Une bibliothèque de typage dynamique en OCaml

Julien Signoles[1] *

1: CEA, LIST, Laboratoire de Sûreté des Logiciels
PC 94, 91191 Gif-sur-Yvette Cedex, France
Julien.Signoles@cea.fr

Résumé

Cet article présente une bibliothèque OCaml fournissant une représentation dynamique des types monomorphes, y compris pour les instances des types polymorphes et les types de données abstraits. Elle permet ainsi de voir les types OCaml comme des citoyens de première classe. Elle comble aussi le fossé séparant OCaml des langages dynamiques en mêlant vérifications statiques et dynamiques des types. Nous nous concentrons ici sur le cœur de son implantation, ses propriétés théoriques et l'usage qui en est fait dans Frama-C, plateforme logicielle libre d'analyse statique de programmes C au sein de laquelle cette bibliothèque est distribuée.

1. Introduction

Un des avantages principaux des systèmes de types statiques des langages à la ML comme Objective Caml (OCaml) est de garantir à la compilation qu'aucun programme n'engendre une erreur lors de son exécution : «*well-typed programs do not go wrong*» [17]. Malheureusement, cette propriété ne peut être préservée lors de l'utilisation de certains traits de programmation pourtant usuels. Ainsi par exemple, les fonctions de sérialisation fournies par la bibliothèque standard d'OCaml autorisent les erreurs à l'exécution en cas de mauvaise utilisation, comme le montre la figure 1.

Les langages de programmation dynamiquement typés — comme Lisp— et ceux offrant des fonctions d'introspection — comme Java— effectuent des vérifications à l'exécution permettant d'éviter ces comportements, ce qui est nettement mieux que le résultat de la figure 1.

Cet article présente une bibliothèque en pur OCaml, sans extension dédiée, de typage dynamique — appelée Type — fournissant une structure de données abstraite sur les types monomorphes OCaml, y compris les instances des types polymorphes et les types de données abstraits. Elle permet ainsi de voir les types OCaml comme des citoyens de première classe. Elle comble aussi de manière pragmatique le fossé séparant OCaml des langages dynamiques : certaines vérifications de types sont faites statiquement par le compilateur grâce à l'interface de la bibliothèque et d'autres dynamiquement, par son implantation.

Cette bibliothèque est intégrée au sein de Frama-C [7], une plateforme libre, destinée à l'analyse de programmes écrits en C, développée en OCaml ($>$ 100 000 lignes de code) et à vocation industrielle [10]. Type reste néanmoins compilable indépendemment du reste des sources.

Dans la suite, la section 2 introduit d'abord les motivations principales pour cette bibliothèque. Ensuite, la section 3 détaille le cœur de son implantation, suivi de quelques extensions dans la section 4. Enfin, la section 5 présente les travaux relatifs principaux et les compare à notre solution.

*. Ce travail a été financé par le projet ANR U3CAT ANR78U3CATZZ.

```
bad_unmarshal.ml:
let () =
  (* saving an integer value in file f.sav *)
  let cout = open_out "f.sav" in
  output_value cout 1;
  close_out cout;
  (* reading the integer value in file f.sav *)
  let cin = open_in "f.sav" in
  let s = input_value cin in
  close_in cin;
  (* using it as a string *)
  print_endline s

$ ocamlc -o bad_unmarshal bad_unmarshal.ml
$ ./bad_unmarshal
Segmentation fault
```

FIGURE 1 – Erreur à l'exécution par désérialisation.

2. Motivations

L'utilisation de la bibliothèque Type dans Frama-C est triple : d'abord, pour enregistrer et accéder aux APIs de greffons chargés dynamiquement, ensuite pour une notion appelée *journalisation* et enfin pour la désérialisation sûre. L'intérêt de cette bibliothèque pour chacune de ces trois fonctionnalités est discuté dans les sections 2.1 à 2.3.

Un point important doit aussi être gardé à l'esprit : toute solution — quelle qu'elle soit — doit être développée uniquement en OCaml, sans utiliser d'extensions, de langages ou de bibliothèques dédiés pour lesquels l'équipe de développement de Frama-C ne serait pas en mesure d'assurer la maintenance sur le long terme. Cette contrainte est due au fait que Frama-C est destiné — entre autres — à être utilisé dans des programmes avioniques à durée de vie longue (potentiellement plusieurs dizaines d'années) et qu'il faut être en mesure d'assurer la maintenance de l'outil et de ces dépendances sur tout le cycle de vie d'un tel programme. C'est la raison principale pour laquelle, par exemple, toute utilisation d'un compilateur dédié ou de Camlp4 est prohibée.

2.1. API dynamique

Frama-C est une plateforme extensible dans laquelle les analyseurs sont autant de greffons venant ajouter des fonctionnalités au noyau de l'outil [24]. Ces greffons peuvent être du code chargé dynamiquement par le noyau *via* le module `Dynlink` de la bibliothèque standard OCaml. Un tel chargement dynamique de code permet d'une part de réduire la taille de l'application et, d'autre part, de reporter au moment de l'exécution le choix de l'implantation effective d'un comportement du programme [6].

Le problème est alors de savoir accéder aux interfaces de ces greffons. En effet, si `Dynlink` autorise un module M à accéder directement à une valeur d'un module P chargé dynamiquement, ceci nécessite d'avoir un accès statique à son interface, réduisant ainsi l'intérêt de charger P dynamiquement pour au moins trois raisons : d'abord il n'est plus possible d'utiliser une implantation alternative de P, possédant une interface compatible mais différente, dans M ; ensuite il n'est plus possible d'exécuter M en l'absence de P (ce qui limite ses possibilités, sans forcément le rendre caduc) ; et enfin il n'est plus possible de définir des greffons mutuellement récursifs (car l'édition de liens prohibe de telles unités de compilation).

Ainsi la manière standard d'accéder à un module `Plugin` chargé dynamiquement au sein d'un module `User` est d'utiliser un module tiers `Share` dans lequel `Plugin` enregistre son interface. La figure 2 résume les liens entre ces modules, tandis que la figure 3 en présente une implantation usuelle

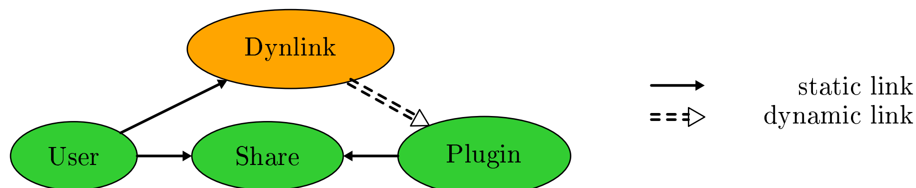

FIGURE 2 – Schéma standard de liaison dynamique en OCaml.

qui fait appel à une table globale `Share.dynamic_functions` pour enregistrer les interfaces des codes chargés dynamiquement (ici seulement la fonction `f` du module `Plugin`).

```
share.ml:
let dynamic_functions: (string, string → unit) Hashtbl.t = Hashtbl.create 7
let get s = Hashtbl.find dynamic_functions s
let register s f = Hashtbl.add dynamic_functions s f

plugin.ml:
let f x = print_endline ("running f with " ^ x)
let () = Share.register "Plugin.f" f

user.ml:
let () = Dynlink.loadfile (Dynlink.adapt_filename "plugin.cmo")
let () = Share.get "Plugin.f" "s"
```

FIGURE 3 – Enregistrement et accès standard à une interface dynamique en OCaml.

