Flow Caml in a Nutshell

Vincent Simonet
INRIA Rocquencourt
Vincent.Simonet@inria.fr

Abstract

Flow Caml is an extension of the Objective Caml language with a type system tracing information flow. It automatically checks information flow within Flow Caml programs, then translates them to regular Objective Caml code that can be compiled by the ordinary compiler to produce secure programs. In this paper, we give a short overview of this system, from a practical viewpoint.

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 *security* or the *integrity* of 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 location. 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, the Flow Caml language is not exactly a subset of the Caml language: because the type system is extended with some security annotations, the type part has to been extended to deal with them. 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/`

For the time being, only source distribution is available. Compilation requires GNU Make and the last version of Objective Caml (available at `http://caml.inria.fr/`). The system should run on almost every UNIX platform.

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 [Sim03b]. 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 requires 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 an integrated programming environment.

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 and 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 has monomorphic types and 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.

2 A taste of Flow Caml

In this section, we give a taste of the Flow Caml language and type system through some small examples. Some experience with programming in the Caml language (or possibly another functional language based on the ML type system, such as SML or Haskell) is assumed. 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].

Our examples reproduce a session of the interactive toplevel: the user types Flow Caml phrases (typeset in `roman typewriter` in this paper), terminated by `;;`, in response to the `#` prompt. The system type-checks them on the fly and prints the inferred type scheme (typeset in *`slanted typewriter`*).

2.1 Security levels

In Flow Caml, ML types are annotated with *security levels* for tracing information flow. Consider this first definition:

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

It 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 3).

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 `!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 `!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 or *principals* (e.g. file i/o or networking) and will be discussed in section 3 of the current tutorial. For the time being, we will only use them in a somewhat artificial way, 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: `'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`.

2.2 An example of data structure: lists

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. They are often recursive, but this does not raise any particular difficulty concerning typing. Here is the function which calculates the length of a list:

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

In this type, the security annotation of the integer 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: the function reveals information about the structure of the list, 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
```

There is naturally nothing magic with lists in Flow Caml, they can be defined as every algebraic datatype by a **`type`** declaration:

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

As in Caml, the definition lists all the possible forms of a value of type `('a, 'b) list`: it is either the constant `[]` or the constructor `::` with two arguments: a head of type `'a` and a tail of type `('a, 'b) list`. The fourth line of the declaration, `# 'b` tells that `'b` is the security level attached to the knowledge of the form of the variant, i.e. whether it is `[]` or `::`. When entering this definition, the system outputs the signature of the type constructor `list`, which mentions the kind of the parameters and their variances: `+'a:type` means that the first parameter must be a type and is covariant, `#'b:level` tells that the second one is a security level which is covariant and guarded (`#` is a distinguished form of `+` which, in short, records `'b` to appear in the `# 'b` clause of the datatype declaration).

2.3 Subtyping and constraints

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 it 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, which give a directed view of programs.

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` constrains `'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 on different integers:

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

From a type-theoretic point of view, the type scheme inferred for `f1` means that this function has every instance of `'a int -> 'a int * 'b int` which satisfies the inequalities `'a < 'b` and `!alice < 'b` as type. This statement cannot be expressed so 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, the function `f1` cannot accept as argument an integer whose level is not known to be less than or equal to `!alice`, e.g. `x2`. 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 two following constraints deal similarly with the second and third arguments of the function, respectively. When one applies the function on the three constants `x1`, `x2` and `x3`, the constraints allow to compute the respective levels of the produced integer:

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

2.4 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, records with mutable fields, usual **`while`** and **`for`** loops, as well as exceptions.

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.)

References In Flow Caml, the type constructor for references, `ref`, has two arguments: the first one is the type of the value stored in the reference; and the second one is a security level which 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 a reference `r1` whose content is declared to be an integer of levels `!alice`:

```
let r1 : (!alice int, 'a) ref = ref 0
val r1 : (!alice int, 'a) ref
```

In the above example, the content of reference `r1` has type `!alice int`. This means it may receive any integer whose security level is less than or equal to `!alice`, i.e. an integer *Alice* is allowed to read. The integer `x1` has level `!alice`. Hence it can legally be stored in `r1`:

```
r1 := x1;;
- : 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. As a result, the `unit` type constructor does not carry any security annotation. On the contrary, the integer `x2` has been declared with the level `!bob`. Because information flow from `!bob` to `!alice` is not allowed (see section 3), assigning it to `r1` raises a typing error:

```
r1 := x2;;
This expression generates the following information flow(s):
  from !bob to !alice
which are not legal.
```

