Sûreté globale des abstractions et sous-typage dans un langage distribué

Pierre-Malo Deniélou[†]

ENS Cachan – INRIA Rocquencourt

13 septembre 2005

Stage du master MPRI
Sous la direction de James J. Leifer[†] et Jean-Jacques Lévy[†]

[†]{Prenom.Nom}@inria.fr

Plan

Sérialisation et désérialisation

Définition : Conversion de valeurs arbitraires vers et à partir de chaînes de caractères.

Utilisation correcte (*P1* || *P2*)

```
P1 : send (marshall (3 : int))
P2 : print_int (unmarshall (receive ()) : int)
```

Utilisation incorrecte (*P1* || *P2*)

```
P1 : send (marshall (3 : int))
P2 : print_string (unmarshall (receive ()) : string)
```

- Donne un `Segfault` en Ocaml
- Rejeté par la comparaison des types dans certains langages.

Sérialisation et désérialisation

Définition : Conversion de valeurs arbitraires vers et à partir de chaînes de caractères.

Utilisation correcte (*P1* || *P2*)

```
P1 : send (marshall (3 : int))
P2 : print_int (unmarshall (receive ()) : int)
```

Utilisation incorrecte (*P1* || *P2*)

```
P1 : send (marshall (3 : int))
P2 : print_string (unmarshall (receive ()) : string)
```

- Donne un `Segfault` en Ocaml
- Rejeté par la comparaison des types dans certains langages.

Types abstraits

Définition : Type dont la représentation interne n'est pas révélée par la signature.

Compteur abstrait

```
module Compteur =                    sig
struct                                type t
 type t = int                         val initial : t
 let initial = 0                :     val incremente : t -> t
 let incremente x = x + 1             val valeur : t -> int
 let valeur x = x                    end
end
```

- Accès impossible de l'extérieur du module
- Invariants internes garantis

Difficultés : Préservation des abstractions

- On veut la sûreté du langage
- On veut la préservation des abstractions

Exemple avec des types abstraits (*P1* || *P2*)

```
P1 : send (marshall (Compteur.initial : Compteur.t))
P2 : print_int (Compteur.valeur
                  (unmarshall (receive ()) : Compteur.t))
```

La comparaison doit aller plus loin que l'égalité des noms :

- Différents types concrets
- Différents invariants internes
- Dépendances vis-à-vis de bibliothèques distinctes

Difficultés : Préservation des abstractions

- On veut la sûreté du langage
- On veut la préservation des abstractions

Exemple avec des types abstraits (*P1* || *P2*)

```
P1 : send (marshall (Compteur.initial : Compteur.t))
P2 : print_int (Compteur.valeur
                  (unmarshall (receive ()) : Compteur.t))
```

La comparaison doit aller plus loin que l'égalité des noms :

- Différents types concrets
- Différents invariants internes
- Dépendances vis-à-vis de bibliothèques distinctes

Solution [1] du hachage et des crochets colorés 1/3

Hachages des codes des modules

- On utilise le résultat du hachage comme identifiant unique et reproductible de chaque type abstrait.
- On remplace `Compteur.t` par `h.t` si `h` est le hachage du module `Compteur`.

- Comparaison des types abstraits = comparaison des hachages.
- Termes clos.
- Permet de gérer les dépendances entre modules.

[1] James J. Leifer et al., *Global abstraction-safe marshalling with hash types* [ICFP-03]

Solution du hachage et des crochets colorés 2/3

```
module Compteur = ... in
send (marshall ( Compteur.initial : Compteur.t))
```
$\longrightarrow_c^*$ `send (marshall ( 0 : h.t))`

avec `h` le hachage du module `Compteur`.

Non typable. Solution :

```
module Compteur = ... in
send (marshall ( Compteur.initial : Compteur.t))
```
$\longrightarrow_c^*$ `send (marshall (`$[0]^{h.t}_{\{h\}}$ `: h.t))`

avec `h` le hachage du module `Compteur`.

Crochets colorés

Dans $[0]^{h.t}_{\{h\}}$, à l'intérieur des crochets, on sait que `h.t == int`.