Malheureusement, cette implantation — quoique standard — a un inconvénient majeur : le type τ de chaque fonction de l'interface de chaque module chargé dynamiquement doit être statiquement défini dans la partie partagée `Share`, ce qui nécessite de tous les connaître à l'avance. Pire encore, il est indispensable de définir une table pour chacun de ces types (ici la table `dynamic_functions` pour les valeurs de types `string` $\rightarrow$ `unit`), ce qui pose un réel problème de passage à l'échelle qui s'avère critique dans un cadre comme celui de Frama-C.

Pour contourner cette double difficulté, on peut être tenté d'utiliser l'implantation du module `Share` proposée par la figure 4. Cette solution utilise une table hétérogène pour enregistrer des valeurs de

```
share.ml:
let dynamic_values: (string, Obj.t) Hashtbl.t = Hashtbl.create 7
let get s = Obj.obj (Hashtbl.find dynamic_values s)
let register s f = Hashtbl.add dynamic_values s (Obj.repr f)
```

FIGURE 4 – Enregistrement et accès puissants mais non sûrs à une interface dynamique.

types quelconques et différents, ce qui résout les deux inconvénients de l'approche précédente. La table hétérogène est obtenue à l'aide du type `Obj.t` et des fonctions `Obj.repr` et `Obj.obj` de la bibliothèque standard. La première, de type $\alpha \rightarrow$ `Obj.t`, permet de convertir toute valeur OCaml en une valeur de type `Obj.t`, tandis que la seconde, de type `Obj.t` $\rightarrow \alpha$, est sa réciproque. Combiner les deux fournit une opération de transtypage (*cast*) en OCaml. Ceci casse malheureusement la sûreté du

typage offerte par le compilateur : en poursuivant notre exemple, l'expression `Share.get "Plugin.f" 1` est bien typée mais génère une erreur à l'exécution. Cette solution est donc pire que la précédente.

La solution que nous proposons dans la figure 5 améliore les deux solutions précédentes, en demeurant néanmoins sur le même principe. Elle utilise une table hétérogène `dynamic_values` indexée

```
share.ml:
let dynamic_values: Type.String_tbl.t = Type.String_tbl.create 7
let get s ty = Type.String_tbl.find dynamic_values s ty
let register s ty f = Type.String_tbl.add dynamic_values s ty f

plugin.ml:
let f x = print_endline ("running f with" ^ x)
let () = Share.register "Plugin.f" (Datatype.func Datatype.string Datatype.unit) f

user.ml:
let () = Dynlink.loadfile (Dynlink.adapt_filename "plugin.cmo")
let () = Share.get "Plugin.f" (Datatype.func Datatype.string Datatype.unit) "s"
```

FIGURE 5 – Enregistrement et accès à une interface dynamique *via* la bibliothèque Type.

par des chaînes et dont l'implantation est fournie par la bibliothèque Type. Une telle hétérogénéité permet de résoudre les deux problèmes engendrés par la solution usuelle de la figure 3. Pour préserver un accès sûr à cette table, une *valeur de type* doit être spécifiée comme argument supplémentaire, ici la valeur `Datatype.func Datatype.string Datatype.unit` correspondant au type statique `string` → `unit`. Même si nous ne détaillerons qu'à la section 3 d'où provient la sûreté de notre approche, cette bonne propriété résout le problème fondamental soulevé par l'approche non sûre de la figure 4.

2.2. Journalisation

La journalisation est un mécanisme de génération d'un script OCaml (appelé *journal*) permettant de rejouer les actions effectuées par l'utilisateur (en particulier dans une interface graphique). L'intérêt est triple. D'abord, un journal facilite le débogage en cas d'erreur interne imprévue. Ensuite, à la manière d'un langage de macros, il permet d'automatiser des séquences d'actions utilisateur. Enfin, il constitue une aide au développeur en permettant un prototypage rapide : à partir d'un squelette généré pour un cas particulier, il lui suffit de généraliser le code et de l'adapter à ses besoins.

Dans Frama-C, le journal est un greffon standard qui peut donc être chargé dynamiquement. À ce titre, il utilise l'API dynamique présentée à la section 2.1. En outre, le développeur doit enregistrer chaque fonction OCaml à consigner dans le journal en cas d'appel. Pour ce faire, une API similaire à celle déjà présentée est fournie. D'ailleurs, il est possible (et conseillé) d'enregistrer simultanément une fonction dans une API dynamique et pour journalisation. Enfin, l'algorithme de journalisation proprement dit procède par récurrence sur le nombre de flèches présentes dans le type de la valeur à journaliser. Pour cela, il utilise des possibilités d'introspection offertes par la bibliothèque Type [1].

2.3. Désérialisation

Comme le montre la figure 1 présentée en introduction, la désérialisation n'est pas sûre en OCaml. Si désérialiser de cette manière quelques valeurs suivant un protocole simple engendre un risque d'erreur limité et facile à déceler par tests, ce n'est malheureusement pas le cas dans Frama-C. Là, cette procédure est complexe, à cause en particulier d'une utilisation massive du *hashconsing* [9] qui nécessite

1. La présentation de cet algorithme dépasse néanmoins le cadre de cet article.

de modifier les valeurs lues sur le disque pour recréer la propriété centrale de *hashconsing* — le partage maximal — entre les valeurs encore présentes en mémoire vive et celles chargées du disque.

Ainsi, et à des fins d'efficacité, Frama-C possède un algorithme de désérialisation *ad hoc*, écrit en OCaml et permettant d'appliquer des traitements particuliers sur chaque valeur chargée, en fonction du type statique de celle-ci [2]. Pour ce faire, cet algorithme utilise intensivement le module `Obj`, y compris dans son interface. Ainsi par exemple, le transformateur à enregistrer pour convertir une valeur chargée d'un type τ donné possède le type `Obj.t` $\rightarrow$ `Obj.t`. La bibliothèque Type permet l'ajout d'une couche logicielle sûre à cet algorithme de désérialisation permettant d'indiquer que le transformateur doit avoir le type $\tau \rightarrow \tau$. En outre, la fonction de désérialisation globale utilisée par Frama-C a le type `in_channel` $\rightarrow \alpha$ `Type.t` $\rightarrow \alpha$, qui permet de la typer correctement par un moyen similaire à celui présenté section 3.2, contrairement à la fonction `input_value` de la figure 1 qui n'empêche pas les erreurs à l'exécution.

