Typer la désérialisation sans sérialiser les types

Grégoire Henry, Michel Mauny et Emmanuel Chailloux

PPS (Univ. P7, P6), INRIA-Rocq/ENSTA

JFLA 2006

La sérialisation, késako ?

Programme source :

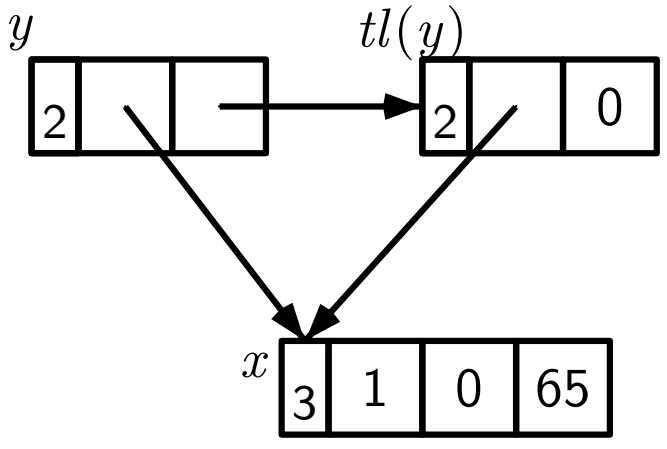
```
let x = (1, false, 'A')
let y =  [x;x]
```

La sérialisation, késako ?

Programme source :

```
let x = (1, false, 'A')
let y =  [x;x]
```

Représentation mémoire :

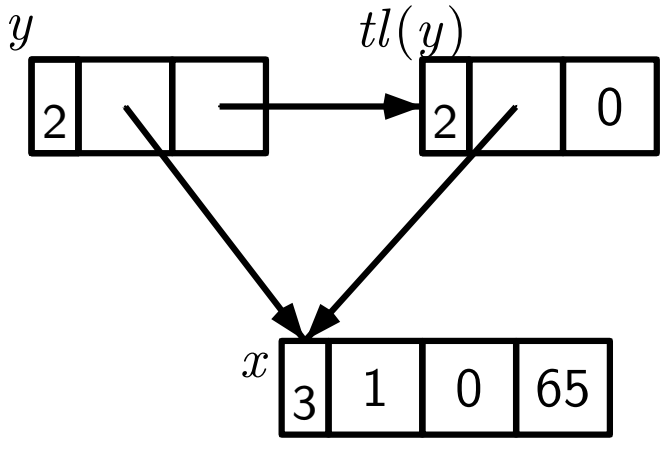


La sérialisation, késako ?

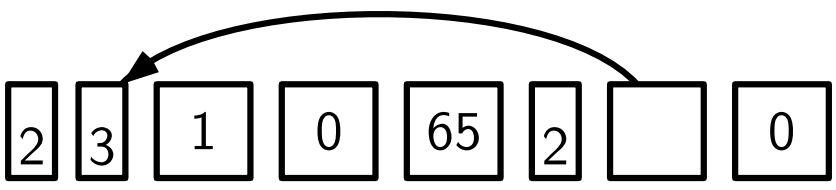
Programme source :

```
let x = (1, false, 'A')
let y = [x;x]
Marshal.to_string y
```

Représentation mémoire :



Représentation sérialisée :



La sérialisation avec des mots

La sérialisation

- c'est la traduction d'une valeur en mémoire vers une suite d'octets ;
- préserve le partage et les cycles ;
- s'effectue par un parcours en profondeur d'abord du graphe mémoire.

Utilisation

- sauvegarde de données sur disque pour reprise ultérieure ;
- transmission de données sur un réseau (applications distribuées) ;
- format commun d'échange de données entre logiciels hétérogènes.

La sérialisation des fonctions est possible en OCaml, mais elles ne peuvent être relues que par une instance du même exécutable.

Désérialiser peut nuire gravement à la santé

La reconstruction d'un graphe mémoire lors de la désérialisation est algorithmiquement facile et efficace.

Mais l'information de types conservée à l'exécution en OCaml est plus pauvre que le système de types du langage :

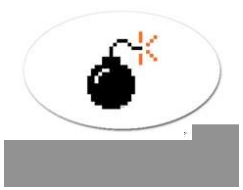
À l'exécution :

- Entier
- Bloc avec taille et *tag*
- Bloc prédéfini

À la compilation :

- Entier, booléen, [ ], ...
- $[23]$, $(23,\text{false})$, $\{x = 23;\ y = 0\}$, ...
- Flottant, chaîne de car., fermeture, ...

Et : `Marshal.from_string` : $\forall\alpha$. `string` $\rightarrow \alpha$



```
let str = Marshal.to_string 0
let array = Marshal.from_string str
array.(1)
```

Objectif

Notre objectif

- étudier la possibilité de vérifier dynamiquement le type des valeurs ;
- fournir une alternative sûre au mécanisme actuel,
- en préservant sa souplesse,
- son format de données,
- et sa complexité linéaire.

Sommaire

Objectif, et sommaire

Notre objectif

- étudier la possibilité de vérifier dynamiquement le type des valeurs ;
- fournir une alternative sûre au mécanisme actuel,
- en préservant sa souplesse,
- son format de données,
- et sa complexité linéaire.

Sommaire

Point de départ : déclaration de type . . .

```
type association = (int * string) list

type 'a tree =
| Leaf
| Unode of 'a * 'a tree
| Bnode of 'a * 'a tree * 'a tree

type 'a stack = { mutable content: 'a list }
```

- On veut pouvoir manipuler les expressions de type ainsi que leurs topographies mémoire (*memory layout* de leurs valeurs).
- On compile une déclaration de type comme une valeur : ajout d'une entrée dans la structure d'un module.

. . . point d'arrivée : typons la représentation des types

On introduit :

- une nouvelle construction syntaxique « τ » pour représenter le type τ à l'exécution ;
- un type paramétré prédéfini (τ `tyRepr`).

Tels que :

- (« τ » : τ `tyRepr`).

Et ainsi :

$$\texttt{Marshal.from\_string} : \forall \alpha \,.\, (\alpha\ \texttt{tyRepr}) \to \texttt{string} \to \alpha$$

Algorithme de vérification – Exemple

Programme source :

type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ \mathit{list} \times t'\ \mathit{list}$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ \mathit{int}\ t \times \mathit{float}\ t\ »\ str$

Algorithme de vérification – Exemple

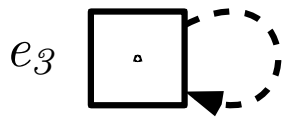
Programme source :

type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

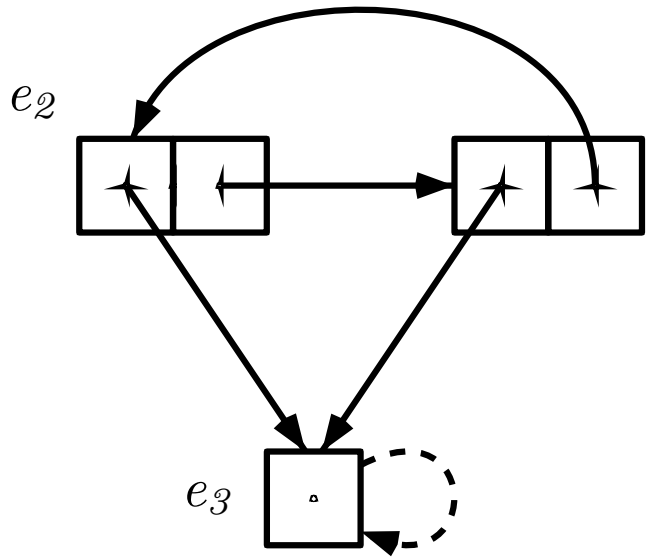
type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$



Algorithme de vérification – Exemple



Programme source :

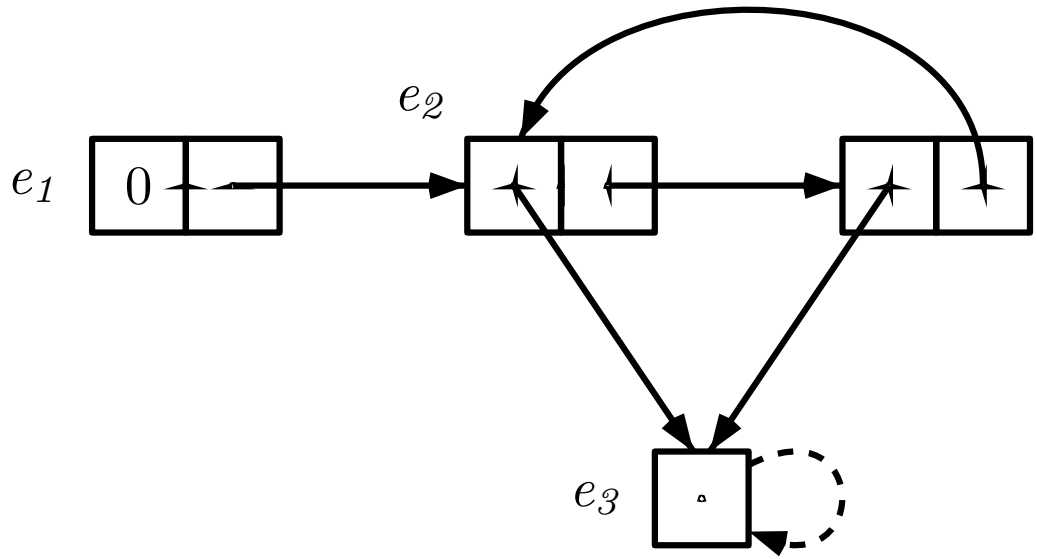
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string \ll int\ t \times float\ t \gg str$

Algorithme de vérification – Exemple



Programme source :

