Chapter 5

Type safe value I/O

5.1 Introduction

Type safe value input and output using extensional polymorphism

All operating systems provide basic input/output facilities that only manipulate elementary pieces of information, namely bytes or characters. Using this simple I/O's, high-level programming languages provide read/write accesses to various devices, including files, screens and keyboards. Usually, the system I/O's are directly reflected into the programmer's interface: the input/output abstractions just handle characters, or at best provide direct input/output of elementary data such as integers or strings. This is far from being satisfactory: in modern programming languages, the programmer can easily define complex data structures; surprisingly, there is no help from the language to input/output those data. The programmer has to write his own specific routines, namely a conversion function from structured data to a stream of characters at output, and the converse conversion from characters to structured data at input. This is a large, complex, and error prone effort. It is also basically not reusable: this code is specific to the data type at hand.

So we claim that the language should also provide a high-level input/output interface that handles directly the complex data that the programmer manipulates. We name "value I/O" such an advanced input/output facility. This new kind of I/O rises new kind of problems, notably safety problems, since a structured value written to a file must be read back as a value of the same type. Or at least of a type somewhat "related" to its original type. Hence, we need to know what is the type of a value when we read it back.

We study here the way we must output a structured value, such as to ensure that the value can be safely read back. In our type safe value I/O system combining an untyped ML value marshaler (Objective Caml unsafe value I/O interface) and the primitives for dynamic values, output values are written with some type information and this type information is used at input to check if the type expected by the type context is safely acceptable.

To exchange values between two or more programs that may not share the same data type definitions, the type information has to be written in some canonical way that can be understood by any program. The simple run time types which attached with the values in dynamic values in the previous chapter are not appropriate for this purpose, since the data types in the run time types are identified just by their names. Such identification works only inside one program, or between different invocation of the some identical program. It is no longer correct between different programs. Therefore, we modify the design of the run time types so that it works correctly through different programs. The new run time types, called *external types* does not contain names of data types, but their digests of data types. The data type digests are encrypted check-sums constructed from type definitions without names in Objective Caml; the names in type definitions do not affect internal value representation of data types. This defines an interesting notion of isomorphism between data type definitions, that captures the notion of being equivalent with respect to external value representation.

Outline of this chapter

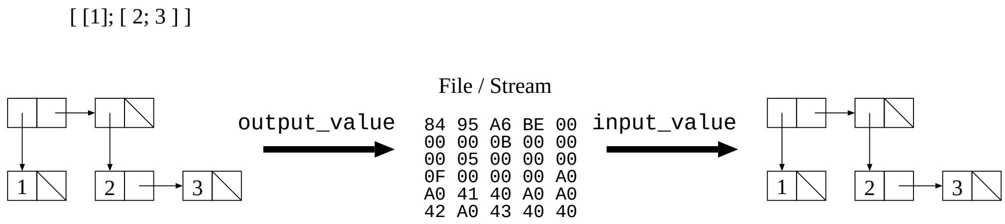
Section 5.2 introduces the notion of value I/O and explains the risk of value I/O without run time type checking, and we present our strategy in Section 5.3. In Section 5.4, we define external types, special type information for value I/O which identify types universally among different programs. Finally in Section 5.5, we define the primitives for value I/O by combining external types and the mechanism of value-type pair encapsulation and decapsulation of dynamic values.

5.2 Basic unsafe value I/O

Objective Caml provides an unsafe value I/O facility, by means of two primitives with the following profile:

$$\begin{array}{ll} \texttt{output\_value :} & \forall\alpha.\texttt{out\_channel} \to \alpha \to \texttt{unit} \\ \texttt{input\_value :} & \forall\alpha.\texttt{in\_channel} \to \alpha \end{array}$$

`output_value` creates a representation of an arbitrary structured value and write it to a file. Conversely, `input_value` decodes the representation and rebuilds in the memory the same structure as the original (preserving sharing).