3. Le cœur de la bibliothèque

Cette section présente le fonctionnement de la bibliothèque Type. La section 3.1 montre comment créer des valeurs de type. La section 3.2 introduit les tables hétérogènes et l'utilisation des valeurs de types qu'elles en font pour garantir leur sûreté. La section 3.3 présente les propriétés générales garanties par la bibliothèque.

3.1. Valeurs de types

La bibliothèque Type définit un type abstrait polymorphe α `Type.t`, qui représente les valeurs de type, et un unique constructeur `register` dont des implantations simplifiées sont présentées figure 6, accompagnées des deux projecteurs `name` et `repr` utilisés par la suite, et d'une fonction d'égalité.

```
type.mli:
type α t
val register: string → α → α t
val equal: α t → β t → bool

type.ml:
type α t = string
type α ty = α t (* alias utile par la suite *)

let types : (string, Obj.t) = Hashtbl.create 7
let register name repr =
  if Hashtbl.mem types name then failwith ("type " ^ name ^ "already registered");
  Hashtbl.add types name (Obj.repr c);
  name

let name ty = ty
let repr ty = try Obj.obj (Hashtbl.find types ty) with Not_found → assert false
let equal = ( = )
```

FIGURE 6 – Vue simplifiée du module `Type`.

2. Comme pour la journalisation, la présentation de cet algorithme dépasse le cadre de cet article.

Pour tout type monomorphe τ, nous appelons $\mathcal{T}_\tau$ l'ensemble des valeurs de type τ, *i.e.* :

$$\mathcal{T}_\tau \triangleq \{ \text{ valeurs de type } ty \mid ty : \tau \; \texttt{Type.t} \; \}.$$

Munis de cette définition, nous pouvons établir la propriété fondamentale de la bibliothèque, qui exprime qu'une valeur du type τ `Type.t` représente le type statique τ.

Propriété 1 (Injection des type monomorphes dans les valeurs de type) *L'égalité des valeurs de type implique l'égalité entre les types statiques monomorphes correspondants,* i.e. *:*

$$\forall \textit{ types monomorphes } \tau \textit{ et } \tau', \forall ty \in \mathcal{T}_\tau, \forall ty' \in \mathcal{T}'_\tau, \; \texttt{Type.equal} \; ty \; ty' \implies \tau = \tau'.$$

Cette propriété est fondée sur le fait que la chaîne de caractères fournie en premier argument de la fonction `Type.register` sert de clé dont on vérifie l'unicité : toutes valeurs créées seront deux à deux différentes. En outre, le second argument — que nous appelons *représentant* du type — sert à établir statiquement un lien entre la valeur de type nouvellement créée et le type statique qu'elle représente, par l'entremise de la variable de type α du type fantôme [19] α `Type.t`.

Ainsi est-il facile de créer des valeurs de type, ce que fait le module `Datatype` — inclus dans la bibliothèque `Type` — pour chacun des types de base du langage. Voici par exemple la valeur de type correspondant au type `string`, et donc de type `string Type.t`.

```
let string = Type.register "string" ""
```

La propriété suivante est par conséquent triviale.

Propriété 2 (Chaque ensemble $\mathcal{T}_\tau$ est habité) *Pour tout type monomorphe τ, il est possible de construire une valeur de type appartenant à $\mathcal{T}_\tau$.*

En revanche, il est théoriquement possible de créer deux valeurs de type différentes représentant le même type. Par exemple, la valeur `string2` suivante est un représentant du type `string`, au même titre que l'est la valeur `string`.

```
let string2 = Type.register "string2" ""
```

Néanmoins, il est aisé d'imposer des règles de nommage évitant ce cas de figure (en utilisant par exemple le nom long du type), ce qui permet d'obtenir en pratique la réciproque de la propriété 1.

Hypothèse 1 (Bijection entre tout type monomorphe τ et $\mathcal{T}_\tau$) *L'égalité `Type.equal` sur les valeurs de type coïncide avec l'égalité sur les types monomorphes,* i.e. *:*

$$\forall \textit{ types monomorphes } \tau \textit{ et } \tau', \forall ty \in \mathcal{T}_\tau, \forall ty' \in \mathcal{T}'_\tau, \; \tau = \tau' \iff \texttt{Type.equal} \; ty \; ty'.$$

En outre, nous verrons que cette hypothèse, valide en pratique, n'est utile que pour garantir la complétude de notre méthode (propriété 5), mais n'est aucunement nécessaire pour sa correction (propriété 4) qui est la propriété la plus importante : si, par une erreur malencontreuse du développeur, un type venait à avoir deux représentants différents, ceci n'aurait pour conséquence potentielle que le rejet de programmes corrects, mais ne permettrait pas d'accepter des programmes engendrant une erreur à l'exécution.

Cette bibliothèque nécessite donc d'enregistrer chaque type qu'elle est amenée à manipuler. Cela peut sembler contraignant au premier abord. Il n'en est rien en réalité. En effet, d'abord, tous les types de la bibliothèque standard sont pré-enregistrés dans le module `Datatype`. Ensuite, enregistrer une nouvelle valeur de type est simple : seuls deux arguments triviaux sont à fournir. Enfin, en pratique, l'enregistrement est effectué *via* le module `Datatype` qui fournit des fonctionnalités supplémentaires (comme des *pretty printers*), réalisant ainsi d'une pierre plusieurs coups.

3.2. Tables hétérogènes

Implanter de manière sûre des tables hétérogènes en OCaml n'est pas possible. La solution standard, quoique très fortement déconseillée, est celle présentée figure 4 et utilisant le module `Obj`. Nous ne dérogerons ici pas à la règle : l'implantation du module `String_tbl` utilisée par la figure 5 est très similaire à l'implantation non sûre et exploite en particulier le module `Obj`. Cependant, et c'est là une différence fondamentale, elle repose sur une interface sûre, fondée sur les valeurs de type, qui garantit qu'une mauvaise utilisation de ce module ne peut provoquer d'erreur à l'exécution. Cette interface est présentée figure 7. Elle est similaire à l'interface des tables de hachage, à la différence près que les

```
module String_tbl : sig
  type t
  val create: int → t
  val add: t → string → α ty → α → unit
  val find: t → string → α ty → α
end
```

FIGURE 7 – Interface des tables hétérogènes.

fonctions `add` et `find` sont gardées par une valeur de type représentant le type statique de la valeur à ajouter ou rechercher dans la table. Ces gardes supplémentaires permettent de vérifier en trois temps qu'on utilise bien la valeur trouvée dans une table t dans un contexte d'évaluation du même type que celui qu'avait la valeur au moment de son ajout dans t.