Similarly, the reference `r1` can be updated in a context whose execution depends on `x1` but not `x2`:

```
if x1 = 0 then r1 := 0 else r1 := 42;;
- : unit
if x2 = 0 then r1 := 0 else r1 := 42;;
This expression generates the following information flow(s):
  from !bob to !alice
which are not legal.
```

Lastly, reading the content of `r1` naturally yields an integer of level `!alice`:

```
!r1;;
- : !alice int
```

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 (* any expression of type !alice bool *) then
  ... (* this branch is type-checked at level !alice *)
else
  if (* any expression of type !bob bool *) then
    ... (* this branch is type-checked at level [> !alice, !bob] *)
  else
    ... (* this branch is type-checked 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.

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. For this purpose, functions types carry a security level which is a lower bound on the effects of the function and an upper bound on the contexts where it can be applied (in the previous examples, there was no constraint on this bound, so it was omitted by the system for the sake of readability). 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. 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. 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. To conclude this overview of references, let us mention that the `ref` type is a case of record datatype with a single mutable field, as in Objective Caml:

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

Exceptions We now briefly 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. 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 ... reraise`**.

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`. 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 (once again, every function we have defined up to now did not raise exceptions, so rows were omitted in arrows types printed by the system). 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 liable to raise exceptions of different names, its row mentions 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`.

Lastly, 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.

This puts an end to this short guided tour of Flow Caml. By lack of space, we naturally omit many features of the language, among whose one may mention pattern-matching, datatypes declarations and the module layer.

3 Writing programs in Flow Caml

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 discuss 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 previous examples, *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.

3.1 The example of the standard input and output

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 current security policy, legal:

```
print_int x1;;
This expression generates the following information flow(s):
  from !alice to !stdout
which are 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. However, 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` 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 relates 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 flows are possible.

3.2 Modeling *principals*

Real programs are liable to communicate with external entities throughout 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 one or several functions for interacting with them, which are implemented in one or several regular Caml modules, using for instance the standard i/o stuff, 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.

3.3 Future work

We plan to extend the Flow Caml system with existential and universal datatypes, in the style of Odersky and Laüfer [OL92]. Indeed, we observed that, because of the presence of security annotations, many programs require to aggregate values of different types, as it is illustrated by the following example. Assume we are modeling the security policy of a bank, where every client stands for a distinct principal. The information concerning each client can be stored in a record, which may be defined as follows:

```
type 'a client_info = {
  cash: 'a int;
  send_message: 'a channel;
  ...
}
```

This definition is identical to that we would have in Caml, with the difference that the record type `client_info` must carry one parameter which is the security level of the client. This security level also appears in the record components types: the field `cash` stores the current balance of the client's account, hence it has the type of an integer labeled by the client's security level. Similarly, `send_message` is an output channel which allows sending numeric messages to the given client. The bank's security policy naturally requires that each client can only receive information about its personal situation, so information sent in the channel `send_message` must have (at most) the security level `'a`. In Caml, all records storing information about clients would have the same unannotated type, `client_info`, so it would be possible to store the clients file in some data structure, such as a list of type `client_info list`. On the contrary, in Flow Caml, records corresponding to distinct clients have incompatible types, because their annotations differ. As a consequence, the clients file forms a heterogeneous set of records which is not representable. Making the client's security level existentially quantified in the declaration is an elegant solution to overcome this problem: because it no longer appears as argument of the type `client_info`, it is again possible to give to all client records the same type and build a list of them.

For this purpose, we recently designed an extension of the HM(X) framework with such existential (and at the same time universal) types [Sim03a] and proposed a type inference algorithm for the case of subtyping featured by Flow Caml. This remains to be implemented and experimented with.

References

[LDG+02a] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system, documentation and user's manual. `http://caml/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.

[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`.

[OL92] Martin Odersky and Konstantin Laüfer. An extension of ML with first-class abstract types. In *Proceedings of the 1992 ACM Workshop on ML and its Applications*, pages 78–91, June 1992. `ftp://ftp.math.luc.edu/pub/laufer/papers/ml+extypes.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.

[Sim03a] Vincent Simonet. An extension of HM(X) with existential and universal datatypes. To appear, mar 2003.

[Sim03b] 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`, February 2003.