | Node (tl, _, tr) -> max (height tl) (height tr)
;;
val height: ('a, 'b) tree -> 'b int
```

As for the function which calculates the length of a list, the inferred type scheme reflects that the height of a tree depends on its structure (i.e. the embedding of constructors `Leaf` and `Node`) but not on the values stored inside the nodes.

Equipping a type definition taken from an Objective Caml program in order to obtain a Flow Caml one which includes security annotations, generally requires to make some design choices, related to the security properties one wants to enforce. For instance, let us consider the definition, in Objective Caml, of binary trees whose nodes are labeled by integers:

```
type int_tree =
    ILeaf
  | INode of int_tree * int * int_tree
;;
```

There are at least two relevant ways to "decorate" this definition in order to obtain a Flow Caml type. A first solution consists in specializing the definition of the type `tree` given above:

```
type ('a, 'b) int_tree =
    ILeaf
  | INode of ('a, 'b) int_tree * 'a int * ('a, 'b) int_tree
  # 'b
;;
type (+'a:level, #'b:level) int_tree =
    ILeaf
  | INode of ('a, 'b) int_tree * 'a int * ('a, 'b) int_tree
  # 'b
;;
```

In this case, the type of a integer tree carries two security annotations, `'a` and `'b`: the former describes information attached to the integers stored in the tree while is related to the structure of the tree. In fact, the type `('a, 'b) int_tree` is isomorphic to `('a int, 'b) tree`. This allows distinguishing the knowledge of the structure of a tree from that of its labels. To illustrate this point, let us define two functions: `size` which calculates the number of nodes of a tree and `sum` which calculates the sum of its labels:

```
let rec size = function
    ILeaf -> 0
  | INode (tl, x, tr) -> size tl + 1 + size tr
;;
val size : ('a, 'b) int_tree -> 'b int
let rec sum = function
    ILeaf -> 0
  | INode (tl, x, tr) -> sum tl + x + sum tr
;;
```

```
val sum : ('a, 'a) int_tree -> 'a int
```

The result of the first function depends only on the structure of the tree, and not on its content. As a result its security level is related only to the second annotation of the tree given as argument. On the other hand, the sum of the labels of a tree carries information about the structure of the tree and about its labels, so in the type of `sum`, the security level of the returned integer must be greater than or equal to the two ones of the tree.

A possible alternative consists in equipping the type of trees labeled by integers with only one security level:

```
type 'a int_tree1 =
    ILeaf1
  | INode1 of 'a int_tree1 * 'a int * 'a int_tree1
  # 'a
;;
type (#'a:level) int_tree1 =
    ILeaf1
  | INode1 of 'a int_tree1 * 'a int * 'a int_tree1
  # 'a
;;
```

Information carried by the labels is no longer distinguished from the knowledge of the structure of the tree. This choice allows more concise and simpler but less precise typings. For instance, with this definition, both functions computing respectively the size of a tree and the sum of its labels have the same types:

```
let rec size1 = function
    ILeaf1 -> 0
  | INode1 (tl, x, tr) -> size1 tl + 1 + size1 tr
;;
val size1 : 'a int_tree1 -> 'a int
let rec sum1 = function
    ILeaf1 -> 0
  | INode1 (tl, x, tr) -> sum1 tl + x + sum1 tr
;;
val sum1 : 'a int_tree1 -> 'a int
```

The type obtained for `size1` gives a less precise description of the behavior of the function w.r.t. information flow than that of `size`: it does not reflect that the size of a tree does not depend on the value of its labels, as reflected by these computations:

```
size (INode (ILeaf, x1, ILeaf));;
- : 'a int
size1 (INode1 (ILeaf1, x1, ILeaf1));;
- : !alice int
```

2.5.2 Records

Records as tuples Records are an improved form of tuples, with named fields. Here, we declare a record type to represent points or vectors in a two-dimensional space.

```
type 'a vector =
  { x: 'a int;
    y: 'a int
  }
;;
```

```
type (#'a:level) vector = { x: 'a int; y: 'a int }
```

As reflected by the output of the toplevel, the type constructor `vector` has one argument which is the common level of the two integers it is made of. This argument is covariant and guarded. As for tuples, there is no particular security level attached to the record structure itself, since it is not really observable in the language.

So, when a vector is built from two integers, its security level is nothing but the union of those of the given integers:

```
let v = { x = x1; y = x2 };;
val v : [> !alice, !bob] vector
```

One can also define some of the classical functions operating on vectors:

```
let add_vector v1 v2 =
  { x = v1.x + v2.x;
    y = v1.y + v2.y
  }
;;
val add_vector: 'a vector -> 'a vector -> 'a vector
let rot_vector v =
  { x = - v.y;
    y = v.x
  }
;;
val rot_vector: 'a vector -> 'a vector
```

The manner we equip the type `vector` with security annotations is somehow arbitrary. Indeed, it is also possible to distinguish the information carried by each of its components and hence have two security levels:

```
type ('a, 'b) vector2 =
  { x2: 'a int;
    y2: 'b int
  }
;;
type (#'a:level, #'b:level) vector = { x2: 'a int; y2: 'b int }
```

Such a declaration allows in some cases more precise, but also more verbose, typings.

```
let add_vector2 v1 v2 =
  { x2 = v1.x2 + v2.x2;
    y2 = v1.y2 + v2.y2
  }
;;
val add_vector2: ('a, 'b) vector -> ('a, 'b) vector -> ('a, 'b) vector

let rot_vector2 v =
  { x2 = - v.y2;
    y2 = v.x2
  }
;;
val rot_vector2: ('a, 'b) vector2 -> ('b, 'a) vector2
```

In particular, the type obtained for the function `rot_vector2` clearly shows that the function performs some permutation of the two components of the vector.

Mutable records As in Objective Caml, records can also have *mutable* fields whose content may be modified in place. They are declared with the **mutable** keyword:

```
type ('a, 'b) mvector =
  { mutable mx: 'a int; mutable my: 'a int } # 'b
;;
type (='a:level, #'b:level) mvector = {
  mutable mx : 'a int;
  mutable my : 'a int;
} # 'b
```

This defines a type for mutable vectors. This declaration calls for two comments. Firstly, a mutable field is at the same time an input and an output channel of a value: it contents can be written and read. Because it is described by an only type, it must be *invariant*. Hence, in the above definition, the parameter `'a` is *invariant*, as reflected in the signature by the `=`. Secondly, a record involving some mutable field is no longer a simple tuple: the information it carries is not entirely contained by its components because its identity (i.e. its address in memory) can be observed in the language. Hence, its type must carry an additional security level which tells how much information is attached to the knowledge of its identity. In our example, this role is played by the argument `'b`, which is specified by the clause `# 'b` at the end of the definition. To illustrate the use of such a datatype, let us define the function `rot_mvector` which rotates in place a vector:

```
let rot_mvector v =
  let x = v.mx in
  v.mx <- v.my;
  v.my <- x
;;
val rot_mvector : ('a, 'a) mvector -{'a ||}-> unit
```

Because of their update, the integers of the vector given as argument contain information about both the context where the function is applied and the identity of the record given as argument.

In section 2.3, we have introduced *references*. However, they are only a particular case of mutable records and can be defined as follows:

```
type ('a, 'b) ref =
  { mutable contents: 'a } # 'b
;;
type (='a:type, #'b:level) ref = { mutable contents: 'a } # 'b
```

What is more, the three primitives operations `ref`, `:=` and `!` on references are regular functions which can be implemented from the record representation of references:

```
let ref x =
  { contents = x }
;;
val ref : 'a -> ('a, _) ref

let (:=) r x =
  r.contents <- x
;;
val ( := ) : ('a, 'b) ref -> 'a -{'b ||}-> unit
             with 'b < level('a)

let ( ! ) r =
  r.contents
;;
```

```
val ( ! ) : ('a, 'b) ref -> 'c
            with 'b < level('c)
            and  'a < 'c
```

2.6 Interacting with the outside world

A whole Flow Caml program may be viewed as a process that receives information from one or several sources, performs some computation and sends its results to one or several receivers. Then, the final purpose of the Flow Caml type system is to check that every information flow from a source to a receiver generated by the execution is legal w.r.t. the security policy of the system. In this section, we describe how such external entities are modeled in Flow Caml and how the desired security policy may be specified by the programmer.

In the literature, holders of information are generally referred to as *principals* (from the program's viewpoint, each of them can be a source, a receiver, or both). Depending on the context, principals may stand for a variety of concepts: (groups of) human beings, security classes (e.g. *public* or *secret*), subsets of the system's memory, communication channels through some peripheral or network interface. However Flow Caml is not concerned with the real existence of such entities, and provides a general and uniform manner to deal with them: in its type system, principals are represented by constant security levels. In the beginning of this tutorial, *Alice*, *Bob* and *Charlie* were examples of principals and represented by the security levels `!alice`, `!bob` and `!charlie`, respectively. However, they remained relatively abstract, because we just declared a series of values to have these levels—thanks to some type constraint—but we did not say how a program can really interact with them.

2.6.1 The example of the standard input and output

More concrete examples of external communication channels for a program consist in its standard input and output. Both can be viewed as principals and we therefore decide to represent them by the two security levels `!stdin` and `!stdout`, respectively. A program can interact with them using the usual functions of the standard library. For instance, `print_int` outputs an integer on the standard output:

```
print_int;;
- : !stdout int -{!stdout ||}-> unit
```

Because the integer provided as argument is sent to the standard output, its security level must be less than or equal to `!stdout`. To print the integer `1`, one writes:

```
print_int 1;;
- : unit
```

The literal constant `1` has type `'a int` for every `'a`; hence one can instantiate `'a < !stdout` and the call to the function is possible. However, printing the integer `x1` (which comes from the principal Alice and hence has the security level `!alice`) is not, in the default security policy, legal:

```
print_int x1;;
This expression generates the following information flow:
  from !alice to !stdout
which is not legal.
```

Indeed, this piece of code generates a flow from Alice to the standard output and hence requires the inequality `!alice < !stdout`. This is not satisfied in the default security policy which is the *empty* one: it never allows any communication from one principal to another. It can be refined using declarations introduced by the keyword **`flow`**:

```
flow !alice < !stdout;;
```

This makes the security level `!alice` less than or equal to `!stdout`. In other words, this allows information flow from the principal represented by `!alice` (Alice) to that of `!stdout` (the standard output). These declarations are naturally "transitive". For instance, if one declares:

```
flow !bob < !alice;;
```

then Bob is allowed to send information to Alice, but also, by transitivity, to the standard output:

```
print_int x2;;
- : unit
```

It is worth noting that the constant security levels are *global* as well as the declarations that relate them. This is natural because the principals and the security policy they represent are so. However, for convenience, the interactive toplevel allows the programmer to refine the security policy incrementally. This is always safe because a piece of code that is legal in some security policy is still allowed in another one where more information flow is possible.

Similarly, the security level `!stdin` intends to represent the standard input in the type system. For instance, the function `read_line` has the following type:

```
read_line;;
- : unit -{[< !stdout, !stdin] | End_of_file: !stdin |}-> !stdin string
```

Quoting the documentation of the standard library, `read_line` "*flushes standard output, then reads characters from standard input until a newline character is encountered [and] returns the string of all characters read, without the newline character at the end*". Thus, invoking `read_line` affects both the standard input and output, which explains the first annotation in the arrow of its type. Furthermore, if the user sent the "end-of-file" sequence (e.g. by typing `^D`), the function raises the exception `End_of_file`, hence the second annotation on the arrow. Lastly, a string obtained by reading on the standard input must have the level `!stdin`:

```
let s1 = read_line ();;
val s1 : !stdin string
Current evaluation context has level !stdin
```

Let us note that, if one wants to print on the standard output a string read on the standard input, the security policy of the program must allow information flow from the latter to the former. This is declared by the following statement:

```
flow !stdin < !stdout;;
```

and then, it is possible to write a function `echo` which "pipes" the standard input to the standard output:

```
let echo () =
  try
    while true do
      let s = read_line () in
      print_string s
    done
  with
    End_of_file -> ();;
val echo : unit -{[< !stdout, !stdin] ||}-> unit
```

2.6.2 Modeling principals

Real programs are liable to communicate with external entities through other channels than the simple standard input and output, e.g. the file system, network interfaces or display devices. However, the Flow Caml library does not provide functions allowing such communications: analyzing these low-level operations with its type system would not yield any relevant information about their behavior w.r.t. the security policy, because fine-grained considerations are in general mandatory to prove they are safe. Then, the interaction with external entities must be modeled in Flow Caml at a higher level.

That is the reason why a program written and verified with the Flow Caml system must generally be divided in two parts. The purpose of the first one is to provide a high level model of the external principals considered by the program. This should consist in a series of functions for interacting with them, which are implemented in one or several regular Caml modules, using for instance the standard i/o interface, the `Unix` library or some graphical toolkit. This part of the code cannot be verified by the Flow Caml system: the programmer must supply itself an interface for these "high level" functions which specifies their behavior w.r.t. the security policy. The second part consists in the body of the program, which interacts with the outside world only with the model of principals provided by the previous modules. This part can be written and type-checked in Flow Caml, which automates its verification. In section 2.8, we will give more details about this programming scheme: because this requires dividing the program at hand into several *compilation units*. Because they are roughly a particular case of *top-level* modules, we first say a few words about the module layer of the Flow Caml language.

2.7 The module language

The broad outline of the Flow Caml module language is the same as that of Objective Caml: it provides *structures*, which are sequences of definitions, and *signatures* which are interfaces for structures. Besides *functors* are "functions" from structures to structures, which allow expressing parameterized structures.

Although, there is no major novelty compared with Objective Caml, this section describes how its module language has been extended to handle the type system of Flow Caml and its information flow analysis; and illustrates with some examples its basic usage.

2.7.1 Structures and signatures

A *structure* consists in an arbitrary sequence of definitions, which are packaged together. It is introduced by the **struct ... end** construct, and is usually given a name with the **module** binding. For instance, one may define a structure implementing sets of integers (with binary trees):

```
module IntSet = struct

  type 'a t =
      Empty
    | Node of 'a t * 'a int * 'a t
    # 'a

  let empty = Empty

  let rec add x = function
      Empty -> Node (Empty, x, Empty)
    | Node (l, y, r) ->
        if x < y then Node (add x l, y, r)
```

```
          else Node (l, y, add x r)

    let rec mem x = function
        Empty -> false
      | Node (l, y, r) ->
          (x = y) || mem x (if x < y then l else r)

  end;;
  module IntSet : sig
    type (#'a:level) t = Empty | Node of 'a t * 'a int * 'a t # 'a
    val empty : 'a t
    val add : 'a int -> 'a t -> 'a t
    val mem : 'a int -> 'a t -> 'a bool
  end
```

This structure comprises one type definition (the type of sets of integers, `t`), and three values: the empty set, `empty`, and two functions operating on sets, `add` (to add an integer to a set) and `mem` (to test whether an integer belongs to a set). The system outputs the *signature* of the structure, which is a list of its components with their declaration. Outside the structure, its components can be referred to using the "dot notation", that is, identifiers qualified by a structure name. For instance, `IntSet.add` refers to the function `add` of this structure.

```
IntSet.add x1 (IntSet.add x2 IntSet.empty);;
- : [> !alice, !bob] IntSet.t
```

Signatures allows to abstract some characteristics of the implementation of a structure by hiding some components or exporting them with a restricted declaration or type. For instance, one may hide the concrete representation of integer sets:

```
module type INTSET = sig
  type (#'a:level) t
  val empty: 'a t
  val add: 'a int -> 'a t -> 'a t
  val mem: 'a int -> 'a t -> 'a bool
end;;
module AbstractIntSet = (IntSet : INTSET);;
module AbstractIntSet : INTSET
```

It is worth noting that parameters in declarations of abstract types must be annotated with their kind and variance in signatures: because the representation of the type is not given, they can no longer be inferred by the system. However, they are required to properly type-check the rest of the program.

2.7.2 Functors

Functors are "functions" from structures to structures. They are used to express parameterized structures. A common example is a definition of a generic set library, parameterized by a structure giving the type of the elements of the set and a function `compare` defining a total order between them:

```
module type ORDERED_TYPE = sig
  type (#'a:level) t
  val compare : 'a t -> 'a t -> 'a int
end;;
```

(As usual in Caml, `compare x y` is expected to return `0` if `x` is equal to `y`, a negative integer if `x` is less than `y` and a positive integer otherwise.) In this signature, the type of the elements, `t`, is parameterized by one security level which describes all the information leaked by a comparison (as reflected by the type of `compare`). However, this does not prevent to instantiate it with more complex data types, which are originally parameterized by several security levels:

```
module IntList : ORDERED_TYPE = struct
  type 'a t = ('a int, 'a) list
  let rec compare l1 l2 =
    match l1, l2 with
      [], [] -> 0
    | [], _ :: _ -> -1
    | _ :: _, [] -> 1
    | hd1 :: tl1, hd2 :: tl2 ->
      let c = Pervasives.compare hd1 hd2 in
      if c = 0 then compare tl1 tl2
      else c
end;;
module IntList : sig
  type (#'a:level) t
  val compare : 'a t -> 'a t -> 'a int
end
```

We now define the functor implementing sets of arbitrary type. This functor takes the structure `Elt` as argument which must have the signature `ORDERED_TYPE`:

```
module Set (Elt: ORDERED_TYPE) = struct

  type 'a element =
    'a Elt.t

  type 'a t =
      Empty
    | Node of 'a t * 'a element * 'a t
    # 'a

  let empty = Empty

  let rec add x = function
      Empty -> Node (Empty, x, Empty)
    | Node (l, y, r) ->
        if Elt.compare x y < 0 then Node (add x l, y, r)
        else Node (l, y, add x r)

  let rec mem x = function
      Empty -> false
    | Node (l, y, r) ->
        let c = Elt.compare x y in
        (c = 0) || mem x (if c < 0 then l else r)

end;;
module Set : functor (Elt : ORDERED_TYPE) -> sig
  type (#'a:level) element = 'a Elt.t
  type (#'a:level) t = Empty | Node of 'a t * 'a Elt.t * 'a t # 'a
  val empty : 'a t
  val add : 'a Elt.t -> 'a t -> 'a t
  val mem : 'a Elt.t -> 'a t -> 'a bool
end
```

As in the `IntSet` example, it would be good style to hide the actual implementation of the type of sets. This can be achieved by restricting `Set` by a suitable functor signature. Firstly, let us define the type of a module implementing a set structure:

```
module type SET = sig

  type (#'a:level) element
  type (#'a:level) t

  val empty: 'a t
  val add: 'a element -> 'a t -> 'a t
  val mem: 'a element -> 'a t -> 'a bool

end;;
```

This signature lists the type of elements and sets as well as the functions operating on them. Then, the `Set` functor takes a structure of signature `ORDERED_TYPE` and returns one of signature `SET`, so it may be declared with the following type:

```
module Set (Elt: ORDERED_TYPE)
          : (SET with type 'a element = 'a Elt.t) = struct
  ...
end
```

The type constraint `with type 'a element = 'a Elt.t` has the same purpose as in Objective Caml: it points out the fact that the sets contain elements of type `Elt.t`, i.e. that the functions `add` and `mem` can be applied with arguments of this type. To conclude with this example, one can retrieve our first implementation of integer sets, the module `IntSet`, as an instance of the functor `Set`:

```
module IntSet' = Set (struct
  type 'a t = 'a int
  let compare = Pervasives.compare
end);;
module IntSet' : sig
  type 'a element = 'a int
  type 'a t
  val empty : 'a t
  val add : 'a element -> 'a t -> 'a t
  val mem : 'a element -> 'a t -> 'a bool
end
```

In the previous examples, the interaction between the module language and the security analysis featured by the type system of Flow Caml remains relatively basic: roughly speaking, it simply consists in including the relevant security annotations in value and type declarations in signatures. However, it is sometimes necessary to parameterize a signature with some security level. For instance, one may define a module type for a structure implementing some input channel:

```
module type IN = sig
  level Data
  level Prompt
  val read : unit -{[< Prompt] ||}-> Data string
end;;
```

This signature involves two abstract levels: Data is the security level of data read on the input channel; and Prompt represents the information leaked on the channel when one starts listening on it. At the time being, nothing is known about these levels, so they remain "abstract". The function read is intended to read one line on the underlying channel. It naturally produces a string whose level is Data (let us remark that, in this model, reading can never fail). An implementation of this signature using the standard input would be as follows:

```
module Stdin = struct
  level Data = !stdin
  level Prompt less than !stdin, !stdout
  let read () =
    try read_line ()
    with End_of_file -> ‘‘
end;;
module Stdin : sig
  level Data = !stdin
  level Prompt less than !stdout, !stdin
  val read : unit -{[< !stdout, !stdin] ||}-> !stdin string
end
```

Strings read on the standard input have level !stdin, so Data is declared to equal to it in Stdin. Invoking read_line affects the standard input and the standard output (because it is flushed), so Prompt must be less than or equal to !stdin and !stdout. Then, the module Stdin implements the signature IN, which may be immediately verified by a type constraint:

```
module AbstractStdin = (Stdin : IN);;
module AbstractStdin : IN
```

Similarly, we declare the module type for output channels, and implement an instance of it devoted to the standard output:

```
module type OUT = sig
  level Data
  val print : Data string -{Data ||}-> unit
end;;
```

In this case, we only need one security level Data which represents the information which may be sent on the channel. (We do not consider the possibility of *receiving* information from an output channel, for instance because of a buffer overflow.) The module Stdout implements this signature for the standard output:

```
module Stdout = struct
  level Data = !stdout
  let print = print_endline
end;;
module Stdout : sig
  level Data = !stdout
  val print : !stdout string -{!stdout ||}-> unit
end;;
```

Now, we aim at writing a functor which takes as argument two structures, defining an input channel and an output channel, respectively. The body of the functor will define a function `copy` whose purpose is simply to read one line on the input channel and print it on the output channel. However, it is not enough to require the two structures parameterizing the functor to have the respective signature `INPUT` and `OUTPUT`: indeed, the function `copy` implemented by the functor generates an information flow from the channel represented by the former to that of the latter. Hence the security level `Data` of the input channel, must be declared to be less than or equal to that of the output channel.

```
module Copier (I : IN)
              (O : OUT with level Data greater than I.Data) = struct
  let copy () =
    O.print (I.read ())
end;;
```

```
module Copier :
  functor (I : IN) ->
    functor
      (O : sig
             level Data greater than I.Data
             val print : Data string -{Data ||}-> unit
           end) ->
      sig
        val copy : unit -{[< O.Data, I.Prompt] ||}-> unit
      end
```

That is the purpose of the constraint **with level** appearing in the type of the second argument of the function. Its semantics is similar to that of **with type** or **with module** in Objective Caml: it refines the definition of the level `Data` in the signature of the module `O`. The clause **greater than** `I.Data` declares that this security level must be that of a principal allowed to "receive" information *from* the channel implemented by the structure `I` whose level is `I.Data`.

It is worth noting that the order in which parameters appear in the functor definition generates some asymmetry, because the constraint is applied on the second structure. Obviously, it is also possible to permute the two arguments:

```
module Copier' (O : OUT)
               (I : IN with level Data less than O.Data) = struct
  let copy () =
    O.print (I.read ())
end;;
```

```
module Copier' :
  functor (O : OUT) ->
    functor
      (I : sig
             level Data less than O.Data
             level Prompt
             val read : unit -{Prompt ||}-> Data string
           end) ->
      sig
        val copy : unit -{[< I.Prompt, O.Data] ||}-> unit
      end
```

The constraint now states that the principal represented by the level `source` in `I` must be allowed to send information to `O.Data`.

To conclude with this example, let us build an instance of `Copier` dedicated to the standard input and output. This requires to allow information flow from the former to the latter, which can be done by the toplevel declaration:

```
flow !stdin < !stdout;;
```

Now, the security level `Stdin.Data` is less than or equal to `Stdout.Data`, so `Stdin` and `Stdout` are legal arguments for `Copier`:

```
module StdCopier = Copier (Stdin) (Stdout);;
module StdCopier :
  sig val copy : unit -{[< Stdout.Data, Stdin.Prompt] ||}-> unit
end
```

2.7.3 Side-effects, exceptions and the module language

Let us briefly explain how the type system of Flow Caml traces information flow due to side-effects and exceptions throughout the evaluation of module expressions. Evaluating a structure consists in considering successively each of its definitions; computation really arises only at **let** definitions and their evaluation may have side-effects or raise exceptions. It is worth noting that an exception which escapes the scope of a top-level **let** definition cannot be trapped further, so it terminates the program.

In fact, Flow Caml's type system associates not only a type to the body of a **let** definition but also two lists of security levels, or *bounds* written `from ... to ...` where each `...` stands for a list of security levels. The *lower bound* (appearing after the `from` keyword) describes the side-effects performed by the evaluation of the definition, roughly speaking it includes the security level(s) of data structures the definition may affect. The *upper bound* (appearing after the **to** keyword) tells how much information is attached to the exceptions the definition may raise. This process is generalized to the whole module language by associating to every definition and module expression a pair of bounds. Because they have no computational content, the bounds of **external**, **type**, **level**, **exception**, **module type**, **open** and **include** definitions are always empty. The bounds of a **module** definition are obtained by considering recursively the module expression which appears in the right-hand-side.

For instance, the evaluation of a structure **struct** $\texttt{def}_1 \ldots \texttt{def}_n$ **end** consists in evaluating successively each of the definitions, then the bounds associated to the whole module expression are naturally obtained by merging those of the definitions $\texttt{def}_1$ to $\texttt{def}_n$. Moreover, while considering this sequence of definitions, $\texttt{def}_i$ is evaluated if and only if none of $\texttt{def}_1$ to $\texttt{def}_{i-1}$ raised an exception. As a result, to prevent any illegal information flow, the upper bounds of the former must be less than or equal to the lower bound of the latter.

However, the toplevel system does not output the bounds of the definitions the user enters. Instead, at every prompt, it considers all definitions entered so far as members of a single structure, whose upper bound corresponds to the evaluation context's security level introduced in section 2.4.2. When one enters a new definition, the system checks that its lower bound is less than or equal to that of the current evaluation context's security level, and then it is extended with the upper bound of the new definition. A message giving the new evaluation context's security level is output in the case it is different from the previous one.

A functor definition does not perform any computation, so it has empty bounds. Indeed, the body of the functor is executed only when the functor is applied. In order to trace bounds from functors definitions to functors applications, the arrow symbols of functors types are annotated—when necessary—by bounds:

```
module type S = sig
  val x : 'a int
end;;
```

```
module F (X: S) = struct
  let _ =
    r1 := X.x;
    if X.x = x2 then raise Exit
end;;
module F : functor (X : S) -{!alice | !bob}-> sig  end
```

This module type means that any application of the functor `F` may generate a side-effect on cells of level `!alice` and may raise an exception at level `!bob`. Then, an application of `F` inserts `!bob` in the evaluation context's security level:

```
module F0 = F (struct let x = 1 end);;
Current evaluation context has level !bob
```

Moreover, it is only possible to instantiate `F` if the evaluation context's security level is less than or equal to `!alice`, which is no longer the case after a first application of `F`:

```
module F1 = F (struct let x = 1 end);;
This expression is executed in a context of level !bob
but has an effect at level !alice.
This yields the following information flow:
  from !bob to !alice
which is not legal.
```

2.8 Standalone programs

All examples given so far were entered under the interactive system. However, Flow Caml code can also be written in separate files: the batch "compiler" `flowcamlc` allows to type-check them, and also translates them into regular Objective Caml source code files, so that they can be compiled using the compilers `ocamlc` or `ocamlopt`, yielding a standard executable.

In this section, we aim at explaining how it is possible to write such programs in Flow Caml. To illustrate our discussion, we consider as example the complete program whose source code is given pages 59 to 60. Let us briefly introduce the operation this example program performs: on `Unix` systems with *shadow* passwords, information about user accounts is stored in two files. The file `/etc/passwd` registers the list of logins, with, for each of them, a password and some administrative information such as a numeric id, the user's home directory and shell. Besides, the file `/etc/shadow` associates to every login a password stored in a encrypted form (with some optional aging information), which is used in place of that in `/etc/passwd`. Our program aims at synchronizing these two files, i.e. generating an entry in `/etc/shadow` for every account which is listed only in `/etc/passwd`. In the forthcoming subsections, we will explain step by step how the source code is organized, how it is verified and compiled thanks to the Flow Caml system. The type system will allow us to check that running the program cannot reveal to the user which invokes the command any information about the passwords stored in the two files.

2.8.1 Compilation units and batch compilation

The source code of a program is generally split into several files, called *compilation units*, that can be compiled separately. In Flow Caml, a compilation unit `A` comprises one or two files, among:

- the implementation file *a*`.fml`, which contains a sequence of definitions, analogous to the inside of a **`struct`**...**`end`** construct;

- the interface file *a*.fmli, which contains a sequence of specifications, analogous to the inside of a **sig**...**end** construct.

(In addition to definitions and specifications, the implementation and the interface may include two particular "*headers*", made of a **flow** declaration and optional **affects** and **raises** statements, whose respective purposes will be explained in the sections 2.8.2 and 2.8.3.) Both files define a structure named A (same name as the base name of the two files, with the first letter capitalized), as if the following definition was entered at top-level:

```
module A : sig (* specifications of file a.fmli *) end
         = struct (* definitions of file a.fml *) end;;
```

The files defining a set of compilation units can be handled separately using flowcamlc, following for each unit A one of the three above schemes:

1. The compilation procedure of a unit A defined in files *a*.fmli and *a*.fml is described in figure 2.1. First, the Flow Caml interface *a*.fmli is fed to flowcamlc, which checks its well-formedness and produces a compiled version of it, *a*.fcmi. It also translates the interface file into a regular Objective Caml one, namely *a*.mli. Second, the implementation *a*.fml can be type-checked by flowcamlc. The compiler computes the most general interface for the implementation, and checks it fulfills the declared one (i.e. that stored in *a*.fcmi). Furthermore, the source code of the unit in *a*.fml is translated into a Objective Caml implementation file, *a*.ml. Then, *a*.mli and *a*.ml can be compiled with ocamlc to produce a compiled interface *a*.cmi and a bytecode object file *a*.cmo.

2. However, as in Objective Caml, it is possible to build a unit A by providing only an implementation file *a*.fml but no interface file. This yields the compilation scheme of figure 2.2: the implementation *a*.fml can be directly passed through flowcamlc and the interface computed by type inference is stored itself in *a*.fcmi.

3. Lastly, as we have explained in section 2.6.2, some pieces of code cannot be satisfyingly typed with Flow Caml's system. Such definitions may be provided by a compilation unit with a Flow Caml interface *a*.fmli but only an Objective Caml implementation *a*.ml, as illustrated by the compilation scheme of figure 2.3. Then, the system will not check that the code in *a*.ml fulfills the interface *a*.fmli w.r.t. the security policy—this is left to the programmer's responsibility—but the definitions exported by the unit (and registered in *a*.fcmi) will be available to the rest of the program.

Our example program is made of four compilation units. Passwd and Shadow are low-level modules which implement functions for accessing the /etc/passwd and /etc/shadows files: their implementations are directly written in Objective Caml (files passwd.ml and shadow.ml), and only interfaces are provided in Flow Caml (files passwd.fmli and shadow.fmli). These interfaces assign security levels to the information manipulated by the units: data stored in /etc/passwd has the level !passwd_file, except the passwords, which have level !password. Similarly, information from /etc/shadow receives the level !shadow_file and !shadow_password. The unit Verbose provides a verbose mode: if the user runs the program with the -v option, then the execution is traced on the standard output. The body of the program is in Main. These last two units are fully implemented in Flow Caml: implementation (verbose.fml and main.fml) and interface (verbose.fmli and main.fmli) files are provided for each of them.

2.8.2 flow declarations in implementations and interfaces

In section 2.6.1, we explained how **flow** declarations allow specifying the security policy by setting inequalities between principals. We have seen that the toplevel system allows the programmer

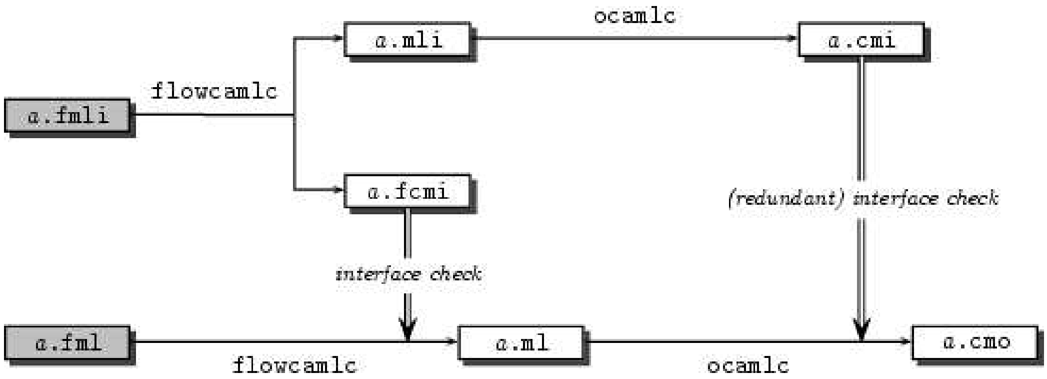


Figure 2.1: Compilation scheme of a unit defined in a.fmli and a.fml

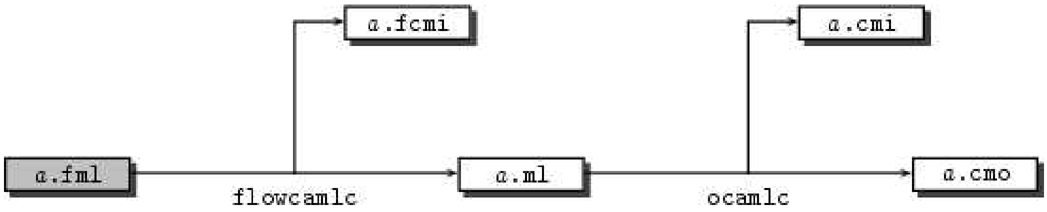


Figure 2.2: Compilation scheme of a unit defined in a.fml

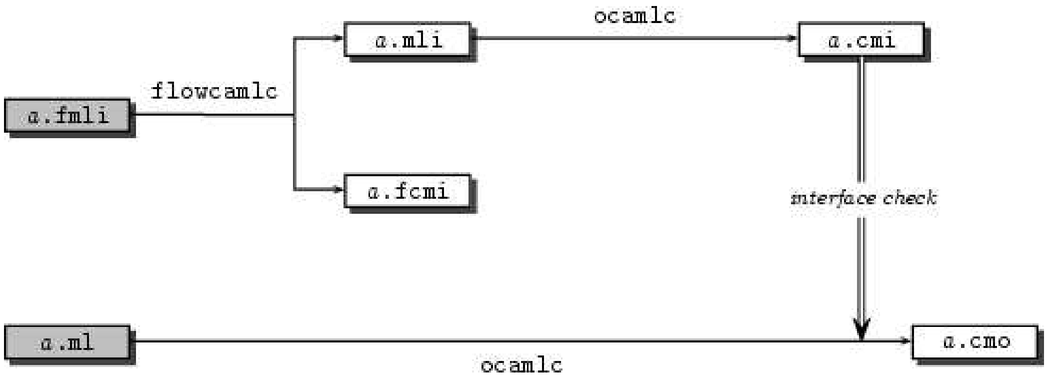


Figure 2.3: Compilation scheme of a unit defined in a.fmli and a.ml

to refine the security policy incrementally, by entering new **flow** declarations which remain valid until the end of the interactive session.

In programs written for the batch compiler `flowcamlc`, every compilation unit must come with its own security policy, i.e. a **flow** declaration which specifies sufficient assumptions on principals for its source code to be well-typed. This declaration must be provided at the beginning of the implementation and interface files. For instance, the compilation unit `Verbose` of our example begins with the following declaration:

```
flow !arg < !stderr, !stdout
```

By convention, the principals `!stderr` and `!stdout` represent the standard error and the standard output of the program, respectively. `!arg` is the security level of the command-line arguments. The declaration is a shorthand for

```
flow !arg < !stderr
and  !arg < !stdout
```

It means that the unit is well-typed under every security policy which enforces at least the two inequalities `!arg < !stdout` and `!arg < !stderr`. When a compilation unit includes no **flow** declaration—as `Passwd` and `Shadow` in the example—this simply means it is well-typed in every security policy.

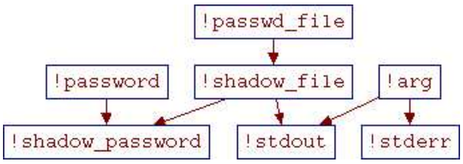
The security policy under which a program made of several compilation units can be considered to be verified is the *union* of those declared in the units, i.e. the least one which satisfies the assumptions made by every unit. This is safe because if a piece of code is well-typed in a given policy, it remains so under a refinement of it, i.e. a lattice which declares more information flow. However, the possibility to provide a different **flow** declaration in every compilation unit of a program is of main importance for modularity of programming and re-usability of code, in the context of separate compilation. Indeed, this allows for instance having libraries (such as the standard one) used in programs which have different security policies. Otherwise, one would have to write or compile a specialized version of these libraries for each program which expects a different policy.

The Flow Caml system provides a tool, `flowcamlpol`, to compute the (minimal) security policy under which a program is (checked to be) safe. The usage of `flowcamlpol` is—to some extent—similar to that of a *linker* of object files: it expects as argument the name of the compiled interfaces of the program's units, in the same order as the corresponding object files will be linked. For our example, one must run the command:

```
flowcamlpol passwd.fcmi shadow.fcmi debug.fcmi main.fcmi
```

which leads the system to sum up all information flow the example program is liable to perform:

```
!shadow_file < !shadow_password
!shadow_file < !stdout
!passwd_file < !shadow_file
!password < !shadow_password
!arg < !stderr
!arg < !stdout
```



The graphical representation can be obtained if the command is run with the `-graph` option. It shows some interesting properties of the information flow graph of the program, which have been established automatically by the type system. For instance, the standard output is not related to sensitive data stored in the `/etc/passwd` and `/etc/shadow` files, i.e. the user passwords.

Lastly, you may wonder why the minimal assumptions necessary for a compilation unit are not inferred by the system while it type-checks the source code. Indeed, doing so is possible for expressions of the core language. For instance, the following piece of code

```
print_string Sys.argv.(0);;
```

is well-typed if and only if the inequality !arg < !stdout is enforced. However, the existence of a “minimal” solution to this problem is no longer ensured when considering the module language. Indeed, typing module expressions requires *comparing* type schemes, i.e. verifying that a given scheme is an instance of another one, which cannot yield principal **flow** declarations. For instance, the comparison of [< !alice, !bob] int with !charlie int requires the least upper bound of the levels !alice and !bob to be less than or equal to !charlie, which is not expressible in a **flow** statement.

2.8.3 affects and raises statements in interfaces

The execution of a program made of one or several compilation units consists in evaluating successively each top-level structure, in the order specified at linking. For instance, our example program is linked with the following command:

```
ocamlc -o passwd2shadow passwd.cmo shadow.cmo verbose.cmo main.cmo
```

then, when running the compiled executable passwd2shadow, the definitions of Passwd, Shadow, Verbose and Main are successively evaluated, until an uncaught exception is raised or the end is reached. Then, when gaining control, each unit acquires the information that the previous ones normally terminated. So, in order to trace this possible information flow, one has to consider the *bounds* of the underlying structures, as if the program where defined in a single unit such as:

```
module Passwd = struct
  ...
end

module Shadow = struct
  ...
end

module Verbose = struct
  ...
end

module Main = struct
  ...
end
```

For this purpose, every interface file can mention the lower and upper *bounds* of the underlying structure thanks to the **affects** and **raises** statements, respectively , which appears between the **flow** declaration and the signature (as usual, omitted bounds are supposed to be empty). For instance, the interface of Verbose declares the following bounds

```
affects !arg
raises !arg
```

which means that the unit Verbose has side-effect of level !arg and can raise an exception at this level. This statement appears only in the interface: when type-checking the implementation, the bounds are inferred from the source code and compared to those provided in the interface.

Then, when linking a series of compilation units Unit$_1$, …, Unit$_n$ to produce an executable, one has to check that, for every i, the upper bounds of Unit$_1$ to Unit$_{i-1}$ are less than or equal to the lower bound of Unit$_i$, which is in fact achieved by the flowcamlpol command, at the same time it computes the security policy:

```
flowcamlpol passwd.fcmi shadow.fcmi debug.fcmi main.fcmi
```

A complete example

passwd.fmli

```
(* An entry of "/etc/passwd" is represented by a record of
   type [(!passwd_file, !password) entry] *)
type (’a, ’b) entry =
    { login: ’a string;
      password: ’b string
    }

(* Input from "/etc/passwd" *)
type noneq in_channel
val open_in: unit -{!passwd_file ||}-> in_channel
val input_entry: in_channel
   -{!passwd_file | End_of_file: !passwd_file |}->
   (!passwd_file, !password) entry
val close_in: in_channel -{!passwd_file ||}-> unit
```

passwd.ml

```
type entry =
    { login: string;
      password: string
    }

type in_channel = Pervasives.in_channel

let open_in () =
  Pervasives.open_in "/etc/passwd"

let rec input_entry chan =
  let line = input_line chan in
  try
    let i1 = String.index line ’:’ in
    let i2 = String.index_from line (i1 + 1) ’:’ in
    { login = String.sub line 0 i1;
      password = String.sub line (i1 + 1) (i2 - i1 - 1)
    }
  with
    Not_found -> input_entry chan

let close_in chan =
  Pervasives.close_in chan
```

shadow.fmli

```
(* An entry of "/etc/shadow" is represented by a record of
   type [(!shadow_file, !shadow_password) entry] *)
type (’a, ’b) entry =
    { login: ’a string;
      password: ’b string;
      rem: ’b string;
    }

(* Input from "/etc/shadow" *)
type noneq in_channel
val open_in: unit -{!shadow_file ||}-> in_channel
val input_entry: in_channel
   -{!shadow_file | End_of_file: !shadow_file |}->
   (!shadow_file, !shadow_password) entry
val close_in: in_channel -{!shadow_file ||}-> unit

(* Output to "/etc/shadow" *)
type noneq out_channel
val open_out: unit -{!shadow_file ||}-> out_channel
val output_entry:
   out_channel -> (!shadow_file, !shadow_password) entry
   -{!shadow_file ||}-> unit
val close_out: out_channel -{!shadow_file ||}-> unit
```

shadow.ml

```
type entry =
    { login: string;
      password: string;
      rem: string;
    }

type in_channel = Pervasives.in_channel

let open_in () =
  Pervasives.open_in "/etc/shadow"

let rec input_entry chan =
  try
    let line = input_line chan in
    let i1 = String.index line ’:’ in
    let i2 = String.index_from line (i1 + 1) ’:’ in
    let ln = String.length line in
    { login = String.sub line 0 i1;
      password = String.sub line (i1 + 1) (i2 - i1 - 1);
      rem = String.sub line (i2 + 1) (ln - i2 - 1)
    }
  with
    Not_found -> input_entry chan

let close_in chan =
  Pervasives.close_in chan

type out_channel = Pervasives.out_channel

let open_out () =
  Pervasives.open_out "/etc/shadow"

let output_entry chan e =
  Printf.fprintf chan "%s:%s:%s\n" e.login e.password e.rem

let close_out chan =
  Pervasives.close_out chan
```

verbose.fmli

```
flow !arg < !stderr
and  !arg < !stdout


affects !arg
raises !arg


val message : !stdout string -{!stdout ||}-> unit
```

verbose.fml

```
flow !arg < !stderr, !stdout

(** [!verbose_mode] is true if the verbose mode is
    active. *)
let verbose_mode : (!arg bool, _) ref = ref false

(** Parse command-line arguments.  If the option "-v" if
    found then [verbose_mode] is set to true.  If any other
    option is encountered then an error message is printed
    and the exception [Exit] is raised. *)
```

```
let _ =
  for i = 1 to Array.length Sys.argv - 1 do
    match Sys.argv.(i) with
      "-v" -> verbose_mode := true
    | option ->
        prerr_string "Invalid option ";
        prerr_endline option;
        raise Exit
  done

(** [print message] print a message on the standard output
    if the verbose mode is enabled. Otherwise, it does
    nothing. *)
let message s =
  if !verbose_mode then print_endline s
```

main.fml

```
flow !passwd_file < !shadow_file
and  !passwd_file, !shadow_file < !stdout
and  !passwd_file, !shadow_file < !shadow_password
and  !password < !shadow_password

(** The module [StringMap] implements association tables
    indexed by strings. *)
module StringMap = Map.Make (struct
  type 'a t = 'a string
  let compare = Pervasives.compare
end)



(** [read_shadow ()] reads the content of /etc/passwd
    and returns a map associating each login to its
    entry. *)
let read_shadow () =

  let in_chan = Shadow.open_in () in

  let rec loop accu =
    try
      let entry = Shadow.input_entry in_chan in
      loop (StringMap.add entry.Shadow.login entry accu)
    with End_of_file ->
      Shadow.close_in in_chan;
      accu
  in
  loop StringMap.empty


(** [read_passwd shadow_map] generates /etc/shadow from
    /etc/passwd and the entries in [shadow_map] *)
let read_passwd shadow_map =

  let in_chan = Passwd.open_in ()
  and out_chan = Shadow.open_out () in

  let rec loop () =

    try

      let passwd_entry = Passwd.input_entry in_chan in
      Verbose.message passwd_entry.Passwd.login;

      let shadow_entry =
        try
          StringMap.find passwd_entry.Passwd.login shadow_map
        with
          Not_found ->
            Verbose.message "  creating an entry";
            { Shadow.login = passwd_entry.Passwd.login;
              Shadow.password = passwd_entry.Passwd.password;
              Shadow.rem = ""
            }
      in

      Shadow.output_entry out_chan shadow_entry

    with

      End_of_file -> ()

  in

  loop ();

  Passwd.close_in in_chan;
  Shadow.close_out out_chan


let _ =
  let shadow_map = read_shadow () in
  read_passwd shadow_map
```

PART II

Reference manual

CHAPTER 3

The Flow Caml language

This chapter is a brief description of the Flow Caml language. It lists the language constructs, and gives their syntax and semantics. However, this description is only *informal*, without the attempt to provide a mathematical formalization of the language. For a more detailed description of the core language and its type system, the reader is refereed to [PS03]. Moreover, many aspects of the Flow Caml language are directly derived from Objective Caml, so they are described only succinctly, the reader is refereed to the Objective Caml's documentation [LDG+02a] for further details.

Notations The syntax of the language is given in BNF-like notation. Terminal symbols are set in typewriter font (`like this`). Non-terminal symbols are set in italic font (*like that*). Square brackets [...] denote optional components. Curly brackets {...} denotes zero, one or several repetitions of the enclosed components. Curly bracket with a trailing plus sign $\{...\}^+$ denote one or several repetitions of the enclosed components. Parentheses (...) denote grouping.

3.1 Lexical conventions

Flow Caml adopts the same lexical conventions as Objective Caml.

$$\textit{ident} ::= (\textit{letter} \mid \texttt{\_})\ \{\textit{letter} \mid \texttt{0...9} \mid \texttt{\_} \mid \texttt{'}\}$$

$$\textit{letter} ::= \texttt{A...Z} \mid \texttt{a...z}$$

$$\textit{infix-symbol} ::= (\texttt{=} \mid \texttt{<} \mid \texttt{>} \mid \texttt{@} \mid \texttt{\^{}} \mid \texttt{|} \mid \texttt{\&} \mid + \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\$} \mid \texttt{\%})\ \{\textit{operator-char}\}$$

$$\textit{prefix-symbol} ::= (\texttt{!} \mid \texttt{?} \mid \texttt{\~{}})\ \{\textit{operator-char}\}$$

$$\textit{operator-char} ::= \texttt{!} \mid \texttt{\$} \mid \texttt{\%} \mid \texttt{\&} \mid + \mid \texttt{-} \mid \texttt{.} \mid \texttt{/} \mid \texttt{:}\ \texttt{<} \mid \texttt{=} \mid \texttt{>} \mid \texttt{?} \mid \texttt{@} \mid \texttt{\^{}} \mid \texttt{|} \mid \texttt{\~{}}$$

Flow Caml integer, float and character literals are identical to those of Objective Caml. There are two flavors of string literals in Flow Caml: mutable ones (*charray-literal*), introduced between backquotes (`‘`) and immutable ones (*string-literal*), between double quotes (`"`). It is worth noting that escape sequences are the same for both. In particular, `\‘` is not a valid escape sequence, even between `‘`, so one has to write `\096`.

$$\begin{aligned}
\textit{integer-literal} ::=\ & [\texttt{-}]\ \{\texttt{0...9}\}^+ \\
\mid\ & [\texttt{-}]\ (\texttt{0x} \mid \texttt{0X})\ \{\texttt{0...9} \mid \texttt{A...F} \mid \texttt{a...f}\}^+ \\
\mid\ & [\texttt{-}]\ (\texttt{0o} \mid \texttt{0O})\ \{\texttt{0...7}\}^+ \\
\mid\ & [\texttt{-}]\ (\texttt{0b} \mid \texttt{0B})\ \{\texttt{0...1}\}^+
\end{aligned}$$

float-literal ::= [-] {0...9}$^+$ [. {0...9}] [(e | E) [+ | -] {0...9}$^+$]

char-literal ::= ’ *regular-char* ’
| ’ *escape-sequence* ’

escape-sequence ::= \ (\ | " | ’ | n | t | b | t)
| \ (0...9) (0...9) (0...9)

string-literal ::= " {*string-character*} "

charray-literal ::= ‘ {*charray-character*} ‘

string-character ::= *regular-char-string*
| *escape-sequence*

charray-character ::= *regular-char-charray*
| *escape-sequence*

The identifiers below are reserved as keywords, and cannot be employed otherwise. The first groups include all Objective Caml keywords while the second group lists Flow Caml additional ones.

```
and         as          assert      asr         begin       class
closed      constraint  do          done        downto      else
end         exception   external    false       for         fun
function    functor     if          in          include     inherit
land        lazy        let         lor         lsl         lsr
lxor        match       method      mod         module      mutable
new         of          open        or          parser      private
rec         sig         struct      then        to          true
try         type        val         virtual     when        while
with

affects     content     finally     flow        greater     less
level       noneq       propagate   raise       raises      row
than
```

The following character sequences are also keywords:

```
#     &     ’     (     )     *     ,     ->    ?
??    .     ..    .(    .[    :     ::    :=    ;
;;    <-    =     [     [|    [<    {<    ]     |]
>]    >}    _     ‘     {     |     }     ~