D'abord, le type de la fonction `add` permet de faire garantir statiquement par le compilateur OCaml que la valeur de type indiquée représente bien le type statique de la valeur ajoutée dans la table. Ainsi, dans l'exemple suivant, la première ligne est-elle bien typée, au contraire de la seconde.

```
let () = Type.String_tbl.add t "key" Datatype.string "toto"
let () = Type.String_tbl.add t "key" Datatype.string 1
```

Ensuite, le type de la fonction `find` permet de faire typer statiquement par le mécanisme d'unification du compilateur OCaml la valeur de retour et, par la même, de vérifier statiquement qu'elle est utilisée dans un contexte d'évaluation du bon type. Ainsi, dans l'exemple suivant, la première ligne est-elle bien typée, au contraire de la seconde.

```
let () = print_string (Type.String_tbl.find t "key" Datatype.string)
let () = print_int (Type.String_tbl.find t "key" Datatype.string)
```

Enfin, l'implantation du module `String_tbl`, présentée figure 8, vérifie dynamiquement que les valeurs de type indiquées aux moments de l'ajout et de la recherche sont égaux [3]. Ainsi, dans l'exemple suivant, la deuxième ligne s'exécute-t-elle sans erreur, au contraire de la troisième qui lève l'exception `Type.String_tbl.Incompatible_type` avec le message *key has type string but is used with type int.*

```
let () = Type.String_tbl.add t "key" Datatype.string "toto"
let () = print_string (Type.String_tbl.find t "key" Datatype.string)
let () = print_int (Type.String_tbl.find t "key" Datatype.int)
```

3.3. Propriétés

Utiliser les fonctions `Obj.obj` et `Obj.repr` peut casser le théorème de correction forte [17] du système de types d'OCaml. Cependant, l'application qui en est faite dans le cadre de notre bibliothèque est sûre, comme l'exprime la propriété suivante.

3. La bibliothèque fournit plusieurs foncteurs génériques de tables hétérogènes dont `Type.String_tbl` n'est en réalité qu'une instance de l'un d'eux, mais nous simplifions ici volontairement l'implantation.

```
module String_tbl = struct

  type dynamic = { ty: string; o: Obj.t }
  type t = (string, dynamic) Hashtbl.t

  exception Already_exists of string
  exception Unbound_value of string
  exception Incompatible_type of string
  let type_error s ty ty' =
    raise (Incompatible_type
      (Format.sprintf "%s has type %s but is used with type %s."
         s (name ty') (name ty)))

  let create = Hashtbl.create

  let add tbl s ty x =
    if Hashtbl.mem tbl s then raise (Already_exists s);
    Hashtbl.add tbl s { ty = ty; o = Obj.repr x }

  let find tbl s ty =
    try
      let data = Hashtbl.find tbl s in
      (* dynamic type checking *)
      if ty <> data.ty then type_error s ty data.ty;
      Obj.obj data.o
    with Not_found →
      raise (Unbound_value s)

end
```

FIGURE 8 – Implantation des tables hétérogènes.

Propriété 3 (Correction forte) *À l'exception éventuelles des expressions non sûres utilisées en dehors du module `Type`, toute expression bien typée ne peut engendrer une erreur à l'exécution.*

En outre, sous l'hypothèse 1, qui est réaliste en pratique, l'utilisation de notre bibliothèque *via* les tables hétérogènes n'introduit aucune restriction particulière, ce qui exprimé par les deux propriétés suivantes.

Propriété 4 (Complétude) *En supposant valide l'hypothèse 1, pour chaque programme p n'étant pas évalué en une erreur et utilisant les tables hétérogènes non sûres, le programme p dans lequel les occurrences de ces tables ont été remplacées par des occurrences similaires de type `Type.String_tbl.t` est bien typé, ne peut être évalué en une erreur et ne lève aucune exception à l'exécution en provenance du module `Type`.*

Propriété 5 (Équivalence sémantique) *Pour chaque programme p utilisant les tables hétérogènes non sûres et étant évalué en une valeur non erronée v, si le programme p dans lequel les occurrences de ces tables ont été remplacées par des occurrences similaires de type `Type.String_tbl.t` est bien typé et ne peut ni être évalué en une erreur ni lever aucune exception en provenance du module `Type`, alors ce programme s'évalue aussi en v.*

4. Extensions

La section 4.1 étend la bibliothèque aux instances monomorphes des types polymorphes, tandis que la section 4.2 l'étend aux types de données abstraits.

4.1. Instances monomorphes de types polymorphes

L'existence de valeurs de types représentant des types polymorphes rendrait la bibliothèque non sûre car, alors, une valeur de type ne correspond plus à un unique type mais à l'ensemble des instances monomorphes de ce type. Par exemple, en considérant une table hétérogène t de type `Type.String_tbl.t` et ty une valeur du type α `list Type.t`, il est possible d'écrire le code suivant.

```
let () = Type.String_tbl.add t "key" ty [ 1 ]
let () = print_string (List.head (Type.String_tbl.find t "key" ty))
```

Ici, ce programme est bien typé car la variable de type α est unifiée différemment au moment de l'ajout (unification avec `int`) et de la recherche (unification avec `string`). Cependant, il génère une erreur à l'exécution. C'est la raison pour laquelle toutes les propriétés ne parlent que de types monomorphes.

Heureusement, une valeur de type polymorphe n'est en réalité jamais constructible. En effet, la variable de type α présente dans le type de la fonction `Type.register` (le seul constructeur de valeurs du type α `Type.t`) n'est pas généralisable, car son second argument *repr* de type α est ajouté dans une table de hachage, qui est une structure impérative [14]. La sûreté de la bibliothèque ne peut donc pas être violée par une utilisation malheureuse de valeurs polymorphes.

La bibliothèque `Type` permet néanmoins d'enregistrer chaque instance monomorphe d'un type polymorphe. Il serait *a priori* possible d'utiliser directement la fonction `Type.register` comme dans l'exemple suivant.

```
let int_list = Type.register "int list" [ 0 ]
```

Une telle solution ne passe cependant pas à l'échelle, car il est alors impossible de pré-enregistrer chacune de ces instances dans la bibliothèque : chacun doit donc le faire selon ces besoins. D'une part le risque d'enregistrer deux fois la même valeur sous des noms différents augmente (ce qui casse la propriété 5) et, d'autre part, si un greffon enregistre par exemple le type `bool list` pour ces besoins propres, un autre greffon ne peut plus le faire (à moins de lui assigner un nom différent, ce qui ramène au problème précédent).