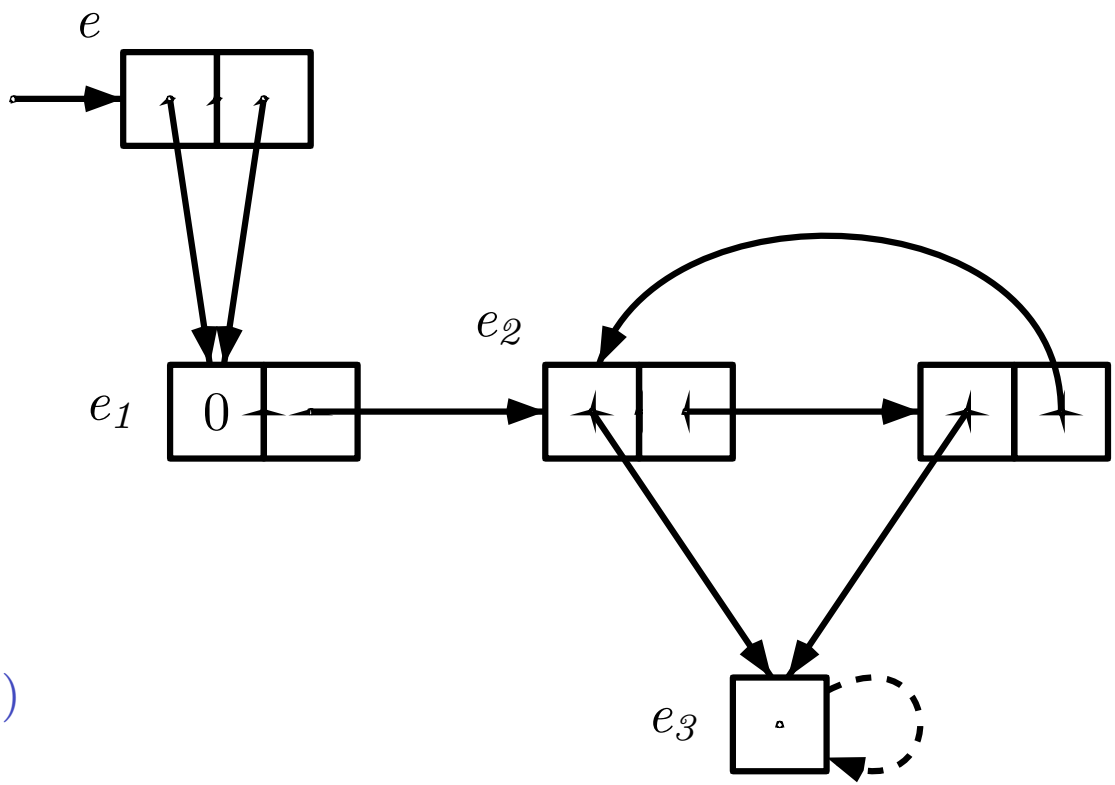
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([\,], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$

Algorithme de vérification – Exemple



Programme source :

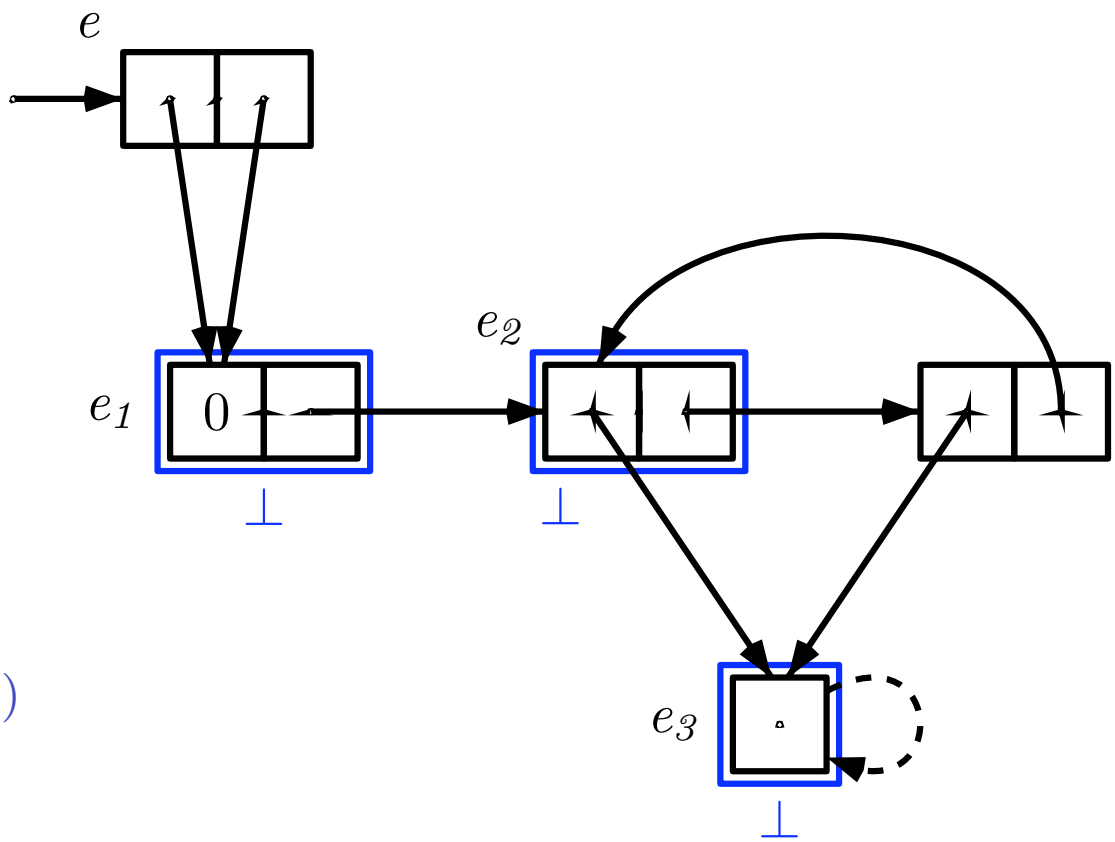
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$

Algorithme de vérification – Exemple



Programme source :

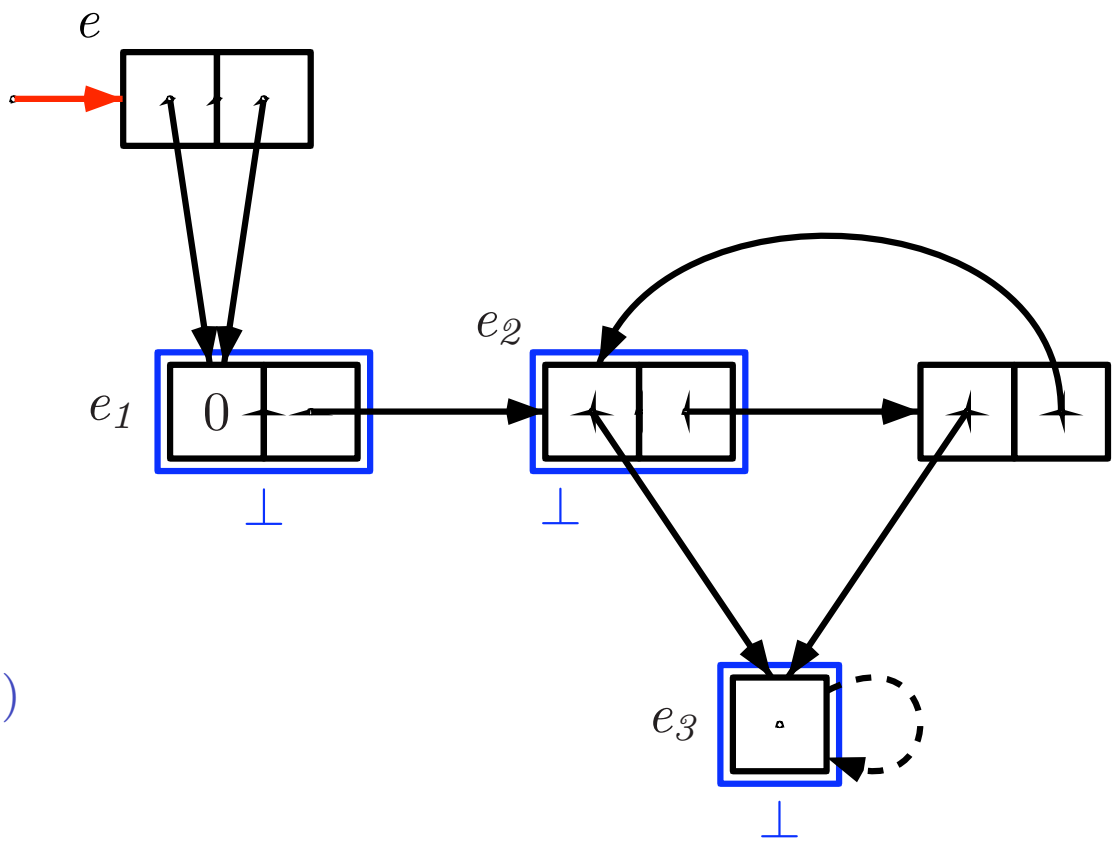
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$

Algorithme de vérification – Exemple



Programme source :

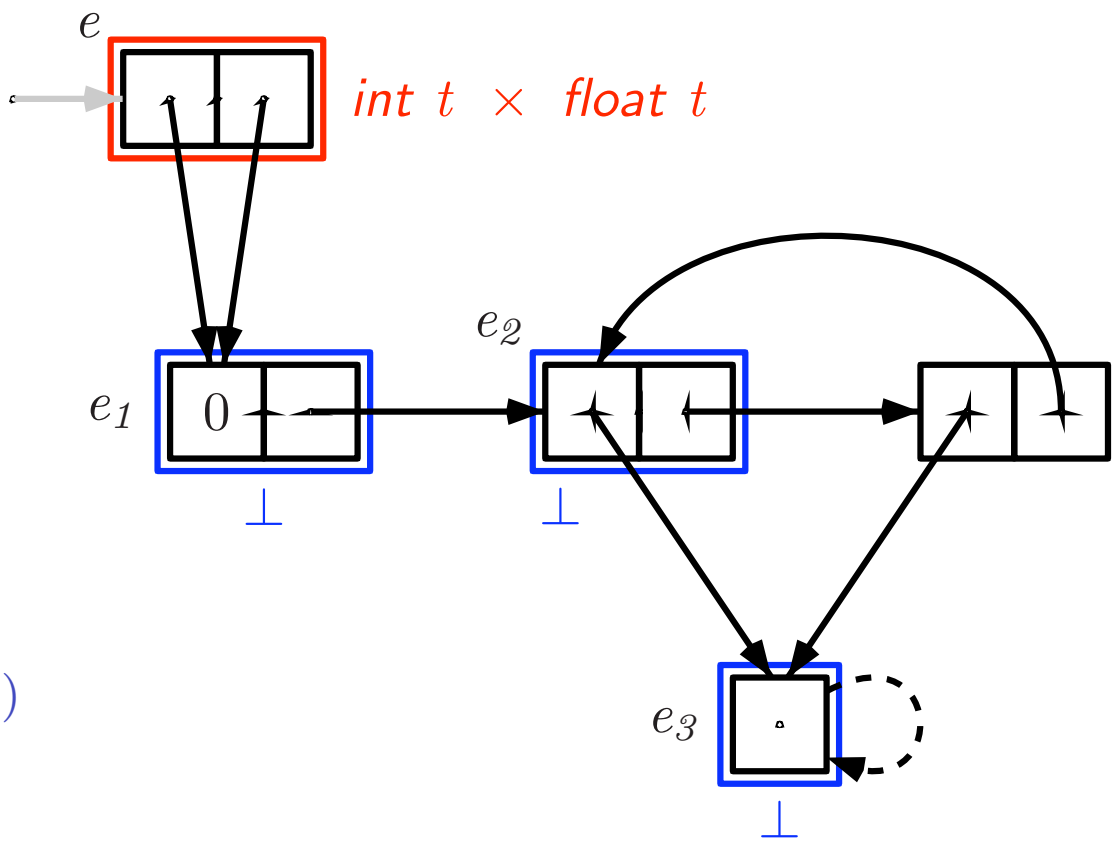
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$