-{    }->   ={    }=>
```

3.2 The core language

3.2.1 Values

Values of Flow Caml are those of Objective Caml: integer numbers, floating-point numbers, characters, character strings, tuples, arrays, variant values and functions. (Polymorphic variants and objects are not supported.)

3.2.2 Names

Naming objects Identifiers are used to give names to several classes of language objects and refer to these objects by name later:

- Value names: *value-name* ::= *lowercase-ident* | `(` *operator-name* `)`
operator-name ::= *prefix-symbol* | *infix-symbol*
| `*` | `=` | `or` | `&` | `:=`
- Value constructors: *constr-name* ::= *capitalized-ident* | `false` | `true` | `[]` | `()` | `::`
- Record fields: *field-name* ::= *lowercase-ident*
- Type constructors: *typeconstr-name* ::= *lowercase-ident*
- Level names: *level-name* ::= *uppercase-ident*
- Principals: *principal* ::= `!` *lowercase-ident*
- Exception names: *exception-name* ::= *capitalized-ident*
- Module name: *module-name* ::= *capitalized-ident*
- Module type names: *modtype-name* ::= *ident*

These nine name-spaces are distinguished both by the context and by the capitalization of the identifier. In comparison with Objective Caml, we have three additional classes: level names and principals (see section 3.2.3), and exception names. Exception names are syntactically distinguished from value constructors because exceptions are not first class values in Flow Caml.

Referring to named objects A named object can be referred to either by its name (following the usual static scoping rules for names) or by an access path *prefix*`.`*name*, where *prefix* designates a module and *name* is the name of an object defined in that module. The first component of the path, *prefix* is either a simple module name or an access path $name_1$`.`$name_2$..., in case the defining module is itself nested inside other modules. For referring to type constructors, levels, exceptions (in type expressions) or module types, the *prefix* can also contain simple functor applications (as in the syntactic class *ext-module-path*), in case the defining module is the result of a functor application.

value-path ::= *value-name* | *module-path* `.` *lowercase-ident*

constr ::= *constr-name* | *module-path* `.` *capitalized-ident*

typeconstr ::= *typeconstr-name* | *extended-module-path* `.` *lowercase-ident*

level ::= *level-name* | *extended-module-path* `.` *capitalized-ident*

exception ::= *exception-name* | *extended-module-path* `.` *capitalized-ident*

field ::= *field-name* | *module-path* `.` *lowercase-ident* *module-path* ::= *module-name* | *module-path* `.` *capitalized-ident*

ext-module-path ::= *module-name*
| *ext-module-path* `.` *capitalized-ident*
| *ext-module-path* `(` *ext-module-path* `)`

modtype-path ::= *modtype-name* | *module-path* `.` *ident*

Principal names need not be qualified because they are global labels.

3.2.3 Security levels

Flow Caml types are annotated by *security levels*, which are supposed to belong to a lattice (the corresponding partial order is denoted by `<`, although it is not strict). The lattice must include principals (*principal*) and levels introduced by `level` declarations (*level*), which form "*(constant) security levels*":

security-level ::= *principal* | *level*

However, in order to preserve the lattice structure, it may also contains other security levels, which are not representable in Flow Caml syntax: they never will appear in type expressions; however, assuming their existence is necessary to interpret type schemes. Principals are intended to represent external entities or channels which programs may interact with.

The partial order on security levels is the smallest order which satisfies:

- The inequalities between principals provided in `flow` declarations (see section 3.3.3),
- and the assumptions made in level definitions (see section 3.2.4).

Information flow is allowed from a source labeled by the security level *security-level*$_1$ to a sink labeled by *security-level*$_2$ if and only if *security-level*$_1$ `<` *security-level*$_2$ holds.

3.2.4 Level definitions

Level names can be bound to security levels of the lattice thanks to `level` definitions.

level-definition ::= `level` *level-name* *level-repr*

level-repr ::= [`greater than` *security-level-list*] [`less than` *security-level-list*]
| `=` *security-level*

security-level-list ::= *security-level* {`,` *security-level*}

A level definition is introduced by the `level` keyword. It consists in a capitalized identifier followed by two optional sets of assumptions or *bounds*. The identifier is the name of the level being defined. The assumptions relate this new level with existing ones:

- If a clause `greater than` *security-level*$_1$ `,` ... `,` *security-level*$_n$ is provided, then the new level is made greater than or equal to *security-level*$_i$, for all i.
- If a clause `less than` *security-level*$_1$ `,` ... `,` *security-level*$_n$ is provided, then the new level is made less than or equal to *security-level*$_i$, for all i.

The representation `=` *security-level* is a shorthand for `greater than` *security-level* `less than` *security-level*. The definition cannot introduce new relationships about levels listed in these assumptions: every level which appears in the former must be known to be less than or equal to each of those appearing in the latter.

3.2.5 Type expressions

Type expressions denote types, security levels and rows in types schemes, definitions of data-types as well as in type constraints over patterns and expressions. A type expression is of one of the three kinds: *level*, *type* or *row* (this last kind is parametrized with a (possibly empty) set of exceptions which is the complementary of its domain).

kind ::= `level` | `type` | `row [` [*exception* {`,` *exception*}] `]`

A *row* of kind `row [`*exception*$_1$`,` ...`,` *exception*$_n$`]` is a mapping from exceptions distinct of *exception*$_1$, ..., *exception*$_n$ to security levels.

typexpr ::= `'` *ident*
| `(` *typeexpr* `)`
| *typexpr* `-{`*typexpr* | *typexpr* | *typexpr* `}->` *typexpr*
| *typexpr* {`*` *typexpr*}$^+$
| *typeconstr*
| *typeexpr* *typeconstr*
| `(` *typexpr* {`,` *typexpr*} `)` *typeconstr*
| *exception* `:` *typexpr* `;` *typexpr*

Type expressions involve **type variables** `'`*ident*. A type variable can be used with any kind (i.e. there is a not a distinguished name-space for every kind of variables); however in a given context (i.e. a type scheme, a data-type definition or a type constraint), every occurrence of a given type variable must be used with the same kind. In type definitions, type variables are names for the type parameters. In type schemes, they are implicitly universally quantified.

A **function type** comprises 5 type expressions: *typexpr*$_1$ `-{`*typexpr*$_2$ | *typexpr*$_3$ | *typexpr*$_4$`}->` *typexpr*$_5$. *typexpr*$_1$ and *typexpr*$_5$ have the kind `type`; the former is the type of the argument of the function and the latter that of its result. *typexpr*$_2$ and *typexpr*$_4$ are levels (of kind `level`). The former is the security level of the context (generally written *pc* in the literature) where the function is called while the latter is the annotation attached to the function's identity. Lastly, *typexpr*$_3$ is the row (of kind `row []`) describing the exceptions raised by the function.

Tuples types *typexpr*$_1$ `*` ... `*` *typexpr*$_n$ is the type of tuples whose elements belong to types *typexpr*$_1$, ..., *typexpr*$_n$ respectively.

Constructed types consists either of a type constructor with no parameter (*typeconstr*) or with one (*typeexpr* *typeconstr*) or several (`(` *typeexpr* {`,` *typeexpr*} `)`) arguments. Every type constructor is supposed to have a *signature* which gives the number of parameters it expects and their respective kind, see section 3.2.7.

The **row term** *exception* `:` *typexpr*$_1$ `;` *typexpr*$_2$ stands for the row whose entry at index *exception* is *typexpr*$_1$ and whose other entries are given by the row *typexpr*$_2$.

3.2.6 Type schemes

Type schemes intend to describe the set of admissible types for some expression or value. Because of the presence of subtyping, this set cannot be represented by a single type expression with (universally quantified) free variables but must involve *constraints*.

type-scheme ::= *typexpr* [`with` *constraint* {`and` *constraint*}]

constraint ::= *left-hand-side* {`,` *left-hand-side*} `<` *right-hand-side* {`,` *right-hand-side*}
| *typexpr* {`~` *typexpr*}$^+$

left-hand-side ::= *typexpr* | `content` `(`*typexpr*`)`

right-hand-side ::= *typexpr* | `level` `(`*typexpr*`)`

A type scheme consists in a type expression (the body) and an optional list of constraints, stating assumptions between the variables appearing in the body. Type variables which appear in a type scheme are implicitly universally quantified: intuitively, the type scheme stands for the set of all instances of the body which satisfy then constraints. A constraint is either an *inequality*, i.e. a pair of a left-hand-side and a right-hand-side, separated by the symbol `<`, or a *same-skeleton* constraint, i.e. a `~`-separated list of type expressions.

Some abbreviations are allowed in type expressions which appear as the body of a type scheme:

- A simple constant level *security-level* stands for a fresh level variable `'a` with the two constraints *security-level* `< 'a` and `'a <` *security-level*.

- `[<` *security-level*$_1$ `,` ... `,` *security-level*$_n$`]` is a shorthand for a fresh level variable `'a` with the constraint `'a <` *security-level*$_1$ `,` ... `,` *security-level*$_n$. Similarly, `[>` *security-level*$_1$ `,` ... `,` *security-level*$_n$`]` is a shorthand for a fresh level variable `'a` with the constraint *security-level*$_1$ `,` ... `,` *security-level*$_n$ `< 'a`. Lastly, `[<` *security-level*$_1$ `,` ... `,` *security-level*$_n$ `|>` *security-level*$_{n+1}$ `,` ... `,` *security-level*$_{n+k}$`]` is a shorthand for a fresh level variable `'a` with the constraints *security-level*$_1$ `,` ... `,` *security-level*$_n$ `< 'a` and `'a <` *security-level*$_{n+1}$ `,` ... `,` *security-level*$_{n+k}$.

- `_` stands for an anonymous fresh variable of any kind.

- On functions arrows, anonymous variables can be omitted. For instance, `-{|` *typexpr*$_1$ | *typexpr*$_2$ `}->` is a shorthand for `-{ 'a |` *typexpr*$_1$ | *typexpr*$_2$ `}->` where `'a` is a fresh variable. Furthermore, an arrow whose three annotations are omitted, `-{ | | | }->`, can be written `->`.

We do not give here a precise description of the interpretation of type schemes and constraints. The reader is referred to section 2.2 for an informal presentation and to [PS03] for a formal definition of the type system.

3.2.7 Type definitions

Type definitions bind type constructors to data types: either variant types, record types, type abbreviations, or abstract data types. They also bind the value constructors and record fields associated with the definition.

type-definition ::= `type` *typedef* {`and` *typedef*}

typedef ::= [`noneq`] [*type-params*] *typeconstr-name* [`=` *typexpr*] [*type-repr*]

type-params ::= *type-param* | `(` *type-param* {`,` *type-param*} `)`

type-param ::= [`+` | `-` | `=` | `#`] `'` *ident* [`:` `(``level` | `type` `row` `[` [*exception-list*] `]` `)`]

type-repr ::= *constr-decl* {| *constr-decl*} [`#` `'`*ident*]
| `{` *field-decl* {`;` *field-decl*} `}` [`#` `'`*ident*]

constr-decl ::= *constr-name* | *constr-name* `of` *typexpr*

field-decl ::= [`mutable`] *field-name* `:` *typexpr*

Type definitions are introduced by the `type` keyword, and consist in one or several simple definitions, possibly mutually recursive, separated by the `and` keyword. Each simple definition defines one type constructor. A simple definition consists in a lowercase identifier, possibly preceded by a `noneq` flag and one or several type parameters, and followed by an optional type equation, and then an optional type representation. The identifier is the name of the type constructor being defined.

The optional type parameters are either one type variable `'`*ident* or a list of type variables `(` `'`*ident*$_1$ `,` ... `,` *ident*$_n$`)` for type constructors with several parameters. Each parameter may be annotated by its variance and its kind. In the case where the type definition introduces an abstract type (i.e. no type equation is provided), these annotations are mandatory and reproduced in the signature of the type constructor. In other cases, the signature computed for the type constructor is the minimal one which fits the type equation, the type representation and these annotations.

The optional type equation `=` *typexpr* makes the defined type equivalent to the type expression *typexpr* on the right of the `=` sign: one can be substituted for the other during typing. If no type equation is given, a new type is generated which is incompatible with any other type.

The optional type representation describes the data structure representing the defined type, by giving the list of associated constructors (if it is a variant type) or associated fields (if it is a record type):

- The type representation `=` *constr-decl* {| constr-decl} [`#` `'` *ident*] describes a variant type. The optional annotation [`#` `'` *ident*] declares the security level attached to variant values. It must be one of the parameters of the type constructor and it is required if the variant type comprises several constructors.

- The type representation `=` `{` *field-decl* {`;` *field-decl* } `}` describes a record type. The optional annotation [`#` `'` *ident*] declares the security level attached to records values. It must be one of the parameter of the type constructor and it is required if the record type comprises one or several mutable fields.

If no type representation is given, nothing is assumed on the structure of the type besides what is stated in the optional type equation.

3.2.8 Exception definitions

Exception definitions introduce new exception names.

exntypexpr ::= `'` *ident*
| *security-level*
| `(` *typeexpr* `)`
| *exntypexpr* `-{`*exntypexpr* | *exntypexpr* | *exntypexpr* `}->` *exntypexpr*
| *exntypexpr* {`*` *exntypexpr*}$^+$
| *typeconstr*
| *typeexpr* *typeconstr*
| `(` *exntypexpr* {`,` *exntypexpr*} `)` *typeconstr*
| *exception* `:` *exntypexpr* `;` *exntypexpr*

exception-definition ::= `exception` *exception-name* [*exception-argument*] [`=` *exception-name*]

exception-argument ::= [`:` `'`*ident*] `of` *exntypexpr*

They consist in an upper case identifier, followed by an optional argument declaration, then an optional equation.

If no argument declaration is provided, then the exception name is constant. Otherwise, the argument declaration consists in an optional parameter and a type expression. The parameter is a level variable `'`*ident* which can appear in the type expression. The type expression *exntypexpr* gives the type of the exception's argument, it is a type expression which may involve constant security levels.

If no equation is provided, then the definition generates a new exception, distinct from all other exceptions in the system. Otherwise, it only gives an alternate name to an existing exception. In this case, the optional argument declaration must be identical to that of the re-binded definition.

3.2.9 Constants

The syntactic class of constants comprises literals from the five base types (integers, floating-point numbers, characters, mutable characters strings and immutable characters strings) and constant constructors.

constant ::= *integer-literal*
| *float-literal*
| *char-literal*
| *string-literal*
| *charray-literal*
| *constr*

3.2.10 Patterns

Flow Caml provides the same patterns as Objective Caml (at the exception of the forms dealing with polymorphic variants). Their respective semantics is naturally preserved.

pattern ::= *value-name*
| `_`
| *pattern* `as` *value-name*
| `(` *pattern* `)`
| `(` *pattern* `:` *type-scheme* `)`
| *pattern* `|` *pattern*
| *constr* *pattern*
| `{` *field* `=` *pattern* {`;` *field* `=` *pattern*} `}`
| `[` *pattern* {`;` *pattern*} `]`
| *pattern* `::` *pattern*
| `[|` *pattern* {`;` *pattern*} `|]`

Because exceptions are not first class values and exception names are not regular value constructors, Flow Caml has a class of patterns dedicated to exceptions:

exception-pattern ::= `_`
| *exception* [*pattern*] {`|` *exception* [*pattern*]}

The first kind of pattern matches every exception. The second matches the exceptions of the given names, with an argument value matching the additional pattern (if any).

3.2.11 Expressions

```
            expr ::= value-path
                  |  constant
                  |  ( expr )
                  |  begin expr end
                  |  ( expr : type-scheme )
                  |  expr , expr {, expr}
                  |  ncconstr expr
                  |  expr :: expr
                  |  [ expr {; expr} ]
                  |  [| expr {; expr} |]
                  |  { field = expr {; field = expr} }
                  |  { expr with field = expr {; field = expr} }
                  |  expr {expr}+
                  |  prefix-symbol expr
                  |  expr (infix-symbol | * | = | or | &) expr
                  |  expr . field
                  |  expr . field <- expr
                  |  expr .( expr )
                  |  expr .( expr ) <- expr
                  |  expr .[ expr ]
                  |  expr .[ expr ] <- expr
                  |  if expr then expr [else expr]
                  |  while expr do expr done
                  |  for ident = expr (to | downto) expr do expr done
                  |  expr ; expr
                  |  match expr with pattern-matching
                  |  function pattern-matching
                  |  fun multiple-matching
                  |  raise (exception | ( exception expr ))
                  |  try expr with [|] handler {| handler}
                  |  try expr finally expr
                  |  let [rec] let-binding {and let-binding} in expr

pattern-matching ::= [|] pattern [when expr] -> expr {| pattern [when expr] -> expr}

multiple-matching ::= {pattern}+ [when expr] -> expr

     let-binding ::= pattern [: type-scheme] = expr
                  |  value-name {pattern}+ [: type-scheme] = expr

         handler ::= exception-pattern -> expr [propagate]
```

The only differences between the syntax of Flow Caml expressions and those of Objective Caml rely in the fact that exceptions are first class values in the former while they are not in the latter. Thus, **`raise`** is no longer a regular function but a construct of the language. Two exceptions handlers are provided: the expression

```
try
  expr
with
  pattern_1 -> expr_1 [propagate]
  ...
  | pattern_n -> expr_n [propagate]
```

evaluates the expression *expr* and returns its value if the evaluation does not raise any exception. If it raises an exception, it is matched against the patterns $pattern_1$ to $pattern_n$. If the matching against $pattern_i$ is the first which succeeds, the associated expression $expr_i$ is evaluated (in an environment enriched by the bindings performed during the matching). If the handler is terminated with the keyword `propagate`, the trapped exception is propagated (in this case, exceptions raised by $expr_i$ are not thrown), otherwise the value produced by the evaluation of $expr_i$ becomes that of the whole `try` expression. If none of the patterns matches the exception raised by *expr*, the exception is raised again, thereby transparently “passing through” the `try` construct.

The expression `try` $expr_1$ `finally` $expr_2$ evaluates the expression $expr_1$. This produces a result which is either a value of a raised exception. In both cases, the expression $expr_2$ is evaluated and its result (value or exception) discarded. Finally, the result produced by $expr_1$ becomes the result of the whole expression.

Evaluation order For the purpose of obtaining a precise information flow analysis, the evaluation order of expressions must be specified. As a result, the right-to-left evaluation order of the current implementation of the Objective Caml language is made part of the specification of the Flow Caml core language.

3.3 The module language

3.3.1 Module types (module specifications)

Module types are the module-level equivalent of type expressions: they specify the general shape and type properties of modules.

```
   module-type ::= modtype-path
                 | sig {specification [;;]} end
                 | functor ( module-name : module-type ) functor-arrow module-type
                 | module-type with mod-constraint {and mod-constraint}
                 | ( module-type )

 specification ::= val value-name : type-scheme
                 | external value-name : type-scheme = external-declaration
                 | type-definition
                 | level-definition
                 | exception-definition
                 | module module-name module-args : module-type
                 | module type modtype-name = [module-type]
                 | open module-path
                 | include module-type

   module-args ::= {( module-name : module-type )}

mod-constraint ::= type [type-parameters] typeconstr = typexpr
                 | level level level-repr
                 | module module-path = ext-module-path
```

functor-arrow ::= `->` | `-{` [*security-level-list*] | [*security-level-list*] `}->`

This language of module types is largely identical to that of Objective Caml, so we only mention the significant differences:

- A new for of specifications, level definitions, may appear in signatures.
- In Flow Caml signatures, exception declarations (introduced by the keyword `exception`) may mention an equality between exceptions (as in structures). This is made necessary because exceptions appears in types for values.
- Functors arrows may mention two lists of security levels, a pair of bounds whose purpose is explained in section 2.7.3 of the tutorial.

3.3.2 Module expressions (module implementations)

Module expressions are the module-level equivalent of value expressions: they evaluate to modules, thus providing implementations for the specifications expressed in module types. There is no difference with Objective Caml to be mentioned.

module-expr ::= *module-path*
| `struct` {*definition* [`;;`]} `end`
| `functor` `(` *module-name* `:` *module-type* `)` `->` *module-expr*
| *module-expr* `(` *module-expr* `)`
| `(` *module-expr* `)`
| `(` *module-expr* `:` *module-type* `)`

definition ::= `let` [`rec`] *let-binding* {`and` *let-binding*}
| `external` *value-name* `:` *type-scheme* `=` *external-declaration*
| *type-definition*
| *level-definition*
| *exception-definition*
| `module` *module-name* *module-args* `=` *module-expr*
| `module type` *modtype-name* `=` *module-type*
| `open` *module-path*
| `include` *module-expr*

Evaluation order The remark about the evaluation order made for the core language also applies to the module language. As a result, the right-to-left evaluation order of the current implementation of the Objective Caml language is made part of the specification of the Flow Caml module language.

3.3.3 Compilation units

flows-declaration [`affects` *security-level-list*] [`raises` *security-level-list*]
interface ::= {*specification*}

implementation ::= *flows-declaration* {*specification*}

flows-declaration ::= `flow` *principal-list* `<` *principal-list* {`and` *principal-list* `<` *principal-list*}

principal-list ::= *principal* {`,` *principal*}

A Flow Caml program can be made of one or several compilation units. Each compilation is made of an interface and an implementation. It also has a name *unit-name*, derived from the names of the files containing the interface and the implementation (see section 4.2 for details).

An implementation consists in two parts: a list of `flow` declarations and a specification. The `flow` declarations define the partial order `<` between principals which is used throughout the type-checking of the specification: it is the smallest one which satisfies all the listed inequalities. The specification is the body of a structure which lists the values, types, levels, exceptions, modules and module types implemented by the unit.

An interface is made of three sections: a list of `flow` declarations, the `affects` and `raises` statements and a specification. The `flow` declarations define the partial order `<` between principals which has been used throughout the type-checking of the unit: in short, it provides a description of the possible information flow generated by the code of the unit. The inequality between principals provided in the interface of a unit must imply (possibly by transitivity) all those mentioned in the implementation of the unit. Lastly, the statements `affects` and `raises` mentions respectively the lower and upper *bounds* of the module expression underlying the compilation unit. They are omitted when the bound is empty.

CHAPTER 4

Tools

4.1 The interactive toplevel (`flowcaml`)

The toplevel tool for Flow Caml, `flowcaml`, permits interactive use of the Flow Caml system through a read–type-check loop. In this mode, the system repeatedly reads Flow Caml phrases from the input, type-checks them and outputs the inferred type, if any. The system prints a `#` (sharp) prompt before reading each phrase.

Input to the toplevel can span several lines. It is terminated by `;;` (a double-semicolon). The toplevel input has the following syntax.

```
toplevel-input ::= {toplevel-phrase} ;;
                 | flows-declaration ;;
                 | # ident [directive-argument];;

toplevel-phrase ::= definition
                  | expr

directive-argument ::= string-literal | integer-literal | value-path | level | typeconstr | exception
```

A toplevel input can consists in a series of definition, similar to those found in implementations of compilation units or in **`struct`** `...` **`end`** module expressions. It can also consist in a `flow` declaration, which extend the current security policy, or in a toplevel directive, starting with `#` (the sharp sign). These directives control the behavior of the toplevel; they are listed below in section 4.1.3.

4.1.1 Graphical output

The toplevel offers a graphical output of inferred type schemes. It is enabled by the `-graph` command line option or the directive `#open_graph`. For a description of the graphical output of type schemes, see section 2.2.6 of the tutorial.

4.1.2 Options

The following command-line options are recognized by the `flowcaml` command.

`-display` *host*:*display*
Specify the host and screen to be used for displaying the graphic window (see the option `-graph`). By default this is obtained from the environment variable `DISPLAY`.

-font *fontname*
Set the font used on the graphical window.

-geometry *width*x*height*+*x-offset*+*y-offset*
Specify the initial geometry of the graphical window. The four parameters are numbers giving (in pixels) the width, height, horizontal offset and vertical offset respectively. Partial specifications of the form *width*x*height* or +*x-offset*+*y-offset* are also allowed.

-graph
Enable the graphic window. The toplevel may produce a graphical representation of inferred type schemes (in addition to the textual standard one), displayed a X graphical window. The display and geometry of this window may be controlled by the options `-display` and `-geometry`.

-I *directory*
Add the given directory to the list of directories searched for source and compiled files. By default, the current directory is searched first, the standard library directory. Directories added with `-I` are searched after the current directory, in the order in which they were given on the command line, but before the standard library directory.

-linewidth *width*
Sets the number of columns of the terminal used for printing messages. Default is 78.

-nopervasives
Does not initially open then `Pervasives` module.

-pprint *flags*
Configure the pretty-print of types. The argument is a string of one or several characters, with the following meaning for each character:

`C/c` enable/disable use of colors for displaying polarities of type variables (on VT100 compatible terminals).

`0/1/2` Set which universally quantified and unconstrained type variables are hidden. In mode `0`, all are hidden or replaced by `_`, in mode `1`, only those which appear on functions arrows are hidden, in mode `2`, all are displayed.

`H/h` enable/disable the hiding of universally quantified and unconstrained type variables (`h` implies `g`).

`V/v` enable/disable printing of the polarities of type variables (with the symbols `+`, `-` and `=`)

Default is `Cv1`.

4.1.3 Toplevel directives

The following directives control the toplevel behavior.

#close_graph
Close the graphical window (see section 4.1.1).

#lookup_exception "*ident*"
Lookup for an exception in the environment and print its declaration.

#lookup_level "*ident*"
Lookup for a level in the environment and print its declaration.

#lookup_type "*ident*"
Lookup for a type in the environment and print its declaration.

#lookup_value *"ident"*
Lookup for a value in the environment and print its description.

#open_graph
Open the graphical window (see section 4.1.1).

#pprint *"flags"*
Set flags for pretty-print (see option `-pprint` above for a description of available flags).

#reset_context
Reset the security level associated to the toplevel evaluation context to its initial value, as if a new program were started.

#quit
Exit the toplevel loop and terminate the `flowcaml` command.

4.2 The batch compiler (`flowcamlc`)

This section is interested with the Flow Caml batch compiler `flowcamlc`. To describe in a few words its working, let us say that it reads Flow Caml files as input, type-checks their content and produces regular Objective Caml code as output. These may be later compiled using the standard `ocamlc` or `ocamlopt` compilers to obtain executables.

4.2.1 Overview

The `flowcamlc` command has a command-line interface similar to that of the Objective Caml compilers. It accepts several types of arguments:

- Arguments ending in `.fmli` are taken to be source files for compilation unit interfaces. From the file *x*`.fmli`, the `flowcamlc` compiler produces a compiled interface in the file *x*`.fcmi` and an Objective Caml compilation unit interface in the file *x*`.mli`.
- Arguments ending in `.fml` are taken to be source files for compilation unit implementations. From the file *x*`.fml`, the `flowcamlc` compiler produces a Objective Caml compilation unit implementation in the file *x*`.ml`.

  If the interface file *x*`.fmli` exists, the implementation *x*`.fml` is checked against the corresponding compiled interface *x*`.fcmi`, which is assumed to exist. If no interface *x*`.fmli` is provided, the compilation of *x*`.ml` produces a compiled interface file *x*`.fcmi` in addition to *x*`.ml`. The file *x*`.fcmi` produced corresponds to an interface that exports everything that is defined in the implementation *x*`.fml`.

4.2.2 Options

The following command-line options are recognized by the `flowcamlc` command:

-dump
Produce dumped abstract syntax tree for Objective Caml as output, instead of literal source code file. These dumped tree can be used as input for the Objective Caml compiler, which won't need to parse again the source code. However, the format of the abstract syntax tree depends on the version of the Objective Caml compiler, so it may be not compatible with yours.

-i
Cause the compiler to print the inferred interface when compiling an implementation (.fml file). This can be useful to check the types inferred by the compiler. Also, since the output follows the syntax of interface files, it can help in writing an explicit interface (.fmli file) for a file: just redirect the standard output of the compiler to a .fmli file, and edit that file to remove all declarations of unexported names.

-I *directory*
Add the given directory to the list of directories searched for source and compiled files. By default, the current directory is searched first, the the standard library directory. Directories added with -I are searched after the current directory, in the order in which they were given on the command line, but before the standard library directory.

-nostdlib
Do not add the standard library directory in the default search path.

-pprint *flags*
Configure the pretty-print of types. See the documentation of the command flowcaml for a description of available flags.

-runlib
Print the location of the run-time library and exit.

-stdlib
Print the location of the standard library and exit.

-v
Print the compiler version and location of libraries and exit.

-version
Print the compiler version and exit.

-nopervasives
Does not initially open the Pervasives module.

4.3 The security policy displayer (flowcamlpol)

4.3.1 Overview

The flowcamlpol tool aims at verifying that a series of compilation units can be linked together in order to produce a secure program. It moreover computes the minimal security policy under which the program can be safely run (which is the "union" of the security policies declared in the compilation units).

The flowcamlpol command takes as argument the list of the compiled interface files of the units which form the program, in the order they should be passed to the linker. For instance, for a program made of the compilation units unit$_1$, unit$_2$, ..., unit$_n$, which is linked with the command ocamlc unit$_1$.cmo unit$_2$.cmo ... unit$_n$.cmo, one have to run:

flowcamlpol unit$_1$.fcmi unit$_2$.fcmi ... unit$_n$.fcmi

This outputs the smallest set of assumptions between principals under which the whole program is type safe, i.e. the smallest security policy which allow its execution. (The tool can give a graphical representation of this set of inequalities with the -graph option.) flowcamlpol also checks that the bounds of the compilation units fit together, i.e. that for every i, the lower bound of unit$_i$ is greater than or equal to the upper bounds of unit$_1$, ..., unit$_{i-1}$, in the printed security policy.

A concrete example of use of flowcamlpol is given in sections 2.8.2 and 2.8.3 of the tutorial.

4.3.2 Options

The following command-line options are recognized by the `flowcamlpol` command.

`-display` *host*: *display*
  Specifies the host and screen to be used for displaying the graphic window (see the option `-graph`). By default this is obtained from the environment variable `DISPLAY`.

`-font` *fontname*
  Sets the font used on the graphical window.

`-geometry` *width*x*height*+*x-offset*+*y-offset*
  Specify the initial geometry of the graphical window. The four parameters are numbers giving (in pixels) the width, height, horizontal offset and vertical offset respectively. Partial specifications of the form *width*x*height* or +*x-offset*+*y-offset* are also allowed.

`-graph`
  Gives a graphical representation of the security lattice. The display and geometry of this windows may be controlled by the options `-display` and `-geometry`.

`-linewidth` *width*
  Sets the number of columns of the terminal used for printing messages. Default is 78.

4.4 The dependency generator (`flowcamldep`)

The `flowcamldep` command scans a set of Flow Caml source files (`.fml` and `.fmli` files) for references to external compilation units, and outputs dependency lines in a format suitable for the `make` utility. This ensures that `make` will compile the source file in the correct order, and recompile those files that need to when a source file is modified.

The typical usage is

```
flowcamldep *.fmli *.fml > Depend.flowcaml
```

where `*.fmli *.fml` expands to all source files in the current directory and `Depend.flowcaml` is the file that should contain the dependencies. (See below for a typical `Makefile`.)

Let us note that `flowcamldep` generates only the dependencies needed to the Flow Caml compiler, that is those relating `.fmli` and `.fml` files to `.mli` and `.ml`. To compile the files generated by Flow Caml with one of the Objective Caml compilers, you will need to run `ocamldep` on the intermediate files `.mli` and `.mli`.

4.4.1 Options

The following command-line options are recognized by `flowcamldep`:

`-I` *directory*
  Add the given directory to the list of directories searched for source files. If a source file `foo.fml` mentions an external compilation unit `Bar`, a dependency on that unit's interface `bar.mli` is generated only if the source for `bar` is found in the current directory or in one of the directories specified with `-I`. Otherwise, `Bar` is assumed to be a module from the standard library, and no dependencies are generated.

4.4.2 A typical Makefile

Here is a template `Makefile` for a Flow Caml program.