Solution du hachage et des crochets colorés 2/3

```
module Compteur = ... in
send (marshall ( Compteur.initial : Compteur.t))
```
$\longrightarrow_c^*$ `send (marshall ( 0 : h.t))`

avec `h` le hachage du module `Compteur`.

Non typable. Solution :

```
module Compteur = ... in
send (marshall ( Compteur.initial : Compteur.t))
```
$\longrightarrow_c^*$ `send (marshall (`$[0]^{h.t}_{\{h\}}$ `: h.t))`

avec `h` le hachage du module `Compteur`.

Crochets colorés

Dans $[0]^{h.t}_{\{h\}}$, à l'intérieur des crochets, on sait que `h.t == int`.

Solution du hachage et des crochets colorés 2/3

```
module Compteur = ... in
send (marshall ( Compteur.initial : Compteur.t))
```
$\longrightarrow_c^*$ `send (marshall ( 0 : h.t))`

avec `h` le hachage du module `Compteur`.

Non typable. Solution :

```
module Compteur = ... in
send (marshall ( Compteur.initial : Compteur.t))
```
$\longrightarrow_c^*$ `send (marshall (`$[0]^{h.t}_{\{h\}}$ `: h.t))`

avec `h` le hachage du module `Compteur`.

Crochets colorés

Dans $[0]^{h.t}_{\{h\}}$, à l'intérieur des crochets, on sait que `h.t == int`.

Solution du hachage et des crochets colorés 3/3

- Annotés par une couleur hm' (i.e. modules dont les types abstraits sont accessibles)

$$\frac{E \vdash_{hm'} e : T}{E \vdash_{hm} [e]^{T}_{hm'} : T}$$

- La révélation des types abstraits s'exprime par :

$$\frac{E \vdash_{hm} \mathrm{ok}}{E \vdash_{hm} h.\mathrm{TYPE} == T} \quad \text{où } h \in hm \text{ et } \mathrm{impl}(h) = T$$

Cadre de travail

Langage considéré

λ-calcul en appel par valeur avec produits et enregistrements. Primitives de sérialisation et désérialisation. Langage de modules simples avec signatures, mais sans foncteurs. Sous-typage implicite et types abstraits et partiellement abstraits.

Objectifs

- Sémantique opérationnelle déterministe à petits pas
- Théorèmes de préservation du typage et de progrès.

Enregistrements et sous-typage

Enregistrements

Expression : `{nom="Dupont"; prenom="Jean"; age=33}`

Type : `{nom:string; prenom:string; age:int}`

Non modifiables, non extensibles.

Le sous-typage et son utilisation

- Une relation d'ordre $<:$ entre les types
- Lorsqu'un type est attendu, on peut mettre un sous-type à la place

$$\frac{}{\{l_1:T_1, ..., l_j:T_j, ..., l_k:T_k\} <: \{l_1:T_1, ..., l_j:T_j\}} \text{ (Largeur)}$$

$$\frac{T_i <: T'_i \quad 1 \leqslant i \leqslant j}{\{l_1:T_1, ..., l_j:T_j\} <: \{l_1:T'_1, ..., l_j:T'_j\}} \text{ (Profondeur)}$$

Enregistrements et sous-typage

Enregistrements

Expression : `{nom="Dupont"; prenom="Jean"; age=33}`

Type : `{nom:string; prenom:string; age:int}`

Non modifiables, non extensibles.

Le sous-typage et son utilisation

- Une relation d'ordre $<:$ entre les types
- Lorsqu'un type est attendu, on peut mettre un sous-type à la place

$$\frac{}{\{l_1 : T_1, ..., l_j : T_j, ..., l_k : T_k\} <: \{l_1 : T_1, ..., l_j : T_j\}} \text{ (Largeur)}$$

$$\frac{T_i <: T'_i \quad 1 \leqslant i \leqslant j}{\{l_1 : T_1, ..., l_j : T_j\} <: \{l_1 : T'_1, ..., l_j : T'_j\}} \text{ (Profondeur)}$$

Abstractions partielles

Définition d'un type partiellement abstrait

Type dont seul un super-type est déclaré dans la signature.