Algorithme de vérification – Exemple



Programme source :

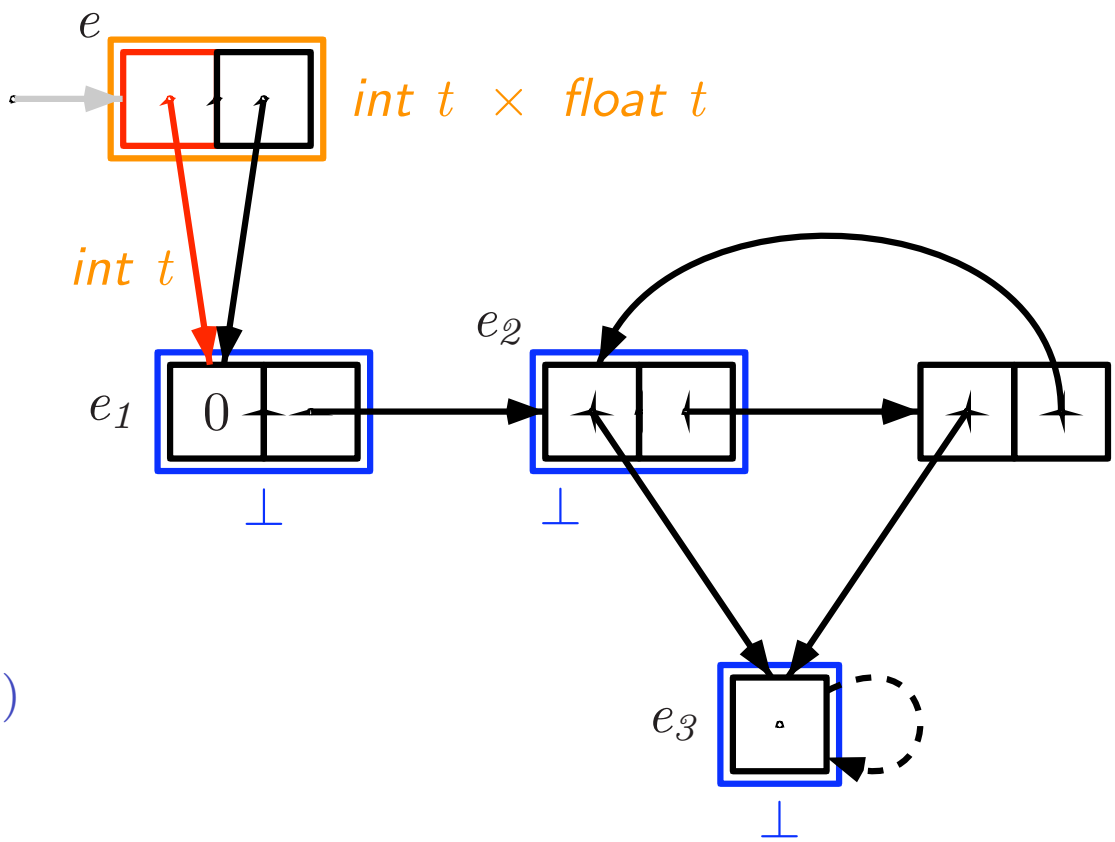
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string$ « *int t × float t* » str

Algorithme de vérification – Exemple



Programme source :

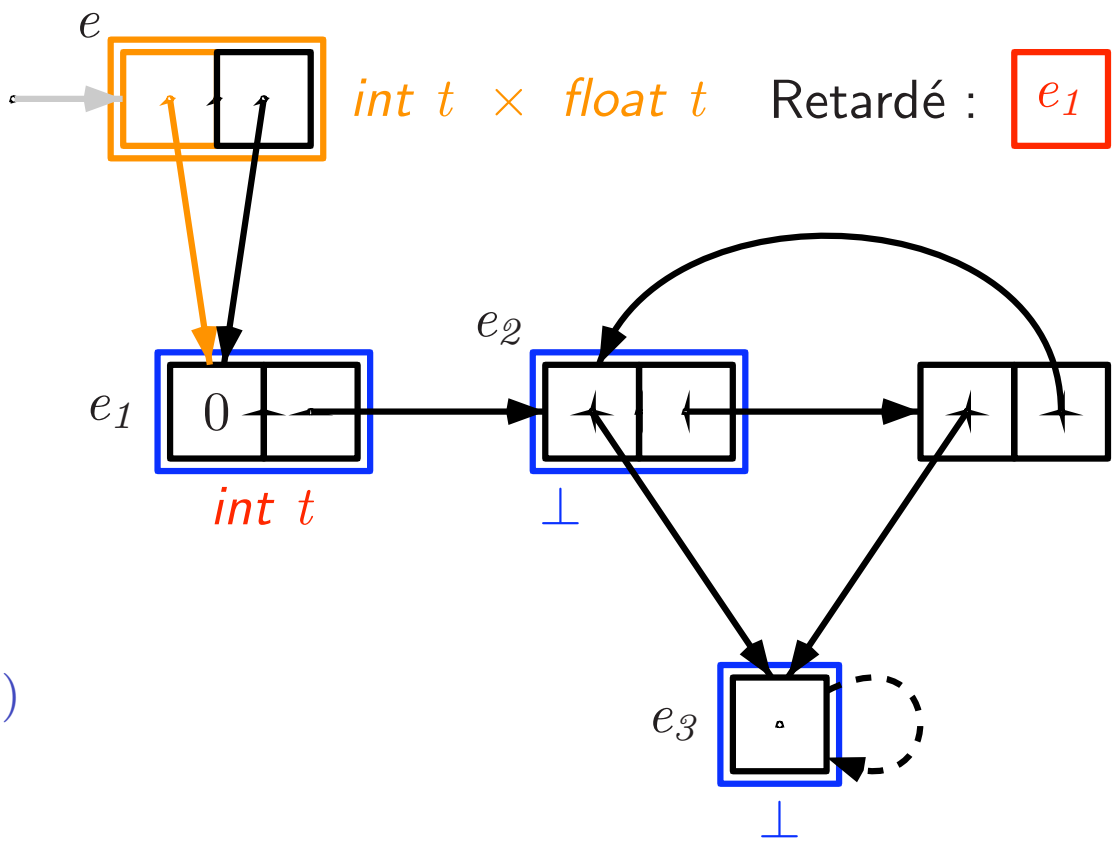
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string$ « *int* $t \times$ *float* t » str

Algorithme de vérification – Exemple



Programme source :

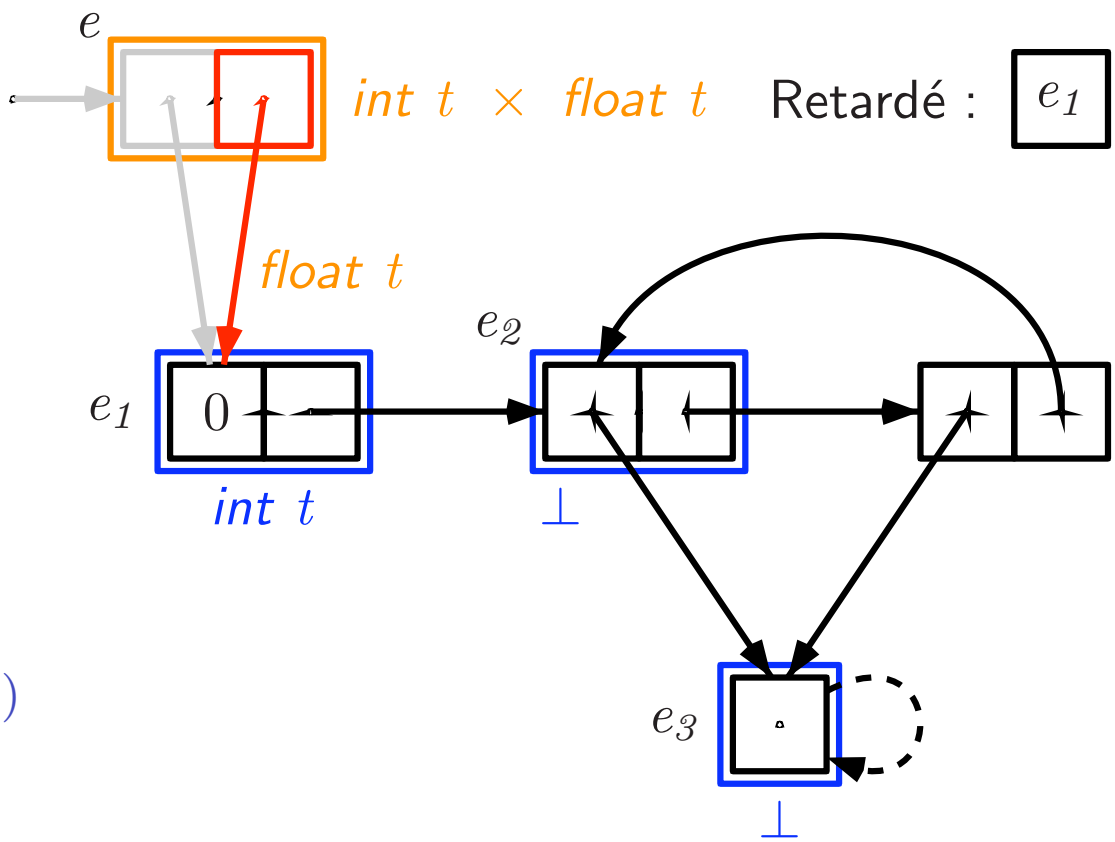
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$

Algorithme de vérification – Exemple



Programme source :