```
# Compilers
OCAMLC=ocamlc -I +flowcamlrun
OCAMLOPT=ocamlopt -I +flowcamlrun
FLOWCAMLC=flowcamlc

OCAMLDEP=ocamldep
FLOWCAMLDEP=flowcamldep

# The list of object files for the program
OBJECTS=mod1.cmo mod2.cmo mod3.cmo

# To check the security policy
pol: $(OBJECTS.cmo=.fcmi)
        flowcamlpol $^

# To build the program
prog: $(OBJECTS)
        $(OCAMLC) -o $@ flowcamlrun.cma $(OBJECTS)

# Common rules
.SUFFIXES: .ml .mli .fml .fmli .cmi .cmo .cmx .fcmi

.ml.cmo:
        $(OCAMLC) -c $<

.mli.cmi:
        $(OCAMLC) -c $<

.ml.cmx:
        $(OCAMLOPT) -c $<

.fmli.mli:
        $(FLOWCAMLC) -c $<

.fml.ml:
        $(FLOWCAMLC) -c $<

# Clean up
        rm -f prog
        rm -f *.cm[iox] *.fcmi
        for i in *.mli; do \
        if test -f `basename $$i .mli`.fmli; then rm -f $$i; fi \
        done
        for i in *.ml; do \
        if test -f `basename $$i .ml`.fml; then rm -f $$i; fi \
        done

# Dependencies
depend-flowcaml:
        $(FLOWCAMLDEP) *.fml *.fmli > Depend.flowcaml

depend-ocaml:
        $(OCAMLDEP) *.ml *.mli > Depend.ocaml

# Dependencies
depend-flowcaml:
        $(FLOWCAMLDEP) *.fml *.fmli > Depend.flowcaml

depend-ocaml: $(patsubst %.fml,%.ml,$(wildcard *.fml))\
              $(patsubst %.fml,%.ml,$(wildcard *.fml))
        $(OCAMLDEP) *.ml *.mli > Depend.ocaml

include Depend.flowcaml
include Depend.ocaml
```

CHAPTER 5

The Flow Caml library

This chapter describes the Flow Caml library. This library is a translation of (a subset of) the Objective Caml standard library for the Flow Caml language, which includes all the necessary security annotations. As in Objective Caml, the `Pervasives` module is automatically "opened" when a compilation starts or when the toplevel is launched.

5.1 Built-in types and predefined exceptions

The following built-in types and predefined exceptions are always defined in the compilation environment, but are not part of any module. As a consequence, they can only be referred by their short names.

5.1.1 Predefined types

These are predefined types :

```
type (#'a:level) int
```

The type of integer numbers.

```
type (#'a:level) int32
```

The type of 32 bits integer numbers.

```
type (#'a:level) int64
```

The type of 64 bits integer numbers.

```
type (#'a:level) nativeint
```

The type of processor-native integer numbers.

```
type (#'a:level) char
```

The type of characters.

```
type (#'a:level) string
```

The type of immutable strings.

```
type (='a:level, #'b:level) charray
```

The type of mutable character strings.

```
type (#’a:level) float
```

The type of floating-point numbers.

```
type (#’a:level) bool
```

The type of booleans (truth values).

```
type noneq unit = ()
```

The type of the unit value.

```
type (+’a:type, #’b:level) array
```

The type of arrays whose elements have type `’a`.

```
type (+’a:type, #’b:level) list = [] | :: of ’a * (’a, ’b) list # ’b
```

The type of lists whose elements have type `’a`.

```
type (+’a:type, #’b:level) option = None | Some of ’a # ’b
```

The type of optional values.

5.1.2 Exceptions

These are predefined exceptions :

```
exception Invalid_argument : ’a of ’a stg
```

Exception raised by library functions to signal that the given arguments do not make sense. **(For compatibility reasons with the Objective Caml library, this exception is uncatchable for some particular values of its string argument.)**

```
exception Failure : ’a of ’a stg
```

Exception raised by library functions to signal that they are undefined on the given arguments. **(For compatibility reasons with the Objective Caml library, this exception is uncatchable for some particular values of its string argument.)**

```
exception Not_found
```

Exception raised by search functions when the desired object could not be found.

```
exception End_of_file
```

Exception raised by input functions to signal that the end of file has been reached.

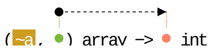
```
exception Division_by_zero
```

Exception raised by division and remainder operations when their second argument is null.

5.2 Module `Array`

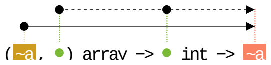
Array operations.

```
val length : ('a, 'b) array -> 'b int
```



Return the length (number of elements) of the given array.

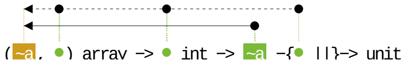
```
val get : ('a, 'b) array -> 'b int -> 'c
          with 'a < 'c
          and 'b < level('c)
```



`Array.get a n` returns the element number `n` of array `a`. The first element has number 0. The last element has number `Array.length a - 1`.

Terminate the program if `n` is outside the range 0 to `(Array.length a - 1)`. You can also write `a.(n)` instead of `Array.get a n`.

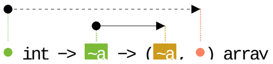
```
val set : ('a, 'b) array -> 'b int -> 'a -{'b ||}-> unit
          with 'b < level('a)
```



`Array.set a n x` modifies array `a` in place, replacing element number `n` with `x`.

Terminate the program if `n` is outside the range 0 to `Array.length a - 1`. You can also write `a.(n) <- x` instead of `Array.set a n x`.

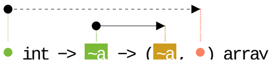
```
val make : 'a int -> 'b -> ('b, 'a) array
```



`Array.make n x` returns a fresh array of length `n`, initialized with `x`. All the elements of this new array are initially physically equal to `x` (in the sense of the `==` predicate). Consequently, if `x` is mutable, it is shared among all elements of the array, and modifying `x` through one of the array entries will modify all other entries at the same time.

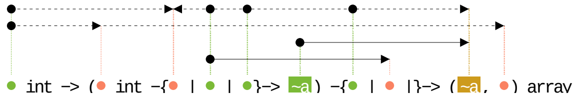
Terminate the program if `n < 0` or `n > Sys.max_array_length`. If the value of `x` is a floating-point number, then the maximum size is divided by 2.

```
val create : 'a int -> 'b -> ('b, 'a) array
```



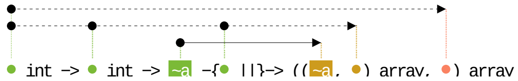
Deprecated. `Array.create` is an alias for `Array.make`.

```
val init : 'a int ->
           ('a int -{'b | 'c | 'd}-> 'e) -{'d | 'c |}-> ('e, 'a) array
           with 'a, content('c), 'd < 'b
           and 'a, content('c), 'd < level('e)
```



`Array.init n f` returns a fresh array of length `n`, with element number `i` initialized to the result of `f i`. In other terms, `Array.init n f` tabulates the results of `f` applied to the integers `0` to `n-1`.

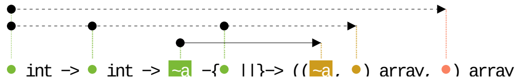
```
val make_matrix : 'a int ->
                  'b int -> 'c -{'b ||}-> (('c, 'b) array, 'a) array
                  with 'a < 'b
```



`Array.make_matrix dimx dimy e` returns a two-dimensional array (an array of arrays) with first dimension `dimx` and second dimension `dimy`. All the elements of this new matrix are initially physically equal to `e`. The element (`x,y`) of a matrix `m` is accessed with the notation `m.(x).(y)`.

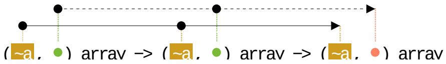
Terminate the program if `dimx` or `dimy` is less than 1 or greater than `Sys.max_array_length`. If the value of `e` is a floating-point number, then the maximum size is only `Sys.max_array_length / 2`.

```
val create_matrix : 'a int ->
                    'b int -> 'c -{'b ||}-> (('c, 'b) array, 'a) array
                    with 'a < 'b
```



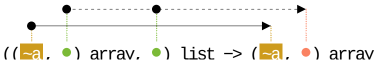
Deprecated. `Array.create_matrix` is an alias for `Array.make_matrix`.

```
val append : ('a, 'b) array -> ('c, 'b) array -> ('d, 'b) array
             with 'a, 'c < 'd
```



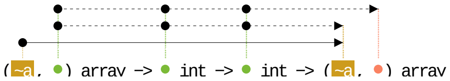
`Array.append v1 v2` returns a fresh array containing the concatenation of the arrays `v1` and `v2`.

```
val concat : (('a, 'b) array, 'b) list -> ('c, 'b) array
             with 'a < 'c
```



Same as `Array.append`, but concatenates a list of arrays.

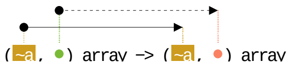
```
val sub : ('a, 'b) array -> 'b int -> 'b int -> ('c, 'b) array
          with 'a < 'c
          and 'b < level('c)
```



Array.sub a start len returns a fresh array of length len, containing the elements number start to start + len - 1 of array a.

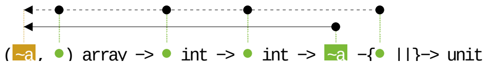
Terminate the program if start and len do not designate a valid subarray of a; that is, if start < 0, or len < 0, or start + len > Array.length a.

```
val copy : ('a, 'b) array -> ('c, 'b) array
       with 'a < 'c
```



Array.copy a returns a copy of a, that is, a fresh array containing the same elements as a.

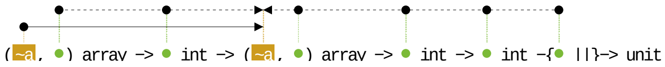
```
val fill : ('a, 'b) array -> 'b int -> 'b int -> 'a -{'b ||}-> unit
       with 'b < level('a)
```



Array.fill a ofs len x modifies the array a in place, storing x in elements number ofs to ofs + len - 1.

Terminate the program if ofs and len do not designate a valid subarray of a.

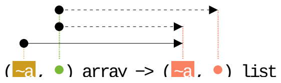
```
val blit : ('a, 'b) array ->
       'b int -> ('c, 'b) array -> 'b int -> 'b int -{'b ||}-> unit
       with 'a < 'c
       and 'b < level('c)
```


(~a, ●) array -> ● int -> (~a, ●) array -> ● int -> ● int -{● ||}-> unit

Array.blit v1 o1 v2 o2 len copies len elements from array v1, starting at element number o1, to array v2, starting at element number o2. It works correctly even if v1 and v2 are the same array, and the source and destination chunks overlap.

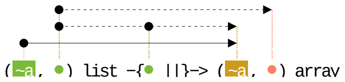
Terminate the program if o1 and len do not designate a valid subarray of v1, or if o2 and len do not designate a valid subarray of v2.

```
val to_list : ('a, 'b) array -> ('c, 'b) list
          with 'a < 'c
          and 'b < level('c)
```



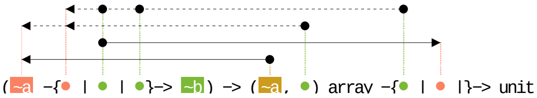
Array.to_list a returns the list of all the elements of a.

```
val of_list : ('a, 'b) list -{'c ||}-> ('a, 'b) array
          with 'b, 'c < level('a)
```



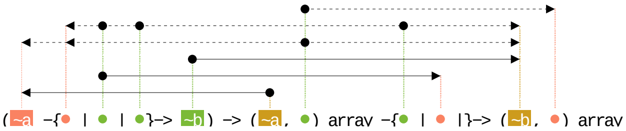
Array.of_list l returns a fresh array containing the elements of l.

```
val iter : ('a -{'b | 'c | 'b}-> 'd) -> ('e, 'f) array -{'b | 'c |}-> unit
        with 'e < 'a
        and 'f < level('a), 'b
        and content('c) < 'b
```



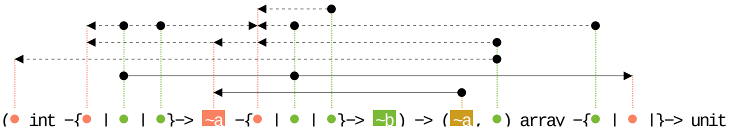
`Array.iter f a` applies function `f` in turn to all the elements of `a`. It is equivalent to `f a.(0); f a.(1); ...; f a.(Array.length a - 1); ()`.

```
val map : ('a -{'b | 'c | 'd}-> 'e) ->
        ('f, 'g) array -{'d | 'c |}-> ('e, 'g) array
        with 'f < 'a
        and 'g < level('a), 'b, level('e)
        and 'd < 'b, level('e)
        and content('c) < 'b, level('e)
```


(~a -{● | ● | ●}-> ~b) -> (~a, ●) array -{● | ● |}-> (~b, ●) array

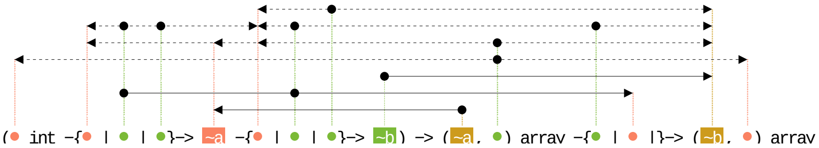
`Array.map f a` applies function `f` to all the elements of `a`, and builds an array with the results returned by `f`: `[| f a.(0); f a.(1); ...; f a.(Array.length a - 1) |]`.

```
val iteri : ('a int -{'b | 'c | 'd}-> 'e -{'f | 'c | 'f}-> 'g) ->
         ('h, 'a) array -{'d | 'c |}-> unit
         with 'd < 'b, 'f
         and 'a < 'b, level('e), 'f
         and content('c) < 'b, 'f
         and 'h < 'e
```


(● int -{● | ● | ●}-> ~a -{● | ● | ●}-> ~b) -> (~a, ●) array -{● | ● |}-> unit

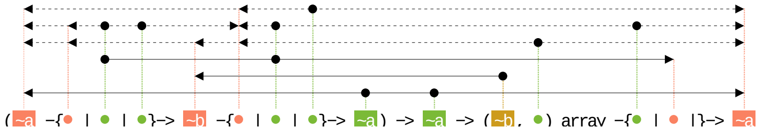
Same as `Array.iter`, but the function is applied to the index of the element as first argument, and the element itself as second argument.

```
val mapi : ('a int -{'b | 'c | 'd}-> 'e -{'f | 'c | 'g}-> 'h) ->
        ('i, 'a) array -{'d | 'c |}-> ('h, 'a) array
        with 'd < 'b, 'f, level('h)
        and 'g < 'f, level('h)
        and 'a < 'b, level('e), 'f, level('h)
        and content('c) < 'b, 'f, level('h)
        and 'i < 'e
```


(● int -{● | ● | ●}-> ~a -{● | ● | ●}-> ~b) -> (~a, ●) array -{● | ● |}-> (~b, ●) array

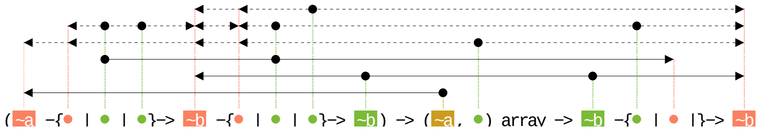
Same as `Array.map`, but the function is applied to the index of the element as first argument, and the element itself as second argument.

```
val fold_left : ('a -{'b | 'c | 'd}-> 'e -{'f | 'c | 'g}-> 'a) ->
                'a -> ('h, 'i) array -{'d | 'c |}-> 'a
                with 'i < level('a), 'b, level('e), 'f
                and 'd < level('a), 'b, 'f
                and 'g < level('a), 'f
                and content('c) < level('a), 'b, 'f
                and 'h < 'e
```


(~a -{● | ● | ●}-> ~b -{● | ● | ●}-> ~a) -> ~a -> (~b, ●) array -{● | ● |}-> ~a

Array.fold_left f x a computes f (... (f (f x a.(0)) a.(1)) ...) a.(n-1), where n is the length of the array a.

```
val fold_right : ('a -{'b | 'c | 'd}-> 'e -{'f | 'c | 'g}-> 'e) ->
                 ('h, 'i) array -> 'e -{'d | 'c |}-> 'e
                 with 'h < 'a
                 and 'i < level('a), 'b, level('e), 'f
                 and 'd < 'b, level('e), 'f
                 and 'g < level('e), 'f
                 and content('c) < 'b, level('e), 'f
```


(~a -{● | ● | ●}-> ~b -{● | ● | ●}-> ~b) -> (~a, ●) array -> ~b -{● | ● |}-> ~b

Array.fold_right f a x computes f a.(0) (f a.(1) (... (f a.(n-1) x) ...)), where n is the length of the array a.

5.3 Module Buffer

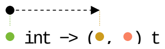
Extensible string buffers.

This module implements string buffers that automatically expand as necessary. It provides accumulative concatenation of strings in quasi-linear time (instead of quadratic time when strings are concatenated pairwise).

```
type (='a:level, #'b:level) t
```

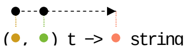
The abstract type of buffers.

```
val create : 'a int -> ('a, 'b) t
```



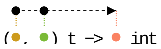
create n returns a fresh buffer, initially empty. The n parameter is the initial size of the internal string that holds the buffer contents. That string is automatically reallocated when more than n characters are stored in the buffer, but shrinks back to n characters when reset is called. For best performance, n should be of the same order of magnitude as the number of characters that are expected to be stored in the buffer (for instance, 80 for a buffer that holds one output line). Nothing bad will happen if the buffer grows beyond that limit, however. In doubt, take n = 16 for instance. If n is not between 1 and Sys.max_string_length, it will be clipped to that interval.

```
val contents : ('a, 'b) t -> 'b string
               with 'a < 'b
```



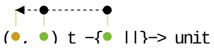
Return a copy of the current contents of the buffer. The buffer itself is unchanged.

```
val length : ('a, 'b) t -> 'b int
             with 'a < 'b
```



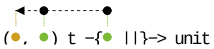
Return the number of characters currently contained in the buffer.

```
val clear : ('a, 'a) t -{'a ||}-> unit
```



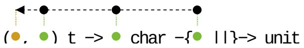
Empty the buffer.

```
val reset : ('a, 'a) t -{'a ||}-> unit
```



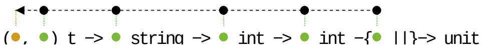
Empty the buffer and deallocate the internal string holding the buffer contents, replacing it with the initial internal string of length `n` that was allocated by `Buffer.create n`. For long-lived buffers that may have grown a lot, `reset` allows faster reclamation of the space used by the buffer.

```
val add_char : ('a, 'a) t -> 'a char -{'a ||}-> unit
```



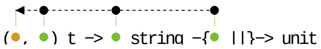
`add_char b c` appends the character `c` at the end of the buffer `b`.

```
val add_substring : ('a, 'a) t ->
                    'a string -> 'a int -> 'a int -{'a ||}-> unit
```



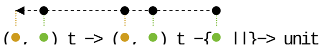
`add_substring b s ofs len` takes `len` characters from offset `ofs` in string `s` and appends them at the end of the buffer `b`.

```
val add_string : ('a, 'a) t -> 'a string -{'a ||}-> unit
```



`add_string b s` appends the string `s` at the end of the buffer `b`.

```
val add_buffer : ('a, 'a) t -> ('b, 'a) t -{'a ||}-> unit
                 with 'b < 'a
```

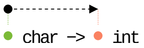


`add_buffer b1 b2` appends the current contents of buffer `b2` at the end of buffer `b1`. `b2` is not modified.

5.4 Module `Char`

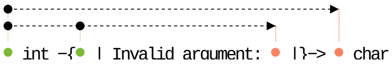
Character operations.

```
val code : ’a char -> ’a int
```



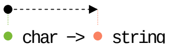
Return the ASCII code of the argument.

```
val chr : ’a int -{’b | Invalid_argument: ’b |}-> ’a char
          with ’a < ’b
```



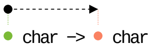
Return the character with the given ASCII code. Raise `Invalid_argument "Char.chr"` if the argument is outside the range 0–255.

```
val escaped : ’a char -> ’a string
```



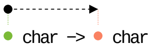
Return a string representing the given character, with special characters escaped following the lexical conventions of Objective Caml.

```
val lowercase : ’a char -> ’a char
```



Convert the given character to its equivalent lowercase character.

```
val uppercase : ’a char -> ’a char
```



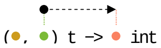
Convert the given character to its equivalent uppercase character.

5.5 Module `Charray`

String operations.

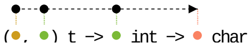
```
type (=’a:level, #’b:level) t = (’a, ’b) charray
```

```
val length : (’a, ’b) t -> ’b int
```



Return the length (number of characters) of the given string.

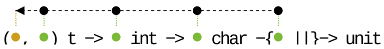
```
val get : (’a, ’b) t -> ’b int -> ’b char
          with ’a < ’b
```



`Charray.get s n` returns character number `n` in string `s`. The first character is character number 0. The last character is character number `Charray.length s - 1`.

Terminate the program if `n` is outside the range 0 to `(Charray.length s - 1)`. You can also write `s.[n]` instead of `Charray.get s n`.

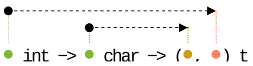
```
val set : ('a, 'a) t -> 'a int -> 'a char -{'a ||}-> unit
```



`Charray.set s n c` modifies string `s` in place, replacing the character number `n` by `c`.

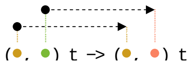
Terminate the program if `n` is outside the range 0 to `(Charray.length s - 1)`. You can also write `s.[n] <- c` instead of `Charray.set s n c`.

```
val make : 'a int -> 'b char -> ('b, 'a) t
```



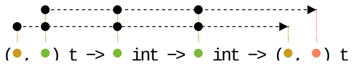
`Charray.make n c` returns a fresh string of length `n`, filled with the character `c`. **Terminate the program** if `n < 0` or `n > Sys.max_string_length`.

```
val copy : ('a, 'b) t -> ('c, 'b) t
        with 'a < 'c
```



Return a copy of the given string.

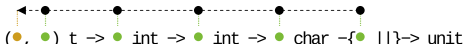
```
val sub : ('a, 'b) t -> 'b int -> 'b int -> ('c, 'b) t
        with 'a, 'b < 'c
```



`Charray.sub s start len` returns a fresh string of length `len`, containing the characters number `start` to `start + len - 1` of string `s`.

Terminate the program if `start` and `len` do not designate a valid substring of `s`; that is, if `start < 0`, or `len < 0`, or `start + len > Charray.length s`.

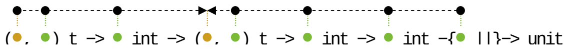
```
val fill : ('a, 'a) t -> 'a int -> 'a int -> 'a char -{'a ||}-> unit
```



`Charray.fill s start len c` modifies string `s` in place, replacing the characters number `start` to `start + len - 1` by `c`.

Terminate the program if `start` and `len` do not designate a valid substring of `s`.

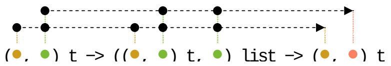
```
val blit : ('a, 'b) t ->
        'b int -> ('b, 'b) t -> 'b int -> 'b int -{'b ||}-> unit
        with 'a < 'b
```


(●, ●) t -> ● int -> (●, ●) t -> ● int -> ● int -{● ||}-> unit

`Charray.blit src srcoff dst dstoff len` copies `len` characters from string `src`, starting at character number `srcoff`, to string `dst`, starting at character number `dstoff`. It works correctly even if `src` and `dst` are the same string, and the source and destination chunks overlap.

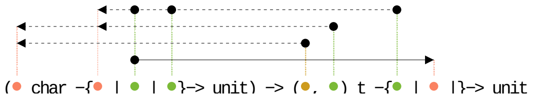
Terminate the program if `srcoff` and `len` do not designate a valid substring of `src`, or if `dstoff` and `len` do not designate a valid substring of `dst`.

```
val concat : (’a, ’b) t -> ((’c, ’b) t, ’b) list -> (’d, ’b) t
             with ’a, ’b, ’c < ’d
```



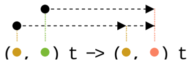
`Charray.concat sep sl` concatenates the list of strings `sl`, inserting the separator string `sep` between each.

```
val iter : (’a char -{’b | ’c | ’b}-> unit) -> (’d, ’e) t -{’b | ’c |}-> unit
           with content(’c), ’e < ’b
           and ’d, ’e < ’a
```



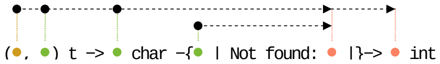
`Charray.iter f s` applies function `f` in turn to all the characters of `s`. It is equivalent to `f s.(0); f s.(1); ...; f s.(Charray.length s - 1); ()`.

```
val escaped : (’a, ’b) t -> (’c, ’b) t
              with ’a < ’b, ’c
```



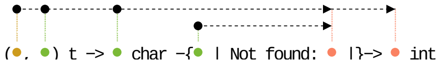
Return a copy of the argument, with special characters represented by escape sequences, following the lexical conventions of Objective Caml. If there is no special character in the argument, return the original string itself, not a copy.

```
val index : (’a, ’b) t -> ’b char -{’c | Not_found: ’c |}-> ’d int
            with ’b < ’c, ’d
            and ’a < ’c, ’d
```



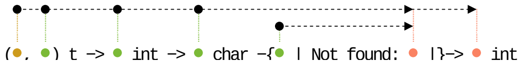
`Charray.index s c` returns the position of the leftmost occurrence of character `c` in string `s`. Raise `Not_found` if `c` does not occur in `s`.

```
val rindex : (’a, ’b) t -> ’b char -{’c | Not_found: ’c |}-> ’d int
             with ’b < ’c, ’d
             and ’a < ’c, ’d
```



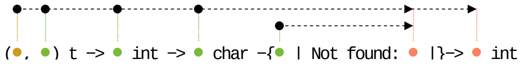
`Charray.rindex s c` returns the position of the rightmost occurrence of character `c` in string `s`. Raise `Not_found` if `c` does not occur in `s`.

```
val index_from : ('a, 'b) t ->
                 'b int -> 'b char -{'c | Not_found: 'c |}-> 'd int
                 with 'b < 'c, 'd
                 and 'a < 'c, 'd
```



Same as `Charray.index`, but start searching at the character position given as second argument. `Charray.index s c` is equivalent to `Charray.index_from s 0 c`.

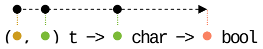
```
val rindex_from : ('a, 'b) t ->
                  'b int -> 'b char -{'c | Not_found: 'c |}-> 'd int
                  with 'b < 'c, 'd
                  and 'a < 'c, 'd
```



Same as `Charray.rindex`, but start searching at the character position given as second argument.

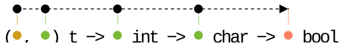
`Charray.rindex s c` is equivalent to `Charray.rindex_from s (Charray.length s - 1) c`.