Pour surmonter cette difficulté, le module `Type` permet d'enregistrer chaque type polymorphe *via* le foncteur `Polymorphic` dont une implantation simplifiée est présentée figure 9. Ce foncteur prend en argument le type polymorphe lui-même, et un générateur de noms de type (*resp.* de représentants) en fonction d'un nom (*resp.* du représentant) d'une instance monomorphe. En sortie, le foncteur fournit une fonction `instantiate` permettant de générer une valeur de type pour chaque instance monomorphe de ce type. Cette fonction utilise la mémoisation [16] pour empêcher de créer deux valeurs de type différentes pour une même instance monomorphe.

La figure 10 considère le type des listes comme exemple d'utilisation de ce foncteur : appliquer une première fois `list int` retourne une valeur v fraîche représentant le type `int list`, tandis que chaque autre application `list int` retourne cette même valeur v.

En pratique, de même que pour les types de base, le module `Datatype` inclus dans la bibliothèque fournit des instanciations de ces foncteurs pour les types polymorphes de la bibliothèque standard, en implantant des générateurs de noms de types minimisant le nombre de parenthèses à afficher afin de préserver des messages d'erreur lisibles.

Le seul inconvénient de cette approche est la nécessité de fournir un foncteur différent par nombre de variables de type libres dans le type polymorphe : le foncteur `Polymorphic` gère les types à une

```
module Polymorphic
  (T:sig
    type α t
    val repr: α → α t
    val name: string → string
  end) :
sig
  val instantiate: α t → α T.t t
end =
struct
  let instances : (string, string) Hashtbl.t = Hashtbl.create 17
  let instantiate ty =
    try Hashtbl.find instances ty
    with Not_found →
      let repr = register (T.name (name ty)) (T.repr (repr ty)) in
      Hashtbl.add instances ty repr;
      repr
end
```

FIGURE 9 – Enregistrement fonctoriel des types polymorphes.

```
module List = Polymorphic
  (struct
     type α t = α list
     let name s = "(" ^ s ^ ") list"
     let repr ty = [ ty ]
   end)
let list = List.instantiate
```

FIGURE 10 – Type des listes.

variable libre (de la forme $\alpha\ t$), la bibliothèque fournit aussi un foncteur `Polymorphic2` pour les types à deux variables libres (de la forme $(\alpha, \beta)\ t$) [4], mais elle ne fournit pas de foncteur pour les types à trois variables libres ou plus.

4.2. Types de données abstraits

Considérons le module ci-dessous qui définit des listes singletons d'entiers.

```
type t = int list
let single n = [ n ]
let head = function [] → assert false | x :: _ → x
```

Fournir l'interface suivante à ce module permet d'implanter un type de données abstrait (ADT) qui garantit statiquement que l'assertion de la fonction **head** ne sera jamais violée.

```
type t
val single: int → t (* constructor *)
val head: t → int   (* getter *)
```

4. La valeur `Datatype.func` utilisée figure 5 est équivalente à `Function.instantiate` qui génère des instances monomorphes pour les types flèches de la forme $\alpha \to \beta$.

Cependant, dans le contexte de la section 2.1 où ce module serait chargé dynamiquement, personne ne pourrait l'utiliser car son type est statiquement inconnu. Ainsi, plaçons nous dans le cadre décrit section 2.1 et continuons cet exemple pour enregistrer dynamiquement les fonctions `single` et `head`.

```
share.ml: (* same as Figure 5 *)
plugin.ml:
(* continuing code at beginning of this section *)
let ty = Type.register "Plugin.t" [ 0 ]
let () = Share.register "Plugin.single" (Datatype.func Datatype.int ty) single
let () = Share.register "Plugin.head" (Datatype.func ty Datatype.int) head
```

La valeur de type `ty` est inaccessible hors du module `Plugin`. Ainsi, personne ne peut utiliser les fonctions `Plugin.single` et `Plugin.head`. Les déclarer en utilisant la valeur de type `Datatype.list Datatype.int` au lieu de `ty` résout ce problème mais brise l'abstraction, permettant ainsi de violer l'invariant de la structure de données.

Pour surmonter cette difficulté, la bibliothèque autorise à déclarer la valeur de type `ty` comme représentant un type abstrait enregistré dynamiquement *via* un argument optionnel booléen de `Type.register`. Cela permet d'accéder à cette valeur de type dans un autre module *via* la fonction `Type.get` de type `string` $\rightarrow$ `abstract Type.t` où `abstract` est un type inhabité. Ainsi, pouvons-nous continuer l'exemple précédent.

```
user.ml:
let ty = Type.get "Plugin.t"
let l = Share.get "Plugin.single" (Datatype.func Datatype.int ty) 1
let () = print_int (Share.get "Plugin.head" (Datatype.func ty Datatype.int) l)
```

Le type de `l` est ici `Type.abstract`, ce qui permet de ne pouvoir l'utiliser qu'en argument d'une fonction attendant un ADT comme `Plugin.head`. Cependant, comme toutes les valeurs de type représentant un ADT partagent le même type `Type.abstract Type.t`, cela peut entraîner des problèmes si une est utilisée à la place d'une autre. Pour cela, quand la fonction `Share.get "Plugin.head" (Type.func ty Type.int)` est appliquée, elle vérifie que son argument a bien été construit en utilisant un constructeur de `Plugin.t`. Dans le cas contraire, une erreur est levée indiquant *argument 1 of Plugin.head not built with a constructor of type Plugin.t*.

Dans ce but, la fonction `Type.String_tbl.add` enregistre une valeur v' observationnellement équivalente à son argument v, mais pouvant être (physiquement) différente. En effet, chaque constructeur f d'un ADT associé à une valeur de type ty enregistre les valeurs qu'il construit sur le principe suivant [5].

```
fun x →
  let y = f x in
  register_value ty y;
  y
```

De manière symétrique, chaque accesseur g de cet ADT vérifie que son argument a bien été construit par un de ses constructeurs sur le principe suivant.

```
fun x →
  if not (mem_value ty x) then fail;
  g x
```

L'algorithme exact est présenté figure 11. Il utilise entre autres des fonctions d'instrospection de la bibliothèque permettant par exemple de savoir si une valeur de type représente un type flèche $\tau_1 \rightarrow \tau_2$ et, le cas échéant, d'obtenir les valeurs de type correspondant aux types τ_1 et τ_2.