Déclaration d'abstraction partielle :

```
module Compte =
  struct
    type t = {france : int ; suisse : int}
    let v = {france = 1 ; suisse = 10000000}
  end :
  sig
    type t <: { france : int }
    val v : t
  end
```

Sous-typage et types abstraits

La comparaison des types après désérialisation utilise le sous-typage.

Question :

Peut-on déduire une relation de sous-typage entre les types abstraits à partir des résultats des hachages de leurs modules d'origine ?

$$\frac{E \vdash_{hm} h <: h'}{E \vdash_{hm} h.\mathrm{TYPE} <: h'.\mathrm{TYPE}}$$

Contraintes :

- Comparaison effectuée entre les hachages à l'exécution
- Sûreté nécessite le sous-typage entre les types concrets
- Préservation des abstractions

Sous-typage et types abstraits

La comparaison des types après désérialisation utilise le sous-typage.

Question :

Peut-on déduire une relation de sous-typage entre les types abstraits à partir des résultats des hachages de leurs modules d'origine ?

$$\frac{E \vdash_{hm} h <: h'}{E \vdash_{hm} h.\mathrm{TYPE} <: h'.\mathrm{TYPE}}$$

Contraintes :

- Comparaison effectuée entre les hachages à l'exécution
- Sûreté nécessite le sous-typage entre les types concrets
- Préservation des abstractions

Sous-typage et types abstraits : Essais

Peut-on définir $h <: h'$ à partir de la comparaison des signatures ?

Signatures contravariantes : Echec

Abstraction de l'emetteur rompue par un accès illicite

Signatures covariantes : Echec

Invariants du receveur non-satisfaits par la valeur reçue

Structure M de compteur

```
struct
 type t = int
 let v  = {initial =  2 ;
           double = function (x: int) -> x * 2;
           incr = function (x: int) -> x + 1;
           get = function (x: int) -> x }
end
```

Sous-typage et types abstraits : Essais

Peut-on définir $h <: h'$ à partir de la comparaison des signatures ?

Signatures contravariantes : Echec

Abstraction de l'emetteur rompue par un accès illicite

Signatures covariantes : Echec

Invariants du receveur non-satisfaits par la valeur reçue

Structure M de compteur

```
struct
 type t = int
 let v  = {initial =  2 ;
           double = function (x: int) -> x * 2;
           incr = function (x: int) -> x + 1;
           get = function (x: int) -> x }
end
```

Sous-typage et types abstraits : Exemple covariant

Programme1 :

```
module CompteurA = M : sig type t
                           val v : { initial : t ;
                                     incr : t -> t ;
                                     double : t -> t ;
                                     get : t -> int }
                       end in
send(marshall(CompteurA.v.incr(CompteurA.v.initial):CompteurA.t))
```

Programme2 :

```
module CompteurB = M : sig type t
                           val v : { initial : t ;
                                     double : t -> t ;
                                     get : t -> int }
                       end in
print_int(CompteurB.v.get(unmarshall(receive ()):CompteurB.t))
```

Programme1 || *Programme2*

Echec (Invariants rompus)

Sous-typage et types abstraits : Exemple covariant

Programme1 :

```
module CompteurA = M : sig type t
                           val v : { initial : t ;
                                     incr : t -> t ;
                                     double : t -> t ;
                                     get : t -> int }
                       end in
send(marshall(CompteurA.v.incr(CompteurA.v.initial):CompteurA.t))
```

Programme2 :

```
module CompteurB = M : sig type t
                           val v : { initial : t ;
                                     double : t -> t ;
                                     get : t -> int }
                       end in
print_int(CompteurB.v.get(unmarshall(receive ()):CompteurB.t))
```

Programme1 || *Programme2* **Echec (Invariants rompus)**

Sous-typage et types abstraits : Spécification

Réponse :

Pas de sous-typage entre types abstraits dans le cas général.

Mais on peut permettre à l'utilisateur de la spécifier lorsqu'il le souhaite et qu'il garantit la préservation des invariants.