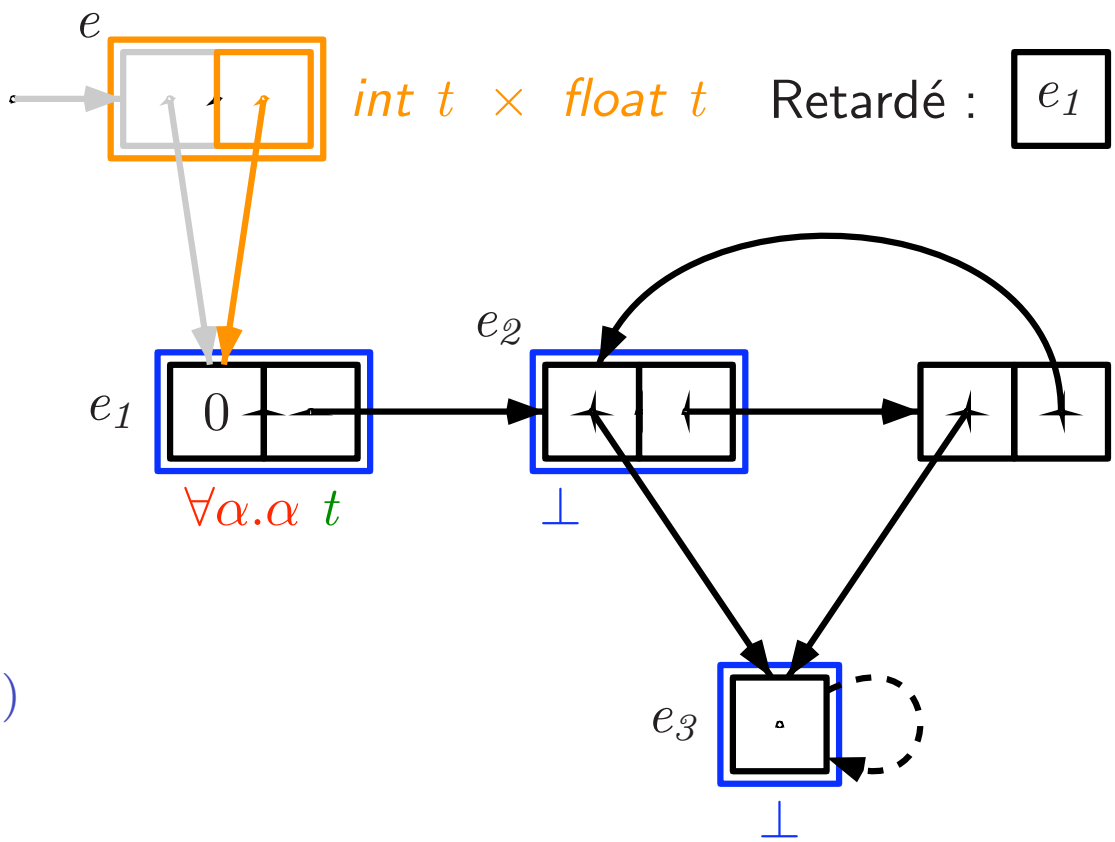
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$

Algorithme de vérification – Exemple



Programme source :

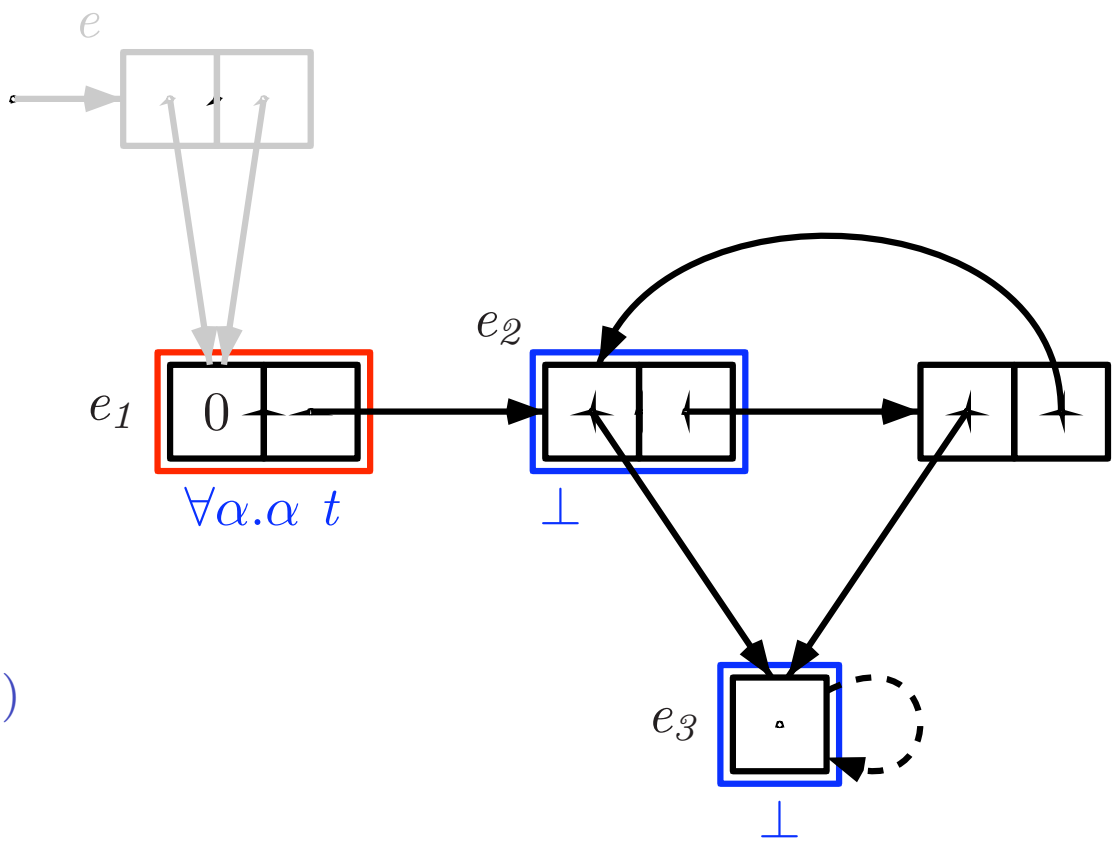
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string$ « *int* $t \times$ *float* t » str

Algorithme de vérification – Exemple



Programme source :

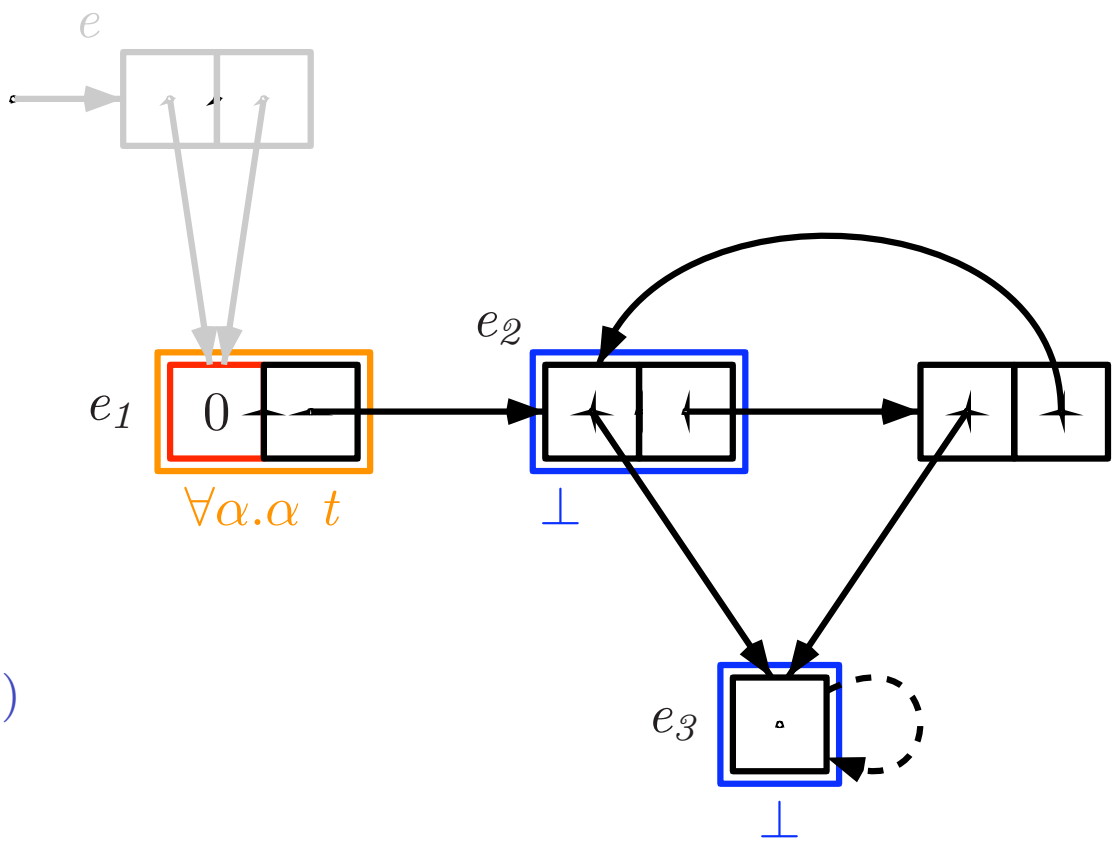
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$

Algorithme de vérification – Exemple



Programme source :

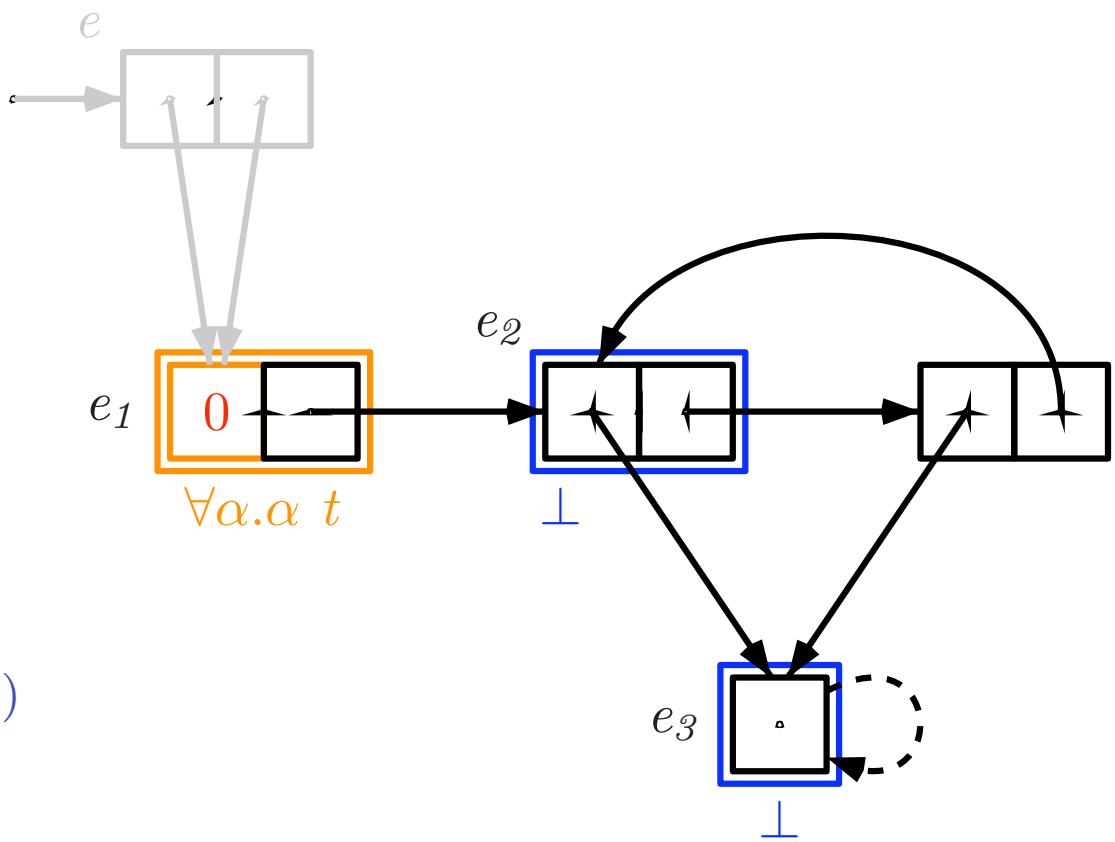
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$

