Entrées/Sorties de valeurs en Caml

J. Furuse & P. Weis

INRIA Rocquencourt

Résumé

Un mécanisme d'"entrées/sorties de valeurs" consiste à offrir au programmeur la possibilité de lire ou d'écrire des valeurs de types de données structurées comme s'il s'agissait d'entités atomiques, caractères ou chaînes, et sans qu'il soit besoin d'un codage explicite dans le programme.

Dans le contexte des langages de programmation fortement typés, ce genre d'entrées/sorties de haut niveau soulève de nouveaux problèmes de sûreté: dès qu'un programme a écrit une valeur structurée dans un fichier, n'importe quel autre programme peut relire cette valeur dans un contexte de typage complètement différent; pour assurer que la sûreté du typage n'est pas compromise, il faut prouver que la valeur lue a nécessairement un type compatible avec le contexte local du programme qui effectue la lecture. Nous étudions de quel genre d'information de type il faut accompagner la valeur écrite, par quel moyen procurer cette information aux primitives d'entrées/sorties, et quels mécanismes de typage à l'exécution peuvent assurer la sûreté du typage à la lecture des valeurs structurées.

Basé sur la théorie du polymorphisme générique, le mécanisme que nous obtenons est un système sûr d'entrées/sorties de valeurs qui permet l'échange de données arbitraires entre des programmes Caml arbitraires.

Cette réalisation nous fournit en outre deux résultats supplémentaires: sur le plan pratique, des valeurs dynamiques sans aucun autre effort d'implémentation, et sur le plan plus théorique la *règle de clôture des valeurs de type* qui autorise un traitement pertinent et original des compositions essentiellement polymorphes des fonctions génériques.

1. Introduction

Tous les systèmes d'exploitation offrent des facilités de lectures/écritures qui manipulent des quantités élémentaires d'information, en général des octets ou caractères. À l'aide de ces entrées/sorties très simples, les langages de programmation de haut niveau fournissent l'accès en lecture/écriture à des périphériques variés, en particulier les fichiers, les écrans et les claviers. En général, le mécanisme d'entrées/sorties du système d'exploitation se retrouve directement dans l'interface de programmation: les abstractions d'entrées/sorties manipulent des caractères, ou au mieux procurent la lecture écriture directe de données simples comme les nombres et les chaînes de caractères. C'est loin d'être satisfaisant: dans les langages de programmation modernes, le programmeur est équipé pour définir le plus facilement du monde les structures de données les plus complexes et pour manipuler sans difficultés leurs éléments; pourtant, dès qu'il s'agit de communiquer ces données à l'extérieur, aucune aide n'est plus fournie par le langage. Ainsi, pour lire ou écrire les valeurs structurées, le programmeur doit écrire ses propres procédures spécifiques, en général une fonction de conversion des valeurs structurées en suites de caractères à l'écriture, et la fonction réciproque à la lecture. C'est une tâche longue, complexe, et une source inépuisable d'erreurs. C'est tout aussi malheureusement une tâche répétitive: le code écrit est absolument spécifique à la structure de données manipulée dont toute modification doit être immédiatement répercutée dans les procédures de lecture/écriture.

C'est pourquoi il nous semble que le langage de programmation se doit de procurer une interface d'entrées/sorties de haut niveau qui traite directement les données complexes que le programmeur manipule. Nous appelons *entrées/sorties de valeurs* ce genre de mécanisme avancé. Ce nouveau trait pose de nouveaux problèmes, en particulier dans un langage fortement typé statiquement, puisqu'une valeur structurée écrite dans un fichier doit être relue comme une valeur du même type pour ne pas invalider les hypothèses faites par le contrôleur de type lors du typage du programme lecteur. Encore peut-on tolérer que la relecture ne s'effectue pas avec *exactement le même* type que l'écriture, pourvu qu'elle se fasse avec un type corrélé au type d'écriture et compatible avec le type attendu par le programme lecteur. En tout état de cause, il faut nécessairement connaître le type d'écriture d'une valeur au moment de sa relecture, pour pouvoir vérifier cette corrélation.

Nous avons mis au point un système d'entrées/sorties de valeurs pour Objective Caml qui assure ainsi la relecture sans erreur des valeurs structurées écrites sur disque. Notre système vérifie automatiquement que la valeur relue a un type compatible avec le contexte de relecture. Il est basé sur le polymorphisme générique, une discipline de typage qui permet l'insertion automatique d'arguments comportant des informations de type calculées statiquement. Ces informations de type sont ainsi fournies aux primitives de lecture/écritures: les primitives d'écriture reçoivent le type de la valeur à écrire et en déduisent une information de type qu'elles écrivent sur le fichier en même temps que la valeur; de manière corrélative, les primitives de lecture reçoivent le type attendu par le contexte de relecture pour la valeur à lire et peuvent ainsi vérifier que ce type attendu et l'information de type qui accompagne la valeur relue sont compatibles, prouvant ainsi que l'utilisation de la valeur relue ne provoquera pas d'erreur à l'exécution.

Le mécanisme précédent est simple et convainquant pour un programme unique qui écrit et relit des valeurs qu'il a lui-même créées. En effet, la compatibilité des types de lecture et d'écriture de ces valeurs se résume à peu près à vérifier l'identité de ces deux types (en passant sous silence les problèmes de polymorphisme, non fondamentaux ici). Une difficulté supplémentaire apparaît lorsque l'on veut échanger des valeurs entre plusieurs programmes différents, compilés dans des contextes différents. En effet, la compatibilité des types de lecture et d'écriture ne peut plus se résumer à vérifier l'identité des noms de types: le type appelé `syntax` défini dans le programme `P1` n'a aucune raison d'être le même que le type `syntax` d'un programme `P2` sans rapport avec `P1`. Pourtant, il faut permettre l'échange de données du type `syntax` entre `P1` et `P2` si le type `syntax` de `P1` et le type `syntax` de `P2` sont effectivement identiques, au sens où ils ont la même définition. On voit donc ici qu'il faut abandonner l'habituelle équivalence par nom de l'algèbre de type de ML pour adopter un autre genre d'équivalence, plus sémantique et plus structurelle.

Nous définissons pour cela une forme canonique des définitions de types où tous les noms ont été effacés: les *définitions sans noms*. Les noms de type sont alors simplement remplacés par une *empreinte* de leur définition sans noms (un «digest MD5» de cette définition).

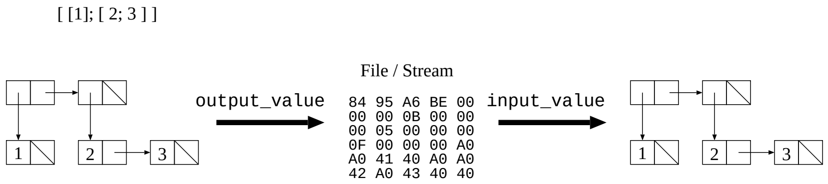
Nous obtenons ainsi une forme canonique des expressions de type: les expressions de type en *représentation externe*, qui font complètement abstraction des noms de types. Cette représentation définit ainsi une notion d'isomorphisme entre expressions de types, l'équivalence pour les représentations externes des valeurs ou (*e-isomorphisme*). Cette notion modélise l'idée intuitive de types compatibles pour les entrées/sorties de valeurs: deux types e-isomorphes possèdent des valeurs qui ont la même représentation sur fichier, ou encore toute valeur de l'un des types est une valeur admissible de l'autre. Ainsi, si les types d'écriture et de lecture d'une valeur sont deux expressions de type e-isomorphes, alors la lecture peut s'effectuer sans danger. La notion d'e-isomorphisme définit ainsi la notion de communication sûre pour les échanges de valeurs entre programmes indépendants.