However, when a value is loaded, there is no way to check whether it has the expected type: input and output types are unrelated, and this way of communicating is essentially untyped. So, this mechanism is unsafe: for instance an integer can be read back as a string, and this leads to a run time error.

```
# let oc = open_out_bin "foo.dat";;
# output_value oc 1;;
...
# let ic = open_in "foo.dat";;
# print_string (input_value ic);;
```
$\Longrightarrow$ **Segmentation fault**

These primitives do not only serve raw ML data saving in files, but allow a simple kind of communication between two or more Objective Caml programs. In this case the type-checking problem is even worse, since a program may erroneously read in a value of a type that is not even defined in the local type context of the reading program. This makes communicating between two or more programs by using `output_value` and `input_value` difficult and error prone.

Our system isolates the user from those unsafe primitives, building a new typed value exchange layer on top of this untyped layer.

5.3 Our Goal

To build a reliable value communication system between different ML programs, we have to connect value I/O with type checking, in order to get a type-safe value I/O facility.

5.3.1 Tag checking at run time is sub-optimal

One of the strongest solution to this problem is to perform run time type checking of the input values by checking the tags of the structures to prove that they are legal value of the type we expect at input[1]. This value I/O with "run time tag checking" rejects all illegal inputs which may crash programs, and is yet completely compatible with the original unsafe version for all legal inputs, since the output data do not require additional run time type information. Unfortunately, this scheme implies that programs have to keep all the definitions of data types, and in addition, we have to recursively check all the tags contained in values. This is indeed inefficient for complicated data types and values, so we designed a more efficient strategy, that is compatible with fast and simple reading of output values.

5.3.2 Exporting with types

Instead of performing this complex and recursive type check of the input values, we may take advantage of the property that output values were once well-typed in writer programs. It is unnecessary to type-check input values again from scratch. When we *export* a value, we also export a witness of well-typedness: encapsulate the value with its type information called *external types* like a value-type pair of dynamic value, and write this pair outside of programs. Then we read back this external type at input, and check whether the external type matches the type expected by the

[1] IoXML[dR] is such a system implemented recently on Objective Caml. It provides printer and parser functions of data types to and from XML[W3C00] data. Lacking of special typing for auto generation of the printers and parsers, users must specify the types they want to export and import manually.

context at *import* time like coercion of dynamic values. We use the words import/export instead of input/output, because our value I/O system has a strong type checking discipline, reminiscent to an airport custom in the real world.

5.4 External types

5.4.1 External types are not run time types

In Chapter 4, we argued an implementation of the dynamic values over extensional polymorphism. Even though they provide a mechanism for encapsulation and decapsulation of values with their types which we also need here, their type information — the run time type representation — cannot directly be used for the safe value I/O which is essentially an inter-program communication, because of the data types.

The data type identification in the dynamic value coercion was done just by its name in Figure 4.1 and it works perfectly inside one program, or between different invocations of the same one program. But once we think about the communication between different ML programs, this identification is no longer valid; types in different programs are not normally identical even if they share the same name. Therefore it cannot achieve the safety of the value I/O just by combining the primitives of the dynamic values and the unsafe value I/O. Still invalid value inputs can crash programs:

Program A

```
type t = Foo | Bar

let oc = open_out_bin "foo.dat" in
output_value oc (dynamic Foo);
close_out oc;;
```

Program B

```
type t = {x : int; y : int}

let ic = open_in_bin "foo.dat" in
coerce (input_value ic) with
| v : t ⇒ print_int v.x;
          print_int v.y
;;
```

Program A output a representation of the value `Foo` to a data file, encapsulating it as a dynamic value `(Foo,t)`. Then a different program B tries to input this dynamic value as its own `t`. Since both data types in the program A and B have the same name `t`, the output data is *wrongly* granted as a value of type `t` of B and the value representation of `Foo` of Program A is used as one for a record type `t` of Program B. It is clearly unsafe and crashes B, since the internal representations of the both data types are incompatible.

Thus, we cannot use the data type identification by their names of the run time types. We can still use the same value-type encapsulation mechanism of dynamic values, but different type information with new representations of data types is required for our purposes, which provides universal identification over programs.

5.4.2 Data type identification by definitions

Data type names (paths) of run time types are not unique universal identifiers for data types. For example, if two programs P and Q are communicating, there is no guarantee that type `t` of P and `t` of Q are equivalent, since they have been defined independently. For value I/O, data types must not be identified by their names, but by their semantics, or more simply by their definitions.