Algorithme de vérification – Exemple



Programme source :

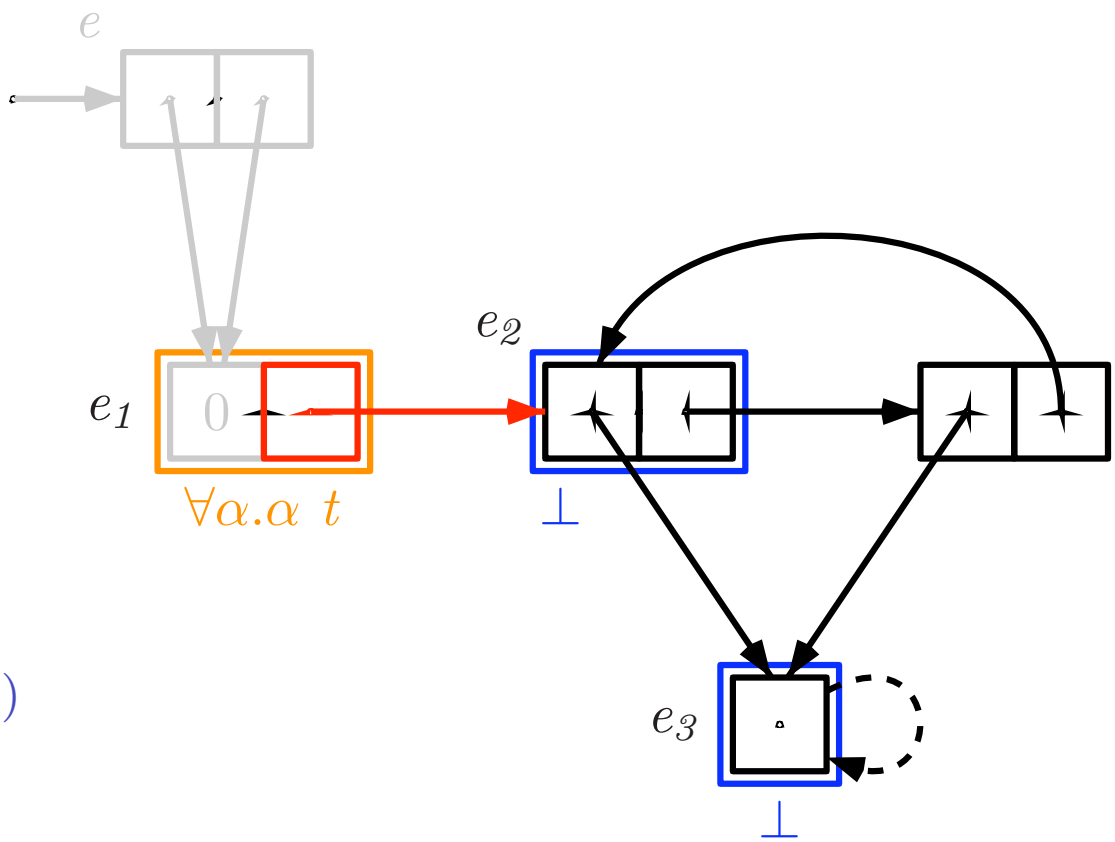
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$

Algorithme de vérification – Exemple



Programme source :

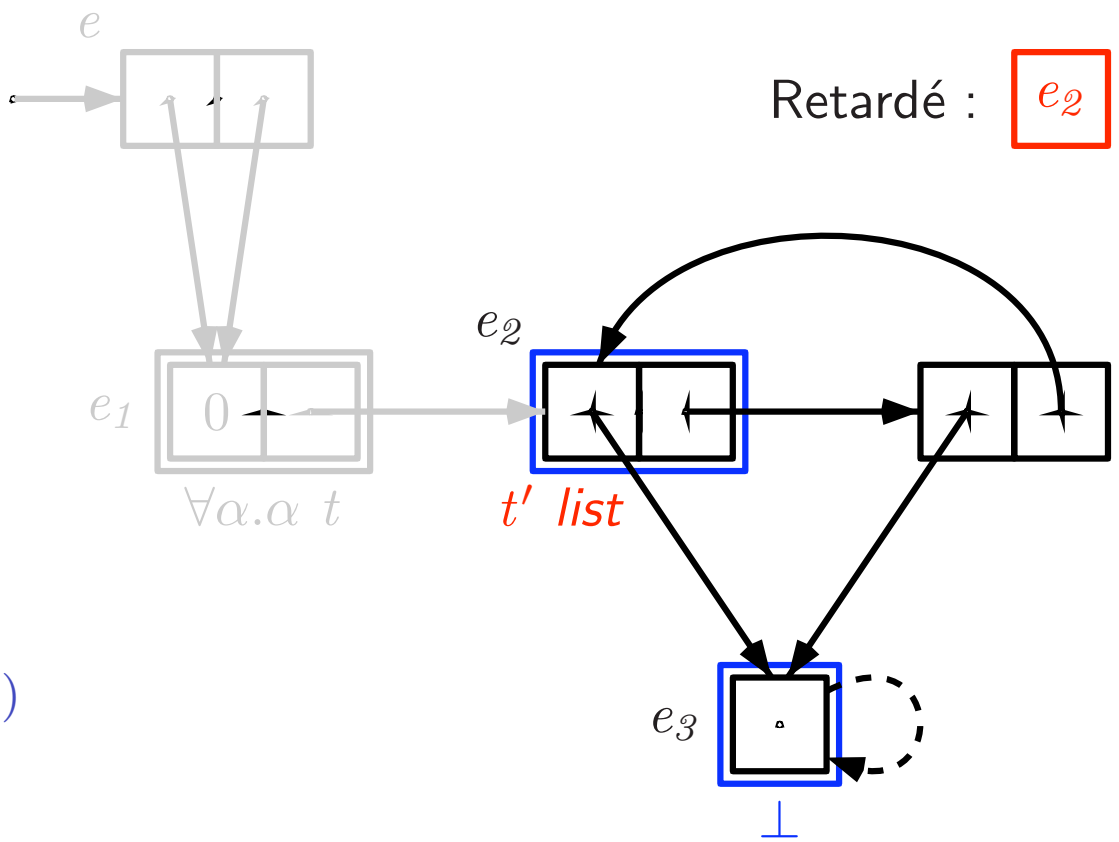
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$

Algorithme de vérification – Exemple



Programme source :

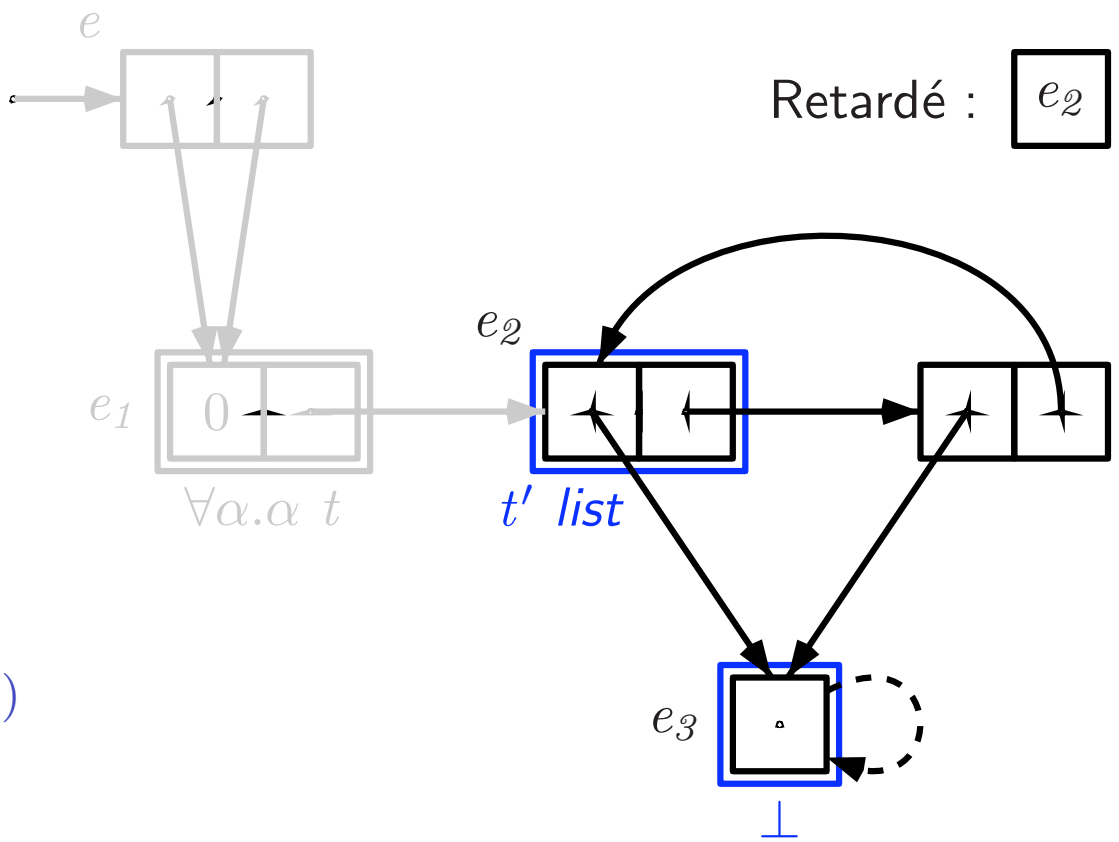
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha$ *list* $\times\ t'$ *list*

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string$ « *int* $t \times$ *float* t » str

Algorithme de vérification – Exemple



Programme source :