2. Primitives non sûres de lecture écriture de valeurs

Le langage Objective Caml fournit une possibilité de lecture/écriture de valeurs sur fichier, aux risques et périls du programmeur; on dispose des deux primitives polymorphes suivantes:

$$
\begin{array}{lcl}
\texttt{output\_value} & : & \texttt{out\_channel} \rightarrow \alpha \rightarrow \texttt{unit} \\
\texttt{input\_value} & : & \texttt{in\_channel} \rightarrow \alpha
\end{array}
$$

`output_value` crée la représentation binaire d'une valeur arbitraire et l'écrit dans un canal de sortie. Symétriquement, `input_value` relit cette représentation, la décode et reconstruit en mémoire la même structure que l'originale écrite sur disque (avec le même partage).



Bien entendu, lorsqu'on relit une valeur, il n'y a aucun moyen de vérifier si la valeur possède le type espéré: les types de lecture et d'écriture ne sont pas corrélés et ce mode de communication est essentiellement non typé. C'est pourquoi ce mécanisme n'est pas sûr: par exemple, on peut par mégarde relire un entier comme une chaîne de caractères et produire une erreur de type à l'exécution.

```
# 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**

Ces primitives sans vérification ne servent pas seulement à sauver des valeurs sur fichiers, mais sont aussi utilisées pour la communication de valeurs entre programmes Caml différents. En ce cas, le problème de typage est encore plus grave, car un programme peut relire par erreur une valeur d'un type qui lui est totalement inconnu puisqu'il ne le définit même pas. L'usage de ces primitives essentiellement non typées est donc dangereuse en pratique et est à la source d'un nombre d'erreur non négligeables. C'est pourquoi, au-dessus de cette couche non typée, nous proposons une couche typée et sûre d'échange de valeurs qui isole complètement l'utilisateur de ces primitives non sûres.

3. Notre but

Nous voulons construire un mécanisme fiable de communication de valeurs entre programmes Caml. Pour cela, nous allons ajouter au mécanisme d'entrées-sorties non typées, les informations qui sont calculées par le contrôleur de types.

Vérification à l'exécution des marqueurs de valeurs

L'une des solutions les plus simples et les plus puissantes de vérification de la cohérence des valeurs lues, consiste à effectuer le «typage» à l'exécution des valeurs relues, en vérifiant que les représentations des valeurs sont des valeurs légales du type attendu à la relecture (vérification des marqueurs du gestionnaire de mémoire). Cette vérification des marqueurs des valeurs (*tag checking*) rejette toutes les lectures de valeurs qui pourraient provoquer des erreurs à l'exécution; de plus cette méthode est parfaitement compatible avec les versions dangereuses des primitives de lecture/écriture puisqu'il n'est pas nécessaire d'écrire sur fichier des informations de type supplémentaires. Cependant le programme lecteur doit disposer de toutes les définitions de tous les types qu'il doit relire (en pratique cela signifie simplement tous les types que définit le programme lecteur, et sans doute, pour simplifier la compilation, tous les types de tous les programmes); de plus, la relecture nécessite un parcours complet de la valeur à relire (avec un mécanisme spécial pour éviter de boucler en cas de valeur cyclique), ce qui est réellement coûteux en pratique.

C'est pourquoi nous avons mis au point un moyen plus efficace d'assurer la sûreté des relectures de valeurs, qui évite à la fois de garder en mémoire toutes les définitions de type et de parcourir récursivement toutes les valeurs lues.

Notion de type *externe*

Au lieu de systématiquement procéder à ce typage récursif complet des valeurs lues, nous pouvons utiliser une propriété que nous assure le typage fort: nous savons que *les valeurs lues étaient bien typées* dans leur programme d'origine. Il est donc inutile de les retyper à partir de zéro, il suffit de vérifier que leur type d'origine est compatible avec le type attendu. C'est pourquoi lorsqu'on *exporte* une valeur, on exporte aussi un «témoin de bon typage»: une information de type appelé le *type externe* de la valeur. À la lecture, on lit aussi ce type externe, puis l'on vérifie simplement que ce type est acceptable par le contexte, au moment de l'*import*. Nous utilisons les termes *import* et *export*, parce que notre système n'accepte les valeurs en entrée qu'après une série de vérifications, un peu comme la douane dans le monde réel. Plus techniquement, notre système assure exactement la même discipline de typage fort que le langage hôte.

4. Polymorphisme générique

Pour effectuer des entrées/sorties de valeurs fortement typées, les primitives de bas niveau qui lisent et écrivent les données doivent évidemment avoir accès au type des valeurs à échanger, aussi bien à l'export pour écrire le témoin de bon typage, qu'à l'import pour disposer du type attendu par le contexte (et le confronter au témoin de bon typage).

Pour transmettre ces types et définir ces primitives de lecture/écriture nous utilisons le *polymorphisme générique* [1]. En effet, le polymorphisme générique permet de typer des fonctions polymorphes *ad hoc* dont le comportement dépend de leur type d'utilisation: une telle fonction *générique* reçoit en argument une information de type, plus précisément une valeur du langage qui représente un type, ce qu'on appelle une *valeur de type*. Dans le polymorphisme générique, cet argument supplémentaire n'est pas écrit dans le programme de l'utilisateur : c'est le compilateur qui le fournit automatiquement en fonction du contexte statique d'utilisation de la fonction générique.

Valeurs de type

Les types statiques du langage sont donc représentés à l'exécution par des valeurs du langage appelées *valeurs de type*. Les valeurs de type appartiennent à un nouveau type de données de base que nous avons appelé `rtype` et dont voici la définition:

```
type rtype =
| RTvar of int                   (* variables de type *)
| RTarrow of rtype * rtype       (* type des fonctions *)
| RTtuple of rtype list          (* type des n-uplets *)
| RTconstr of rpath * rtype list (* types définis *)
```

Bien évidemment, les valeurs du type `rtype` permettent de représenter n'importe quelle expression de type Caml:

- une variable de type est représentée par un entier qui l'identifie;

- les cas des types de fonctions et de n-uplets sont évidents;
- les arguments des constructeurs de type sont tout simplement représentés par une liste de valeurs de type.

Les constructeurs de type eux-mêmes sont identifiés par une représentation canonique de leur *chemin d'accès* ou *path*. Au niveau du langage de base, un chemin d'accès utilise la notation en point pour identifier de façon unique toutes les entités définies dans le langage. Par exemple, le chemin d'accès `List.map` désigne l'identificateur `map` du module `List`. De même, les types de données d'un programme sont uniquement déterminés par leur chemin d'accès qui désigne le module qui les définit et le nom utilisé dans ce module pour les définir. Nos *valeurs de chemins d'accès*, ou `rpath`, ne sont autres que des codages des chemins d'accès du langage.

Nous notons «τ», la valeur de type qui représente le type statique τ. Par exemple, «`int` $\rightarrow$ `int`» est la valeur de type qui représente le type `int` $\rightarrow$ `int`, et l'on a

«`int` $\rightarrow$ `int`» = `RTarrow (RTconstr (Pervasives.int, []), RTconstr (Pervasives.int, []))`

Pour comprendre comment marche le polymorphisme générique, et comment il va nous permettre de transmettre les types des valeurs écrites et lues aux primitives d'entrées/sorties, étudions brièvement l'exemple emblématique de la fonction `print`.

Application de fonctions génériques

`print` a pour schéma de type $\forall\delta.\delta \rightarrow$ `unit`, où δ est une variable de type *dynamique*. Les variables de type dynamiques sont une nouvelle classe de variables de type introduite par le polymorphisme génériques: elles dénotent la présence des arguments de type implicites des fonctions génériques. Ainsi, quand `print` est appliquée à une donnée, le compilateur ajoute un argument supplémentaire à l'application: le type utilisé pour δ(plus exactement la *valeur de type* correspondante à l'occurrence d'utilisation de δ). En bref, cet argument supplémentaire est tout simplement le type de l'argument de la fonction `print`.

Par exemple, si nous appliquons `print` à une liste d'entiers, le programme compilé recevra deux arguments: la valeur de type «`int list`» et la liste elle-même. En pseudo-code Caml, cela donne:

```
print : ∀δ.δ → unit
print [1;2;3]
 ⟹ print_comp «int list» [1;2;3]
```

Définition de fonctions génériques

La fonction `print` est évidemment définie par une «analyse de cas de types», selon le type de son argument. C'est ce que nous appelons une définition de *fonction générique*:

```
generic rec print = case δ  → unit of
| «int» → print_int
| «float» → print_float
| «string» → print_string
| «τ list» → (fun l → List.iter print l)
| ...
```

La sémantique des fonctions génériques est définie par une traduction vers les fonctions Caml ordinaires. Ainsi la sémantique de la fonction générique `print` ci-dessus est la définition de la fonction Caml `print_comp` suivante:

```
let rec print_comp = fonction
| RTconstr (Pervasives.int, []) → print_int
| RTconstr (Pervasives.float, []) → print_float
| RTconstr (Pervasives.string, []) → print_string
| RTconstr (Pervasives.list, [tau]) → (fun l → List.iter (print_comp tau) l)
| ...
```

La fonction `print_comp`, sémantique (ou implémentation) de `print`, reçoit une valeur de type et l'argument à imprimer, et appelle la fonction d'impression correspondante. Pour les listes, la fonction s'appelle récursivement pour chaque élément de la liste, avec une valeur de type qui correspond au type de l'élément.

Valeurs dynamiques

Dans le papier [1], l'un des auteurs avait remarqué que le polymorphisme générique offre gratuitement les valeurs dynamiques, puisqu'il permet de définir très simplement les deux primitives `dynamic` and `coerce`, qui respectivement créent et déstructurent les *valeurs dynamiques*[3, 4]. En effet, leur compilation est très simplement:

```
dynamic : ∀δ.δ → dyn
let dynamic_comp ty v = (ty, v)

coerce : ∀δ.dyn → δ
let coerce_comp t (ty,v) =
  if is_compatible t ty then v
  else raise (Invalid_coercion (t, ty))
```

`dynamic` retourne une valeur de type `dyn` (un nouveau type abstrait pour les valeurs dynamiques). Cette fonction reçoit en fait une valeur `v` et sa valeur de type `ty` et les regroupe simplement en un couple. Symétriquement, `coerce` reçoit une valeur dynamique créée par `dynamic` et un type attendu `t`, et vérifie que la valeur peut s'utiliser comme un objet de type `t`. Le prédicat `is_compatible` est une primitive qui vérifie que le type encapsulé `ty` dans la valeur dynamique est plus général que `t` (plus général ayant ici la signification habituelle du prédicat "plus général" des typeurs de ML).

De façon analogue, notre interface pour les entrées/sorties de valeurs est simplement constituée des deux fonctions, `export_value` et `import_value` qui sont définies à peu près comme les primitives pour les valeurs dynamiques: `export_value` crée le couple d'une valeur et d'une information de type, son type «externe», et l'écrit dans un fichier, tandis qu'`import_value` compare l'information de type externe de la valeur lue avec le type attendu, puis retourne la valeur lue si les informations de type sont compatibles.

5. Types externes

Quel genre d'information de type devons-nous associer aux valeurs écrites ? Comme nous l'avons déjà dit, cette information doit représenter une expression de type quelconque qui puisse être interprétée sans erreur par des programmes qui n'ont pas de relation entre eux. Cela implique en particulier que cette information ne contienne rien de spécifique à un programme donné. Il ne suffit donc pas d'écrire et de lire des valeurs dynamiques pour résoudre le problème des entrées-sorties de valeurs.

5.1. L'identification des types par nom n'est pas possible

L'identification des types en ML s'effectue par noms uniques, au sens où deux types de même dénomination externe ne sont pas confondus par le langage. Un cas simple est celui d'un type `tree` qui serait défini dans deux modules différents. On considère que les deux types n'ont en fait pas le même nom en les qualifiant avec les noms de leur module de définition: `M1.tree` est évidemment différent de `M2.tree`. En général, c'est ce qu'on attend du langage, car on veut pas confondre le type `tree` donnant la liste des arbres poussant en Europe avec le type `tree` des arbres pagodes de l'algorithmique. Cependant, le langage refuse dans tous les cas de confondre deux types: les définitions de type sont génératives, ce qui signifie que toute nouvelle définition crée un nouveau type incompatible avec tous les autres. Ainsi `M1.tree` et `M2.tree` sont incompatibles *même si la définitions de* `M1.tree` *est une copie de celle de* `M2.tree`. Si l'on pousse cette logique au cas de deux programmes indépendants, il n'y a aucune raison de considérer un type quelconque du programme `P1` comme compatible avec un type quelconque du programme `P2`, et *même si les deux types ont exactement la même définition*. La compatibilité des types serait alors restreinte aux types de baser ou aux types ayant le même nom qualifié au sein du *même programme*. Cela empêcherait la communication de valeurs de types de données complexes entre des programmes différents, limitant cette communication à un seul programme ou, à la limite, à deux incarnations du même exécutable. C'est évidemment inacceptable.

5.2. L'identification des types par leurs noms n'est pas sûre

Les *valeurs de type* que nous avons définies plus haut ne suffisent donc malheureusement pas: en effet si ces valeurs de type transportent les noms qualifiés des types. Elles conviendraient donc parfaitement à l'échange de données entre des fonctions définies au sein d'un même programme, elles ne sont pas adéquates pour l'échange de données entre des programmes différents. Cela tient toujours à l'identification des types par leurs noms (leurs chemins d'accès) qui crée maintenant des problèmes de refus de communication pourtant licite, ou pire d'acceptation de communication interdite. En effet, considérons deux programmes P et Q qui communiquent; P écrit une valeur de type t; Q tente de relire cette valeur avec le type t'; plusieurs cas d'erreurs peuvent se produire,

- les types t' et t sont exactement équivalents car leurs définitions dans les deux programmes sont mots à mots les mêmes; malheureusement, les noms t et t' sont différents, si bien que les types sont considérés comme incompatibles et la valeur lue est rejetée.
- les noms t et t' sont par hasard les mêmes, alors que leurs définitions sont sans rapport; la valeur lue est acceptée alors qu'elle n'est pas une valeur possible du type t de Q. Il y a de grandes chances qu'une erreur à l'exécution intervienne ...

En bref, il n'y a aucun moyen de garantir que le type appelé `t` dans P et le type appelé `t` dans Q sont équivalents ou compatibles, puisqu'ils ont été définis de façon indépendantes (peut-être par des programmeurs différents en des lieux différents!).

5.3. L'identification des types par nom n'est pas souhaitable

Allons plus loin: des structures de données isomorphes (au sens où elles définissent les mêmes valeurs) peuvent avoir des noms différents; pourtant elles doivent être considérées comme parfaitement compatibles au sens de la relecture des données: il est donc nécessaire d'identifier des types de noms différents.

Nous allons encore plus loin, car nous considérons que les autres noms associés aux définitions de type n'ont pas plus d'importance que le nom du type défini. Par exemple, les deux définitions de type suivantes représentent exactement la même structure d'arbres, bien que tous les noms définis soient différents:

Programme E:	Programme F:
`type` α `tree =`	`type` α `arbre =`
`| Leaf of` α	`| Feuille of` α
`| Branch of` α `tree list`	`| Branche of` α `arbre list`

Or, la représentation interne des valeurs de ces deux types sont exactement identiques en Objective Caml. C'est pourquoi une valeur exportée avec le type `int tree` peut être importé sans danger comme une valeur du type `int arbre` et vice versa.

Ce genre de "confusion" de types est-il utile et désirable pour les entrées-sorties de valeurs ? Nous en sommes complètement persuadés: dans notre expérience de programmation d'application de grandes tailles comportant des bases de données simplifiées, nous utilisons souvent les entrées/sorties de valeurs pour sauver les structures de données sur fichiers, parce que justement ce mécanisme évite d'écrire des imprimeurs pour sauver les données et des analyseurs syntaxiques pour les relire. Souvent, il arrive que les noms de type et les étiquettes soient modifiés lors des changements de versions du logiciel, nous voulons alors absolument pouvoir relire les fichiers de données écrits par les anciennes versions de l'application, pourvu que la représentation interne des données ne soient pas modifiée.

Pour les entrées/sorties de valeurs, il faut donc abandonner la méthode habituelle d'identification des types de ML par leurs noms uniques, mais on doit recourir à une identification plus subtile liée à la sémantique des types, ou plus simplement à leur définition.

Nous formalisons maintenant ces idées avec le concept de *définitions sans noms.*

Définition de type sans noms et *empreintes* de types

Pour expliquer les décisions prises pour fabriquer notre systèmes nous énonçons, d'abord informellement, les propriétés du compilateur Objective Caml qui nous sont indispensables. Ces propriétés seront énoncées formellement et prouvées dans le lemme 2.

Hypothèse 1 *En Objective Caml, les informations suivantes n'affectent pas les représentations des valeurs manipulées:*

- *les noms des types de données*
- *les noms des constructeurs des types somme*
- *les noms des labels des types enregistrement*

En s'appuyant sur ces propriétés, notre système identifie les types par leurs *définitions sans noms,* c'est-à-dire par leurs définitions desquelles tous les noms ont été enlevés. Intuitivement, deux types de données qui ont la même définition sans noms sont sémantiquement équivalents et la communication de valeurs entre ces types ne peut pas provoquer d'erreur à l'exécution. Deux types ayant la même définition sans noms sont donc qualifiés de types *isomorphes.*

Définitions de type sans noms

nameless-typedefs	::=	`type` *typedef* `et ...et` *typedef*	Définition sans noms
typedef	::=	$(\alpha_1, \ldots, \alpha_m)$ $\#n$ `=` *defbody*	
defbody	::=	`_ of` τ_e $\mid \ldots \mid$ `_ of` τ_e	Types somme
	$\mid$	`{_:`τ_e`; ...; _:`τ_e`}`	Enregistrements

Les définitions de types sans noms ont exactement la même structure que les définitions de types habituelles, mais les noms de constructeurs, de labels et de types sont omis. Les noms des types définis sont remplacés par de simples entiers: ainsi l'entier $\#n$ qui représente le nom du type `t` est le numéro de la définition du type `t` dans la liste des définitions (éventuellement récursives) où `t` est défini.

Par exemple, le type `arbre` a la définition sans noms suivante:

```
type α #1 =
   | * of α
   | * of α #1 ⟨empreinte de list⟩
```

Calcul des définitions sans nom

Pour obtenir une définition sans nom il suffit de

- supprimer tous les noms de constructeurs et de labels,
- remplacer les noms des types de données définis mutuellement récursivement par des numéros $\#1$, $\#2$, etc;
- remplacer les autres noms de types qui apparaissent par leur empreinte $\mathit{Digest}(\Gamma_P, \mathtt{t})$.
- trier les définitions ainsi obtenues pour obtenir un ordre canonique indépendant de celui des noms de la définition d'origine.

Types externes

$$\begin{array}{rcll} \tau_e & ::= & \alpha & \\ & | & \tau_e \mathtt{\,*\,} \ldots \mathtt{\,*\,} \tau_e & \\ & | & (\tau_e, \ldots, \tau_e)\ \#n & \text{Noms des types dans les définitions de types sans noms} \\ & | & (\tau_e, \ldots, \tau_e)\ \mathit{digest} & \text{Empreintes de types} \end{array}$$

Les types externes sont les informations de types qui sont écrites avec les valeurs exportées; c'est pourquoi les types externes permettent de représenter toutes les expressions de type: pour les variables et les n-uplets les types externes sont équivalents aux expressions de type habituelles, sauf qu'il n'y a pas de type externe correspondant aux fonctions puisqu'il n'y a pas de moyens d'écrire les fonctions sur fichiers. Nous disposons aussi d'un moyen de désigner par un entier les types définis récursivement dans les définitions de types. C'est un artifice technique et ces indices ne sont utilisés que pendant les calculs des types externes: ils disparaissent des types attachés aux valeurs. En effet, nous identifions le nom d'un type avec le couple de sa définition sans noms et de l'entier correspondant à l'indice du type dans la définition qui le définit. Dans les types externes, ces couples sont ensuite remplacés par leur empreinte qui est une «empreinte digitale» du couple (indice, définition) qui désigne le type. L'utilisation de ces empreintes MD5 de 128 bits réduit de façon drastique la taille des informations de type: un nom de type est maintenant désigné par 128 bits qu'elle que soit la taille de la définition du type.

Lemme 1 *La taille d'un type externe est proportionnelle à la longueur de l'expression de type qu'il dénote.*

L'algorithme $\mathit{Digest}(\Gamma_P, \mathtt{t})$ calcule l'empreinte d'un type `t` dans l'environnement de typage Γ_P du programme P:

- si `t` est un type primitif on renvoie simplement la chaîne de caractères représentant son nom. (Par exemple, $Digest(\Gamma_P, \texttt{t}) = \texttt{"int"}$.)
- sinon,
  1. on cherche dans l'environnement de typage la définition de type, **`def`**, où apparaît la définition du type `t`
  2. on calcule la définition sans noms de **`def`**,
  3. on renvoie le couple du numéro de la définition du type `t` dans la définition de type **`def`**, et de l'empreinte de la définition sans noms de **`def`** (plus précisément l'empreinte MD5 de la représentation textuelle de la définition sans nom).

Substitutions sur les types externes

Les substitutions sur les types externes sont définies comme d'habitude: τ_e est plus général que τ'_e ($\tau_e > \tau'_e$), si et seulement si, il existe une substitution S telle que $S(\tau_e) = \tau'_e$.

Définition 1 Isomorphisme externe *Deux types de données* `t` *du programme* P *et* `t'` *du programmes* Q *sont* e-isomorphes *si et seulement si* $Digest(\Gamma_P, \texttt{t}) = Digest(\Gamma_Q, \texttt{t'})$

Par exemple, les deux types **`tree`** et **`arbre`** de l'exemple précédent sont isomorphes, car ils ont la même définition sans noms:

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

Notez que cette égalité repose sur le fait que l'empreinte du type `list` est le même pour tous les programmes Caml, $Digest(\Gamma_E, \texttt{tree})$ et $Digest(\Gamma_F, \texttt{arbre})$ ont la même valeur.

Représentation externe d'une expression de type

Finalement, nous définissons une fonction de conversion $Ext(\Gamma_P, \tau)$ qui calcule la représentation externe d'une expression de type τ d'un program P.

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

Par la suite, nous omettrons l'environnement Γ_P de $Ext(\Gamma_P, \tau)$ et $Digest(\Gamma_P, \texttt{t})$, si Γ_P se déduit facilement du contexte.

6. Importabilité

Nous définissons la notion d'*importabilité*, une relation binaire entre expressions de type de programmes différents qui modélise le fait que toute valeur de l'un des types dans l'un des programmes est lisible comme une valeur de l'autre type dans l'autre programme.

Nous prouvons ensuite la sûreté de cette transmission des valeurs d'un type τ à un type τ', lorsque τ est importable comme τ', en introduisant un système de typage des représentations internes des valeurs Caml (les données binaires).

Définition 2 *On dit qu'une expression de type τ d'un programme P est* importable *dans un programme Q avec le type τ' et l'on note $\tau_P \Rightarrow \tau'_Q$, si et seulement si $Ext(\Gamma_P, \tau) > Ext(\Gamma_Q, \tau')$.*

Pour prouver la sûreté du mécanisme d'importation des valeurs, nous étudions la structure interne des représentations des données en Caml, les *données binaires*. Puis, nous créons un système de typage des données binaires, et prouvons que le bon typage d'une donnée binaire est équivalent à la sûreté de lecture de cette donnée binaire.

Représentation interne des valeurs Caml

Les représentations internes des valeurs Caml, les *données binaires*, sont définies comme suit:

$bdata$	$::=$	n	Entier
	$\mid$	C_p	Constante primitive de type p, (`bool`, `float`, `string`, etc)
	$\mid$	$B_n[bdata;\ \ldots;\ bdata]$	Bloc de mémoire avec marqueur n

Les données binaires sont formées de constantes entières, d'autres constantes primitives, et de blocs de mémoire. Un bloc mémoire est une liste de données binaires marqué par un entier: le marqueur du bloc ou *tag*. Les blocs représentent les données structurées, valeurs de types somme ou de types produit. Cette modélisation est une simplification des représentations réelles, parce que nous les avons limitées aux représentations qui intéressent les entrées/sorties de valeurs (par exemple les fonctions sont omises car on ne sait pas les écrire sur disque).

Compilation des données en données binaires

D'après le manuel d'Objective Caml[5], la fonction *bin* de traduction des données ML vers les données binaires est la suivante:

$bin(\Gamma, n) = n$

$bin(\Gamma, C_p) = C_p$

$bin(\Gamma, \texttt{(}e_1, \ldots, e_n\texttt{)}) = B_0[bin(\Gamma, e_1);\ \ldots;\ bin(\Gamma, e_n)]$

$bin(\Gamma, \texttt{\{}l_1 = e_1\texttt{;}\ \ldots\texttt{;}\ l_n = e_n\texttt{\}}) = B_0[bin(\Gamma, e_1);\ \ldots;\ bin(\Gamma, e_n)]$
les labels des enregistrements $l_1 \ldots l_n$ sont écrits dans l'ordre de leur définition dans Γ

$bin(\Gamma, \texttt{Cnst}) = n - 1$
si `Cnst` est le $n$ième constructeur constant du type somme `t` dans Γ

$bin(\Gamma, \texttt{Cnst(}e_1 \ldots, e_m\texttt{)}) = B_{n-1}[bin(\Gamma, e_1);\ \ldots;\ bin(\Gamma, e_m)]$
si `Cnst` est le $n$ième constructeur non-constant du type somme `t` dans Γ

La traduction des constantes primitives est évidente; celle des n-uplets est simple: des blocs avec marqueur 0 contenant les composantes; de même pour les enregistrements, avec toutefois la précaution d'ordonner les champs de façon canonique dans l'ordre de la définition du type. Enfin, la traduction des valeurs des types somme fait la différence entre les constructeurs constants et les autres: si `Cnst` est le $n^{\text{ième}}$ constructeur du type somme alors

- s'il est constant, on le représente par l'entier $n - 1$
- sinon, ce constructeur a des arguments qu'on écrit dans un bloc qui est marqué de l'entier $n - 1$.

Par exemple, le type liste est prédéfini, et sa définition est équivalente à