Internally equivalent type definitions

On the other hand, the same data type structures may have different names and different labels or constructors, while being equivalent. For example, the following two type definitions represent the same data structure for trees, even if their names are different.

Program E:

```
type α tree =
| Leaf of α
| Branch of α tree list
```

Program F:

```
type α arbre =
| Feuille of α
| Branche of α arbre list
```

In Objective Caml, the internal representations of these two types are equal[2]. Therefore, the output value of an `int tree` is safely input as a value of type `int arbre` and vice versa. We call such types *internally equivalent* each other. For type safety of value I/O, these differences of type names can be ignored. Only structural equivalence of type definitions without names specifies compatibility of internal value representations.

Is this kind of "confusion" of data types practical for value I/O? We definitively believe so: in our experience of large scale application programming with some sort of data base, we often use the unsafe version of value I/O to save data structures in files because the value I/O get rid of the boring effort of writing printer functions for saving and parsers for loading. Sometimes, it occurs that names of types, labels, modules which defines data types, etc. are modified through application version-updates, but we can still use the data files created by the previous versions of the application, as far as the internal data representation is not modified.

In the cases where this practical confusion of types would not be preferred, we could integrate type and label names into our system without extra cost to make the value I/O more strict, so that only types with exactly the same type definitions would be equivalent. We formalize these ideas with the minimum requirements for safe value I/O, by ignoring type names.

We now formalize these ideas.

[2] Of course, this is not always true for other functional language implementations.

5.4.3 Nameless data type definitions and data type digests

In Objective Caml, using the following *data type definitions*, we can define variants and records, sums and products of types with labels, as user-defined data types:

typedefs	::=	`type` *typedef* `and` ... `and` *typedef*	Data type declarations
typedef	::=	$(\alpha_1, \ldots, \alpha_m)$ `t` $=$ *defbody*	Type definitions
defbody	::=	C `of` $\tau \mid \ldots \mid C$ `of` τ	Variants
	$\mid$	$\{l : \tau;\ \ldots;\ l : \tau\}$	Records

Variants and records take labels C and l. A type definition takes one of these variant or record definition bodies with type variable parameters $\alpha_1, \ldots, \alpha_n$. The types τ which appears in the right hand of the equation must not contain other variables. A type declaration can consist of more than one type definitions to define mutual recursive data types. For simplicity the variants must take always one type argument like C `of` τ: the constant variant definitions without type arguments like `type t = Foo | Bar` can be encoded into this form of definition adding a dummy type argument `unit` for example: `type t = Foo of unit | Bar of unit`. We also omit the type aliases here.

We assume the following property of the Objective Caml implementation:

Assumption 1 *In Objective Caml, the following information in data type definitions never affects the representation of values.*

- *Data type names*
- *Constructor names of variants*
- *Labels names of records*

Therefore, in our system, data types are identified by their definitions without these irrelevant names, their *nameless type definitions* in short. Intuitively, data types which have the same nameless type definition are internally equivalent. Under Assumption 1, value communication between such equivalent data types never leads to run time crashes.

Nameless type definitions

nameless-typedefs	::=	`type` *typedef* `and` ... `and` *typedef*	Nameless definition
typedef	::=	$(\alpha_1, \ldots, \alpha_m)\ \#n =$ *defbody*	
defbody	::=	$\_$ `of` $\tau_e \mid \ldots \mid \_$ `of` τ_e	Variants
	$\mid$	$\{\_ : \tau_e;\ \ldots;\ \_ : \tau_e\}$	Records

External types

$$\begin{array}{rcll} \tau_e & ::= & \alpha & \\ & | & \tau_e * \tau_e & \\ & | & (\tau_e, \ldots, \tau_e)\ \#n & \text{Recursive index in type definitions} \\ & | & (\tau_e, \ldots, \tau_e)\ \mathit{digest} & \text{Data type digest} \end{array}$$

Nameless type definitions have the same structure as normal ML type definitions. They are just type definitions without the constructor names and record labels names. Names of types are also omitted, just replaced by an integer index $\#n$. Index $\#n$ indicates the position of the type `t` in its mutual recursively defined types. Each data type `t` is identified by a pair (l, n) where l is the list of the mutually recursive nameless type definitions where `t` appears at n-th position.