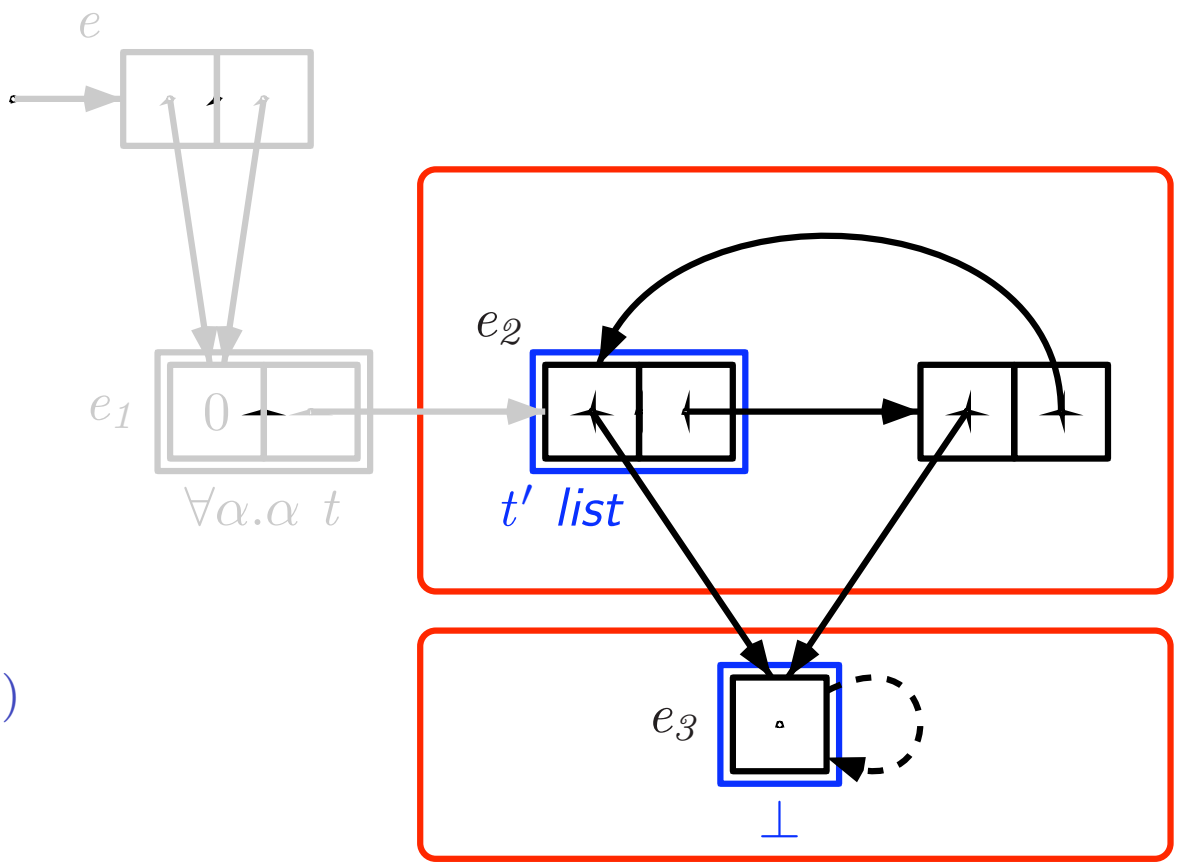
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha$ *list* $\times\ t'$ *list*

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string$ « *int* $t \times$ *float* t » str

Algorithme de vérification – Exemple



Programme source :

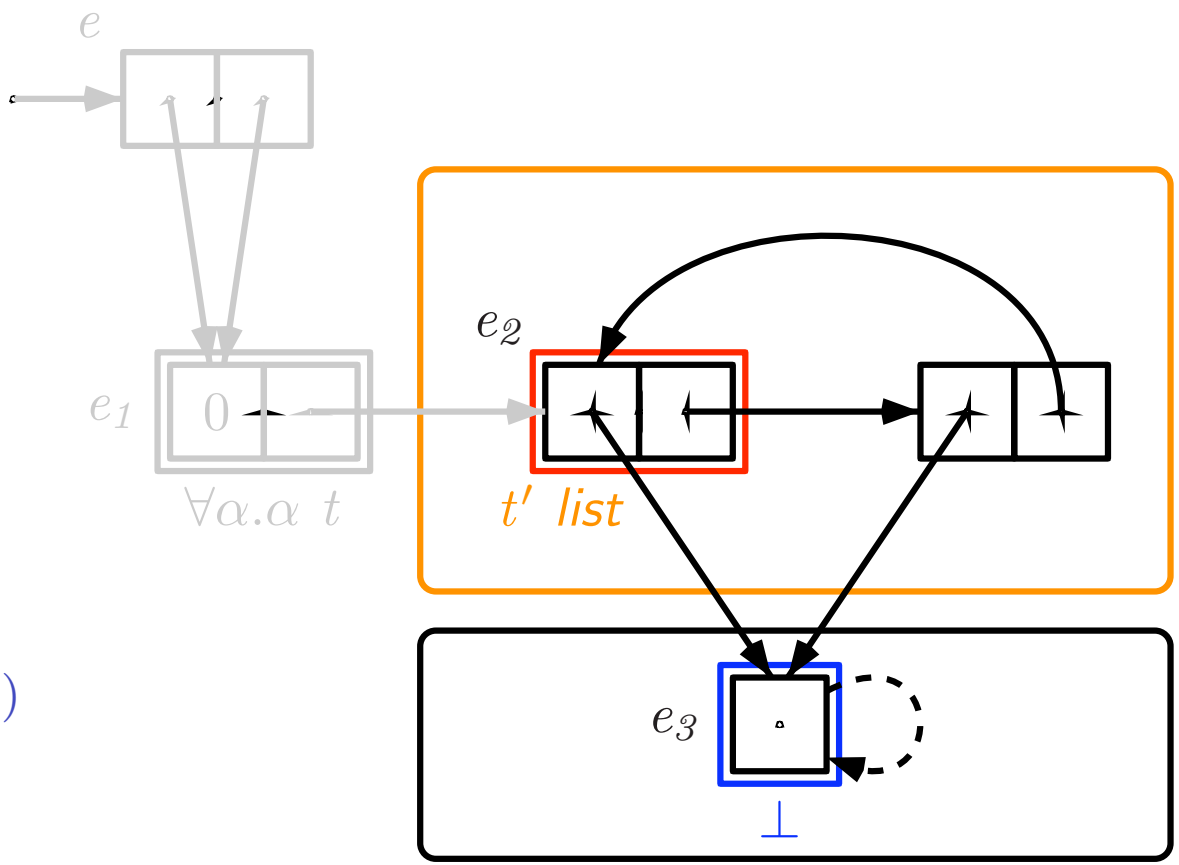
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$

Algorithme de vérification – Exemple



Programme source :

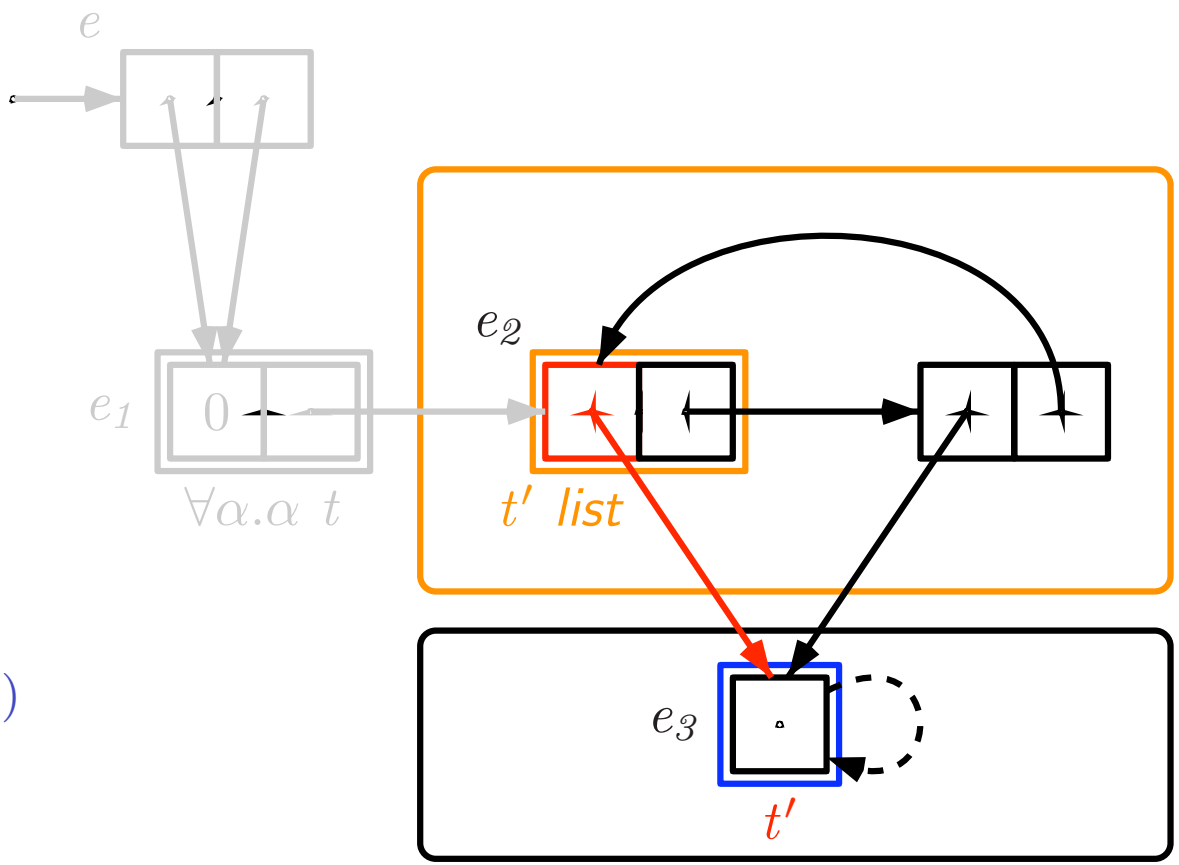
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$

Algorithme de vérification – Exemple



Programme source :