```
type 'a list =
| ( [] )
| ( :: ) of 'a * 'a list
```

la traduction binaire de la liste `[1; 2]` est donc:

$$bin(\Gamma, \texttt{[1; 2]}) = bin(\Gamma, \texttt{1::2::[]}) = B_0[1;\ B_0[2;\ 0]]$$

Typage des représentations binaires

Dans la figure 1, nous donnons le système de typage pour les données binaires. Il est très similaire au typage habituel, si ce n'est qu'il traite de valeurs distinguées par des marqueurs:

$$\Gamma \rhd n : \texttt{int}$$

$$\Gamma \rhd C_p : p \quad p \text{ est un type de données primitif, } \texttt{bool}, \texttt{float}, \text{etc.}$$

$$\frac{\Gamma \rhd e : \tau \quad \tau > \tau'}{\Gamma \rhd e : \tau'}$$

$$\frac{\Gamma \rhd bdata_1 : \tau_1 \quad \ldots \quad \Gamma \rhd bdata_n : \tau_n}{\Gamma \rhd B_0[bdata_1;\ \ldots;\ bdata_n] : \tau_1 \texttt{ * } \ldots \texttt{ * } \tau_n}$$

$$\frac{\begin{array}{c}\Gamma \rhd bdata_1 : \tau_1 \quad \ldots \quad \Gamma \rhd bdata_n : \tau_n \\ \texttt{type } (\alpha_1, \ldots, \alpha_m) \texttt{ t = \{}l_1 : \tau'_1\texttt{;} \ \ldots\texttt{;}\ l_n : \tau'_n\texttt{\}} \in \Gamma \text{ , tel que } S(\tau'_i) = \tau_i\end{array}}{\Gamma \rhd B_0[bdata_1;\ \ldots;\ bdata_n] : (S(\alpha_1), \ldots, S(\alpha_m)) \texttt{ t}}$$

$$\frac{\begin{array}{l}\text{Il existe une définition de type somme } (\alpha_1, \ldots, \alpha_m) \texttt{ t} \text{ dans } \Gamma \\ \text{avec au moins } (n+1) \text{ constructeurs constants}\end{array}}{\Gamma \rhd n : (\alpha_1, \ldots, \alpha_m) \texttt{ t}}$$

$$\frac{\begin{array}{l}\Gamma \rhd bdata_1 : \tau_1 \quad \ldots \quad \Gamma \rhd bdata_n : \tau_n \\ \text{Il existe une définition de type somme } (\alpha_1, \ldots, \alpha_m) \texttt{ t} \text{ dans } \Gamma \\ \text{et le } (n+1)\text{th constructeur non-constant est } \texttt{Cnst(}\tau'_1, \ldots \tau'_n\texttt{)} \text{ , tel que } S(\tau'_i) = \tau_i\end{array}}{\Gamma \rhd B_n[bdata_1;\ \ldots;\ bdata_n] : (S(\alpha_1), \ldots, S(\alpha_m)) \texttt{ t}}$$

Figure 1: Règles de typage des données binaires

Lemme 2 *Si une valeur e a le type $\Gamma \vdash e : \tau$, alors sa compilation en données binaires est aussi typable avec le type τ, c'est-à-dire $\Gamma \rhd bin(\Gamma, e) : \tau$. Réciproquement, si une expression binaire bdata a le type binaire $\Gamma \rhd bdata : \tau$, alors il existe une valeur ML e, telle que $bin(\Gamma, e) = bdata$ et qui a le type τ, $\Gamma \vdash e : \tau$.*

PROOF: (Plan) Par récurrence sur l'arbre de typage de l'un des système, on peut obtenir un arbre de typage dans l'autre en utilisant les règles de typage correspondantes, tout en réécrivant les expressions pour passer d'une représentation à l'autre.

Ce lemme 2 indique que si l'on peut trouver un typage binaire de $bdata$, $\Gamma \rhd bdata : \tau$ dans un programme et son environnement de typage Γ, alors $bdata$ peut être importé sans danger dans le programme comme une valeur de type τ. Ce lemme prouve donc ce que nous avions supposé admis au sujet des noms dans 1.

Lemme 3 *Si le type* `t` *d'un programme P et le type* `t'` *d'un programme Q sont e-isomorphes, alors pour toute donnée binaire bdata qui admet le typage binaire $\Gamma_P \rhd bdata : \mathtt{t}$, il existe un typage binaire $\Gamma_Q \rhd bdata : \mathtt{t'}$.*

PROOF: Pour prouver ce lemme, nous devons supposer que les empreintes MD5 sont fiables: les empreintes de deux types sont égales si et seulement si ces deux types ont la même définition sans noms. Alors on peut construire l'arbre de dérivation produisant le type `t'` en réécrivant l'arbre de dérivation donné pour `t`.

Ce lemme assure que la communication entre types e-isomorphes est sûre. Nous étendons maintenant cette propriété à la relation d'importabilité:

Lemme 4 *Si $\tau_P \Rightarrow \tau'_Q$ et que la représentation binaire bdata a un typage binaire dans l'environnement de type Γ_P du programme P, $\Gamma_P \rhd bdata : \tau$, alors il existe un typage binaire $\Gamma_Q \rhd bdata : \tau'$.*

PROOF: (Plan) On obtient un typage binaire $\Gamma_Q \rhd bdata : \tau'$ en réécrivant la dérivation de typage de $\Gamma_P \rhd bdata : \tau$

.

Avec les lemmes 2 et 4, nous concluons facilement:

Théorème 1 *Si $\tau_P \Rightarrow \tau'_Q$ alors pour toute valeur ML e dont le type est $\Gamma_P \vdash e : \tau$, il existe une valeur ML e' dont le type est $\Gamma_Q \vdash e' : \tau'$, et telle que $bin(\Gamma_P, e) = bin(\Gamma_Q, e')$.*

Ce théorème établit la sûreté du système d'entrées/sorties de valeurs: si le type τ du programme P est importable dans le programme Q comme le type τ', alors toute valeur ML ayant été typée de type τ dans P, peut être importée comme un valeur ML bien typée de type τ' dans Q.

7. Les primitives d'entrées/sorties de valeurs

Compilation des empreintes

Les empreintes des types de données sont calculées statiquement par le compilateur à partir de leur définition, puis elles sont incorporées dans le code compilé comme des chaînes de caractères. Par exemple l'empreinte du type `tree` est définie comme suit:

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

Cependant, le compilateur ne peut pas toujours accéder à la définition des types, car certains types sont *abstraits*. Considérez par exemple le cas de deux modules `M` et `N`, tels que l'implémentation de `M` n'est pas encore écrite: on ne dispose que de l'interface `m.mli` au moment de la compilation de `n.ml`.

m.mli	n.ml
`type t`	`type t' =`
	`| Boo of M.t (* abstract *)`
	`| Far`

L'empreinte de `M.t` n'est évidemment pas disponible puisque le fichier d'implémentation `m.ml` n'existe pas encore. Pour ces types abstraits, nous considérons deux cas:

Types complètement abstraits, sans définition Caml

Ces types n'ont aucune implémentation en Caml car ils sont définis dans un autre langage; on les utilise pour communiquer avec d'autres langages, C par exemple. Il n'y a évidemment aucun moyen de calculer une empreinte basée sur la structure du type, puisque cette structure n'est pas celle des types Caml. Il faut donc utiliser une méthode plus simple, par exemple l'empreinte du fichier source qui définit les opérations sur le type abstrait. Nous utilisons donc une nouvelle déclaration de type abstrait "external type" qui permet de calculer l'empreinte. Par exemple, si les opérations du type `t` sont définies dans le fichier `impl.c`, on écrit:

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

et l'empreinte de `t` est celle du fichier source `impl.c`.

Types abstraits ayant une définition Caml

Les définitions de ces types existent (ou existeront), mais elles ne sont pas disponibles au moment de la compilation. Pour toutes les définitions qui font intervenir ce genre de types abstrait, le compilateur génère du code qui calculera les empreintes à l'exécution, lorsque toutes les empreintes des types abstraits seront disponibles. Par exemple, l'empreinte du type `N.t'` ci-dessus est définie par une expression qui utilise la valeur de l'empreinte du type `M.t`, qui est nécessairement défini et lié avant `N.t'`.

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

Primitives sûres d'entrées/sorties de valeurs

Armés des types externes, nous définissons les primitives `export_value` et `import_value`.

Pour cela, nous définissons d'abord la fonction `rtype_to_etype`, qui calcule le représentant externe canonique d'une valeur de type; cette fonction se contente de remplacer les chemins d'accès des types par les empreintes correspondantes, nous omettons donc sa définition. La définition des primitives sûres d'entrées/sorties de valeurs est alors une composition des primitives concernant les valeurs dynamiques et des primitives d'entrées/sorties non-sûres originales.