The *external types* τ_e appear in the nameless type definitions are also nameless. So the external types can represent any Caml type expression: for variables and tuples, the external types are equivalent to the normal type expressions; there is no functional types in τ_e, since there is no function encoding for value I/O[3]. We have an additional construct for recursively defined data types in type definitions. In the definitions, those types are designated with an index n. As the nameless type definitions, each data type name `t` must be replaced by something nameless in the nameless types. However, we do not use its pair of corresponding mutual recursive nameless definitions and its position index (l, n) directly, since embedding nameless definitions inside types would cause size explosion. Therefore, in the nameless types, instead of pairs of nameless definitions and a position index (l, n), we embed their *digests*. These digests are defined as *MD5 digests*[Riv92] of the string representation of the corresponding nameless definitions and indices (l, n). MD5 algorithm produces a 128-bit "fingerprint" check-sum of strings of any arbitrary length. The use of digests reduces the size of the type information radically; the size of the data type information is now constant, no matter how definitions are complex.

5.4.4 Digest calculation

The algorithm $\mathit{Digest}(\Gamma_P, \mathtt{t})$ to calculate data type digest for type `t` in the type environment Γ_P of program P is described as follows:

- If `t` is one of the primitive type like `int` and `float`, just returns its name `"int"` or `"float"` as $\mathit{Digest}(\Gamma_P, \mathtt{t})$.
- Otherwise, obtain the type definitions of `t` and the data types mutually recursively defined with `t` from the typing environment Γ_P,
- Sort the definitions to have unique data type digests. The ordering of definitions must not depend on the names contained in the definitions.
- Erase all the constructor names and record labels from the sorted type definitions.

[3]Objective Caml provides a facility to output functional values, but these values must be input in the same program. If we had some encoding for functional values which can be input in different programs, it would be easy to add functional types to our system, because our type check system is independent of the value encoding.

- Replace mutually defined data type names by indices #1, #2, and so on; other type names `t'` are simply replaced by their digests, $Digest(\Gamma_P, \mathtt{t}')$.

- Finally, calculate and return the MD5 digest of the string representation of these nameless definitions and an integer of the position of `t` in them.

Using this data type digest, we can define the notion of the equivalent data types more formally:

Definition 9 Equivalent data types *Two data types* `t` *of program* P *and* `t'` *of program* Q *are* internally equivalent *iff* $Digest(\Gamma_P, \mathtt{t}) = Digest(\Gamma_Q, \mathtt{t}')$

For example, the two types `tree` and `arbre` in the previous example are internally equivalent, because they have the same nameless definition:

```
type α #1 =
| * of α
| * of α #1 ⟨digest for list⟩
```

Because the digest for `list` can be considered same throughout all the Caml programs[4], $Digest(\Gamma_E, \mathtt{tree})$ and $Digest(\Gamma_F, \mathtt{arbre})$ have the same value.

5.4.5 Digest compilation

The data type digests are computed by the compiler from their definitions. Then they are stored in Caml object (`.cmo`) files as regular strings. For example, the digest for data type `tree` is compiled as the following definition:

```
let digest_tree =
  let definition_string =
    sprintf "type α #1 = * of α | * of α #1 %s" digest_list
  in
  compute_md5 (definition_string, 1)
```

The digest of `tree`, `digest_tree` is defined as the MD5 digest checksum of the string representation of its nameless type definition and its position index (here it is 1). The data type digest compilations depend on the already computed digests of the dependent data types which occur in their definitions, like in this example: the digest of `tree` is defined using the digest of `list`.

The data type digests can be statically compiled and stored, if all the dependent digests are accessible. However, in the separate module compilation environment, it is not always the case, and we may have to compute them at run time. If some data type used in our target type definition is abstracted and its definition is not available for the module we compile, the digest of the data

[4]The type `list` is a predefined type and has no actual definition. Therefore data type digest of `list` is just a string `"list"`.

type is unknown to us. Consider for instance, two module `M` and `N`, such that the implementation of `M` is not yet written or just does not exist: we just have the interface file `m.mli` when we compile `n.ml`.

m.mli	n.ml

`type t`	`type t' =`
	`| Boo of M.t (* abstract *)`
	`| Far`

In such case we cannot compute the digest of `N.t'` statically, since the implementation of `M.t` is not available and its digest is still undefined.