5. L'utilisation d'une table de hachage faible [9] permet de ne pas conserver de valeurs inutiles en mémoire.

```
let rec abstract s n ty x : Obj.t =
  if is_abstract ty then Abstract_values.add ty x;
  if Function.is_instance_of ty then
    let ty1, ty2 = Function.get_instance ty in
    (* ok: [x] is a function here *)
    let f : α → β = Obj.magic x in
    Obj.repr (fun (y: α) →
      if is_abstract ty1 && not (Abstract_values.mem a y) then
        raise (Incompatible_type
          (Format.sprintf
            "argument %d of %s not built with a constructor of type %s"
            n s (name ty1)));
      (Obj.obj (abstract s (succ n) ty2 (f y)) : β))
   else
     Obj.repr x

let add tbl s ty x =
  if Hashtbl.mem tbl s then raise (Already_exists s);
  Hashtbl.add tbl s { ty = ty; o = abstract s 1 ty x }
```

Figure 11 – Vérification pour ADT.

5. Travaux relatifs

Mêler typages statique et dynamique est un sujet qui a déjà donné lieu à de nombreux travaux, particulièrement pour les langages de programmation fonctionnels.

Ainsi, est-il possible d'étendre les langages de programmation dynamiquement typés avec des annotations de types pour effectuer des vérifications statiques et partielles de types. Ces extensions sont connus sous le nom de *typage graduel*[6] [23]. Nous adoptons ici l'approche inverse : introduire des vérifications dynamiques à un langage de programmation statiquement typé.

De ce côté, la plupart des approches sont orientées langages : soit un nouveau langage de programmation dédié est inventé, soit un existant comme Haskell ou ML est étendu avec des traits additionnels. Dans les deux cas, le but est d'intégrer du typage dynamique directement au sein d'un langage et de son système de types statique. Ces approches, quoique fort intéressantes, ne peuvent malheureusement pas être utilisées dans notre cadre, comme nous l'avons expliqué en introduction de la section 2 : nous ne pouvons utiliser de langages et d'extensions dédiés à la maintenance incertaine sur le long terme. Il n'en demeure pas moins intéressant de les comparer à la nôtre.

La plupart de ces approches sont fondées sur la notion de *dynamic*, introduite par Luca Cardelli dans le langage Amber [5]. Un *dynamic* est une paire constituée d'une valeur v et de son type statique τ. Comme le montre le type `dynamic` de la figure 8, un encodage d'une telle paire peut être construit en utilisant notre bibliothèque, qui ne supporte néanmoins pas les types polymorphes. Le plus souvent, une construction `typecase` primitive est également fournie afin de choisir l'action à exécuter en fonction du type statique présent dans le *dynamic*. L'égalité sur les valeurs de type et les fonctions `is_instance_of` fournies par la bibliothèque Type permettent d'encoder cette construction de manière certes assez peu élégante (ce qui peut être corrigé si on s'autorise l'utilisation de Camlp4). Étendre ML avec des *dynamics* a été assez étudié dans les années 1990, sans [1] et avec [2, 15] support du polymorphisme. En particulier, les travaux de Xavier Leroy et Michel Mauny [15] apportent une solution au problème de la sérialisation sûre et semblent même avoir été implantés dans une ancienne version de CAML. Néanmoins, les *dynamics* n'existent plus dans OCaml aujourd'hui. Dans les années

6. *gradual typing*

2000, d'autres langages fonctionnels comme Alice ML [20, 21] fondée sur Standard ML, Clean [18] fondée sur Haskell ou GML [12] ont également offert un support des *dynamics*.

D'autres approches orientées langages ne sont pas fondées sur cette notion de *dynamic*. Ainsi Acute [22] et son extension HashCaml [4] fournissent la sérialisation sûre en utilisant des représentations de type calculées à partir des *hashes* des définitions de modules, tandis que Grégoire Henry et *al* proposent une extension de OCaml pour la désérialisation sûre sans modifier le *runtime* OCaml [13]. Par ailleurs, des travaux plus théoriques ont aussi étudiés ces questions. En particulier, Karl Crary et *al* propose un lambda-calcul permettant de représenter les types par des termes à la façon de nos valeurs de type, ce qui leur permet d'exécuter dynamiquement des routines différentes en fonction du type statique [8]. Ils gèrent en particulier les types polymorphes tout en préservant l'abstraction. Néanmoins, les traits impératifs ne sont pas évoqués et ne semblent pas du tout évidents à ajouter à leur lambda-calcul pur.

Parmi les travaux sus-mentionnés, certains [4, 12, 13, 22] ne semblent pas pouvoir être appliqués dans tous les cas de figure qui nous intéressent. En outre, seuls quelques-uns cherchent à préserver l'abstraction [4, 20, 21].

Même si les approches orientées langages sont les plus étudiées, certaines autres sont orientées bibliothèques. D'abord, les distributions Haskell Hugs et GHC fournissent une implantation des *dynamics* sous forme d'une bibliothèque. Comme la nôtre, elles utilisent des traits non sûrs du langage et ne supporte que les instances monomorphes de types polymorphes. Néanmoins, aucun support des types de données abstraits n'est fourni. Une implantation totalement sûre de cette bibliothèque a été proposée [3], mais elle n'est malheureusement pas transposable en OCaml car elle requiert l'utilisation de types et de constructeurs universellement et existentiellement quantifiés.

Aucun langage ML ne possède par défaut de bibliothèque de *dynamics*. L'extension OCaml Deriving [28] fournit un préprocesseur et une bibliothèque pour la sérialisation sûre et le typage dynamique. Elle se rapproche de la nôtre sur certains points : on y trouve par exemple des combinateurs similaires à nos fonctions `Datatype.list` et `Datatype.func` pour gérer les instances monomorphes des types polymorphes. Elle est néanmoins intrinsèquement fondée sur des extensions syntaxiques Camlp4, ce qui l'éloigne d'une approche bibliothèque en pure OCaml. Du même ordre, Dyn [25] est une autre extension Camlp4 implantant les *dynamics*. Par ailleurs, des encodages sûrs des types universels en ML [26, 27] sont connus, vers lesquels les valeurs de chaque type peuvent être injectées ou surjectées. De tels types sont utiles pour définir des structures hétérogènes, mais ils ne peuvent être utilisés dans le contexte du chargement dynamique car les injecteurs et surjecteurs doivent être définis simultanément. Or, nous avons vu dans la section 2.1 que l'injecteur devait être défini dans le module `Plugin`, tandis que le surjecteur devait l'être dans le module `User`. Par ailleurs, l'assistant de preuves Coq possède de son côté une implantation non sûre des types universels *via* son module `Dyn` [7].

Un autre travail proche du nôtre est effectué chez LexiFi, un fournisseur d'applications logicielles et d'infrastructures technologiques pour les établissements financiers. LexiFi définit un type α `ttype` représentant la structure interne des types monomorphes à l'exécution [11]. La partie de la bibliothèque utilisée pour la désérialisation sûre et non présentée ici (section 2.3) possède une telle représentation, ce qui devrait permettre de l'utiliser dans les cas de LexiFi. En revanche, le type α `ttype` ne préserve pas l'abstraction, ce qui est problématique dans notre cas. Comme dans Type, les types fantômes sont utilisés pour garantir la sûreté de l'approche. En outre, nos objectifs respectifs sont différents : LexiFy utilise cette représentation pour automatiser la génération de parties d'interface graphique, pour décrire des schémas SQL, pour des communications inter-processus et pour du débogage. Une autre différence majeure est que les valeurs de type α `ttype` sont automatiquement générées par une version dédiée du compilateur OCaml.