```
rtype_to_etype : rtype → etype
val export_value : ∀δ. out_channel → δ → unit
⟹  let export_value_comp =
      fun ty oc v →
        unsafe_output_value oc (rtype_to_etype ty, v)

val import_value : ∀δ. in_channel → δ
⟹  let import_value_comp =
      fun ty ic →
        let ety, v =  unsafe_input_value ic in
        if is_importable (rtype_to_etype ty) ety then v
        else raise Invalid_import (ty, ety)
```

`export_value`, tout comme `dynamic`, crée le couple d'une valeur et de son type externe et l'écrit sur le canal de sortie. `import_value` lit un tel couple et vérifie son importabilité par rapport au type attendu par le contexte. Le théorème 1 nous assure qu'il n'y aura jamais d'erreurs à l'exécution.

Notons que les types en représentation externe ne sont utilisés que dans le code de ces primitives, puisque les représentations externes sont calculées juste avant l'écriture sur disque à l'export, et disparaissent à l'import après la vérification de cohérence. Les types en représentation externe ne servent donc qu'à assurer la sûreté de fonctionnement et sont donc ignorés de l'utilisateur (et du reste du compilateur).

Entrées/sorties de valeurs et échange de valeurs dynamiques

Une idée très simple pour obtenir ces primitives à peu de frais serait de les définir par échange de valeurs dynamiques:

```
export_value : out_channel → dyn → unit
import_value : in_channel → dyn
```

Cela ne marche malheureusement pas, pour deux raisons:

- Le type encapsulé dans une valeur de type `dyn` est une valeur de type normale et pas un type externe: elle contient évidemment des informations spécifiques au programme qui fabrique le dynamique et ne permet donc pas l'échange de données entre programmes différents.
- Les valeurs dynamiques doivent être converties pour devenir des valeurs Caml normales; le procédé serait donc plus lourd qu'avec les primitives `import_value` et `export_value` qui renvoie directement une valeur utilisable.

8. Entrées/sorties de valeurs polymorphes

Caml est un langage polymorphe, il nous faut donc traiter le cas des valeurs polymorphes. Le problème ici concerne essentiellement le typage, il exactement similaire à celui de la généralisation aux frontières de modules dans le langage ordinaire. On le traite d'une façon analogue mais plus générale, au sens où l'on ne refuse pas les valeurs exportées dont le type n'est pas complètement généralisable: on exporte alors ces valeur avec le type `dyn`.

Le problème

Dans le polymorphisme générique, il n'y a pas de schémas de types **explicites** dans les valeurs de types. De même que dans les types habituels écrits par le compilateur, toute variable de type qui apparaît dans une valeur de type est automatiquement considérée comme universellement quantifiée, c'est-à-dire que toute valeur de type est en fait un schéma de type universellement quantifié sur toutes les variables qui apparaissent dans cette valeur de type. C'est pourquoi, il ne faut inclure de variables de type dans les valeurs de type qu'à la condition expresse que ces variables soient généralisables dans le contexte statique.

Montrons par un exemple que cette précaution est indispensable, en faisant apparaître une erreur due à l'inclusion d'une variable de type non généralisable dans une valeur de type:

```
# let x = ref [];;
val x : _α list ref = {contents=[]}
# let export_x () =
     let oc = open_out_bin "foo" in
     export_value oc !x;
     close_out oc;;
val export_x : unit → unit = <fun>
```

Comme on s'y attend, le type de la référence `x` contient une variable de type α non généralisable. Comme nous supposons qu'aucune précaution n'est prise au moment de la compilation de la fonction `export_x`, l'expression `export_value oc !x` est compilée en:

```
export_value_comp «α list» oc !x
```

Cela signifie donc que la référence `x` est maintenant considérée comme "polymorphe à l'export", c'est-à-dire que l'information de type écrite lors de son écriture sur le disque indique $\forall\alpha.\alpha\,\mathrm{list}$. C'est évidemment faux, puisque le contexte statique indique que la variable α n'est pas généralisable, et que toute affectation ultérieure de `x` est susceptible de modifier α. Il serait alors aisé de mettre en évidence une erreur, par exemple en écrivant:

```
# x := [1];;
- : unit = ()
# export_x ();;
- : unit = ()
```

La dernière phrase exporte la liste `[1]` avec le type «αlist», ce qui est évidemment faux.

Clôture des valeurs de type

Nous empêchons ce genre d'erreur en effectuant, pendant la phase de typage statique, une vérification de la nature des variables de type dynamique insérées dans les valeurs de types: la *clôture des valeurs de type*.

Définition 3 Clôture des valeurs de type *Pour toutes les variables de type dynamiques apparaissant dans une valeur de type, effectuer le traitement suivant:*

- *si la variable de type dynamique est généralisable dans le contexte statique, la généraliser.*
- *sinon, l'unifier avec le type* `dyn`.

La clôture des valeurs de type assure que ces types sont clos, puisque toutes leurs variables de type sont effectivement universellement quantifiées. En effet, pour toute variable libre, nous la généralisons dans le contexte statique et l'incorporons dans la valeur de type dans le premier cas de la règle; sinon la variable disparaît purement et simplement. Quand ce second cas s'applique, la valeur de type n'est donc pas faussement généralisée, mais on obtient un bénéfice supplémentaire: l'expression qui a pour type la variable qui nous occupe devient monomorphe, et de type `dyn`; la valeur de cette expression à l'exécution sera donc un dynamique qui enregistrera correctement le type statique de l'expression. La règle de clôture des valeurs de type permet la fabrication de valeurs dynamiques polymorphes, puisque la généralisation du contexte statique est maintenant fidèlement reproduit dans les valeurs de types. Par exemple:

```
# let d = dynamic [];;
val d : dyn = ([] : α list)
```

Le type de `[]` est généralisable, donc généralisé, et l'on obtient une valeur dynamique polymorphe.

Pour les entrées/sorties de valeurs, le premier cas de la règle de clôture des valeurs de types correspond à l'export de valeurs effectivement polymorphes, tandis que le second cas est indispensable pour ne pas indûment généraliser les variables de type dynamiques dans les valeurs de type; elle supprime donc le problème décrit ci-dessus, **sans pour autant rejeter le programme**:

```
# let export_x () =
    let oc = open_out_bin "foo" in
    export_value oc !x; (* δ est unifié à dyn *)
    close_out oc;;
val export_x : unit → unit = <fun>
# x;;
- : dyn list ref = {contents=[]}
```

En effet, le type d'utilisation de `export_value` dans `export_x` devient `out_channel` $\rightarrow$ `dyn` $\rightarrow$ `unit`, et le type de `x` devient `dyn list ref`. La fonction `export_x` est maintenant sûre et utile, puisque la référence `x` peut être modifiée avec des valeurs de type `dyn`, puis exportée et relue sans erreur.

```
# x := [dyn 1];;
- : unit = ()
# export_x ();; (* Export de [(1:int)] avec le type «dyn list» *)
- : unit = ()
```

Ce traitement par la règle de clôture des valeurs de types n'est évidemment pas spécifique aux entrées/sorties de valeurs, c'est une solution générale au problème des variables libres encapsulées dans les types de valeurs: il s'applique parfaitement à toutes les fonctions génériques polymorphes.

Application aux compositions polymorphes de fonctions génériques polymorphes

La règle de clôture des valeurs de type résoud élégamment un cas épineux du typage des fonctions génériques: celui des variables de type dynamiques "totalement libres", c'est-à-dire des variables de type dynamiques qui n'apparaissent qu'à l'interface de la composition de deux fonctions.

Supposons disposer de deux fonctions génériques polymorphes `print` (de type $\forall\beta.\beta \rightarrow$ `unit`) qui écrit son argument quel qu'il soit, et `eval` (de type $\forall\beta.$`syntax` $\rightarrow \beta$) qui type puis évalue un arbre de syntaxe dont elle retourne la valeur; supposons donnée aussi une fonction interactive d'analyse syntaxique `read` (de type `unit` $\rightarrow$ `syntax`); alors une composition monomorphe de ces fonctions est sûre et utile: par exemple `print (1 + (eval (read ())))` lira une expression quelconque, dont `eval` vérifiera d'abord qu'elle est de type entier, puis dont elle calculera la valeur qu'elle fera finalement imprimer comme un entier par la fonction `print`. Rien que de très normal, et ce processus se déroulerait de la même manière pour tout autre type statiquement attendu en résultat de l'évaluation de l'arbre de syntaxe lu. Mais considérons maintenant une composition polymorphe de ces fonctions, par exemple une tentative de définition d'une boucle interactive:

```
# let toplevel () = print (eval (read ()));;
toplevel : unit -> unit = <fun>
```

La composition est évidemment bien typée et la fonction `toplevel` reçoit le type attendu. Cependant, les fonctions `eval` et `print` échangent une valeur dont le type est statiquement complètement inconnu (et pour cause, puisqu'on ne dispose pas de l'arbre de syntaxe qui permettrait d'effectuer ce typage), c'est une simple variable de type dynamique β. Nous appelons ce genre de composition de fonctions génériques où une variable de type reste complètement polymorphe des *compositions essentiellement polymorphes*. En l'occurrence, cette composition ne crée pas de problèmes d'erreurs à l'exécution, mais donne une sémantique surprenante à la fonction composée. Dans notre cas, la fonction `eval` est sûre et ne produira pas d'erreur puisqu'elle typera l'arbre de syntaxe avant de l'exécuter. Cependant la fonction `print` écrira le résultat de l'appel à `eval` avec le type statique que le compilateur lui aura indiqué, c'est-à-dire une variable de type dynamique libre (ce qui correspond donc à la valeur de type représentant le schéma de type $\forall\beta.\beta$); malheureusement cette variable libre ne contient bien évidemment aucune information sur la valeur à imprimer. Encore une fois, pas d'erreur à l'exécution: `print` écrira systématiquement `<poly>`. Ce n'est évidemment pas le comportement attendu de la composition de ces fonctions. Avec notre traitement des variables dynamiques libres dans les valeurs de type, la variable β sera unifiée à `dyn`, puisqu'elle ne peut pas être généralisée. Ainsi `eval` sera utilisée avec le type `syntax` $\rightarrow$ `dyn`; elle encapsulera donc le type calculé par la phase de typage avec la valeur évaluée dans une valeur de type `dyn`; et la fonction `print`, maintenant utilisée avec le type `dyn` $\rightarrow$ `unit`, pourra connaître le type de la valeur évaluée et l'écrire correctement cette fois au terminal.

Conversion polymorphe

Il nous reste à aborder l'implémentation des primitives `coerce` et `import_value`, qui doivent déstructurer les valeurs éventuellement polymorphes contenues dans les valeurs dynamiques et les valeurs importées. À cause de la restriction du polymorphisme aux expressions non expansives, `coerce` et `import_value` ne sont pas définissables comme de simples fonctions génériques. On introduit donc de nouvelles constructions syntaxiques qui heureusement simplifient et généralisent les conversions.

Voyons une tentative naïve de conversion polymorphe de la liste vide:

```
# let nil = (coerce (dynamic []) : δ list);;
nil : _δ list = []
```

La valeur obtenue est évidemment monomorphe: en effet l'application de `coerce` n'est pas généralisable, comme toutes les applications de fonctions en Caml. Il n'y a aucun moyen de forcer la généralisation en indiquant au contrôleur de types qu'il est sûr d'effectuer des conversions polymorphes; c'est pourquoi on introduit des constructions syntaxiques typées spécialement:

```
coerce e with (x:τ) → e'
import ic with (x:τ) → e'
```

L'évaluation de l'expression `coerce` e `with x:`$\tau \rightarrow e'$ tente de convertir la valeur de e en le type τ; si l'information de type contenue dans e est plus générale que τ, `x` est lié à la partie valeur de e, avec le schéma de type $\forall FV(\tau).\tau$ où $FV(\tau)$ l'ensemble des variables libres de τ; sinon une erreur est signalée. L'expression `import` opère de même pour les valeurs importées depuis un canal de lecture ic, mais la vérification de type utilise des types externes au lieu de valeurs de type. Plus précisément:

9. Implémentation

Ce système d'entrées/sorties de valeurs est implémenté au dessus d'une version expérimentale d'Objective Caml 2.02 qui connaît les fonctions génériques et leur système de typage. Ce compilateur traite en outre les valeurs dynamiques et les nouvelles constructions `coerce` et `import`, généralisées à un nombre arbitraire de clauses.

Les calculs des empreintes de définitions de types à l'exécution sont effectués paresseusement, et il n'y a donc aucun ralentissement notable de la vitesse des programmes. L'augmentation de la taille des exécutables, due aux empreintes des définitions de types, est négligeable: on observe une croissance de 0 à 10 percent, selon le nombre de définitions de type qui apparaissent dans le programme.

$$\frac{\Gamma \vdash e : \texttt{dyn} \quad \Gamma[\texttt{x} : \sigma] \vdash e' : \tau' \text{ où } \sigma = \forall FV(\tau).\tau}{\texttt{coerce } e \texttt{ with (x:}\tau\texttt{)} \rightarrow e' : \tau'}$$

$$\frac{\Gamma \vdash ic : \texttt{in\_channel} \quad \Gamma[\texttt{x} : \sigma] \vdash e' : \tau' \text{ où } \sigma = \forall FV(\tau).\tau}{\texttt{import } ic \texttt{ with (x:}\tau\texttt{)} \rightarrow e' : \tau'}$$

```
coerce e with (x:τ) → e'
⟹ let (ty,v) = e in
     if is_compatible «τ» ty then (fun x → e') v
     else raise (Invalid_coercion («τ», ty))


import ic with (x:τ) → e'
⟹ let (ty,v) = input_value ic in
     if is_importable Ext(«τ») ty then (fun x → e') v
     else raise (Invalid_import («τ», ty))
```

Figure 2: Règles de typage et compilation de `coerce` et `import`

10. Conclusion

Nous avons proposé et implémenté un système d'entrées/sorties de valeurs pour ML qui traite le polymorphisme et dont la correction est prouvée.

Les informations nécessaires à l'exécution ne concernent que les définitions de types et elles sont réduites à de simples empreinte MD5. En outre, les expressions de type véhiculées avec les valeurs échangés par les programmes sont de petite taille, et les types de données y sont identifiés par une abstraction de leur définition qui représente leur sémantique.

Cela donne à notre système une propriété intéressante: les programmes qui se communiquent des valeurs n'ont pas besoin de partager les mêmes définitions de types.

En implémentant les entrées/sorties de valeurs, nous avons obtenu des valeurs dynamiques à peu de frais. Réciproquement, l'introduction des valeurs dynamiques nous a permis d'étendre le polymorphisme générique pour permettre le traitement des valeurs polymorphes dans les entrées/sorties de valeurs. Finalement, cette extension du système de types nous a permis à nouveau de trouver une solution simple et élégante au difficile problème de la composition essentiellement polymorphe des fonctions génériques.

Bibliographie

[1] Catherine Dubois, François Rouaix and Pierre Weis. *Extensional polymorphism* Proceedings of the 22th ACM Conference on Principles of Programming Languages, January 1995

[2] M. Abadi, L. Cardelli, B.Pierce and G.Plotkin. *Dynamic typing in a statically typed language* ACM Transactions on Programming Languages and Systems, 13(2):237-268, Apr 1991.

[3] M. Abadi, L. Cardelli, B. Pierce and D. Rémy. *Dynamic typing in polymorphic languages.* Proceedings of the 1992 workshop on ML and its applications, 1992.

[4] Xavier Leroy and Michel Mauny. *Dynamics in ML.* Journal of Functional Programming, 3(4):431-463, 19944

[5] Xavier Leroy, Didier Rémy, Jérôme Vouillon and Damien Doligez. *The Objective CAML reference manual* Available at `http://pauillac.inria.fr/caml/ocaml/htmlman/index.html`

[6] R. Rivest, MIT Laboratory for Computer Science and RSA Data Security, Inc. *The MD5 Message-Digest Algorithm* Network Working Group Request for Comments: 1321, April 1992

[7] Mark Shields, Tim Sheard, Simon Peyton Jones. Dynamic Typing as Staged Type Inference (25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages)

[8] M.Pil First Class File I/O IFL'96.