```
val contains : ('a, 'b) t -> 'b char -> 'b bool
               with 'a < 'b
```



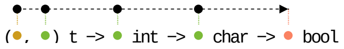
`Charray.contains s c` tests if character `c` appears in the string `s`.

```
val contains_from : ('a, 'b) t -> 'b int -> 'b char -> 'b bool
                    with 'a < 'b
```



`Charray.contains_from s start c` tests if character `c` appears in the substring of `s` starting from `start` to the end of `s`. **Terminate the program** if `start` is not a valid index of `s`.

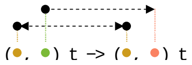
```
val rcontains_from : ('a, 'b) t -> 'b int -> 'b char -> 'b bool
                     with 'a < 'b
```



`Charray.rcontains_from s stop c` tests if character `c` appears in the substring of `s` starting from the beginning of `s` to index `stop`.

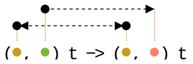
Terminate the program if `stop` is not a valid index of `s`.

```
val uppercase : ('a, 'b) t -> ('a, 'b) t
```



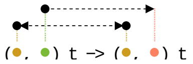
Return a copy of the argument, with all lowercase letters translated to uppercase, including accented letters of the ISO Latin-1 (8859-1) character set.

```
val lowercase : ('a, 'b) t -> ('a, 'b) t
```



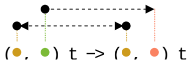
Return a copy of the argument, with all uppercase letters translated to lowercase, including accented letters of the ISO Latin-1 (8859-1) character set.

```
val capitalize : ('a, 'b) t -> ('a, 'b) t
```



Return a copy of the argument, with the first letter set to uppercase.

```
val uncapitalize : ('a, 'b) t -> ('a, 'b) t
```



Return a copy of the argument, with the first letter set to lowercase.

5.6 Module Complex

Complex numbers.

This module provides arithmetic operations on complex numbers. Complex numbers are represented by their real and imaginary parts (cartesian representation). Each part is represented by a double-precision floating-point number (type `float`*).*

```
type (#'a:level) t = { re : 'a float; im : 'a float; }
```

The type of complex numbers. `re` is the real part and `im` the imaginary part.

```
val zero : 'a t
```

The complex number `0`.

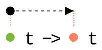
```
val one : 'a t
```

The complex number `1`.

```
val i : 'a t
```

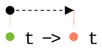
The complex number `i`.

```
val neg : 'a t -> 'a t
```



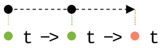
Unary negation.

```
val conj : 'a t -> 'a t
```



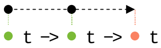
Conjugate: given the complex `x + i.y`, returns `x - i.y`.

```
val add : ’a t -> ’a t -> ’a t
```



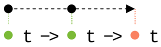
Addition

```
val sub : ’a t -> ’a t -> ’a t
```



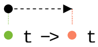
Subtraction

```
val mul : ’a t -> ’a t -> ’a t
```



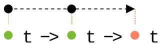
Multiplication

```
val inv : ’a t -> ’a t
```



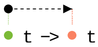
Multiplicative inverse (`1/z`).

```
val div : ’a t -> ’a t -> ’a t
```



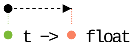
Division

```
val sqrt : ’a t -> ’a t
```



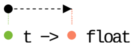
Square root. The result `x + i.y` is such that `x > 0` or `x = 0` and `y >= 0`. This function has a discontinuity along the negative real axis.

```
val norm2 : ’a t -> ’a float
```



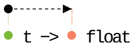
Norm squared: given `x + i.y`, returns `x^2 + y^2`.

```
val norm : ’a t -> ’a float
```



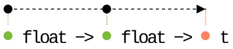
Norm: given `x + i.y`, returns `sqrt(x^2 + y^2)`.

```
val arg : ’a t -> ’a float
```



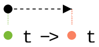
Argument. The argument of a complex number is the angle in the complex plane between the positive real axis and a line passing through zero and the number. This angle ranges from `-pi` to `pi`. This function has a discontinuity along the negative real axis.

```
val polar : 'a float -> 'a float -> 'a t
```



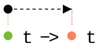
`polar norm arg` returns the complex having norm `norm` and argument `arg`.

```
val exp : 'a t -> 'a t
```



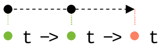
Exponentiation. `exp z` returns `e` to the `z` power.

```
val log : 'a t -> 'a t
```



Natural logarithm (in base `e`).

```
val pow : 'a t -> 'a t -> 'a t
```



Power function. `pow z1 z2` returns `z1` to the `z2` power.

5.7 Module `Digest`

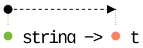
MD5 message digest.

This module provides functions to compute 128-bit "digests" of arbitrary-length strings or files. The digests are of cryptographic quality: it is very hard, given a digest, to forge a string having that digest. The algorithm used is MD5.

```
type (#'a:level) t = 'a string
```

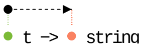
The type of digests: 16-character strings.

```
val string : 'a string -> 'a t
```



Return the digest of the given string.

```
val to_hex : 'a t -> 'a string
```

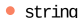


Return the printable hexadecimal representation of the given digest.

5.8 Module `Filename`

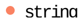
Operations on file names.

```
val current_dir_name : 'a string
```



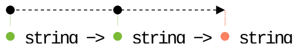
The conventional name for the current directory (e.g. `.` in Unix).

```
val parent_dir_name : 'a string
```



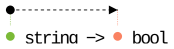
The conventional name for the parent of the current directory (e.g. `..` in Unix).

```
val concat : 'a string -> 'a string -> 'a string
```



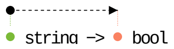
`concat dir file` returns a file name that designates file `file` in directory `dir`.

```
val is_relative : 'a string -> 'a bool
```



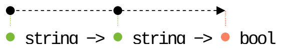
Return `true` if the file name is relative to the current directory, `false` if it is absolute (i.e. in Unix, starts with `/`).

```
val is_implicit : 'a string -> 'a bool
```



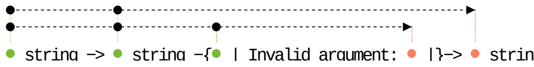
Return `true` if the file name is relative and does not start with an explicit reference to the current directory (`./` or `../` in Unix), `false` if it starts with an explicit reference to the root directory or the current directory.

```
val check_suffix : 'a string -> 'a string -> 'a bool
```



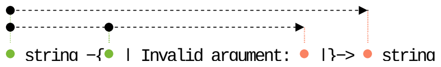
`check_suffix name suff` returns `true` if the filename `name` ends with the suffix `suff`.

```
val chop_suffix : 'a string ->
                  'a string -{'b | Invalid_argument: 'b |}-> 'a string
                  with 'a < 'b
```


● string -> ● string -{● | Invalid argument: ● |}-> ● string

`chop_suffix name suff` removes the suffix `suff` from the filename `name`. The behavior is undefined if `name` does not end with the suffix `suff`.

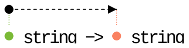
```
val chop_extension : 'a string -{'b | Invalid_argument: 'b |}-> 'a string
                     with 'a < 'b
```



Return the given file name without its extension. The extension is the shortest suffix starting with a period, `.xyz` for instance.

Raise `Invalid_argument` if the given name does not contain a period.

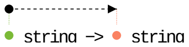
```
val basename : 'a string -> 'a string
```



Split a file name into directory name / base file name.

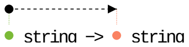
`concat (dirname name) (basename name)` returns a file name which is equivalent to `name`.

```
val dirname : 'a string -> 'a string
```



See `Filename.basename`.

```
val quote : 'a string -> 'a string
```



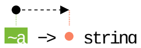
Return a quoted version of a file name, suitable for use as one argument in a shell command line, escaping all shell meta-characters.

5.9 Module `Fmarshal`

Marshaling of data structures.

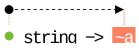
This module provides functions to encode arbitrary data structures as sequences of bytes, which can then be written on a file or sent over a pipe or network connection. The bytes can then be read back later, possibly in another process, and decoded back into a data structure. The format for the byte sequences is compatible across all machines for a given version of Objective Caml.

```
val to_string : 'a -> 'b string
                with content('a) < 'b
```



`Fmarshal.to_string v flags` returns a string containing the representation of `v` as a sequence of bytes.

```
val from_string : 'a string -> 'b
                  with 'a < level('b)
```



`Fmarshal.from_string buff ofs` unmarshals a structured value stored in `stg`.

5.10 Module Hashtbl

Hash tables and hash functions.

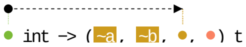
Hash tables are hashed association tables, with in-place modification.

Generic interface

```
type (='a:type, ='b:type, ='c:level, #'d:level) t
```

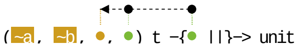
The type of hash tables from type `'a` to type `'b`.

```
val create : 'a int -> ('b, 'c, 'a, 'd) t
```



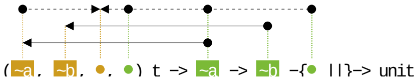
`Hashtbl.create n` creates a new, empty hash table, with initial size `n`. For best results, `n` should be on the order of the expected number of elements that will be in the table. The table grows as needed, so `n` is just an initial guess.

```
val clear : ('a, 'b, 'c, 'c) t -{'c ||}-> unit
```



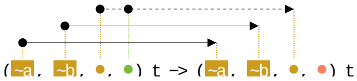
Empty a hash table.

```
val add : ('a, 'b, 'c, 'c) t -> 'd -> 'b -{'c ||}-> unit
          with 'd < 'a
          and content('a), content('d) < 'c
```



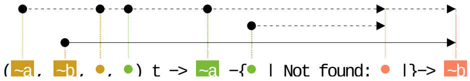
`Hashtbl.add tbl x y` *adds a binding of* `x` *to* `y` *in table* `tbl`*. Previous bindings for* `x` *are not removed, but simply hidden. That is, after performing* `Hashtbl.remove tbl x`*, the previous binding for* `x`*, if any, is restored. (Same behavior as with association lists.)*

```
val copy : ('a, 'b, 'c, 'd) t -> ('e, 'f, 'd, 'g) t
           with 'a < 'e
           and 'b < 'f
           and 'c < 'd
```



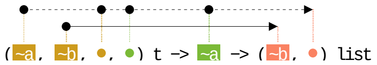
Return a copy of the given hashtable.

```
val find : ('a, 'b, 'c, 'd) t -> 'e -{'f | Not_found: 'f |}-> 'g
           with 'e ~ 'a
           and content('a), 'c, 'd, content('e) < level('g)
           and 'b < 'g
           and content('a), 'c, 'd, content('e) < 'f
```



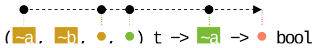
`Hashtbl.find tbl x` returns the current binding of `x` in `tbl`, or raises `Not_found` if no such binding exists.

```
val find_all : (’a, ’b, ’c, ’d) t -> ’e -> (’b, ’d) list
               with ’a ~ ’e
               and content(’a), ’c, content(’e) < ’d
```



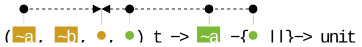
`Hashtbl.find_all tbl x` returns the list of all data associated with `x` in `tbl`. The current binding is returned first, then the previous bindings, in reverse order of introduction in the table.

```
val mem : (’a, ’b, ’c, ’d) t -> ’e -> ’d bool
          with ’a ~ ’e
          and content(’a), ’c, content(’e) < ’d
```



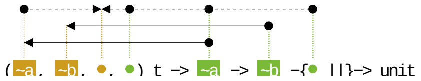
`Hashtbl.mem tbl x` checks if `x` is bound in `tbl`.

```
val remove : (’a, ’b, ’c, ’c) t -> ’d -{’c ||}-> unit
             with ’a ~ ’d
             and content(’a), content(’d) < ’c
```



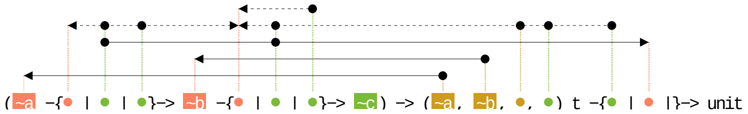
`Hashtbl.remove tbl x` removes the current binding of `x` in `tbl`, restoring the previous binding if it exists. It does nothing if `x` is not bound in `tbl`.

```
val replace : (’a, ’b, ’c, ’c) t -> ’d -> ’b -{’c ||}-> unit
              with ’d < ’a
              and content(’a), content(’d) < ’c
```



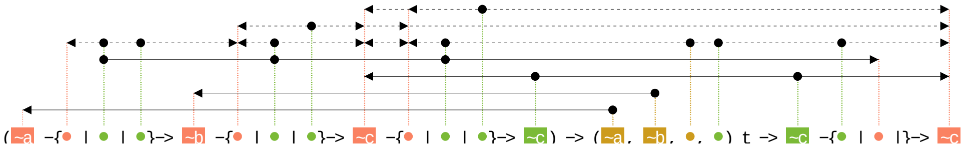
`Hashtbl.replace tbl x y` replaces the current binding of `x` in `tbl` by a binding of `x` to `y`. If `x` is unbound in `tbl`, a binding of `x` to `y` is added to `tbl`. This is functionally equivalent to `Hashtbl.remove tbl x` followed by `Hashtbl.add tbl x y`.

```
val iter : (’a -{’b | ’c | ’d}-> ’e -{’f | ’c | ’f}-> ’g) ->
           (’a, ’e, ’h, ’d) t -{’d | ’c |}-> unit
           with content(’c), ’d, ’h < ’f
           and content(’c), ’d, ’h < ’b
```


(~a -{ ● | ● | ● }-> ~b -{ ● | ● | ● }-> ~c) -> (~a , ~b , ● , ●) t -{ ● | ● |}-> unit

`Hashtbl.iter f tbl` applies `f` to all bindings in table `tbl`. `f` receives the key as first argument, and the associated value as second argument. The order in which the bindings are passed to `f` is unspecified. Each binding is presented exactly once to `f`.

```
val fold : ('a -{'b | 'c | 'd}-> 'e -{'f | 'c | 'g}-> 'h -{'i | 'c | 'j}-> 'h) ->
           ('a, 'e, 'k, 'd) t -> 'h -{'d | 'c |}-> 'h
           with content('c), 'd, 'g, 'j, 'k < 'i
           and content('c), 'd, 'g, 'k < 'f
           and content('c), 'd, 'k < 'b
           and content('c), 'd, 'g, 'j, 'k < level('h)
```



`Hashtbl.fold f tbl init` computes `(f kN dN ... (f k1 d1 init)...)`, where `k1 ... kN` are the keys of all bindings in `tbl`, and `d1 ... dN` are the associated values. The order in which the bindings are passed to `f` is unspecified. Each binding is presented exactly once to `f`.

Module type `HashedType`

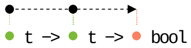
The input signature of the functor `Hashtbl.Make`.

```
module type HashedType = sig
```

```
    type (#'a:level) t
```

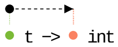
The type of the hashtable keys.

```
    val equal : 'a t -> 'a t -> 'a bool
```



The equality predicate used to compare keys.

```
    val hash : 'a t -> 'a int
```



A hashing function on keys. It must be such that if two keys are equal according to `equal`, then they have identical hash values as computed by `hash`. Examples: suitable (`equal`, `hash`) pairs for arbitrary key types include (`(=)`, `Hashtbl.hash`) for comparing objects by structure, and (`(==)`, `Hashtbl.hash`) for comparing objects by addresses (e.g. for mutable or cyclic keys).

```
end
```

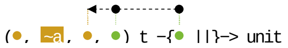
Module type `S`

The output signature of the functor `Hashtbl.Make`.

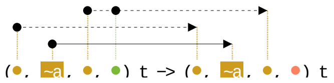
```
module type S = sig
```

```
    type (#'a:level) key

    type (='a:level, ='b:type, ='c:level, #'d:level) t

    val create : 'a int -> ('b, 'c, 'a, 'd) t
```

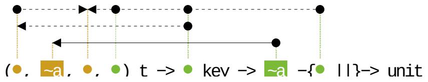
```
val clear : (’a, ’b, ’c, ’c) t -{’c ||}-> unit
```



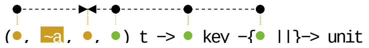
```
val copy : (’a, ’b, ’c, ’d) t -> (’e, ’f, ’d, ’g) t
           with ’c < ’d
           and ’a < ’e
           and ’b < ’f
```



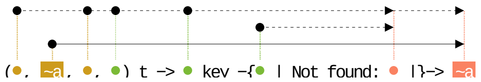
```
val add : (’a, ’b, ’c, ’c) t -> ’a key -> ’b -{’c ||}-> unit
          with ’a < ’c
```



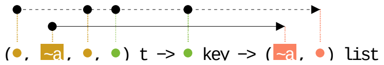
```
val remove : (’a, ’b, ’c, ’c) t -> ’c key -{’c ||}-> unit
             with ’a < ’c
```



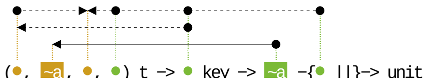
```
val find : (’a, ’b, ’c, ’d) t -> ’d key -{’e | Not_found: ’e |}-> ’f
           with ’a, ’c, ’d < ’e
           and ’a, ’c, ’d < level(’f)
           and ’b < ’f
```



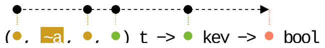
```
val find_all : (’a, ’b, ’c, ’d) t -> ’d key -> (’b, ’d) list
               with ’a, ’c < ’d
```



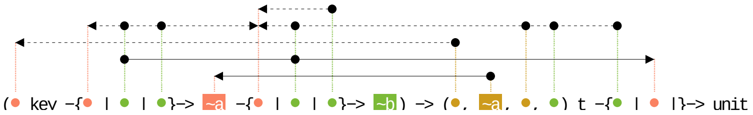
```
val replace : (’a, ’b, ’c, ’c) t -> ’a key -> ’b -{’c ||}-> unit
              with ’a < ’c
```



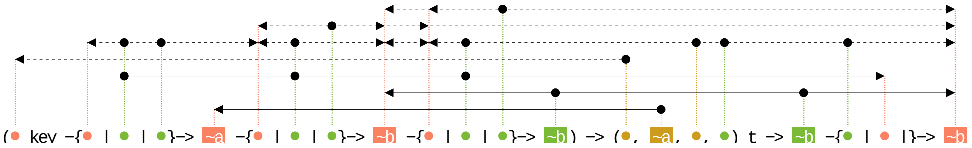
```
val mem : (’a, ’b, ’c, ’d) t -> ’d key -> ’d bool
          with ’a, ’c < ’d
```



```
    val iter : ('a key -{'b | 'c | 'd}-> 'e -{'f | 'c | 'f}-> 'g) ->
               ('a, 'e, 'h, 'd) t -{'d | 'c |}-> unit
               with content('c), 'd, 'h < 'f
               and content('c), 'd, 'h < 'b
```


(● key -{● | ● | ●}-> ~a -{● | ● | ●}-> ~b) -> (●, ~a, ●, ●) t -{● | ● |}-> unit

```
    val fold : ('a key -{'b | 'c | 'd}->
                'e -{'f | 'c | 'g}-> 'h -{'i | 'c | 'j}-> 'h) ->
               ('a, 'e, 'k, 'd) t -> 'h -{'d | 'c |}-> 'h
               with content('c), 'd, 'g, 'j, 'k < 'i
               and content('c), 'd, 'g, 'k < 'f
               and content('c), 'd, 'k < 'b
               and content('c), 'd, 'g, 'j, 'k < level('h)
```


(● key -{● | ● | ●}-> ~a -{● | ● | ●}-> ~b -{● | ● | ●}-> ~b) -> (●, ~a, ●, ●) t -> ~b -{● | ● |}-> ~b

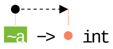
```
end
```

```
module Make : functor (H : HashedType) -> S with type 'a key = 'a H.t
```

Functor building an implementation of the hashtable structure. The operations perform similarly to those of the generic interface, but use the hashing and equality functions specified in the functor argument `H` instead of generic equality and hashing.

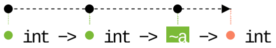
The polymorphic hash primitive

```
val hash : 'a -> 'b int
           with content('a) < 'b
```



`Hashtbl.hash x` associates a positive integer to any value of any type. It is guaranteed that if `x = y`, then `hash x = hash y`. Moreover, `hash` always terminates, even on cyclic structures.

```
val hash_param : 'a int -> 'a int -> 'b -> 'a int
                 with content('b) < 'a
```



`Hashtbl.hash_param n m x` computes a hash value for `x`, with the same properties as for `hash`. The two extra parameters `n` and `m` give more precise control over hashing. Hashing performs a depth-first, right-to-left traversal of the structure `x`, stopping after `n` meaningful nodes were encountered, or `m` nodes, meaningful or not, were encountered. Meaningful nodes are: integers; floating-point numbers; strings; characters; booleans; and constant constructors. Larger values of `m` and `n` means that more nodes are taken into account to compute the final hash value, and therefore collisions are less likely to happen. However, hashing takes longer. The parameters `m` and `n` govern the tradeoff between accuracy and speed.

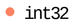
5.11 Module Int32

32-bit integers.

This module provides operations on the type `int32` *of signed 32-bit integers. Unlike the built-in* `int` *type, the type* `int32` *is guaranteed to be exactly 32-bit wide on all platforms. All arithmetic operations over* `int32` *are taken modulo* 2^{32}.

Performance notice: values of type `int32` *occupy more memory space than values of type* `int`, *and arithmetic operations on* `int32` *are generally slower than those on* `int`. *Use* `int32` *only when the application requires exact 32-bit arithmetic.*

```
val zero : 'a int32
```



The 32-bit integer 0.

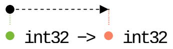
```
val one : 'a int32
```

The 32-bit integer 1.

```
val minus_one : 'a int32
```

The 32-bit integer -1.

```
val neg : 'a int32 -> 'a int32
```



Unary negation.

```
val add : 'a int32 -> 'a int32 -> 'a int32
```

● int32 -> ● int32 -> ● int32

Addition.

```
val sub : 'a int32 -> 'a int32 -> 'a int32
```

● int32 -> ● int32 -> ● int32

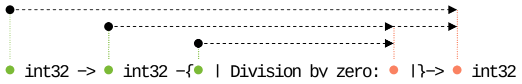
Subtraction.

```
val mul : 'a int32 -> 'a int32 -> 'a int32
```

● int32 -> ● int32 -> ● int32

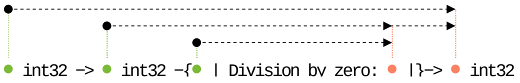
Multiplication.

```
val div : 'a int32 -> 'b int32 -{'c | Division_by_zero: 'c |}-> 'a int32
          with 'b < 'a, 'c
```



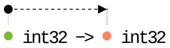
Integer division. Raise `Division_by_zero` if the second argument is zero. This division rounds the real quotient of its arguments towards zero, as specified for `Pervasives.(/)`.

```
val rem : 'a int32 -> 'b int32 -{'c | Division_by_zero: 'c |}-> 'a int32
          with 'b < 'a, 'c
```



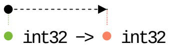
Integer remainder.

```
val succ : 'a int32 -> 'a int32
```



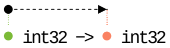
Successor. `Int32.succ x` is `Int32.add x Int32.one`.

```
val pred : 'a int32 -> 'a int32
```



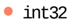
Predecessor. `Int32.pred x` is `Int32.sub x Int32.one`.

```
val abs : 'a int32 -> 'a int32
```



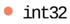
Return the absolute value of its argument.

```
val max_int : 'a int32
```



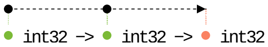
The greatest representable 32-bit integer, 2^{31} - 1.

```
val min_int : 'a int32
```



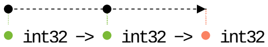
The smallest representable 32-bit integer, -2^{31}.

```
val logand : 'a int32 -> 'a int32 -> 'a int32
```



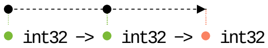
Bitwise logical and.

```
val logor : 'a int32 -> 'a int32 -> 'a int32
```



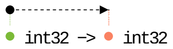
Bitwise logical or.

```
val logxor : 'a int32 -> 'a int32 -> 'a int32
```



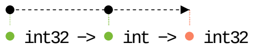
Bitwise logical exclusive or.

```
val lognot : 'a int32 -> 'a int32
```



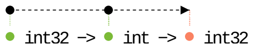
Bitwise logical negation

```
val shift_left : 'a int32 -> 'a int -> 'a int32
```



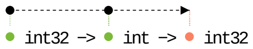
`Int32.shift_left x y` shifts `x` to the left by `y` bits. The result is unspecified if `y < 0` or `y >= 32`.

```
val shift_right : 'a int32 -> 'a int -> 'a int32
```



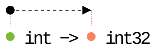
`Int32.shift_right x y` shifts `x` to the right by `y` bits. This is an arithmetic shift: the sign bit of `x` is replicated and inserted in the vacated bits. The result is unspecified if `y < 0` or `y >= 32`.

```
val shift_right_logical : 'a int32 -> 'a int -> 'a int32
```



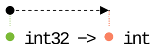
`Int32.shift_right_logical x y` shifts `x` to the right by `y` bits. This is a logical shift: zeroes are inserted in the vacated bits regardless of the sign of `x`. The result is unspecified if `y < 0` or `y >= 32`.

```
val of_int : 'a int -> 'a int32
```



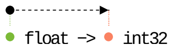
Convert the given integer (type `int`) to a 32-bit integer (type `int32`).

```
val to_int : 'a int32 -> 'a int
```



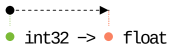
Convert the given 32-bit integer (type `int32`) to an integer (type `int`). On 32-bit platforms, the 32-bit integer is taken modulo 2^{31}, i.e. the high-order bit is lost during the conversion. On 64-bit platforms, the conversion is exact.

```
val of_float : 'a float -> 'a int32
```



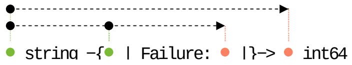
Convert the given floating-point number to a 32-bit integer, discarding the fractional part (truncate towards 0). The result of the conversion is undefined if, after truncation, the number is outside the range [`Int32.min_int`, `Int32.max_int`].

```
val to_float : 'a int32 -> 'a float
```



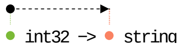
Convert the given 32-bit integer to a floating-point number.

```
val of_string : 'a string -{'b | Failure: 'b |}-> 'a int64
                with 'a < 'b
```



Convert the given string to a 32-bit integer. The string is read in decimal (by default) or in hexadecimal, octal or binary if the string begins with `0x`, `0o` or `0b` respectively. Raise `Failure "int_of_string"` if the given string is not a valid representation of an integer.

```
val to_string : 'a int32 -> 'a string
```

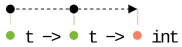


Return the string representation of its argument, in signed decimal.

```
type (#'a:level) t = 'a int32
```

An alias for the type of 32-bit integers.

```
val compare : 'a t -> 'a t -> 'a int
```



The comparison function for 32-bit integers, with the same specification as `Pervasives.compare`. Along with the type `t`, this function `compare` allows the module `Int32` to be passed as argument to the functors `Set.Make` and `Map.Make`.

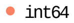
5.12 Module `Int64`

64-bit integers.

This module provides operations on the type `int64` *of signed 64-bit integers. Unlike the built-in* `int` *type, the type* `int64` *is guaranteed to be exactly 64-bit wide on all platforms. All arithmetic operations over* `int64` *are taken modulo* 2^{64}

Performance notice: values of type `int64` *occupy more memory space than values of type* `int`*, and arithmetic operations on* `int64` *are generally slower than those on* `int`*. Use* `int64` *only when the application requires exact 64-bit arithmetic.*

```
val zero : 'a int64
```



The 64-bit integer 0.

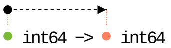
```
val one : 'a int64
```

The 64-bit integer 1.