7. De manière cohérente avec le nom du module, sa documentation utilise le mot «*dynamics*» au lieu de «type universel». Cependant, le code effectif semble réellement fournir une implantation non sûre des types universels.

6. Conclusion

Cet article a présenté la bibliothèque Type qui fournit une représentation dynamique des types OCaml statiques, y compris pour les instances monomorphes des types polymorphes et les types de données abstraits. Elle permet ainsi de voir les types OCaml comme des citoyens de première classe. Elle comble aussi le fossé séparant OCaml des langages dynamiques en mêlant vérifications statiques et dynamiques des types.

Cette bibliothèque a été créée pour, et est incluse dans, l'outil Frama-C, une plateforme d'analyses de programmes C à vocation industrielle. En son sein, trois utilisations différentes en sont faites. D'abord, elle permet d'enregistrer et d'accéder à des interfaces chargées et enregistrées dynamiquement. Ensuite, elle sert à interfacer un algorithme efficace dédié à la désérialisation en présence de *hashconsing*. Enfin, elle autorise l'implantation d'un algorithme de journalisation permettant de générer un code OCaml correct correspondant aux actions effectuées par l'utilisateur. Dans ces trois cas, Type permet de garantir l'absence d'erreur à l'exécution en cas d'erreur de développement, tandis que des possibilités d'instropection, même limitées, permettent d'élargir son champ d'application. Cet article n'a présenté que les points clés critiques de la bibliothèque. L'implantation réelle fait de l'ordre de 2300 lignes de code et possèdent des fonctionnalités supplémentaires, comme la gestion des labels et des arguments optionnels [8].

Des possibilités potentiellement utiles ne sont néanmoins pas encore implantées. La plus importante est l'enregistrement de types ouverts, comme les types polymorphes, les variants polymorphes ou les objets. Au delà de la difficulté intrinsèque de cet ajout, l'enregistrement de tels types n'est jamais requis dans Frama-C aujourd'hui et ne devrait pas l'être au moins à moyen terme, car les valeurs enregistrées servent principalement à paramétrer les analyseurs et à les exécuter, ce qui ne nécessite que des types monomorphes clos et relativement simples.

Références

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2) :237–268, April 1991.

[2] Martín Abadi, Luca Cardelli, Benjamin Pierce, Didier Rémy, and Robert W. Taylor. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5 :92–103, 1994.

[3] Arthur I. Baars and S. Doaitse Swierstra. Typing dynamic typing. *SIGPLAN Notices*, 37(9) :157–166, 2002.

[4] John Billings, Perter Sewell, Mark Shinwell, and Rok Strniša. Type-safe distributed programming for OCaml. In *Proceedings of the 2006 workshop on ML*, pages 20–31, New York, NY, USA, 2006. ACM.

[5] Luca Cardelli. Amber. In *Combinators and Functional Programming Languages*, pages 21–47, 1985.

[6] Emmanuel Chailloux, Pascal Manoury, and Bruno Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, 2000.

[7] Loïc Correnson, Pascal Cuoq, Armand Puccetti, and Julien Signoles. *Frama-C User Manual*, April 2010. http://frama-c.com.

[8] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. *Journal of Functional Programming*, 12(6) :567–600, 2002.

8. La partie liée à la désérialisation sûre et le module Datatype n'existent pas encore sous cette forme dans la version publique actuelle (Frama-C Boron), mais seront disponibles dans la suivante.

[9] Pascal Cuoq and Damien Doligez. Hashconsing in an incrementally garbage-collected system : a story of weak pointers and hashconsing in OCaml 3.10.2. In *Proceedings of the 2008 Workshop on ML*, pages 13–22, New York, NY, USA, 2008. ACM.

[10] Pascal Cuoq and Julien Signoles. Experience Report : OCaml for an Industrial-Strength Static Analysis Framework. In *Proceedings of the 2009 International Conference on Functional Programming*, pages 281–286. ACM, September 2009.

[11] Alain Frisch. Communication personnelle par courrier électronique, September 2009.

[12] Jun Furuse. *Extensional polymorphism : theory and applications*. PhD thesis, University Paris 7, December 2002.

[13] Grégoire Henry, Michel Mauny, and Emmanuel Chailloux. Typer la désérialisation sans sérialiser les types. *Technique et Science Informatiques*, 26(9) :1067–1090, 2007.

[14] Xavier Leroy. *Typage polymorphe d'un langage algorithmique*. Thèse de doctorat, Université Paris 7, 1992.

[15] Xavier Leroy and Michel Mauny. Dynamics in ML. *Journal of Functional Programming*, 3(4) :431–463, 1993.

[16] Donald Michie. Memo functions and machine learning. *Nature*, 218 :19–22, 1968.

[17] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17 :348–375, 1978.

[18] Rinus Plasmeijer and Marko van Eekelen. *Clean Language Report, version 2.1*, November 2002.

[19] Morten Rhiger. A foundation for embedded languages. *ACM Transactions on Programming Languages and Systems*, 25(3) :291–315, 2003.

[20] Andreas Rossberg, Didier Le Botlan, Guido Tack, Thorsten Brunklaus, and Gert Smolka. Alice through the looking glass. In *Trends in Functional Programming*, pages 79–95, 2004.

[21] Andreas Rossberg, Guido Tack, and Leif Kornstaedt. Status report : HOT pickles, and how to serve them. In Claudio Russo and Derek Dreyer, editors, *ML*, pages 25–36. ACM, November 2007.

[22] Peter Sewell, James J. Leifer, Keith Wansbrough, Francesco Zappa Nardelli, Mair Allen-Williams, Pierre Habouzit, and Viktor Vafeiadis. Acute : high-level programming language design for distributed computation. In *Proceedings of the tenth International Conference on Functional Programming*, pages 15–26, New York, NY, USA, 2005. ACM.

[23] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *Workshop on Scheme and Functional Programming*, September 2006.

[24] Julien Signoles, Loïc Correnson, and Virgile Prevosto. *Frama-C Plug-in Development Guide*, April 2010.

[25] Till Varoquaux. The Dyn project, October 2009. Affiliated to Jane Street Capital. `https://forge.ocamlcore.org/projects/dyn`.

[26] Stephen Weeks, Andrej Bauer, and Alain Frisch. A universal type ?, March 2008. Blog entry at `http://janestcapital.com/?q=node/18`.

