Chapter 6

Unification of dynamic values and safe value I/O

6.1 Introduction

Marriage of dynamic values and safe value I/O

To demonstrate the power of extensional polymorphism, in the last two chapters we implemented the basic functionalities of dynamic values and type safe value I/O as generic primitives. They were formalized differently, but their internal mechanisms are identical: they are both systems of encapsulation and decapsulation of values with their types. Only the difference between them is the type informations stored in the encapsulation. In this chapter, we make both systems into one, by unifying their type information for the encapsulation. The unified *external values* are value-type pairs which carry both data type names and their digests. The primitives of dynamic values and safe value I/O are updated to handle this external values. It is not just a mere simplification, it also makes them cover their shortages each other. Having data type digests inside, dynamic values are secure even if they are exported outside of programs. In the value I/O, having another run time type checking which compares the name of data types, we can use more strict value importation.

Outline of this chapter

In Section 6.2, we explain the necessity of unification of dynamic values and safe value I/O, showing possible type insecurity at the combination of these two functionalities. Section 6.3 and 6.4 extend run time types and dynamic values so that they can also carry related data type digests. The two different dynamic decapsulation schemes which come from the decapsulation primitives of dynamic values and safe value I/O, `coerce` and `import`, are explained in Section 6.5. In Section 6.6, we define the primitives for the unified system using the extended run time types with digests and two decapsulation schemes.

6.2 Dynamic values in value I/O

In the last implementation of dynamic values, we did not suppose that dynamic values can be exported outside of programs. Inside one program, data types can be identified by their names. But this s no longer correct in the combination of dynamic values and value I/O. For example, imagine an exportation of a dynamic value $(v, \mathtt{t})$, a dynamic value of a data type `t`:

```
let oc = open_out "foo.dat" in
let d = dynamic (v : t) in
export_value oc d;;
```

The `export_value` primitive creates a value-type pair $((v, \mathtt{t}),$ `ETconst("dyn", [])` $)$ and export this pair outside of the program. In a reader program, we import back this pair as a dynamic value:

```
let ic = open_in "foo.dat" in
let d = (import_value ic : dyn);;
```

The type digest `"dyn"` is verified against the expected type `dyn` and the importation is granted as $(v, \mathtt{t})$. But this imported dynamic value is not a valid type-value pair for the reader program, since the data type name `t` is came from different program.

Run time types encapsulated in dynamic values must also contain data type digests and be universal throughout different programs, since dynamic values can now be exported outside of the program which created them by value I/O. The coercion of dynamic values must also check the data type digests.

6.3 Run time types with digests

Dynamics require data type digests and value I/O want data type names. Then why not merge them and obtain unique run time type representation? In the new *run time types with digests*, additional to the initial definition at Section 3.3.1, the run time representation of data type `RTconst` has a new member `digest`, the digest of the data type:

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

type rtype =
| RTbase  of rpath                                        (* basic types *)
| RTvar   of int                                      (* type variables *)
| RTarrow of rtype * rtype                            (* function types *)
| RTtuple of rtype * rtype                                     (* tuples *)
| RTconst of rpath * digest * rtype list                   (* data types *)
```

The conversion function $\ll \cdot \gg$ which is for translating the normal ML type expressions to run time types is now extended to embed data type digests:

$$\begin{array}{lcl}
\ll base \gg & = & \texttt{RTbase}\ "base" \\
\ll \delta_i \gg & = & \texttt{RTvar}\ i \\
\ll \tau_1 \rightarrow \tau_2 \gg & = & \texttt{RTarrow(}\ll \tau_1 \gg\texttt{, } \ll \tau_2 \gg\texttt{)} \\
\ll \tau_1 * \tau_2 \gg & = & \texttt{RTtuple(}\ll \tau_1 \gg\texttt{, } \ll \tau_2 \gg\texttt{)} \\
\ll (\tau_1, \ldots, \tau_n)\ \texttt{t} \gg & = & \texttt{RTconst("t", } \mathit{Digest}(\mathit{\Gamma}_P, \texttt{t}), \texttt{ [}\ll \tau_1 \gg\texttt{; } \ldots \texttt{; } \ll \tau_n \gg\texttt{])}
\end{array}$$

For example, «`int tree`» returns a run time type with digests: `RTconst("tree", digest_tree, [ RTbase "int" ])` where the variable `digest_tree` stores $\mathit{Digest}(\Gamma_P, \texttt{tree})$.

Rest of this chapter, run time types with digests are called run time types.

6.4 External dynamic values

The internal representation of dynamic values and exported values in the value I/O is unified to *external dynamic values*, pairs of a value and the new run time type representation of its type. We keep using the abstract type `dyn` for the unified external dynamic values.

6.5 Two modes of value importation

Data type identification of the previous implementation of the safe value I/O is by their semantics based on the internal value representation, i.e. nameless definitions. If they define data with the same internal structure, data types can be compatible even when their definitions can have different names, For example, we can exchange between the following data types via value I/O:

```
type α tree =                 type α arbre =
| Leaf of α                   | Feuille of α
| Branch of α tree list       | Branche of α arbre list
```

This confusion of data types provides an interesting and practical safe data exchange between different data types, but in some cases this confusion is not welcome. In the normal semantics of data types, above two types are completely different. This strict distinction of data types may be necessary in value I/O sometimes, too. By encapsulating values as external dynamic values at export, and comparing also data type names at import, we can introduce an additional mode of value importation with this more strict data type identification. We call these two decapsulation of external dynamic values as *coercion* and *importation*. Both decapsulations check data type digests equality in the data type identification, while at coercion data type path names are additionally taken into account in the identification.

We implement two levels of run time type instance relationships in the unified system of external dynamic values, corresponding to the two decapsulation schemes: `is_coerce_instance` and `is_import_instanace` in Figure 6.1. They are different only in the equivalence check of data types,

```
(* basic function is_instance, shared by both *)
let is_instance p ty ty' =
  let subst = ref [] in
  let rec is_instance_aux ty ty' =
    match ty, ty' with
    | «τ», «α» →
        begin try
            if List.assoc «α» !subst = ty then true else false
          with
            Not_found → subst := («α», ty) :: !subst; true
        end
    | «τ1 → τ2», «τ'1 → τ'2» →
        is_instance_aux «τ1» «τ'1» && is_instance_aux «τ2» «τ'2»
    | «τ1 * τ2», «τ'1 * τ'2» →
        is_instance_aux «τ1» «τ'1» && is_instance_aux «τ2» «τ'2»
    | «base1», «base2» → base1 = base2
    | «(τ1,...,τn) t» «(τ'1,...,τ'n') t'» →
        (* comparison of data types are delegated to the predicate p *)
        p ty ty'
    | _ → false
  in
  is_instance_aux t t'

(* data type comparison for importation *)
let import_equivalence
  (RTconst(_, d, [ty1;...;tyn])) (RTconst(_, d', [ty'1;...;ty'n'])) =
    d = d' && n = n' && ∀1 ≤ i ≤ n, is_instance_aux tyi ty'i

(* data type comparison for coercion = import_equivalence + path check *)
let coerce_equivalence (RTconst(p, d, tys)) (RTconst(p', d', tys')) =
  p = p' && import_equivalence (RTconst(p, d, tys)) (RTconst(p', d', tys'))

(* the two levels of instance relationships for importation and coercion *)
let is_import_instance = is_instance import_equivalence
let is_coerce_instance = is_instance coerce_equivalence
```

Figure 6.1: Two versions of instance predicate `is_*_instance`

and share the rest in `is_instance`. Different from Figure 4.1, this `is_instance` takes an additional argument `p`, a data type equivalence predicate. By applying different equivalence predicates, we obtain the two instance relationships. `import_equivalence` for the importation compares the data type digests, arities and arguments of given two data types. `coerce_equivalence` checks also the path name equality additionally to those of `import_equivalence`.

6.6 New primitives

6.6.1 Dynamic value primitives

The dynamic primitives and their compilations are updated using the new run time type instance relationships.