We can split such abstract data types into the following two groups. For each group we take different approach:

Completely abstract data types (foreign data types)

These data types have no implementation at all, and are used for communication between other languages such as C. Since ML programs do not know the structure of such types, their data type digests are undefined. The digests of the data types derived from these abstract data types are undefined neither. We can create dynamic values of such digest-less data types, treating their digests are null strings. But their exportation must be forbidden for type security reasons.

Instead of just rejecting their exportation, we may be able to give their digest in a somewhat explicit way. For example, if the operations of an abstract data type `t` is defined in a file `impl.c`, we can use the MD5 digest of the content of the file as the digest of `t`, using a new syntax:

```
type t with "impl.c"
```

But of course programmers must give these explicit digests with great care, unless they can easily break the value I/O security by giving the same digest for different non-equivalent data types.

Abstract data types with Caml definitions

In this case, we have the implementations for these data types but they are not accessible when we are compiling different modules. Their digests are available to the other modules when these implementations are compiled and linked. For a data type definition which contains such abstract data types inside, the compiler generates a code to compute its digest at program load time (i.e. the very beginning of program execution). For example, the digest of `N.t'` in the above example is defined as an expression using the value of the digest of `M.t` which will be linked together:

```
let digest_t' = compute_md5 (sprintf "type #1 = * of %s | *" M.digest_t, 1)
```

5.4.6 Run time representation of external types

We define the ML value representation of external types like the run time type representation at Section 3.3.1. Instead of the data type names of run time types external types carry data type digests. data types can carry their definitions as data type digests.

```
type rpath                                   (* type path name (≒ string) *)
type digest = string

type etype =
| ETvar   of int                                          (* type variables *)
| ETtuple of etype * etype                                         (* tuples *)
| ETconstr of digest * etype list      (* data types including basic types *)
```

The following conversion function $Ext(\Gamma_P, \tau)$ is a canonical mapping from type expressions to external types, replacing data type names by the corresponding digests of type environment Γ_P of program P:

$$\begin{aligned} Ext(\Gamma_P, \alpha) &= \alpha \\ Ext(\Gamma_P, \tau_1 * \ldots * \tau_n) &= Ext(\Gamma_P, \tau_1) * \ldots * Ext(\Gamma_P, \tau_n) \\ Ext(\Gamma_P, (\tau_1, \ldots, \tau_n)\ \mathtt{t}) &= (Ext(\Gamma_P, \tau_1), \ldots, Ext(\Gamma_P, \tau_n))\ Digest(\Gamma_P, \mathtt{t}) \end{aligned}$$

This conversion function is implemented in the following pseudo ML code of translation of ML value representations:

```
let rec rtype_to_etype = function
  | RTbase s → ETconst (s, [])
  | RTvar id → ETvar id
  | RTtuple (rt1,rt2) → ETtuple (rtype_to_type rt1, rtype_to_type rt2)
  | RTconstr (path, rts) →
      ETconstr (Digest(ΓP,path), List.map rtype_to_etype rts)
```

In the above definition of `rtype_to_etype`, we suppose that the data type digests $Digest(\Gamma_P, \mathtt{path})$ are globally available and calculated at run time, when the run time types are actually constructed. For example, «`int tree`» = `RTconstr("tree", [RTbase "int"])` is converted to a value of external type: `ETconst(digest_tree, [ RTconst ("int", []) ])` where the variable `digest_tree` stores $Digest(\Gamma_P, \mathtt{tree})$.