[27] Stephen Weeks, Matthew Fluet, and Vesa Karvonen. MLton UniversalType, May 2005. `http://mlton.org/UniversalType`.

[28] Jeremy Yallop. Practical generic programming in ocaml. In *Proceedings of the 2007 workshop on Workshop on ML*, pages 83–94, New York, NY, USA, 2007. ACM.

A. Preuves

Propriété 1 (Injection des type monomorphes dans les valeurs de type) *L'égalité des valeurs de type implique l'égalité entre les types statiques monomorphes correspondants,* i.e. *:*

$$\forall \textit{ types monomorphes } \tau \textit{ et } \tau', \forall ty \in \mathcal{T}_\tau, \forall ty' \in \mathcal{T}'_\tau, \texttt{Type.equal}\ ty\ ty' \implies \tau = \tau'.$$

Preuve Soit τ, τ', $ty \in \mathcal{T}_\tau$, $ty' \in \mathcal{T}'_\tau$ tels que *Type.equal ty ty'*. La fonction `Type.register` est le seul constructeur de valeurs de type α `Type.t`. Deux appels successifs ne génèrent jamais deux valeurs équivalentes car elle vérifie que l'argument *name* est unique. Par suite, *Type.equal ty ty'* implique que *ty* et *ty'* représentent exactement la même valeur et donc qu'elles ont le même type statique. ■

La propriété 2 est triviale : elle n'est pas prouvée ici.

Propriété 4 (Correction forte) *À l'exception éventuelles des expressions non sûres utilisées en dehors du module* `Type`*, toute expression bien typée ne peut engendrer une erreur à l'exécution.*

Preuve Le seul cas non trivial est l'évaluation d'une expression e de la forme `Type.StringTbl.find tbl s ty` dans un contexte d'évaluation de type τ dans lequel *tbl*, *s* et *ty* ont respectivement les types *Type.String_tbl.t*, *string* et τ'' *Type.t*. Le type statique de `Type.String_tbl.find` impose que $\tau = \tau''$.

- S'il n'existe aucun s', ty' et v tels que $s = s'$ et `Type.StringTbl.add tbl s' ty' v` n'ait été précédemment évalué, alors l'exception `Unbound_value s` est levée : par suite e n'engendre pas une erreur à l'exécution (au sens de Milner [17]).
- Supposons à présent qu'il existe de tels s', ty' and v. Par typage statique de la fonction `Type.StringTbl.add`, il existe un type τ' tel que ty' et v aient respectivement les types τ' *Type.t* et τ'. Distinguons deux cas.
  - Si *not* (*Type.equal ty ty'*), alors la function `type_error` est appelée et l'exception `Incompatible_type` est levée : e n'engendre donc aucune erreur à l'exécution (au sens de Milner).
  - Si *Type.equal ty ty'*, alors, d'après la propriété 1, $\tau = \tau'$. Par suite, aucune erreur de type ne survient et l'expression `Obj.obj data.o` est évaluée dans un contexte d'évaluation de type τ. Cette expression est réductible en `Obj.obj (Obj.repr v)` qui est évalué de manière sûre en v, de type $\tau' = \tau$. Là encore, e ne génère pas d'erreur à l'exécution. ■

Propriété 5 (Complétude) *En supposant valide l'hypothèse 1, pour chaque programme p n'étant pas évalué en une erreur et utilisant les tables hétérogènes non sûres, le programme p dans lequel les occurrences de ces tables ont été substituées par des occurrences similaires de type* `Type.String_tbl.t` *est bien typé, ne peut être évalué en une erreur et ne lève aucune exception à l'exécution en provenance du module* `Type`*.*

Preuve Le seul cas non trivial est l'évaluation d'une expression e de la forme `Hashtbl.find dynamic_values s` vers une valeur v dans un contexte de type τ dans lequel *dynamic_values* et s ont pour types respectifs $(string, Obj.t)$ *Hashtbl.t* et *string*.

Soit e' l'expression e dans laquelle les occurrences des tables hétérogènes non sûres (en particulier *dynamic_values*) ont été substituées par des occurrences similaires du `Type.String_tbl`.

Comme l'évaluation de e n'engendre pas d'erreur à l'exécution, v est de type τ et l'expression `Hashtbl.add dynamic_values s v` a été précédemment évaluée. Ainsi, comme e' et e sont similaires à substitution près, il existe une table *tbl* de type *Type.String.t*, *ty* de type τ *Type.t* et *ty'* de type τ *Type.t* tels que e' soit l'expression `Type.String_tbl.find tbl s ty` et que l'expression

`Type.String_tbl.add tbl s ty' v` ait été précédemment évaluée. On peut noter que, d'après la propriété 2, ty and ty' existent forcément.

Par stabilité du typage par substitution (voir par exemple la thèse de Xavier Leroy [14]), le type de e' est τ. Donc e' est bien typé dans un contexte de type τ.

Par l'hypothèse 1, on déduit que $Type.equal\ ty\ ty'$. Ainsi, d'après l'implantation du module `String_tbl`, e' n'engendre aucune erreur à l'exécution et ne lève aucune exception provenant de la bibliothèque `Type`. ■

Propriété 6 (Équivalence sémantique) *Pour chaque programme p utilisant les tables hétérogènes non sûres et étant évalué en une valeur non erronée v, si le programme p dans lequel les occurrences de ces tables ont été substituées par des occurrences similaires de type `Type.String_tbl.t` est bien typé et ne peut ni être évalué en une erreur ni lever aucune exception en provenance du module `Type`, alors ce programme s'évalue aussi en v.*

Preuve Le seul cas non trivial est l'évaluation d'une expression e de la forme `Hashtbl.find dynamic_values s` vers une valeur v, dans un contexte de type τ dans lequel $dynamic_values$ et s ont pour types respectifs $(string, Obj.t)\ Hashtbl.t$ et $string$.

Soit e' l'expression e dans laquelle les occurrences des tables hétérogènes (en particulier $dynamic_values$) ont été substituées par des occurrences similaires du module `Type.StringTbl`.

Comme e' est bien typé, n'engendre aucune erreur à l'exécution et ne lève aucune exception en provenance du module `Type`, elle est évaluée vers la valeur `Obj.obj (Obj.repr v)` qui est équivalente à v (le schéma de la preuve est similaire à celui de la preuve de la propriété 5).

Nous pouvons juste noter que si nous utilisons la fonction `abstract` présentée figure 11, e' est évaluée vers la valeur `abstract s 0 ty v` (pour un certain s et ty) qui est une valeur observationnellement équivalente à v dès que e' est bien typé, n'engendre aucune erreur à l'exécution et ne lève aucune exception en provenance du module `Type`, ce qui est le cas ici. ■