```
val minus_one : 'a int64
```

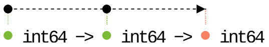
The 64-bit integer -1.

```
val neg : 'a int64 -> 'a int64
```



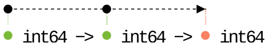
Unary negation.

```
val add : 'a int64 -> 'a int64 -> 'a int64
```



Addition.

```
val sub : 'a int64 -> 'a int64 -> 'a int64
```

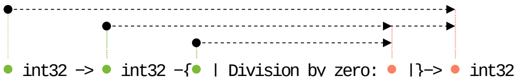


Subtraction.

```
val mul : 'a int64 -> 'a int64 -> 'a int64
```

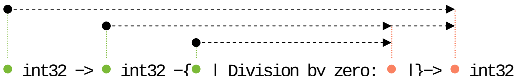
Multiplication.

```
val div : 'a int32 -> 'b int32 -{'c | Division_by_zero: 'c |}-> 'a int32
          with 'b < 'a, 'c
```



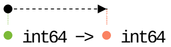
Integer division. Raise `Division_by_zero` if the second argument is zero. This division rounds the real quotient of its arguments towards zero, as specified for `Pervasives.(/)`.

```
val rem : 'a int32 -> 'b int32 -{'c | Division_by_zero: 'c |}-> 'a int32
          with 'b < 'a, 'c
```



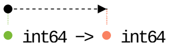
Integer remainder.

```
val succ : 'a int64 -> 'a int64
```



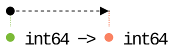
Successor. `Int64.succ x` is `Int64.add x Int64.one`.

```
val pred : 'a int64 -> 'a int64
```



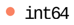
Predecessor. `Int64.pred x` is `Int64.sub x Int64.one`.

```
val abs : 'a int64 -> 'a int64
```



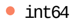
Return the absolute value of its argument.

```
val max_int : ’a int64
```



The greatest representable 64-bit integer, 2^{63} - 1.

```
val min_int : ’a int64
```



The smallest representable 64-bit integer, -2^{63}.

```
val logand : ’a int64 -> ’a int64 -> ’a int64
```

Bitwise logical and.

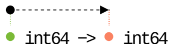
```
val logor : ’a int64 -> ’a int64 -> ’a int64
```

Bitwise logical or.

```
val logxor : ’a int64 -> ’a int64 -> ’a int64
```

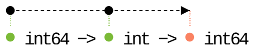
Bitwise logical exclusive or.

```
val lognot : ’a int64 -> ’a int64
```



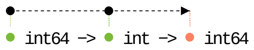
Bitwise logical negation

```
val shift_left : ’a int64 -> ’a int -> ’a int64
```



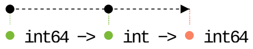
`Int64.shift_left x y` shifts `x` to the left by `y` bits. The result is unspecified if `y < 0` or `y >= 64`.

```
val shift_right : ’a int64 -> ’a int -> ’a int64
```



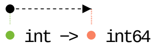
`Int64.shift_right x y` shifts `x` to the right by `y` bits. This is an arithmetic shift: the sign bit of `x` is replicated and inserted in the vacated bits. The result is unspecified if `y < 0` or `y >= 64`.

```
val shift_right_logical : ’a int64 -> ’a int -> ’a int64
```



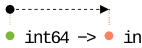
`Int64.shift_right_logical x y` shifts `x` to the right by `y` bits. This is a logical shift: zeroes are inserted in the vacated bits regardless of the sign of `x`. The result is unspecified if `y < 0` or `y >= 64`.

```
val of_int : 'a int -> 'a int64
```



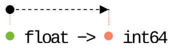
Convert the given integer (type `int`) to a 64-bit integer (type `int64`).

```
val to_int : 'a int64 -> 'a int
```



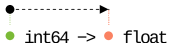
Convert the given 64-bit integer (type `int64`) to an integer (type `int`). On 64-bit platforms, the 64-bit integer is taken modulo 2^{63}, i.e. the high-order bit is lost during the conversion. On 32-bit platforms, the 64-bit integer is taken modulo 2^{31}, i.e. the top 33 bits are lost during the conversion.

```
val of_float : 'a float -> 'a int64
```



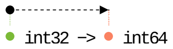
Convert the given floating-point number to a 64-bit integer, discarding the fractional part (truncate towards 0). The result of the conversion is undefined if, after truncation, the number is outside the range [`Int64.min_int`, `Int64.max_int`].

```
val to_float : 'a int64 -> 'a float
```



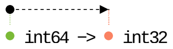
Convert the given 64-bit integer to a floating-point number.

```
val of_int32 : 'a int32 -> 'a int64
```



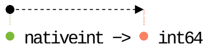
Convert the given 32-bit integer (type `int32`) to a 64-bit integer (type `int64`).

```
val to_int32 : 'a int64 -> 'a int32
```



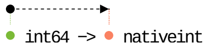
Convert the given 64-bit integer (type `int64`) to a 32-bit integer (type `int32`). The 64-bit integer is taken modulo 2^{32}, i.e. the top 32 bits are lost during the conversion.

```
val of_nativeint : 'a nativeint -> 'a int64
```



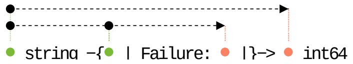
Convert the given native integer (type `nativeint`) to a 64-bit integer (type `int64`).

```
val to_nativeint : 'a int64 -> 'a nativeint
```



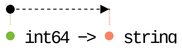
Convert the given 64-bit integer (type `int64`) to a native integer. On 32-bit platforms, the 64-bit integer is taken modulo 2^{32}. On 64-bit platforms, the conversion is exact.

```
val of_string : ’a string -{’b | Failure: ’b |}-> ’a int64
                with ’a < ’b
```



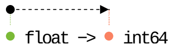
Convert the given string to a 64-bit integer. The string is read in decimal (by default) or in hexadecimal, octal or binary if the string begins with `0x`, `0o` or `0b` respectively. Raise `Failure "int_of_string"` if the given string is not a valid representation of an integer.

```
val to_string : ’a int64 -> ’a string
```



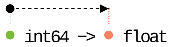
Return the string representation of its argument, in decimal.

```
val bits_of_float : ’a float -> ’a int64
```



Return the internal representation of the given float according to the IEEE 754 floating-point “double format” bit layout. Bit 63 of the result represents the sign of the float; bits 62 to 52 represent the (biased) exponent; bits 51 to 0 represent the mantissa.

```
val float_of_bits : ’a int64 -> ’a float
```

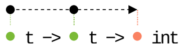


Return the floating-point number whose internal representation, according to the IEEE 754 floating-point “double format” bit layout, is the given `int64`.

```
type (#’a:level) t = ’a int64
```

An alias for the type of 64-bit integers.

```
val compare : ’a t -> ’a t -> ’a int
```



The comparison function for 64-bit integers, with the same specification as `Pervasives.compare`. Along with the type `t`, this function `compare` allows the module `Int64` to be passed as argument to the functors `Set.Make` and `Map.Make`.

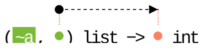
5.13 Module `List`

List operations.

Some functions are flagged as not tail-recursive. A tail-recursive function uses constant stack space, while a non-tail-recursive function uses stack space proportional to the length of its list argument, which can be a problem with very long lists. When the function takes several list arguments, an approximate formula giving stack usage (in some unspecified constant unit) is shown in parentheses.

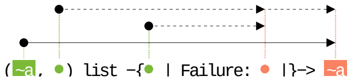
The above considerations can usually be ignored if your lists are not longer than about 10000 elements.

```
val length : ('a, 'b) list -> 'b int
```



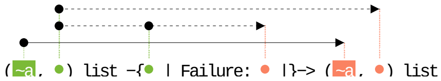
Return the length (number of elements) of the given list.

```
val hd : ('a, 'b) list -{'c | Failure: 'c |}-> 'a
         with 'b < level('a), 'c
```



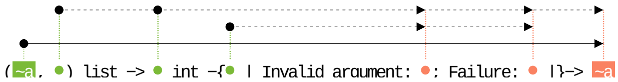
Return the first element of the given list. Raise `Failure "hd"` if the list is empty.

```
val tl : ('a, 'b) list -{'c | Failure: 'c |}-> ('a, 'b) list
         with 'b < 'c
```



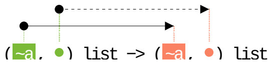
Return the given list without its first element. Raise `Failure "tl"` if the list is empty.

```
val nth : ('a, 'b) list ->
          'b int -{'c | Invalid_argument: 'c; Failure: 'c |}-> 'a
          with 'b < level('a), 'c
```


(~a , ●) list -> ● int -{● | Invalid argument: ● ; Failure: ● |}-> ~a

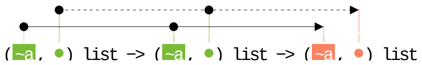
Return the n-th element of the given list. The first element (head of the list) is at position 0. Raise `Failure "nth"` if the list is too short.

```
val rev : ('a, 'b) list -> ('a, 'b) list
```



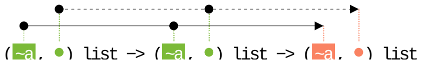
List reversal.

```
val append : ('a, 'b) list -> ('a, 'b) list -> ('a, 'b) list
```



Catenate two lists. Same function as the infix operator `@`. Not tail-recursive (length of the first argument). The `@` operator is not tail-recursive either.

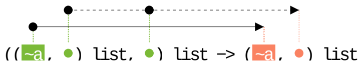
```
val rev_append : ('a, 'b) list -> ('a, 'b) list -> ('a, 'b) list
```



`List.rev_append l1 l2` reverses `l1` and concatenates it to `l2`.

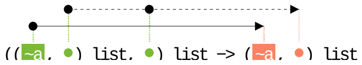
This is equivalent to `List.rev l1 @ l2`, but `rev_append` is tail-recursive and more efficient.

```
val concat : (('a, 'b) list, 'b) list -> ('a, 'b) list
```



Concatenate a list of lists. Not tail-recursive (length of the argument + length of the longest sub-list).

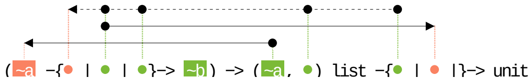
```
val flatten : (('a, 'b) list, 'b) list -> ('a, 'b) list
```



Flatten a list of lists. Not tail-recursive (length of the argument + length of the longest sub-list).

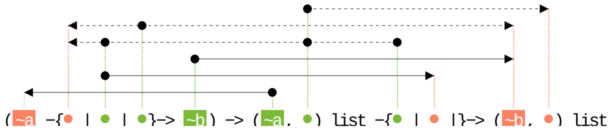
5.13.1 Iterators

```
val iter : ('a -{'b | 'c | 'b}-> 'd) -> ('a, 'b) list -{'b | 'c |}-> unit
           with content('c) < 'b
```



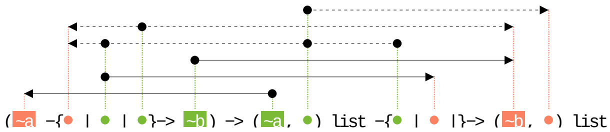
`List.iter f [a1; ...; an]` applies function `f` in turn to `a1; ...; an`. It is equivalent to `begin f a1; f a2; ...; f an; () end`.

```
val map : ('a -{'b | 'c | 'd}-> 'e) ->
          ('a, 'f) list -{'b | 'c |}-> ('e, 'f) list
          with content('c), 'd, 'f < 'b
          and 'd < level('e)
```


(~a -{● | ● | ●}-> ~b) -> (~a, ●) list -{● | ● |}-> (~b, ●) list

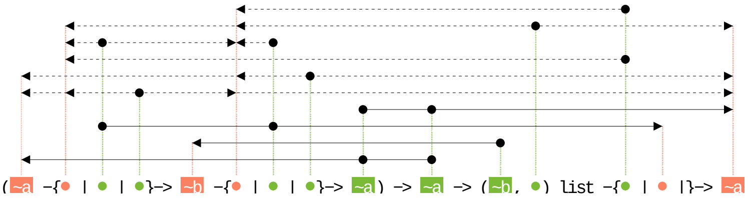
`List.map f [a1; ...; an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`. Not tail-recursive.

```
val rev_map : ('a -{'b | 'c | 'd}-> 'e) ->
              ('a, 'f) list -{'b | 'c |}-> ('e, 'f) list
              with content('c), 'd, 'f < 'b
              and 'd < level('e)
```


(~a -{● | ● | ●}-> ~b) -> (~a, ●) list -{● | ● |}-> (~b, ●) list

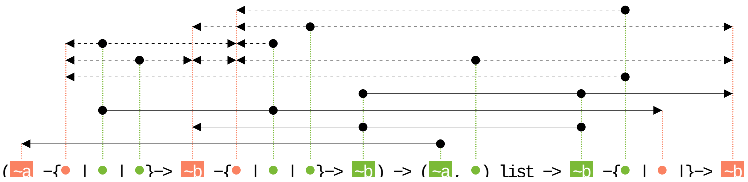
`List.rev_map f l` gives the same result as `List.rev (List.map f l)`, but is tail-recursive and more efficient.

```
val fold_left : ('a -{'b | 'c | 'd}-> 'e -{'f | 'g | 'h}-> 'a) ->
                'a -> ('e, 'i) list -{'b | 'g |}-> 'j
                with content('c), 'd, content('g), 'i < 'b
                and 'b, content('c), 'd, content('g), 'h, 'i < 'f
                and 'c < 'g
                and 'd, 'h, 'i < level('j)
                and 'a < 'j
                and 'd, 'h < level('a)
```


(~a -{● | ● | ●}-> ~b -{● | ● | ●}-> ~a) -> ~a -> (~b, ●) list -{● | ● |}-> ~a

List.fold_left f a [b1; ...; bn] is f (... (f (f a b1) b2) ...) bn.

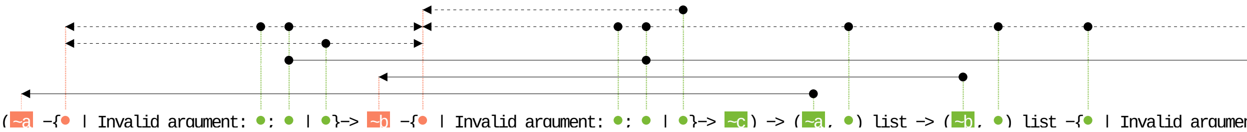
```
val fold_right : ('a -{'b | 'c | 'd}-> 'e -{'f | 'g | 'h}-> 'i) ->
                 ('a, 'd) list -> 'i -{'b | 'g |}-> 'i
                 with content('c), 'd, content('g) < 'b
                 and 'b, content('c), 'd, content('g), 'h < 'f
                 and 'c < 'g
                 and 'd, 'h < level('e)
                 and 'i < 'e
                 and 'd, 'h < level('i)
```


(~a -{● | ● | ●}-> ~b -{● | ● | ●}-> ~b) -> (~a, ●) list -> ~b -{● | ● |}-> ~b

List.fold_right f [a1; ...; an] b is f a1 (f a2 (... (f an b) ...)). Not tail-recursive.

5.13.2 Iterators on two lists

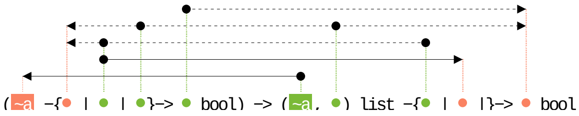
```
val iter2 : ('a -{'b | Invalid_argument: 'c; 'd | 'e}->
             'f -{'g | Invalid_argument: 'c; 'h | 'g}-> 'i) ->
            ('a, 'c) list ->
            ('f, 'c) list -{'c | Invalid_argument: 'j; 'h |}-> unit
            with 'c, content('d), content('h) < 'j
            and 'c, content('d), 'e, content('h) < 'b
            and 'c, content('d), 'e, content('h) < 'g
            and 'd < 'h
```


(~a -{● | Invalid argument: ●; ● | ●}-> ~b -{● | Invalid argument: ●; ● | ●}-> ~c) -> (~a, ●) list -> (~b, ●) list -{● | Invalid argumen

List.iter2 f [a1; ...; an] [b1; ...; bn] calls in turn f a1 b1; ...; f an bn. Raise Invalid_argument if the two lists have different lengths.

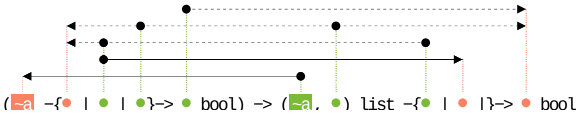
5.13.3 List scanning

```
val for_all : ('a -{'b | 'c | 'd}-> 'e bool) ->
              ('a, 'd) list -{'b | 'c |}-> 'e bool
              with 'd < 'b, 'e
              and content('c) < 'b
```


(~a -{● | ● | ●}-> ● bool) -> (~a, ●) list -{● | ● |}-> ● bool

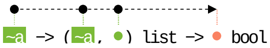
for_all p [a1; ...; an] checks if all elements of the list satisfy the predicate p. That is, it returns (p a1) && (p a2) && ... && (p an).

```
val exists : ('a -{'b | 'c | 'd}-> 'e bool) ->
             ('a, 'd) list -{'b | 'c |}-> 'e bool
             with 'd < 'b, 'e
             and content('c) < 'b
```


(~a -{● | ● | ●}-> ● bool) -> (~a, ●) list -{● | ● |}-> ● bool

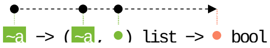
exists p [a1; ...; an] checks if at least one element of the list satisfies the predicate p. That is, it returns (p a1) || (p a2) || ... || (p an).

```
val mem : 'a -> ('a, 'b) list -> 'b bool
          with content('a) < 'b
```



mem a l is true if and only if a is equal to an element of l.

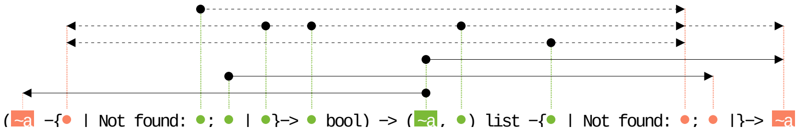
```
val memq : 'a -> ('a, 'b) list -> 'b bool
           with content('a) < 'b
```



Same as List.mem, but uses physical equality instead of structural equality to compare list elements.

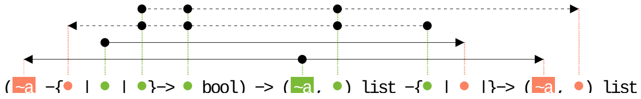
5.13.4 List searching

```
val find : ('a -{'b | Not_found: 'c; 'd | 'e}-> 'e bool) ->
           ('a, 'e) list -{'f | Not_found: 'c; 'd |}-> 'g
           with 'a < 'g
           and 'f < 'b, 'c
           and 'e < 'b, 'c, level('g)
```


(~a -{● | Not found: ●; ● | ●}-> ● bool) -> (~a, ●) list -{● | Not found: ●; ● |}-> ~a

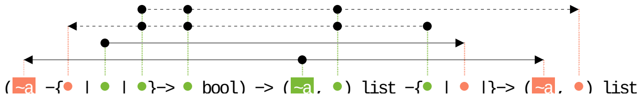
find p l returns the first element of the list l that satisfies the predicate p. Raise Not_found if there is no value that satisfies p in the list l.

```
val filter : (’a -{’b | ’c | ’d}-> ’d bool) ->
             (’a, ’d) list -{’b | ’c |}-> (’a, ’d) list
             with ’d < ’b
```


(~a -{● | ● | ●}-> ● bool) -> (~a, ●) list -{● | ● |}-> (~a, ●) list

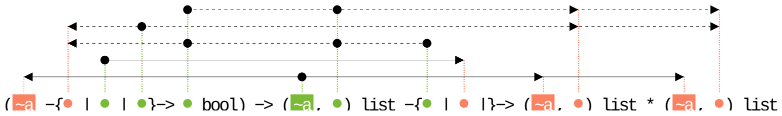
filter p l returns all the elements of the list l that satisfy the predicate p. The order of the elements in the input list is preserved.

```
val find_all : (’a -{’b | ’c | ’d}-> ’d bool) ->
               (’a, ’d) list -{’b | ’c |}-> (’a, ’d) list
               with ’d < ’b
```


(~a -{● | ● | ●}-> ● bool) -> (~a, ●) list -{● | ● |}-> (~a, ●) list

find_all is another name for List.filter.

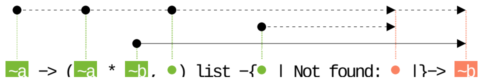
```
val partition : (’a -{’b | ’c | ’d}-> ’e bool) ->
                (’a, ’e) list -{’b | ’c |}-> (’a, ’e) list * (’a, ’e) list
                with ’e < ’b
                and ’d < ’b, ’e
```


(~a -{● | ● | ●}-> ● bool) -> (~a, ●) list -{● | ● |}-> (~a, ●) list * (~a, ●) list

partition p l returns a pair of lists (l1, l2), where l1 is the list of all the elements of l that satisfy the predicate p, and l2 is the list of all the elements of l that do not satisfy p. The order of the elements in the input list is preserved.

5.13.5 Association lists

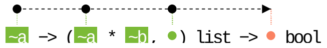
```
val assoc : ’a -> (’a * ’b, ’c) list -{’d | Not_found: ’d |}-> ’b
            with content(’a) < level(’b), ’d
            and ’c < level(’b), ’d
```



assoc a l returns the value associated with key a in the list of pairs l.

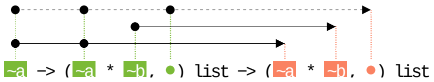
That is, assoc a [...; (a,b); ...] = b if (a,b) is the leftmost binding of a in list l. Raise Not_found if there is no value associated with a in the list l.

```
val mem_assoc : ’a -> (’a * ’b, ’c) list -> ’c bool
                with content(’a) < ’c
```



Same as List.assoc, but simply return true if a binding exists, and false if no bindings exist for the given key.

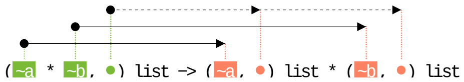
```
val remove_assoc : 'a -> ('a * 'b, 'c) list -> ('a * 'b, 'c) list
                   with content('a) < 'c
```



remove_assoc a l returns the list of pairs l without the first pair with key a, if any. Not tail-recursive.

5.13.6 Lists of pairs

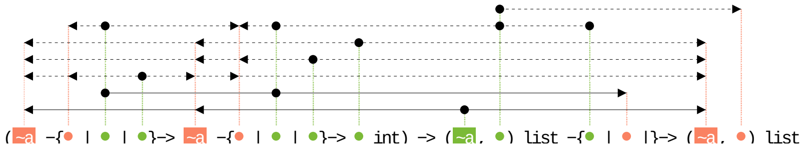
```
val split : ('a * 'b, 'c) list -> ('a, 'c) list * ('b, 'c) list
```



Transform a list of pairs into a pair of lists:
split [(a1,b1); ...; (an,bn)] is ([a1; ...; an], [b1; ...; bn]). Not tail-recursive.

5.13.7 Sorting

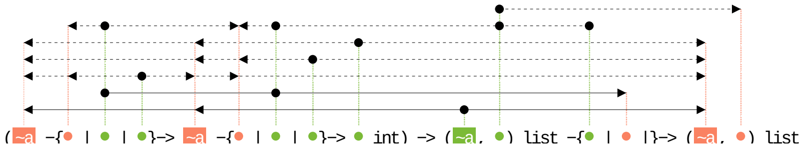
```
val sort : ('a -{'b | 'c | 'd}-> 'a -{'e | 'c | 'f}-> 'g int) ->
           ('a, 'h) list -{'i | 'c |}-> ('a, 'h) list
           with 'd, 'f, 'g < level('a)
           and content('c), 'd, 'f, 'h, 'i < 'e
           and content('c), 'd, 'h, 'i < 'b
```


(~a -{● | ● | ●}-> ~a -{● | ● | ●}-> ● int) -> (~a, ●) list -{● | ● |}-> (~a, ●) list

Sort a list in increasing order according to a comparison function. The comparison function must return 0 if it arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller. For example, the compare function is a suitable comparison function. The resulting list is sorted in increasing order. List.sort is guaranteed to run in constant heap space (in addition to the size of the result list) and logarithmic stack space.

The current implementation uses Merge Sort and is the same as List.stable_sort.

```
val stable_sort : ('a -{'b | 'c | 'd}-> 'a -{'e | 'c | 'f}-> 'g int) ->
                  ('a, 'h) list -{'i | 'c |}-> ('a, 'h) list
                  with 'd, 'f, 'g < level('a)
                  and content('c), 'd, 'f, 'h, 'i < 'e
                  and content('c), 'd, 'h, 'i < 'b
```


(~a -{● | ● | ●}-> ~a -{● | ● | ●}-> ● int) -> (~a, ●) list -{● | ● |}-> (~a, ●) list

Same as `List.sort`, but the sorting algorithm is stable.

The current implementation is Merge Sort. It runs in constant heap space and logarithmic stack space.

5.14 Module `Map`

Module type `OrderedType`

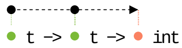
Input signature of the functor `Map.Make`.

```
module type OrderedType = sig
```

```
type (#'a:level) t
```

The type of the map keys.

```
val compare : 'a t -> 'a t -> 'a int
```



A total ordering function over the keys. This is a two-argument function `f` such that `f e1 e2` is zero if the keys `e1` and `e2` are equal, `f e1 e2` is strictly negative if `e1` is smaller than `e2`, and `f e1 e2` is strictly positive if `e1` is greater than `e2`. Example: a suitable ordering function is the generic structural comparison function `Pervasives.compare`.

```
end
```

Module type `S`

Output signature of the functor `Map.Make`.

```
module type S = sig
```

```
type (#'a:level) key
```

The type of the map keys.

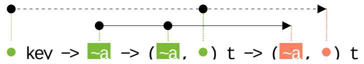
```
type (+'a:type, #'b:level) t
```

The type of maps from type `key` to type `'a`.

```
val empty : ('a, 'b) t
```

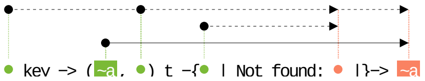
The empty map.

```
val add : 'a key -> 'b -> ('b, 'a) t -> ('b, 'a) t
```



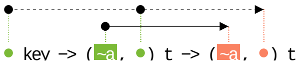
`add x y m` returns a map containing the same bindings as `m`, plus a binding of `x` to `y`. If `x` was already bound in `m`, its previous binding disappears.

```
val find : 'a key -> ('b, 'a) t -{'c | Not_found: 'c |}-> 'b
        with 'a < level('b), 'c
```



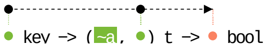
find x m returns the current binding of x in m, or raises Not_found if no such binding exists.

```
val remove : 'a key -> ('b, 'a) t -> ('b, 'a) t
```



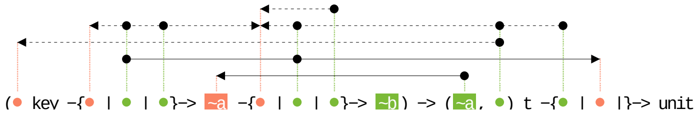
remove x m returns a map containing the same bindings as m, except for x which is unbound in the returned map.