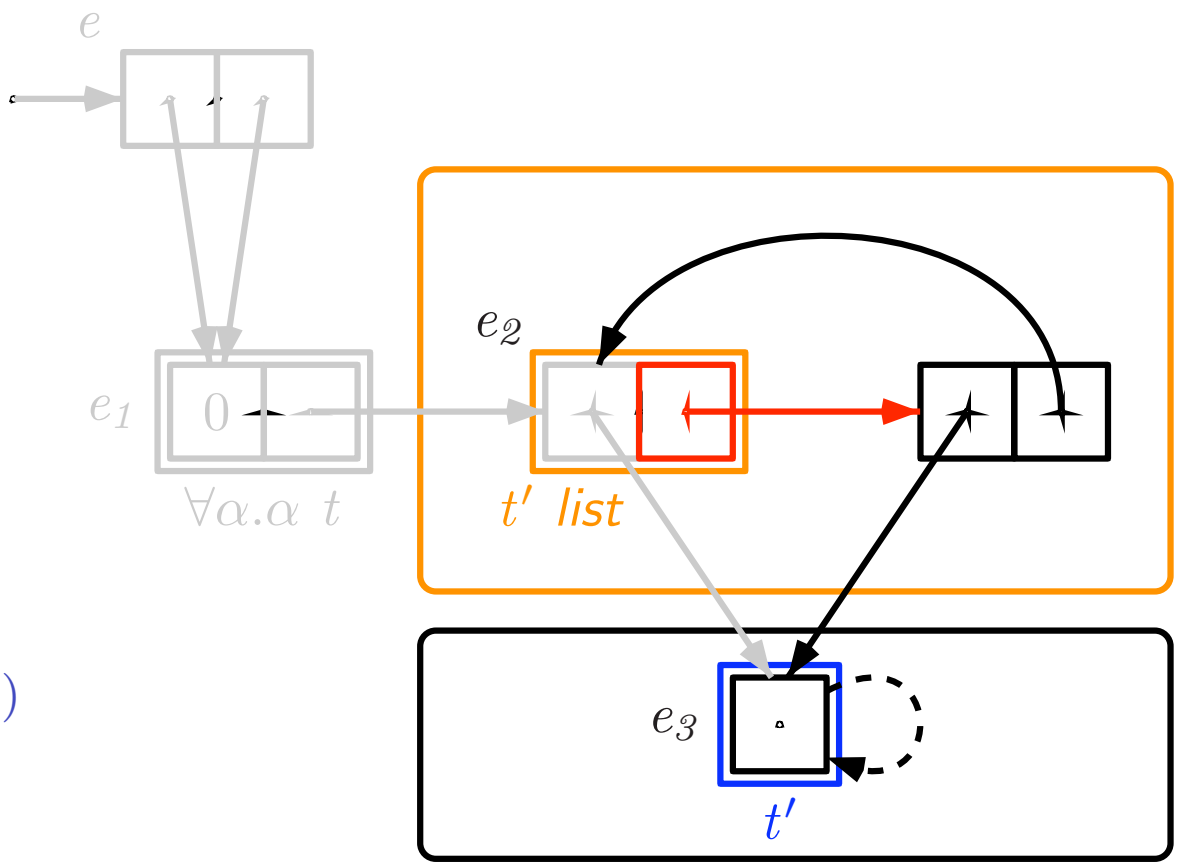
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$

Algorithme de vérification – Exemple



Programme source :

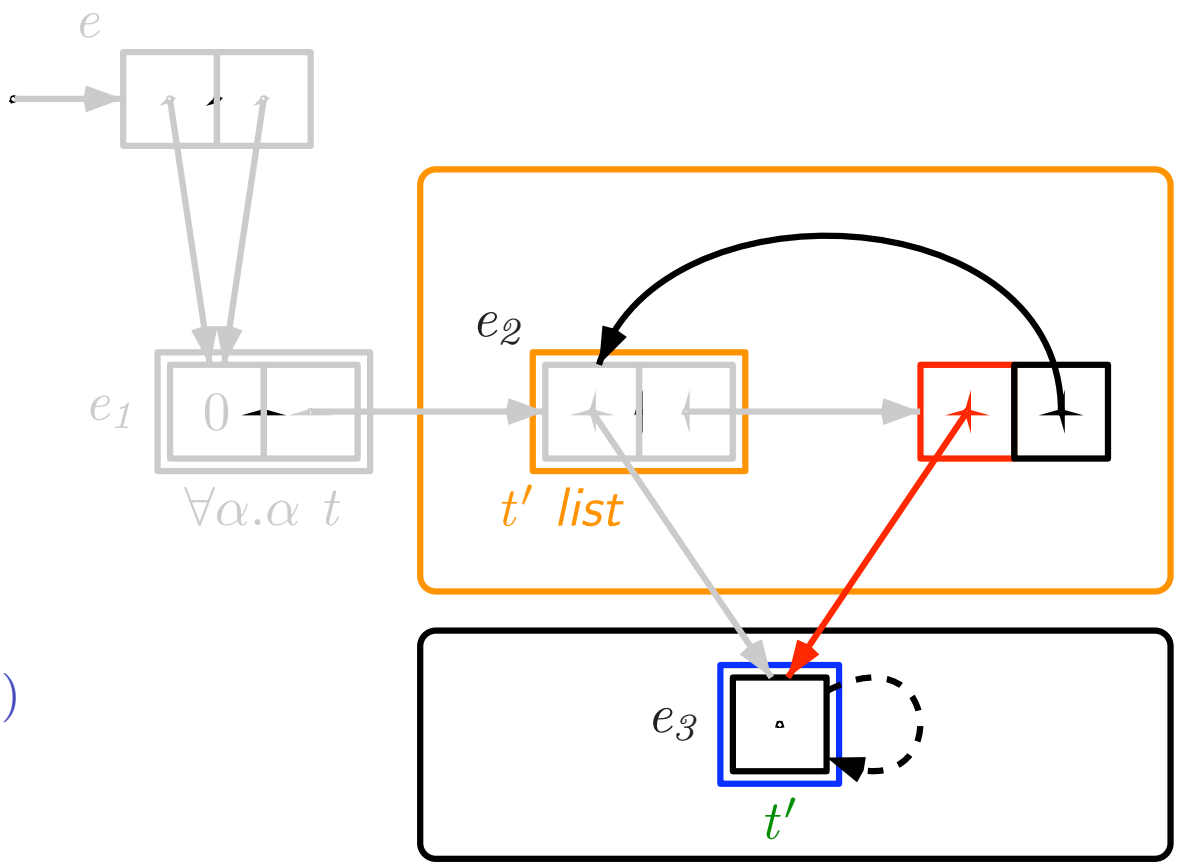
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$

Algorithme de vérification – Exemple



Programme source :

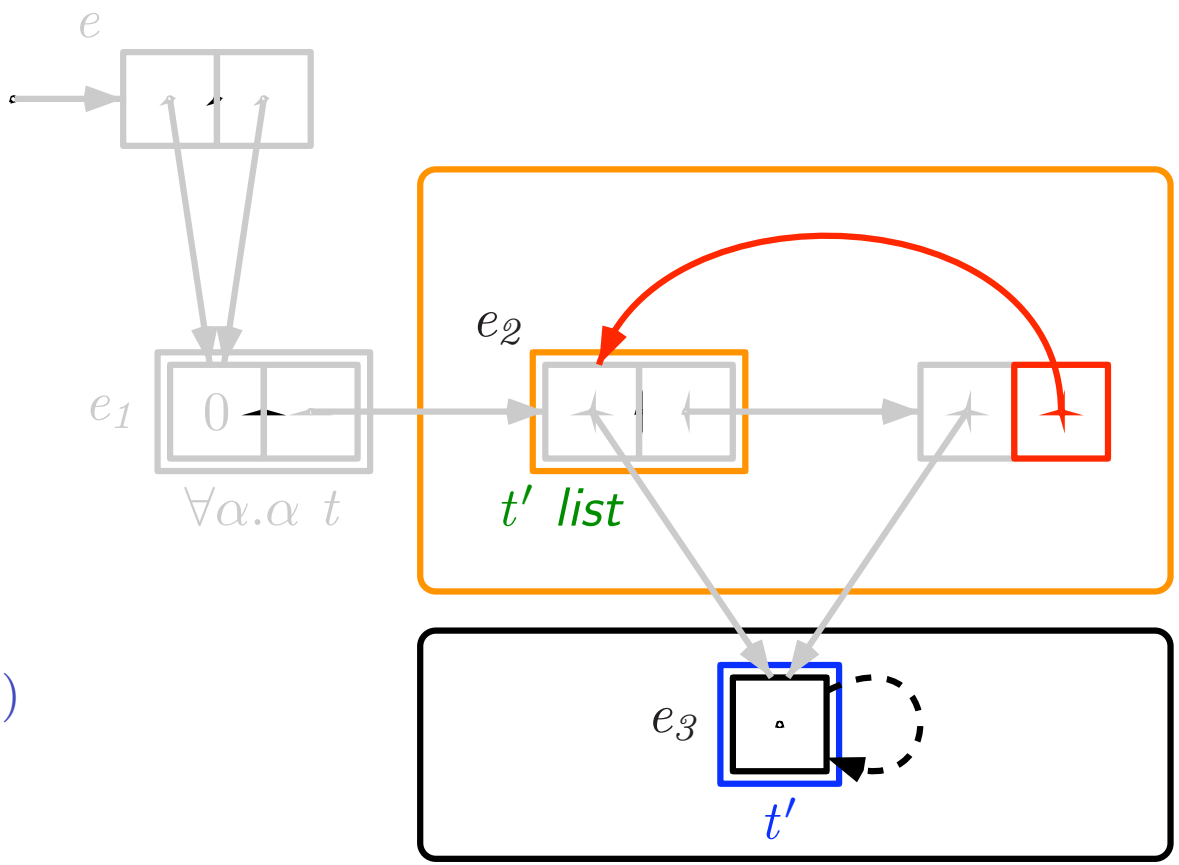
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$

Algorithme de vérification – Exemple



Programme source :