Exemple d'utilisation

```
module Compteur1 = M : sig type t
                           val v : {initial : t ;
                                    incremente : t -> t;
                                    valeur : t -> int }
                       end in
module Compteur2 restricts Compteur1 =
                     M :  sig type t
                              val v : {...
                                       double : t -> t;
                                       ... }
                          end
```

Crochets et abstractions partielles (1/3)

Les crochets protègent les abstractions mais, dans la sémantique, ils interagissent avec les constructeurs et destructeurs.

- Quelle influence pour le sous-typage et les abstractions partielles ?

Destructeur et crochets colorés

```
module Compte =
  struct  type t = { france : int ; suisse : int }
          let v = { france = 1 ; suisse = 10000000 }
  end :
  sig  type t <: { france : int }
       val v : t
  end in
print_int ((Compte.v).france )
```

On veut pouvoir ouvrir les crochets en cas d'informations partielles.

Crochets et abstractions partielles (2/3)

Essai de réduction (n-uplets) :

$$\mathbf{proj}_i\ [e]^{h_0.\mathrm{TYPE}}_{hm_1} \longrightarrow_{hm} [\mathbf{proj}_i\ e]^{T_i}_{hm_1} \qquad \text{où } \vdash_{hm} h_0.\mathrm{TYPE} <: (T_1 * \ldots * T_k)$$

Pas de déterminisme ni de progrès

Quel type T_i écrire dans le membre de droite ?

Difficulté provient des diverses origines du sous-typage

On introduit du sous-typage explicite :

- notation : ${}_{(T_1 <: T_2)}e$

Crochets et abstractions partielles (2/3)

Essai de réduction (n-uplets) :

$$\mathbf{proj}_i\ [e]^{h_0.\mathrm{TYPE}}_{hm_1} \longrightarrow_{hm} [\mathbf{proj}_i\ e]^{T_i}_{hm_1} \qquad \text{où } \vdash_{hm} h_0.\mathrm{TYPE} <: (T_1 * \ldots * T_k)$$

Pas de déterminisme ni de progrès

Quel type T_i écrire dans le membre de droite ?

Difficulté provient des diverses origines du sous-typage

On introduit du sous-typage explicite :

- notation : ${}_{(T_1 <: T_2)}e$

Crochets et abstractions partielles (2/3)

Essai de réduction (n-uplets) :

$$\mathbf{proj}_i\ [e]^{h_0.\mathrm{TYPE}}_{hm_1} \longrightarrow_{hm} [\mathbf{proj}_i\ e]^{T_i}_{hm_1} \qquad \text{où } \vdash_{hm} h_0.\mathrm{TYPE} <: (T_1 * \ldots * T_k)$$

Pas de déterminisme ni de progrès

Quel type T_i écrire dans le membre de droite ?

Difficulté provient des diverses origines du sous-typage

On introduit du sous-typage explicite :

- notation : ${}_{(T_1 <: T_2)}e$

Crochets et abstractions partielles (3/3)

Interaction crochets – sous-typage explicite

$$(T <: T')([e]^{h_0.\mathrm{TYPE}}_{hm_1}) \longrightarrow_{hm} [(T <: T')e]^{T'}_{hm_1}$$

Pas de préservation du typage

$T <: T'$ est vrai dans la couleur hm.
$T <: T'$ n'est pas toujours dérivable dans la couleur hm_1.

Additivité des crochets colorés

$$\frac{E \vdash_{hm \cup hm'} e : T}{E \vdash_{hm} [e]^{T}_{hm'} : T}$$

Crochets et abstractions partielles (3/3)

Interaction crochets – sous-typage explicite

$$({T <: T'})([e]^{h_0.\mathrm{TYPE}}_{hm_1}) \longrightarrow_{hm} [({T <: T'})e]^{T'}_{hm_1}$$

Pas de préservation du typage

$T <: T'$ est vrai dans la couleur hm.

$T <: T'$ n'est pas toujours dérivable dans la couleur hm_1.