```
val mem : 'a key -> ('b, 'a) t -> 'a bool
```



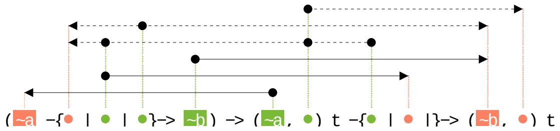
mem x m returns true if m contains a binding for x, and false otherwise.

```
val iter : ('a key -{'b | 'c | 'd}-> 'e -{'f | 'c | 'f}-> 'g) ->
        ('e, 'a) t -{'d | 'c |}-> unit
        with 'a, content('c), 'd < 'f
        and 'a, content('c), 'd < 'b
```


(● kev -{● | ● | ●}-> ~a -{● | ● | ●}-> ~b) -> (~a, ●) t -{● | ● |}-> unit

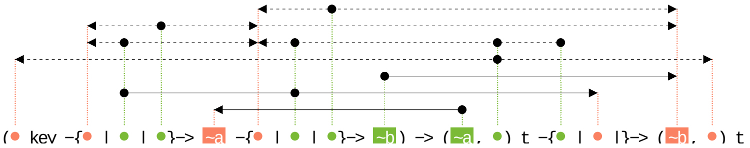
iter f m applies f to all bindings in map m. f receives the key as first argument, and the associated value as second argument. The order in which the bindings are passed to f is unspecified. Only current bindings are presented to f: bindings hidden by more recent bindings are not passed to f.

```
val map : ('a -{'b | 'c | 'd}-> 'e) -> ('a, 'f) t -{'b | 'c |}-> ('e, 'f) t
       with content('c), 'd, 'f < 'b
       and 'd < level('e)
```



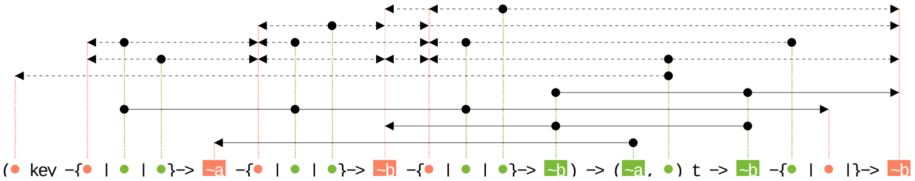
map f m returns a map with same domain as m, where the associated value a of all bindings of m has been replaced by the result of the application of f to a. The order in which the associated values are passed to f is unspecified.

```
val mapi : ('a key -{'b | 'c | 'd}-> 'e -{'f | 'g | 'h}-> 'i) ->
        ('e, 'a) t -{'j | 'g |}-> ('i, 'a) t
        with 'a, content('c), 'd, content('g), 'h, 'j < 'f
        and 'a, content('c), 'd, content('g), 'j < 'b
        and 'c < 'g
        and 'd, 'h < level('i)
```



Same as `Map.S.map`, but the function receives as arguments both the key and the associated value for each binding of the map.

```
val fold : ('a key -{'b | 'c | 'd}->
            'e -{'f | 'c | 'g}-> 'h -{'i | 'c | 'j}-> 'k) ->
           ('e, 'a) t -> 'k -{'l | 'c |}-> 'k
           with 'a, content('c), 'd, 'g, 'j, 'l < 'i
           and 'a, content('c), 'd, 'g, 'l < 'f
           and 'a, content('c), 'd, 'l < 'b
           and 'a, 'd, 'g, 'j < level('h)
           and 'k < 'h
           and 'a, 'd, 'g, 'j < level('k)
```


(● key -{● | ● | ●}-> ~a -{● | ● | ●}-> ~b -{● | ● | ●}-> ~b) -> (~a, ●) t -> ~b -{● | ● |}-> ~b

`fold f m a` computes `(f kN dN ... (f k1 d1 a)...)`, where `k1 ... kN` are the keys of all bindings in `m`, and `d1 ... dN` are the associated data. The order in which the bindings are presented to `f` is unspecified.

```
end
```

```
module Make : functor (Ord : OrderedType) -> S with type 'a key = 'a Ord.t
```

Functor building an implementation of the map structure given a totally ordered type.

5.15 Module `Nativeint`

Processor-native integers.

This module provides operations on the type `nativeint` *of signed 32-bit integers (on 32-bit platforms) or signed 64-bit integers (on 64-bit platforms). This integer type has exactly the same width as that of a* `long` *integer type in the C compiler. All arithmetic operations over* `nativeint` *are taken modulo 2^{32} or 2^{64} depending on the word size of the architecture.*

Performance notice: values of type `nativeint` *occupy more memory space than values of type* `int`*, and arithmetic operations on* `nativeint` *are generally slower than those on* `int`*. Use* `nativeint` *only when the application requires the extra bit of precision over the* `int` *type.*

```
val zero : 'a nativeint
```

● nativeint

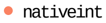
The native integer 0.

```
val one : 'a nativeint
```

● nativeint

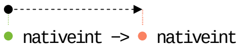
The native integer 1.

```
val minus_one : 'a nativeint
```



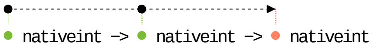
The native integer -1.

```
val neg : 'a nativeint -> 'a nativeint
```



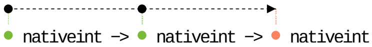
Unary negation.

```
val add : 'a nativeint -> 'a nativeint -> 'a nativeint
```



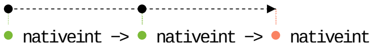
Addition.

```
val sub : 'a nativeint -> 'a nativeint -> 'a nativeint
```



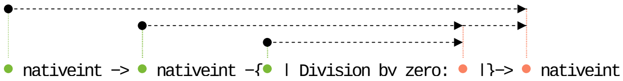
Subtraction.

```
val mul : 'a nativeint -> 'a nativeint -> 'a nativeint
```



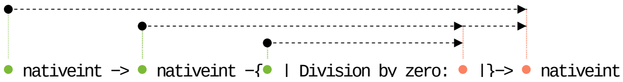
Multiplication.

```
val div : 'a nativeint ->
          'b nativeint -{'c | Division_by_zero: 'c |}-> 'a nativeint
          with 'b < 'a, 'c
```


nativeint -> nativeint -{ | Division by zero: |}-> nativeint

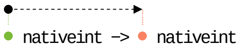
Integer division. Raise Division_by_zero if the second argument is zero. This division rounds the real quotient of its arguments towards zero, as specified for Pervasives.(/).

```
val rem : 'a nativeint ->
          'b nativeint -{'c | Division_by_zero: 'c |}-> 'a nativeint
          with 'b < 'a, 'c
```


nativeint -> nativeint -{ | Division by zero: |}-> nativeint

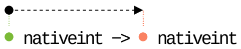
Integer remainder.

```
val succ : 'a nativeint -> 'a nativeint
```



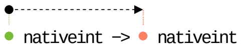
Successor. Nativeint.succ x is Nativeint.add x Nativeint.one.

`val pred : 'a nativeint -> 'a nativeint`



Predecessor. `Nativeint.pred x` is `Nativeint.sub x Nativeint.one`.

`val abs : 'a nativeint -> 'a nativeint`



Return the absolute value of its argument.

`val size : 'a int`

● int

The size in bits of a native integer. This is equal to `32` on a 32-bit platform and to `64` on a 64-bit platform.

`val max_int : 'a nativeint`

● nativeint

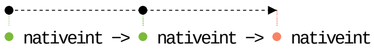
The greatest representable native integer, either 2^{31} - 1 on a 32-bit platform, or 2^{63} - 1 on a 64-bit platform.

`val min_int : 'a nativeint`

● nativeint

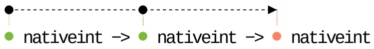
The greatest representable native integer, either -2^{31} on a 32-bit platform, or -2^{63} on a 64-bit platform.

`val logand : 'a nativeint -> 'a nativeint -> 'a nativeint`



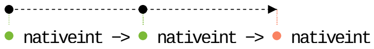
Bitwise logical and.

`val logor : 'a nativeint -> 'a nativeint -> 'a nativeint`



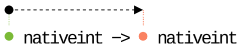
Bitwise logical or.

`val logxor : 'a nativeint -> 'a nativeint -> 'a nativeint`



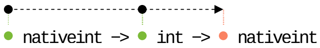
Bitwise logical exclusive or.

`val lognot : 'a nativeint -> 'a nativeint`



Bitwise logical negation

```
val shift_left : 'a nativeint -> 'a int -> 'a nativeint
```



`Nativeint.shift_left x y` shifts `x` to the left by `y` bits. The result is unspecified if `y < 0` or `y >= bitsize`, where `bitsize` is `32` on a 32-bit platform and `64` on a 64-bit platform.

```
val shift_right : 'a nativeint -> 'a int -> 'a nativeint
```

`Nativeint.shift_right x y` shifts `x` to the right by `y` bits. This is an arithmetic shift: the sign bit of `x` is replicated and inserted in the vacated bits. The result is unspecified if `y < 0` or `y >= bitsize`.

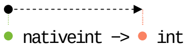
```
val shift_right_logical : 'a nativeint -> 'a int -> 'a nativeint
```

`Nativeint.shift_right_logical x y` shifts `x` to the right by `y` bits. This is a logical shift: zeroes are inserted in the vacated bits regardless of the sign of `x`. The result is unspecified if `y < 0` or `y >= bitsize`.

```
val of_int : 'a int -> 'a nativeint
```

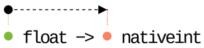
Convert the given integer (type `int`) to a native integer (type `nativeint`).

```
val to_int : 'a nativeint -> 'a int
```



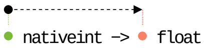
Convert the given native integer (type `nativeint`) to an integer (type `int`). The high-order bit is lost during the conversion.

```
val of_float : 'a float -> 'a nativeint
```



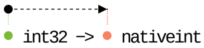
Convert the given floating-point number to a native integer, discarding the fractional part (truncate towards 0). The result of the conversion is undefined if, after truncation, the number is outside the range [`Nativeint.min_int`, `Nativeint.max_int`].

```
val to_float : 'a nativeint -> 'a float
```



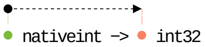
Convert the given native integer to a floating-point number.

```
val of_int32 : 'a int32 -> 'a nativeint
```



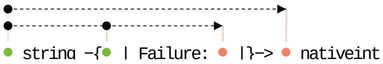
Convert the given 32-bit integer (type `int32`) to a native integer.

```
val to_int32 : 'a nativeint -> 'a int32
```



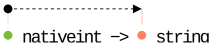
Convert the given native integer to a 32-bit integer (type `int32`). On 64-bit platforms, the 64-bit native integer is taken modulo 2^{32}, i.e. the top 32 bits are lost. On 32-bit platforms, the conversion is exact.

```
val of_string : 'a string -{'b | Failure: 'b |}-> 'a nativeint
                with 'a < 'b
```



Convert the given string to a native integer. The string is read in decimal (by default) or in hexadecimal, octal or binary if the string begins with `0x`, `0o` or `0b` respectively. Raise `Failure "int_of_string"` if the given string is not a valid representation of an integer.

```
val to_string : 'a nativeint -> 'a string
```

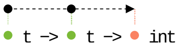


Return the string representation of its argument, in decimal.

```
type (#'a:level) t = 'a nativeint
```

An alias for the type of native integers.

```
val compare : 'a t -> 'a t -> 'a int
```



The comparison function for native integers, with the same specification as `Pervasives.compare`. Along with the type `t`, this function `compare` allows the module `Nativeint` to be passed as argument to the functors `Set.Make` and `Map.Make`.

5.16 Module Pervasives

The initially opened module.

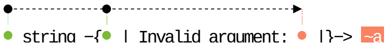
This module provides the built-in types (numbers, booleans, strings, exceptions, references, lists, arrays, input-output channels, ...) and the basic operations over these types.

This module is automatically opened at the beginning of each compilation. All components of this module can therefore be referred by their short name, without prefixing them by `Pervasives`.

```
exception Exit
```

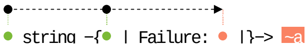
The `Exit` exception is not raised by any library function. It is provided for use in your programs.

```
val invalid_arg : 'a string -{'a | Invalid_argument: 'a |}-> 'b
```



Raise exception `Invalid_argument` with the given string.

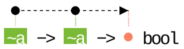
```
val failwith : 'a string -{'a | Failure: 'a |}-> 'b
```



Raise exception `Failure` with the given string.

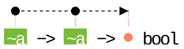
5.16.1 Comparisons

```
val ( = ) : 'a -> 'a -> 'b bool
            with content('a) < 'b
```



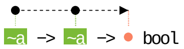
`e1 = e2` tests for structural equality of `e1` and `e2`. Mutable structures (e.g. references and arrays) are equal if and only if their current contents are structurally equal, even if the two mutable objects are not the same physical object. Equality between cyclic data structures may not terminate.

```
val ( <> ) : 'a -> 'a -> 'b bool
             with content('a) < 'b
```



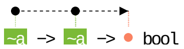
Negation of `Pervasives.=`.

```
val ( < ) : 'a -> 'a -> 'b bool
            with content('a) < 'b
```



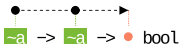
See `Pervasives.>=`.

```
val ( > ) : 'a -> 'a -> 'b bool
            with content('a) < 'b
```



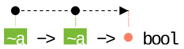
See `Pervasives.>=`.

```
val ( <= ) : 'a -> 'a -> 'b bool
             with content('a) < 'b
```



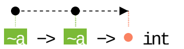
See `Pervasives.>=`.

```
val ( >= ) : 'a -> 'a -> 'b bool
             with content('a) < 'b
```



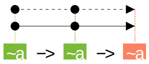
Structural ordering functions. These functions coincide with the usual orderings over integers, characters, strings and floating-point numbers, and extend them to a total ordering over all types. The ordering is compatible with `(=)`. As in the case of `(=)`, mutable structures are compared by contents. Comparison between cyclic structures may not terminate.

```
val compare : 'a -> 'a -> 'b int
              with content('a) < 'b
```



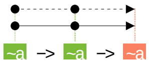
compare x y returns 0 if x=y, a negative integer if x<y, and a positive integer if x>y. The same restrictions as for = apply. compare can be used as the comparison function required by the Set and Map modules.

```
val min : 'a -> 'a -> 'b
          with 'a < 'b
          and content('a) < level('b)
```



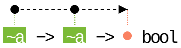
Return the smaller of the two arguments.

```
val max : 'a -> 'a -> 'b
          with 'a < 'b
          and content('a) < level('b)
```



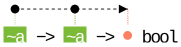
Return the greater of the two arguments.

```
val ( == ) : 'a -> 'a -> 'b bool
             with content('a) < 'b
```



e1 == e2 tests for physical equality of e1 and e2. On integers and characters, it is the same as structural equality. On mutable structures, e1 == e2 is true if and only if physical modification of e1 also affects e2. On non-mutable structures, the behavior of (==) is implementation-dependent, except that e1 == e2 implies e1 = e2.

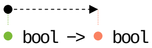
```
val ( != ) : 'a -> 'a -> 'b bool
             with content('a) < 'b
```



Negation of Pervasives.==.

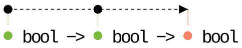
5.16.2 Boolean operations

```
val not : 'a bool -> 'a bool
```



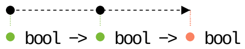
The boolean negation.

```
val ( && ) : 'a bool -> 'a bool -> 'a bool
```



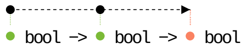
The boolean "and". Evaluation is sequential, left-to-right: in e1 && e2, e1 is evaluated first, and if it returns false, e2 is not evaluated at all.

```
val ( & ) : 'a bool -> 'a bool -> 'a bool
```



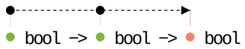
Deprecated. `Pervasives.&&` should be used instead.

```
val ( || ) : 'a bool -> 'a bool -> 'a bool
```



See `Pervasives.or`.

```
val ( or ) : 'a bool -> 'a bool -> 'a bool
```

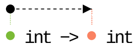


The boolean "or". Evaluation is sequential, left-to-right: in `e1 || e2`, `e1` is evaluated first, and if it returns `true`, `e2` is not evaluated at all.

5.16.3 Integer arithmetic

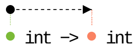
Integers are 31 bits wide (or 63 bits on 64-bit processors). All operations are taken modulo 2^{31} (or 2^{63}). They do not fail on overflow.

```
val ( ~- ) : 'a int -> 'a int
```



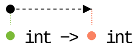
Unary negation. You can also write `-e` instead of `~-e`.

```
val succ : 'a int -> 'a int
```



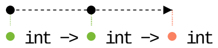
`succ x` is `x+1`.

```
val pred : 'a int -> 'a int
```



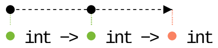
`pred x` is `x-1`.

```
val ( + ) : 'a int -> 'a int -> 'a int
```



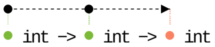
Integer addition.

```
val ( - ) : 'a int -> 'a int -> 'a int
```



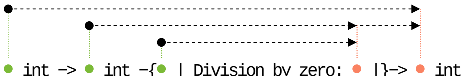
Integer subtraction.

```
val ( * ) : 'a int -> 'a int -> 'a int
```



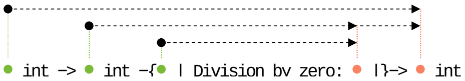
Integer multiplication.

```
val ( / ) : 'a int -> 'b int -{'c | Division_by_zero: 'c |}-> 'a int
            with 'b < 'a, 'c
```



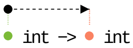
Integer division. Raise `Division_by_zero` if the second argument is 0. Integer division rounds the real quotient of its arguments towards zero. More precisely, if `x >= 0` and `y > 0`, `x / y` is the greatest integer less than or equal to the real quotient of `x` by `y`. Moreover, `(-x) / y = x / (-y) = -(x / y)`.

```
val ( mod ) : 'a int -> 'b int -{'c | Division_by_zero: 'c |}-> 'a int
              with 'b < 'a, 'c
```



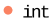
Integer remainder. If `y` is not zero, the result of `x mod y` satisfies the following properties: `x = (x / y) * y + x mod y` and `abs(x mod y) < abs(y)`. If `y = 0`, `x mod y` raises `Division_by_zero`. Notice that `x mod y` is negative if `x < 0`.

```
val abs : 'a int -> 'a int
```



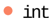
Return the absolute value of the argument.

```
val max_int : 'a int
```



The greatest representable integer.

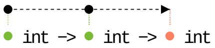
```
val min_int : 'a int
```



The smallest representable integer.

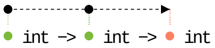
Bitwise operations

```
val ( land ) : 'a int -> 'a int -> 'a int
```



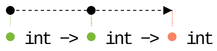
Bitwise logical and.

```
val ( lor ) : 'a int -> 'a int -> 'a int
```



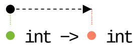
Bitwise logical or.

```
val ( lxor ) : 'a int -> 'a int -> 'a int
```



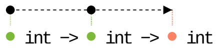
Bitwise logical exclusive or.

```
val lnot : 'a int -> 'a int
```



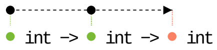
Bitwise logical negation.

```
val ( lsl ) : 'a int -> 'a int -> 'a int
```



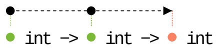
`n lsl m` shifts `n` to the left by `m` bits. The result is unspecified if `m < 0` or `m >= bitsize`, where `bitsize` is `32` on a 32-bit platform and `64` on a 64-bit platform.

```
val ( lsr ) : 'a int -> 'a int -> 'a int
```



`n lsr m` shifts `n` to the right by `m` bits. This is a logical shift: zeroes are inserted regardless of the sign of `n`. The result is unspecified if `m < 0` or `m >= bitsize`.

```
val ( asr ) : 'a int -> 'a int -> 'a int
```

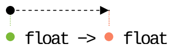


`n asr m` shifts `n` to the right by `m` bits. This is an arithmetic shift: the sign bit of `n` is replicated. The result is unspecified if `m < 0` or `m >= bitsize`.

5.16.4 Floating-point arithmetic

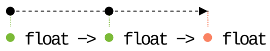
Caml's floating-point numbers follow the IEEE 754 standard, using double precision (64 bits) numbers. Floating-point operations never raise an exception on overflow, underflow, division by zero, etc. Instead, special IEEE numbers are returned as appropriate, such as `infinity` *for* `1.0 /. 0.0`, `neg_infinity` *for* `-1.0 /. 0.0`, *and* `nan` *("not a number") for* `0.0 /. 0.0`. *These special numbers then propagate through floating-point computations as expected: for instance,* `1.0 /. infinity` *is* `0.0`, *and any operation with* `nan` *as argument returns* `nan` *as result.*

```
val ( ~-. ) : 'a float -> 'a float
```



Unary negation. You can also write `-.e` instead of `~-.e`.

```
val ( +. ) : 'a float -> 'a float -> 'a float
```



Floating-point addition

```
val ( -. ) : 'a float -> 'a float -> 'a float
```

● float -> ● float -> ● float

Floating-point subtraction

```
val ( *. ) : 'a float -> 'a float -> 'a float
```

● float -> ● float -> ● float

Floating-point multiplication

```
val ( /. ) : 'a float -> 'a float -> 'a float
```

● float -> ● float -> ● float

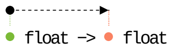
Floating-point division.

```
val ( ** ) : 'a float -> 'a float -> 'a float
```

● float -> ● float -> ● float

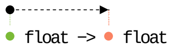
Exponentiation

```
val sqrt : 'a float -> 'a float
```


● float -> ● float

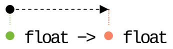
Square root

```
val exp : 'a float -> 'a float
```


● float -> ● float

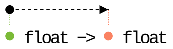
Exponential.

```
val log : 'a float -> 'a float
```


● float -> ● float

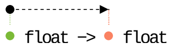
Natural logarithm.

```
val log10 : 'a float -> 'a float
```


● float -> ● float

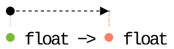
Base 10 logarithm.

```
val cos : 'a float -> 'a float
```


● float -> ● float

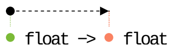
See `Pervasives.atan2`.

```
val sin : 'a float -> 'a float
```



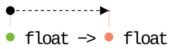
See `Pervasives.atan2`.

```
val tan : ’a float -> ’a float
```



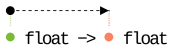
See `Pervasives.atan2`.

```
val acos : ’a float -> ’a float
```



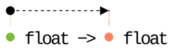
See `Pervasives.atan2`.

```
val asin : ’a float -> ’a float
```



See `Pervasives.atan2`.

```
val atan : ’a float -> ’a float
```

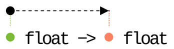


See `Pervasives.atan2`.

```
val atan2 : ’a float -> ’a float -> ’a float
```

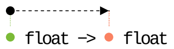
The usual trigonometric functions.

```
val cosh : ’a float -> ’a float
```



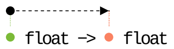
See `Pervasives.tanh`.

```
val sinh : ’a float -> ’a float
```



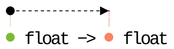
See `Pervasives.tanh`.

```
val tanh : ’a float -> ’a float
```



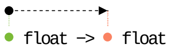
The usual hyperbolic trigonometric functions.

```
val ceil : ’a float -> ’a float
```



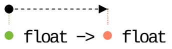
See `Pervasives.floor`.

```
val floor : 'a float -> 'a float
```



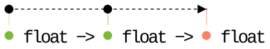
Round the given float to an integer value. `floor f` returns the greatest integer value less than or equal to `f`. `ceil f` returns the least integer value greater than or equal to `f`.

```
val abs_float : 'a float -> 'a float
```



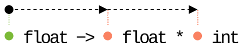
Return the absolute value of the argument.

```
val mod_float : 'a float -> 'a float -> 'a float
```



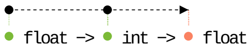
`mod_float a b` returns the remainder of `a` with respect to `b`. The returned value is `a -. n *. b`, where `n` is the quotient `a /. b` rounded towards zero to an integer.

```
val frexp : 'a float -> 'a float * 'a int
```



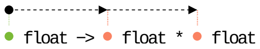
`frexp f` returns the pair of the significant and the exponent of `f`. When `f` is zero, the significant `x` and the exponent `n` of `f` are equal to zero. When `f` is non-zero, they are defined by `f = x *. 2 ** n` and `0.5 <= x < 1.0`.

```
val ldexp : 'a float -> 'a int -> 'a float
```



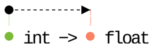
`ldexp x n` returns `x *. 2 ** n`.

```
val modf : 'a float -> 'a float * 'a float
```



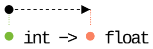
`modf f` returns the pair of the fractional and integral part of `f`.

```
val float : 'a int -> 'a float
```



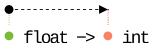
Same as `Pervasives.float_of_int`.

```
val float_of_int : 'a int -> 'a float
```



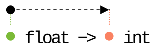
Convert an integer to floating-point.

```
val truncate : 'a float -> 'a int
```



Same as `Pervasives.int_of_float`.

```
val int_of_float : 'a float -> 'a int
```



Truncate the given floating-point number to an integer. The result is unspecified if it falls outside the range of representable integers.

```
val infinity : 'a float
```

● float

Positive infinity.

```
val neg_infinity : 'a float
```

● float

Negative infinity.

```
val nan : 'a float
```

● float

A special floating-point value denoting the result of an undefined operation such as `0.0 /. 0.0`. Stands for “not a number”.

```
val max_float : 'a float
```

● float

The largest positive finite value of type `float`.

```
val min_float : 'a float
```

● float

The smallest positive, non-zero, non-denormalized value of type `float`.

```
val epsilon_float : 'a float
```

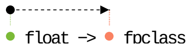
● float

The smallest positive float `x` such that `1.0 +. x <> 1.0`.

```
type (#'a:level) fpclass =
    FP_normal
  | FP_subnormal
  | FP_zero
  | FP_infinite
  | FP_nan # 'a
```

The five classes of floating-point numbers, as determined by the `Pervasives.classify_float` function.

```
val classify_float : 'a float -> 'a fpclass
```

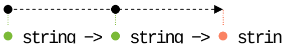


Return the class of the given floating-point number: normal, subnormal, zero, infinite, or not a number.

5.16.5 String operations

More string operations are provided in the modules `String` *(immutable strings) and* `Charray` *(mutable strings).*

```
val ( ^ ) : 'a string -> 'a string -> 'a string
```



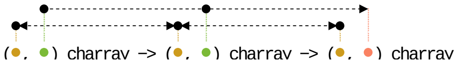
String concatenation.

```
val ( $$ ) : 'a string -> 'a int -> 'a char
```

string -> int -> char

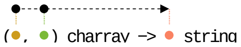
Character access.

```
val ( ^^ ) : ('a, 'b) charray -> ('a, 'b) charray -> ('a, 'b) charray
```



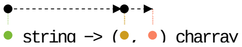
Charray concatenation.

```
val string_of_charray : ('a, 'b) charray -> 'b string
                        with 'a < 'b
```



Coerces a mutable string into an immutable one.

```
val charray_of_string : 'a string -> ('a, 'a) charray
```

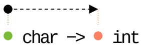


Creates a mutable string from an immutable one.

5.16.6 Character operations

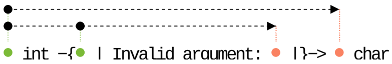
More character operations are provided in module `Char`.

```
val int_of_char : 'a char -> 'a int
```



Return the ASCII code of the argument.