```
val dynamic : ∀δ.δ → dyn
val coerce : ∀δ.dyn → δ
val import : ∀δ.dyn → δ
```

$$
\begin{array}{rcll}
gen'(\texttt{dynamic}) & = & \texttt{fun t v} & \rightarrow\ (\texttt{v},\texttt{t}) \\
gen'(\texttt{coerce}) & = & \texttt{fun t}\ (\texttt{v},\texttt{t}') & \rightarrow\ \texttt{if is\_coerce\_instance t t}'\ \texttt{then v} \\
 & & & \ \texttt{else raise Coercion\_failure} \\
gen'(\texttt{import}) & = & \texttt{fun t}\ (\texttt{v},\texttt{t}') & \rightarrow\ \texttt{if is\_import\_instance t t}'\ \texttt{then v} \\
 & & & \ \texttt{else raise Coercion\_failure}
\end{array}
$$

The compilation of the `dynamic` primitive is not modified and same as before, but it receives newly defined run time types with digests which are obtained by the new run time type translation function « · ». The `coerce` primitive is diverged to two primitives, `coerce` and `import`, using different instance relationships for each. These primitives themselves cannot treat the polymorphic contents of external dynamic values well, but the same technique we discussed in Section 4.3 can be applied to them, too. Thanks to the data type digests, we can coerce and import external dynamic values safely:

```
# let v = Branch [Leaf 1; Leaf 2];;
v : int tree
# let d = dynamic v;;                          (* creates (v, int tree) *)
d : dyn = <abstr>
# (coerce d : int tree);;
- : int tree = Branch [Leaf 1; Leaf 2]
# (coerce d : int arbre);;              (* coerce checks the type names *)
Uncaught exception: Coercion_failure
# (import d : int tree);;                              (* same as coerce *)
- : int tree = Branch [Leaf 1; Leaf 2]
# (import d : int arbre);;             (* import can rename data types ! *)
- : int arbre = Branche [Feuille 1; Feuille 2]
```

6.6.2 Safe I/O primitives

Once the data type digests provide the universal run time type representations of external dynamic values amongst the different programs, the primitives of the type safe value I/O can be obtained for free. We just restrict the polymorphic types of the unsafe version of the value I/O functions to monomorphic, so that they can just accept external dynamic values:

```
let export_dyn = (output_value : out_channel → dyn → unit)
let import_dyn = (input_value : in_channel → dyn)
```

After these primitives are introduced, the original `input_value` and `output_value` functions must be inaccessible to users since they are unsafe.

We can also shift encapsulation of the output values into the `export_dyn` and make it a derived dynamizer `export_value` as follows:

```
val export_value : ∀∂̲.out_channel → ∂ → unit
let export_value oc v = export_dyn oc (dynamic v)
```

6.7 Conclusion

Systems of dynamic values and type safe value I/O were formalized and implemented independently, though their internal mechanism share many concepts. We unified them to external dynamic values, value-type pairs which can be used for both purposes of dynamics and value I/O. This unification is not a simplification, but extends both systems. The dynamic values are now safe even they are exported by value I/O outside of programs. Having a new decapsulation scheme, we can perform more strict data type identification at value importation.

The run time types are modified to carry data type digests like external types of the type safe value I/O, so that exported dynamic values can be coerced safely inside different reader programs. By checking data type path names additional to digest comparison, imported values can be decapsulated more strictly, using the `coerce` primitive. On the other hand, the `import` primitive weaken the condition of dynamic value decapsulation and enables exchanging between different data types of the same internal structures via external dynamic values.