Additivité des crochets colorés

$$\frac{E \vdash_{hm \cup hm'} e : T}{E \vdash_{hm} [e]^{T}_{hm'} : T}$$

Crochets et abstractions partielles (3/3)

Interaction crochets – sous-typage explicite

$$({}_{T <: T'})([e]^{h_0.\mathrm{TYPE}}_{hm_1}) \longrightarrow_{hm} [({}_{T <: T'})e]^{T'}_{hm_1}$$

Pas de préservation du typage

$T <: T'$ est vrai dans la couleur hm.

$T <: T'$ n'est pas toujours dérivable dans la couleur hm_1.

Additivité des crochets colorés

$$\frac{E \vdash_{hm \cup hm'} e : T}{E \vdash_{hm} [e]^{T}_{hm'} : T}$$

Résumé (1/3) : langage final

Syntaxe du calcul

$$
\begin{array}{lll}
e ::= x \mid \lambda x.e \mid \ldots & & \lambda\text{-calcul avec produits} \\
\quad \mid U.\mathrm{term} & & \text{champ d'un module} \\
\quad \mid \{l_1 = e_1, \ldots, l_j = e_j\} \mid e.l_i & & \text{enregistrements} \\
\quad \mid !\, e \mid ? & & \text{envoi et réception} \\
\quad \mid \mathbf{mar}\,(e{:}T) \mid \mathbf{marshalled}_H(e{:}T) & & \text{sérialisation} \\
\quad \mid \mathbf{unmar}\, e{:}T \mid \mathbf{Unmarfailure}^T & & \text{désérialisation et erreur} \\
\quad \mid [e]^T_{hm} & & \text{crochets colorés} \\
\quad \mid {}_{(T_1 <: T_2)}e & & \text{sous-typage explicite}
\end{array}
$$

$$
\begin{array}{lll}
m ::= \mathbf{module}\, N_U\, \mathbf{extends}\, \kappa\, \mathbf{restricts}\, \kappa' = M{:}S\, \mathbf{in}\, m & & \text{déclaration de module} \\
\quad \mid e & & \text{expression}
\end{array}
$$

Résumé (2/3) : sémantiques finales

Système de typage

Correction, sous-typage, équivalences, typage (86 règles)

Sémantique

- $H, m \longrightarrow_c H', m'$: réduction des machines (2 règles)
- $H, e \longrightarrow_{hm} H', e$: réduction des expressions (21 règles)
- $n \longrightarrow n'$: réduction des réseaux (7 règles)

Résumé (3/3) : théorèmes (75% – 40 p.)

Préservation du typage pour la réduction des expressions

Si $\vdash^{H}_{hm} e : T$ et $H, e \longrightarrow_{hm} H', e'$ alors $\vdash^{H'}_{hm} e' : T$.

Progrès de la réduction des expressions

Si $\vdash^{H}_{hm} e : T$ alors de deux choses l'une :

- e est légalement bloquée dans la couleur hm.
- e peut se réduire, i.e. il existe e' et H' tels que $H, e \longrightarrow_{hm} H', e'$.

Déterminisme de la reduction des expressions

La réduction des expressions et des machines est déterministe, i.e. si $H, e \longrightarrow_{hm} H', e'$ et $H, e \longrightarrow_{hm} H'', e''$ alors $e' = e''$ et $H' = H''$ et les deux réductions utilisent la même règle sur le même redex.

Contributions

- Enrichissement du langage avec enregistrement, sous-typage et abstractions partielles
- Choix de conception : déclaration explicite de compatibilité entre modules
- Conséquences sur la sémantique opérationnelle : sous-typage explicite et additivité des crochets colorés
- Système de typage adapté
- Preuve partielle des théorèmes de préservation du typage, de progrés et de déterminisme des réductions

Travaux en cours et perspectives

En cours ou à court terme

- Sous-typage en profondeur
- Modules plus généraux
- Preuves (plus effacement des annotations)
- Inférence
- Implémentation

Perspectives

- Polymorphisme
- Types plus riches (objets, types XML, ...)
- Intégration dans Acute ou autre langage

Remerciements

- James J. Leifer
- Jean-Jacques Lévy
- Francesco Zappa Nardelli
- Gilles Peskine
- INRIA/bat. 8