```
val char_of_int : 'a int -{'b | Invalid_argument: 'b |}-> 'a char
                  with 'a < 'b
```



Return the character with the given ASCII code. Raise `Invalid_argument "char_of_int"` if the argument is outside the range 0–255.

5.16.7 Unit operations

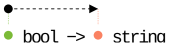
```
val ignore : 'a -> unit
```

~a -> unit

Discard the value of its argument and return `()`. For instance, `ignore(f x)` discards the result of the side-effecting function `f`. It is equivalent to `f x; ()`, except that the latter may generate a compiler warning; writing `ignore(f x)` instead avoids the warning.

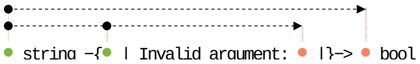
5.16.8 String conversion functions

```
val string_of_bool : 'a bool -> 'a string
```



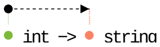
Return the string representation of a boolean.

```
val bool_of_string : 'a string -{'b | Invalid_argument: 'b |}-> 'a bool
                      with 'a < 'b
```



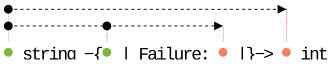
Convert the given string to a boolean. Raise `Invalid_argument "bool_of_string"` if the string is not `"true"` or `"false"`.

```
val string_of_int : 'a int -> 'a string
```



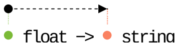
Return the string representation of an integer, in decimal.

```
val int_of_string : 'a string -{'b | Failure: 'b |}-> 'a int
                     with 'a < 'b
```



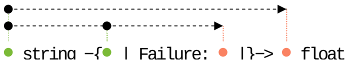
Convert the given string to an integer. The string is read in decimal (by default) or in hexadecimal, octal or binary if the string begins with `0x`, `0o` or `0b` respectively. Raise `Failure "int_of_string"` if the given string is not a valid representation of an integer.

```
val string_of_float : 'a float -> 'a string
```



Return the string representation of a floating-point number.

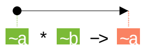
```
val float_of_string : 'a string -{'b | Failure: 'b |}-> 'a float
                       with 'a < 'b
```



Convert the given string to a float. Raise `Failure "float_of_string"` if the given string is not a valid representation of a float.

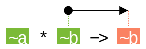
5.16.9 Pair operations

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



Return the first component of a pair.

```
val snd : 'a * 'b -> 'b
```

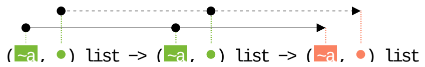


Return the second component of a pair.

5.16.10 List operations

More list operations are provided in module List.

```
val ( @ ) : ('a, 'b) list -> ('a, 'b) list -> ('a, 'b) list
```

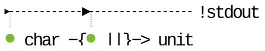


List concatenation.

5.16.11 Input/output

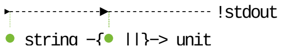
Output functions on standard output

```
val print_char : !stdout char -{!stdout ||}-> unit
```



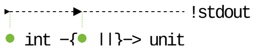
Print a character on standard output.

```
val print_string : !stdout string -{!stdout ||}-> unit
```



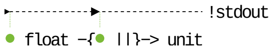
Print a string on standard output.

```
val print_int : !stdout int -{!stdout ||}-> unit
```



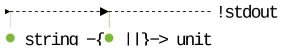
Print an integer, in decimal, on standard output.

```
val print_float : !stdout float -{!stdout ||}-> unit
```



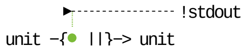
Print a floating-point number, in decimal, on standard output.

```
val print_endline : !stdout string -{!stdout ||}-> unit
```



Print a string, followed by a newline character, on standard output.

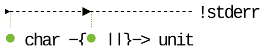
```
val print_newline : unit -{!stdout ||}-> unit
```



Print a newline character on standard output, and flush standard output. This can be used to simulate line buffering of standard output.

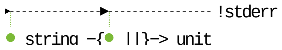
Output functions on standard error

```
val prerr_char : !stderr char -{!stderr ||}-> unit
```



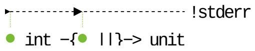
Print a character on standard error.

```
val prerr_string : !stderr string -{!stderr ||}-> unit
```



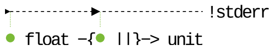
Print a string on standard error.

```
val prerr_int : !stderr int -{!stderr ||}-> unit
```



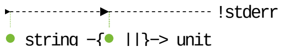
Print an integer, in decimal, on standard error.

```
val prerr_float : !stderr float -{!stderr ||}-> unit
```



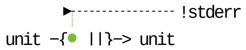
Print a floating-point number, in decimal, on standard error.

```
val prerr_endline : !stderr string -{!stderr ||}-> unit
```



Print a string, followed by a newline character on standard error and flush standard error.

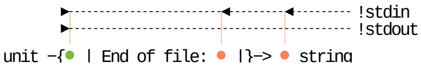
```
val prerr_newline : unit -{!stderr ||}-> unit
```



Print a newline character on standard error, and flush standard error.

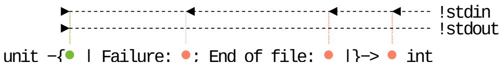
Input functions on standard input

```
val read_line : unit -{[< !stdout, !stdin] | End_of_file: !stdin |}->
                !stdin string
```



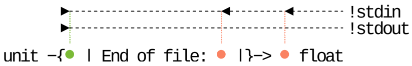
Flush standard output, then read characters from standard input until a newline character is encountered. Return the string of all characters read, without the newline character at the end.

```
val read_int : unit -{[< !stdout,
               !stdin] | Failure: !stdin; End_of_file: !stdin |}-> !stdin int
```



Flush standard output, then read one line from standard input and convert it to an integer. Raise `Failure "int_of_string"` if the line read is not a valid representation of an integer.

```
val read_float : unit -{[< !stdout, !stdin] | End_of_file: !stdin |}->
                 !stdin float
```



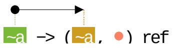
Flush standard output, then read one line from standard input and convert it to a floating-point number. The result is unspecified if the line read is not a valid representation of a floating-point number.

5.16.12 References

```
type (='a:type, #'b:level) ref = { mutable contents : 'a; } # 'b
```

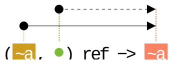
The type of references (mutable indirection cells) containing a value of type `'a`.

```
val ref : 'a -> ('a, 'b) ref
```



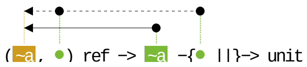
Return a fresh reference containing the given value.

```
val ( ! ) : ('a, 'b) ref -> 'c
            with 'a < 'c
            and 'b < level('c)
```



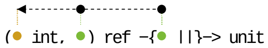
`!r` returns the current contents of reference `r`. Equivalent to `fun r -> r.contents`.

```
val ( := ) : ('a, 'b) ref -> 'a -{'b ||}-> unit
             with 'b < level('a)
```



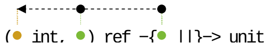
r := a stores the value of a in reference r. Equivalent to fun r v -> r.contents <- v.

```
val incr : ('a int, 'a) ref -{'a ||}-> unit
```



Increment the integer contained in the given reference. Equivalent to fun r -> r := succ !r.

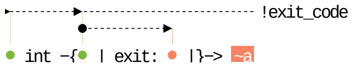
```
val decr : ('a int, 'a) ref -{'a ||}-> unit
```



Decrement the integer contained in the given reference. Equivalent to fun r -> r := pred !r.

5.16.13 Program termination

```
val exit : !exit_code int -{'a | exit: 'a |}-> 'b
           with 'a < !exit_code
```



Flush all pending writes on Pervasives.stdout and Pervasives.stderr, and terminate the process, returning the given status code to the operating system (usually 0 to indicate no errors, and a small positive integer to indicate failure.) An implicit exit 0 is performed each time a program terminates normally (but not if it terminates because of an uncaught exception).

5.17 Module Queue

First-in first-out queues.

This module implements queues (FIFOs), with in-place modification.

```
type (='a:type, ='b:level, #'c:level) t
```

The type of queues containing elements of type 'a.

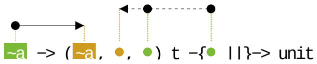
```
exception Empty
```

Raised when Queue.take or Queue.peek is applied to an empty queue.

```
val create : unit -> ('a, 'b, 'c) t
```

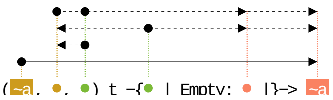
Return a new queue, initially empty.

```
val add : 'a -> ('a, 'b, 'b) t -{'b ||}-> unit
```



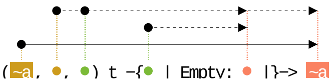
add x q adds the element x at the end of the queue q.

```
val take : ('a, 'b, 'b) t -{'c | Empty: 'd |}-> 'e
           with 'a < 'e
           and 'c < 'b, 'd, level('e)
           and 'b < 'd, level('e)
```



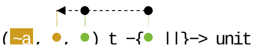
`take q` removes and returns the first element in queue `q`, or raises `Empty` if the queue is empty.

```
val peek : ('a, 'b, 'c) t -{'d | Empty: 'd |}-> 'e
           with 'a < 'e
           and 'c < 'd, level('e)
           and 'b < 'd, level('e)
```



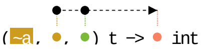
`peek q` returns the first element in queue `q`, without removing it from the queue, or raises `Empty` if the queue is empty.

```
val clear : ('a, 'b, 'b) t -{'b ||}-> unit
```



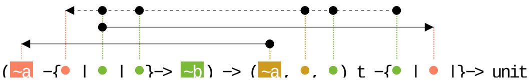
Discard all elements from a queue.

```
val length : ('a, 'b, 'c) t -> 'c int
             with 'b < 'c
```



Return the number of elements in a queue.

```
val iter : ('a -{'b | 'c | 'b}-> 'd) -> ('a, 'e, 'b) t -{'b | 'c |}-> unit
           with content('c), 'e < 'b
```

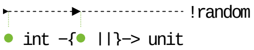


`iter f q` applies `f` in turn to all elements of `q`, from the least recently entered to the most recently entered. The queue itself is unchanged.

5.18 Module Random

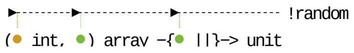
Pseudo-random number generator (PRNG).

```
val init : !random int -{!random ||}-> unit
```



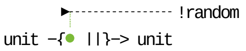
Initialize the generator, using the argument as a seed. The same seed will always yield the same sequence of numbers.

```
val full_init : ([< !random] int, !random) array -{!random ||}-> unit
```



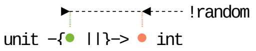
Same as `Random.init` but takes more data as seed.

```
val self_init : unit -{!random ||}-> unit
```



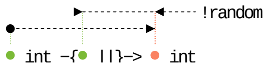
Initialize the generator with a more-or-less random seed chosen in a system-dependent way.

```
val bits : unit -{!random ||}-> !random int
```



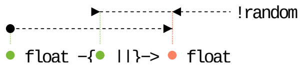
Return 30 random bits in a nonnegative integer.

```
val int : 'a int -{!random ||}-> 'a int
      with !random < 'a
```



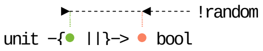
`Random.int bound` returns a random integer between 0 (inclusive) and `bound` (exclusive). `bound` must be more than 0 and less than 2^{30}.

```
val float : 'a float -{!random ||}-> 'a float
        with !random < 'a
```



`Random.float bound` returns a random floating-point number between 0 (inclusive) and `bound` (exclusive). If `bound` is negative, the result is negative. If `bound` is 0, the result is 0.

```
val bool : unit -{!random ||}-> !random bool
```

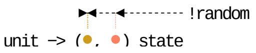


`Random.bool ()` returns `true` or `false` with probability 0.5 each.

```
type (='a:level, #'b:level) state
```

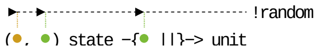
Values of this type are used to store the current state of the generator.

```
val get_state : unit -> (!random, !random) state
```



Returns the current state of the generator. This is useful for checkpointing computations that use the PRNG.

```
val set_state : ([< !random], !random) state -{!random ||}-> unit
```

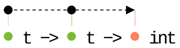


Resets the state of the generator to some previous state returned by `Random.get_state`.

5.19 Module Set

Module type OrderedType

```
module type OrderedType = sig

    type (#'a:level) t

    val compare : 'a t -> 'a t -> 'a int
```

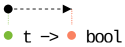


```
end
```

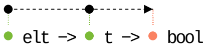
Module type S

```
module type S = sig

    type (#'a:level) elt

    type (#'a:level) t

    val empty : 'a t
```

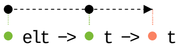
```
    val is_empty : 'a t -> 'a bool
```



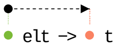
```
    val mem : 'a elt -> 'a t -> 'a bool
```



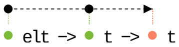
```
    val add : 'a elt -> 'a t -> 'a t
```



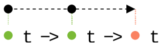
```
    val singleton : 'a elt -> 'a t
```



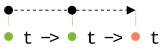
```
    val remove : 'a elt -> 'a t -> 'a t
```



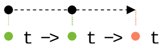
```
    val union_ : 'a t -> 'a t -> 'a t
```



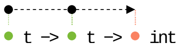
```
val inter_ : 'a t -> 'a t -> 'a t
```



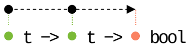
```
val diff : 'a t -> 'a t -> 'a t
```



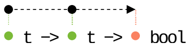
```
val compare : 'a t -> 'a t -> 'a int
```



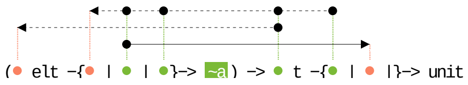
```
val equal : 'a t -> 'a t -> 'a bool
```



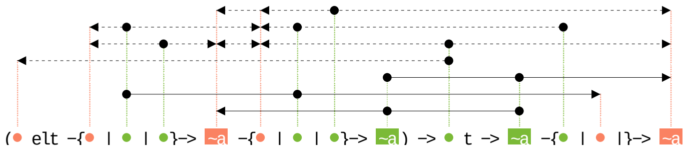
```
val subset : 'a t -> 'a t -> 'a bool
```



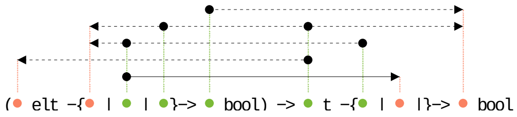
```
val iter : ('a elt -{'b | 'c | 'b}-> 'd) -> 'a t -{'b | 'c |}-> unit
           with 'a, content('c) < 'b
```



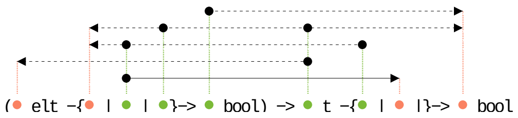
```
val fold : ('a elt -{'b | 'c | 'd}-> 'e -{'f | 'c | 'g}-> 'h) ->
           'a t -> 'h -{'i | 'c |}-> 'h
           with 'a, content('c), 'd, 'g, 'i < 'f
           and 'a, content('c), 'd, 'i < 'b
           and 'a, 'd, 'g < level('e)
           and 'h < 'e
           and 'a, 'd, 'g < level('h)
```



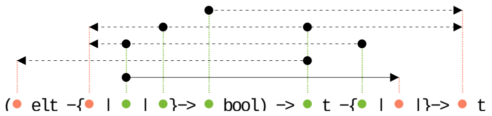
```
val for_all : ('a elt -{'b | 'c | 'd}-> 'e bool) ->
              'a t -{'b | 'c |}-> 'e bool
              with 'a, 'd < 'e
              and 'a, content('c), 'd < 'b
```



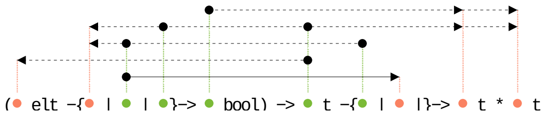
```
    val exists : ('a elt -{'b | 'c | 'd}-> 'e bool) ->
                 'a t -{'b | 'c |}-> 'e bool
                 with 'a, 'd < 'e
                 and 'a, content('c), 'd < 'b
```



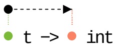
```
    val filter : ('a elt -{'b | 'c | 'd}-> 'e bool) -> 'a t -{'b | 'c |}-> 'e t
                 with 'a, 'd < 'e
                 and 'a, content('c), 'd < 'b
```



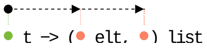
```
    val partition : ('a elt -{'b | 'c | 'd}-> 'e bool) ->
                    'a t -{'b | 'c |}-> 'e t * 'e t
                    with 'a, 'd < 'e
                    and 'a, content('c), 'd < 'b
```



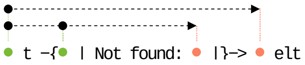
```
    val cardinal : 'a t -> 'a int
```



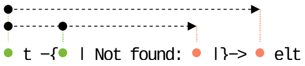
```
    val elements : 'a t -> ('a elt, 'a) list
```



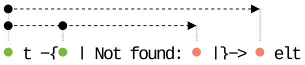
```
    val min_elt : 'a t -{'b | Not_found: 'b |}-> 'a elt
                  with 'a < 'b
```



```
    val max_elt : 'a t -{'b | Not_found: 'b |}-> 'a elt
                  with 'a < 'b
```



```
    val choose : 'a t -{'b | Not_found: 'b |}-> 'a elt
                 with 'a < 'b
```



```
end


module Make : functor (Ord : OrderedType) -> S with type 'a elt = 'a Ord.t
```

5.20 Module Stack

Last-in first-out stacks.

This module implements stacks (LIFOs), with in-place modification.

```
type (='a:type, ='b:level, #'c:level) t
```

The type of stacks containing elements of type `'a`.

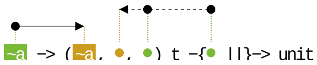
```
exception Empty
```

Raised when `Stack.pop` or `Stack.top` is applied to an empty stack.

```
val create : unit -> ('a, 'b, 'c) t
```

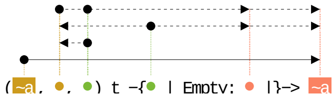
Return a new stack, initially empty.

```
val push : 'a -> ('a, 'b, 'b) t -{'b ||}-> unit
```



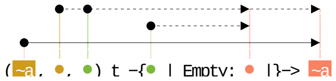
`push x s` adds the element `x` at the top of stack `s`.

```
val pop : ('a, 'b, 'b) t -{'c | Empty: 'd |}-> 'e
          with 'a < 'e
          and 'c < 'b, 'd, level('e)
          and 'b < 'd, level('e)
```



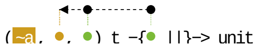
`pop s` removes and returns the topmost element in stack `s`, or raises `Empty` if the stack is empty.

```
val top : ('a, 'b, 'c) t -{'d | Empty: 'd |}-> 'e
          with 'a < 'e
          and 'c < 'd, level('e)
          and 'b < 'd, level('e)
```



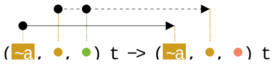
`top s` returns the topmost element in stack `s`, or raises `Empty` if the stack is empty.

```
val clear : ('a, 'b, 'b) t -{'b ||}-> unit
```



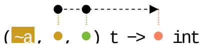
Discard all elements from a stack.

```
val copy : ('a, 'b, 'c) t -> ('d, 'c, 'e) t
           with 'a < 'd
           and 'b < 'c
```



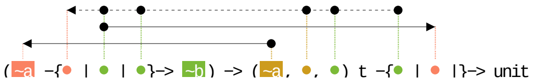
Return a copy of the given stack.

```
val length : ('a, 'b, 'c) t -> 'c int
             with 'b < 'c
```



Return the number of elements in a stack.

```
val iter : ('a -{'b | 'c | 'b}-> 'd) -> ('a, 'e, 'b) t -{'b | 'c |}-> unit
           with content('c), 'e < 'b
```

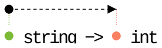


`iter f s` applies `f` in turn to all elements of `s`, from the element at the top of the stack to the element at the bottom of the stack. The stack itself is unchanged.

5.21 Module String

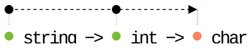
String operations.

```
val length : 'a string -> 'a int
```



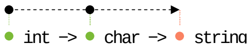
Return the length (number of characters) of the given string.

```
val get : 'a string -> 'a int -> 'a char
```



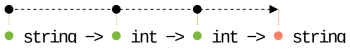
`String.get s n` returns character number `n` in string `s`. The first character is character number 0. The last character is character number `String.length s - 1`. **Terminate the program** if `n` is outside the range 0 to `(String.length s - 1)`. You can also write `s.[n]` instead of `String.get s n`.

```
val make : 'a int -> 'a char -> 'a string
```



`String.make n c` returns a fresh string of length `n`, filled with the character `c`. **Terminate the program** if `n < 0` or `n > Sys.max_string_length`.

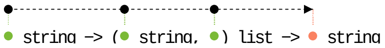
```
val sub : 'a string -> 'a int -> 'a int -> 'a string
```



`String.sub s start len` returns a fresh string of length `len`, containing the characters number `start` to `start + len - 1` of string `s`.

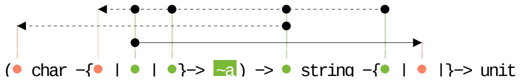
Terminate the program if `start` and `len` do not designate a valid substring of `s`; that is, if `start < 0`, or `len < 0`, or `start + len > String.length s`.

```
val concat : 'a string -> ('a string, 'a) list -> 'a string
```



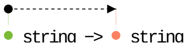
String.concat sep sl concatenates the list of strings sl, inserting the separator string sep between each.

```
val iter : ('a char -{'b | 'c | 'b}-> 'd) -> 'a string -{'b | 'c |}-> unit
         with 'a, content('c) < 'b
```



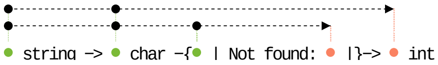
String.iter f s applies function f in turn to all the characters of s. It is equivalent to f s.(0); f s.(1); ...; f s.(String.length s - 1); ().

```
val escaped : 'a string -> 'a string
```



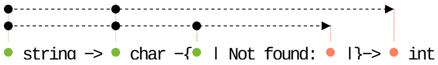
Return a copy of the argument, with special characters represented by escape sequences, following the lexical conventions of Objective Caml. If there is no special character in the argument, return the original string itself, not a copy.

```
val index : 'a string -> 'a char -{'b | Not_found: 'b |}-> 'a int
          with 'a < 'b
```



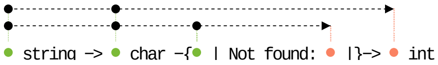
String.index s c returns the position of the leftmost occurrence of character c in string s. Raise Not_found if c does not occur in s.

```
val rindex : 'a string -> 'a char -{'b | Not_found: 'b |}-> 'a int
           with 'a < 'b
```



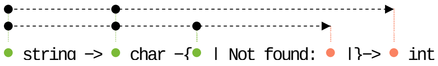
String.rindex s c returns the position of the rightmost occurrence of character c in string s. Raise Not_found if c does not occur in s.

```
val index_from : 'a string -> 'a char -{'b | Not_found: 'b |}-> 'a int
               with 'a < 'b
```



Same as String.index, but start searching at the character position given as second argument. String.index s c is equivalent to String.index_from s 0 c.

```
val rindex_from : 'a string -> 'a char -{'b | Not_found: 'b |}-> 'a int
                with 'a < 'b
```



Same as `String.rindex`, but start searching at the character position given as second argument.

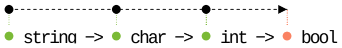
`String.rindex s c` is equivalent to `String.rindex_from s (String.length s - 1) c`.

```
val contains : 'a string -> 'a char -> 'a bool
```

string -> char -> bool

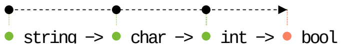
`String.contains s c` tests if character `c` appears in the string `s`.

```
val contains_from : 'a string -> 'a char -> 'a int -> 'a bool
```



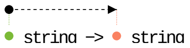
`String.contains_from s start c` tests if character `c` appears in the substring of `s` starting from `start` to the end of `s`. **Terminate the program** if `start` is not a valid index of `s`.

```
val rcontains_from : 'a string -> 'a char -> 'a int -> 'a bool
```



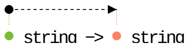
`String.rcontains_from s stop c` tests if character `c` appears in the substring of `s` starting from the beginning of `s` to index `stop`. **Terminate the program** if `stop` is not a valid index of `s`.

```
val uppercase : 'a string -> 'a string
```



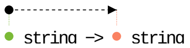
Return a copy of the argument, with all lowercase letters translated to uppercase, including accented letters of the ISO Latin-1 (8859-1) character set.

```
val lowercase : 'a string -> 'a string
```



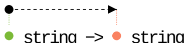
Return a copy of the argument, with all uppercase letters translated to lowercase, including accented letters of the ISO Latin-1 (8859-1) character set.

```
val capitalize : 'a string -> 'a string
```



Return a copy of the argument, with the first letter set to uppercase.

```
val uncapitalize : 'a string -> 'a string
```

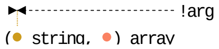


Return a copy of the argument, with the first letter set to lowercase.

5.22 Module Sys

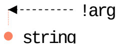
System interface.

```
val argv : (!arg string, 'a) array
```



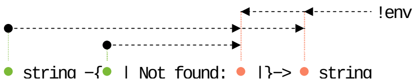
The command line arguments given to the process. The first element is the command name used to invoke the program. The following elements are the command-line arguments given to the program.

```
val executable_name : !arg string
```



The name of the file containing the executable currently running.

```
val getenv : 'a string -{'b | Not_found: 'b |}-> 'c string
             with !env < 'c
             and !env < 'b
             and 'a < 'b, 'c
```

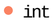


Return the value associated to a variable in the process environment. Raise `Not_found` if the variable is unbound.

```
val os_type : 'a bool
```

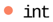
Operating system currently executing the Caml program. One of - `"Unix"` (for all Unix versions, including Linux and Mac OS X), - `"Win32"` (for MS-Windows, OCaml compiled with MSVC++ or Mingw), - `"Cygwin"` (for MS-Windows, OCaml compiled with Cygwin), - `"MacOS"` (for MacOS 9).

```
val word_size : 'a int
```



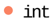
Size of one word on the machine currently executing the Caml program, in bits: 32 or 64.

```
val max_string_length : 'a int
```



Maximum length of a string.

```
val max_array_length : 'a int
```



Maximum length of an array.

```
val ocaml_version : 'a string
```

`ocaml_version` is the version of Objective Caml.

It is a string of the form `"major.minor[additional-info]"` Where major and minor are integers, and `additional-info` is a string that is empty or starts with a '+'.

Appendix

Bibliography

[LDG$^+$02a] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system, documentation and user's manual. http://caml.inria.fr/ocaml/htmlman/, 2002.

[LDG$^+$02b] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system, release 3.06. http://caml.inria.fr/, 2002.

[MNZZ01] Andrew C. Myers, Nathaniel Nystrom, Lantian Zheng, and Steve Zdancewic. Jif: Java + information flow. http://www.cs.cornell.edu/jif/, September 2001.

[MTHM97] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[Mye99] Andrew C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, January 1999. Technical Report MIT/LCS/TR-783. http://www.cs.cornell.edu/andru/release/tr783.ps.gz.

[PS02] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM Symposium on Principles of Programming Languages*, pages 319–330, Portland, Oregon, January 2002. ACM Press. http://cristal.inria.fr/~simonet/publis/fpottier-simonet-popl02.ps.gz.

[PS03] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Transactions on Programming Languages and Systems*, 25(1):117–158, January 2003. http://cristal.inria.fr/~simonet/publis/fpottier-simonet-toplas.ps.gz.

[Sim02] Vincent Simonet. *Dalton*, an efficient implementation of type inference with structural subtyping. http://cristal.inria.fr/~simonet/soft/dalton/, October 2002.

[Sim03] Vincent Simonet. Type inference with structural subtyping: A faithful formalization of an efficient constraint solver. Submitted for publication. http://cristal.inria.fr/~simonet/publis/simonet-structural-subtyping.ps.gz, March 2003.