5.4.7 External instance relationship

Like the normal type expressions, we can extend the notion of external types to quantified external type schemes σ_e, and define instance relationship $\tau_e \leq \sigma_e$ without problem. This instance relationship between external types and type schemes is implemented in Figure 5.1 just like as `is_instance` for run time types. `is_external_instance` τ_{e1} τ_{e2} returns true if and only if τ_{e1} is an instance of the external type scheme $\forall FV(\tau_{e2}).\tau_{e2}$. When `is_external_instance` receives two data types, it checks they have the same data type digests. If they are different, they are not instance of each other.

```
let is_external_instance t t' =
  let subst = ref [] in
  let rec is_instance_aux t t' =
    match t, t' with
    | _, ETvar id →
        begin try
            if List.assoc id !subst = t then true else false
          with
            Not_found → subst := (id, t) :: !subst; true
        end
    | ETtuple (t1,t2), ETtuple(t'1,t'2) →
        is_instance_aux t1 t'1 && is_instance_aux t2 t'2
    | ETconstr (d1, [t11;...;t1n]), ETconstr(d2, [t21;...;t2n]) →
        d1 = d2 && is_instance_aux t11 t21 && ... && is_instance_aux t1n t2n
    | _ → false
  in
  is_instance_aux t t'
```

Figure 5.1: Definition of the predicate `is_external_instance`

5.5 Value I/O primitives

With external types, we can define our safe value I/O primitives, `export_value` and `import_value`:

```
val export_value : ∀δ. out_channel → δ → unit
val import_value : ∀δ. in_channel → δ
```

$$\begin{array}{lcl} gen'(\texttt{export\_value}) & = & \texttt{fun t oc v} \;\rightarrow\; \texttt{output\_value oc (v, rtype\_to\_etype t)} \\ gen'(\texttt{import\_value}) & = & \texttt{fun t ic} \;\rightarrow\; \texttt{let}\ (\texttt{v}, \texttt{t}') \;=\; \texttt{input\_value ic in} \\ & & \qquad\qquad \texttt{if is\_external\_instance}\ \texttt{t}'\ \texttt{t then v} \\ & & \qquad\qquad \texttt{else raise Coercion\_failure} \end{array}$$

The compilation of the primitives are defined as mixtures of the compilation of dynamic value primitives and the original unsafe I/O primitive functions. `export_value`, like `dynamic`, creates a pair of the internal value representation of the given argument `v` and its external type representation and write it out to a channel. `import_value` reads such a value-type pair and checks whether it is an instance of the external type representation of the expected type. If the check succeeds the internal value representation of the input value is coerced into the importing program side as a value of the expected type.

5.6 Notes

Polymorphic encapsulation

We can use the same technique studied in Chapter 4 to make the safe I/O primitives possible to encapsulate polymorphic values correctly. The `export_value` primitive will take full poly-dynamic quantification $\forall\partial$ instead of the current normal one $\forall\delta$, and its second argument will be checked its generalization with a special typing rule. To import back exported values with polymorphic external type scheme to polymorphic, `import_value` expression must be a special syntax construct like the `coerce` expression.

Digest computation at link time

In the separate module compilation environment, data type digests of type definitions cannot be computed statically at each module compilation, due to the existence of the abstract types. The definitions of abstract types are inaccessible outside of the module which defines them, or even worse, the definitions might be written yet. Therefore, in Section 5.4.3, for such statically unavailable data type digests at each module compilation, we embed small piece of code to compute them at load time. They are calculated when programs are launched each time. But actually, they can be computable statically at the link time, since all the data type definitions are available at that time. This kind of link time optimization reduces the size of the executables and the load time overhead, but requires additional framework to retrieve type definitions from the compilation result of modules, even they are abstracted by signatures.

5.7 Conclusion

We proposed a type safe ML value I/O over extensional polymorphism. The values are output with small size type information, where data types are identified by an abstraction of their definitions, which represents their universal semantics. This gives to our safe I/O system an interesting property – two or more different programs communicating via this value I/O do not need to share the same type definitions. The data type definitions information is compressed using MD5 digests. This allow extremely compact type information in exported values.

We have already implemented a prototype of the safe value I/O functionality over an extensional polymorphism implementation as an extension of Objective Caml version 2.02 system. It also has polymorphic type-safe value I/O, and its `coerce` and `import` constructs handle multiple type cases. We cannot compute all the data type digests statically due to the abstract data types, but their calculation is lazy-evaluated, so there is no meaningless overhead at run time. The executable (object code) size growth due to data type digest embedding is usually about 0 to 10 percent (0 for programs with no type definitions, 10 for `ocamlc`, which might have a high typedefs/code ratio). This is perfectly acceptable.