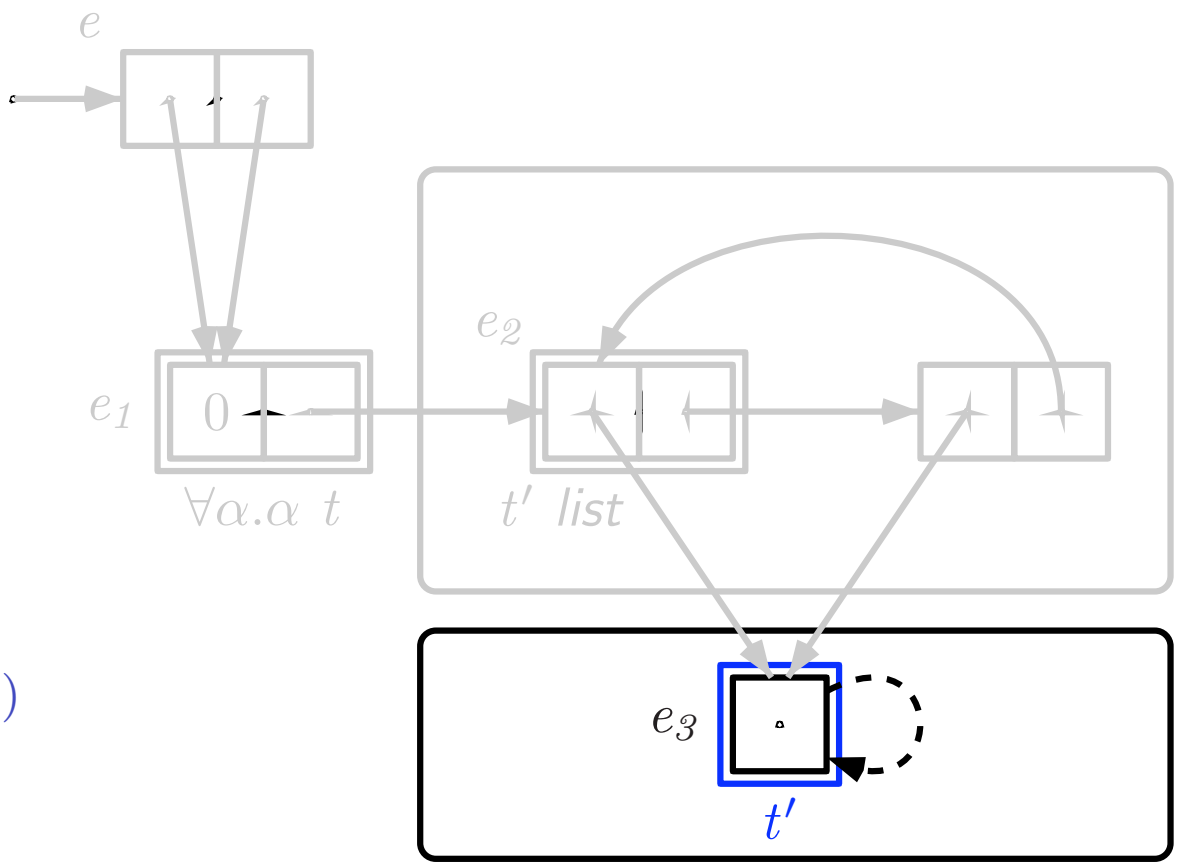
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$

Algorithme de vérification – Exemple



Programme source :

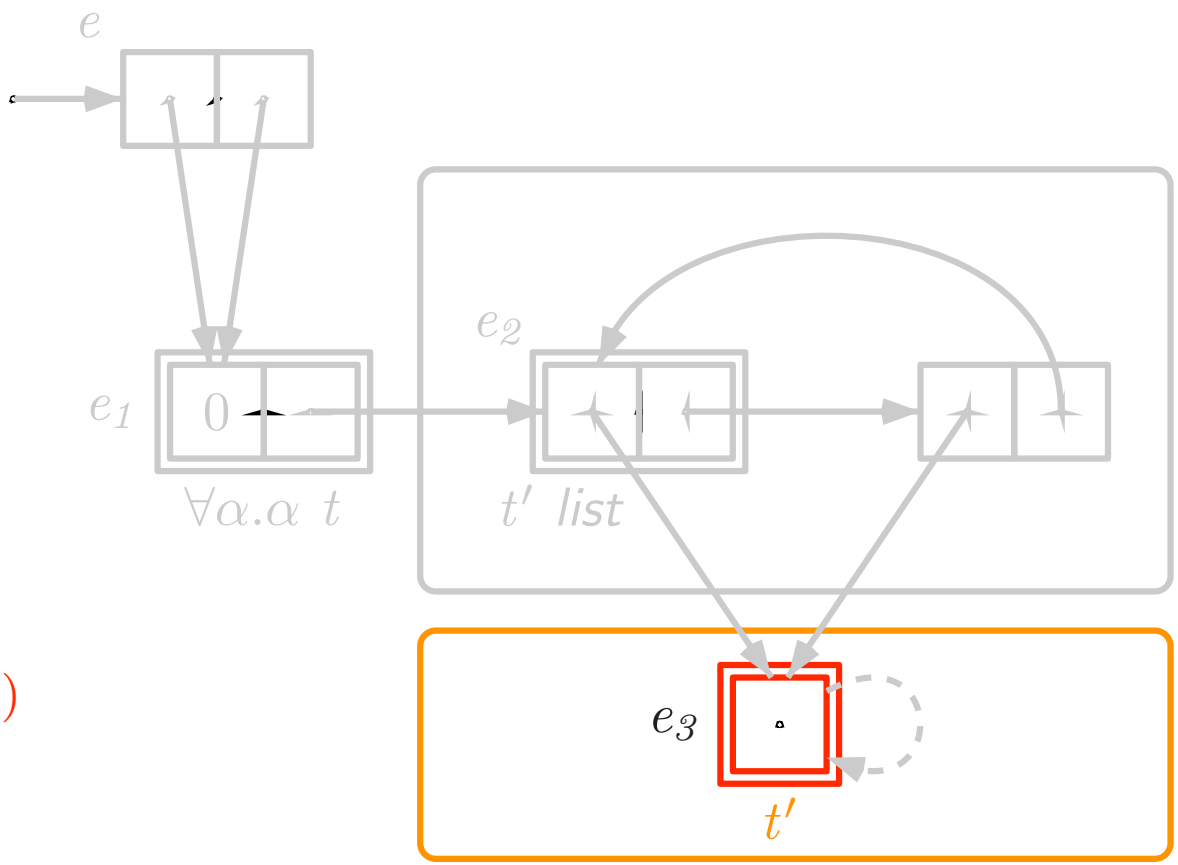
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$

Algorithme de vérification – Exemple



Programme source :

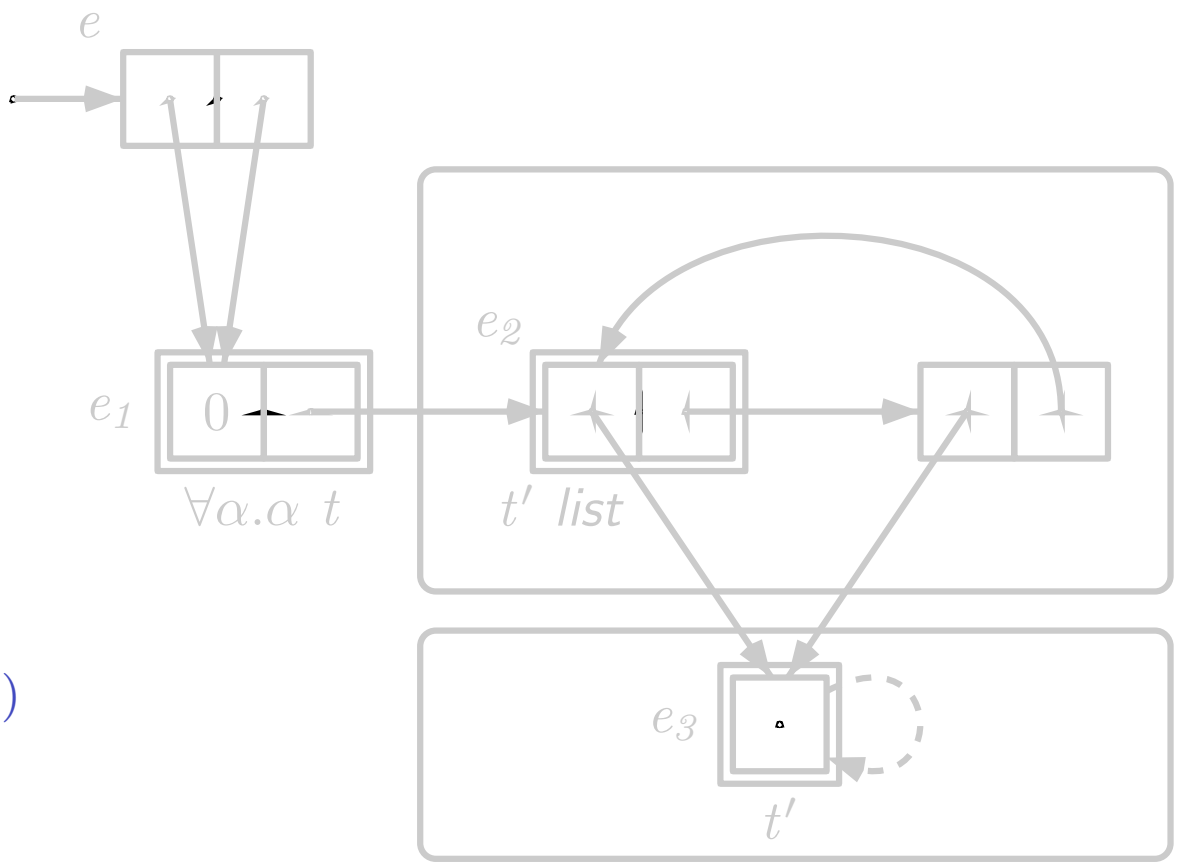
type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ list \times t'\ list$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([], e_2)$
let $e = (e_1, e_1)$

let $str = Marshal.to\_string\ e$
$Marshal.from\_string\ «\ int\ t \times float\ t\ »\ str$

Algorithme de vérification – Exemple



Programme source :

type $t' = \ldots$
let rec $e_3 = (\ldots\ e_3\ \ldots : t')$

type $\alpha\ t = \alpha\ \mathit{list} \times t'\ \mathit{list}$

let rec $e_2 = e_3 :: e_3 :: e_2$
let $e_1 = ([\,], e_2)$
let $e = (e_1, e_1)$

let $str = \mathit{Marshal.to\_string}\ e$
$\mathit{Marshal.from\_string} \ll \mathit{int}\ t \times \mathit{float}\ t \gg \mathit{str}$

Algorithme de vérification – En résumé

- Le partage résolu par anti-unification, introduit du polymorphisme
- Parcours en profondeur d'abord, avec :
  - retardement de la vérification des valeurs partagées jusqu'à exploration complète de leur contexte
  - recherche de composantes fortement connexes et tri topologique
  - récursivement, traitement de chacune des composantes fortement connexes

Algorithme de vérification – En résumé

- Le partage résolu par anti-unification, introduit du polymorphisme
- Parcours en profondeur d'abord, avec :
  - retardement de la vérification des valeurs partagées jusqu'à exploration complète de leur contexte
  - recherche de composantes fortement connexes et tri topologique
  - récursivement, traitement de chacune des composantes fortement connexes

- Ce parcours permet d'abstraire la représentation mémoire par le langage suivant :

$$
\begin{array}{lll}
v ::= & n \mid p \mid r & \textit{entier, pointeurs} \\
\mid & \mathsf{Block}(i, v_1, \ldots, v_n) & \textit{bloc alloué} \\
\mid & \mathsf{let}\ (p_1, \ldots, p_n) = (v_1, \ldots, v_n)\ \mathsf{in}\ w & \textit{partage} \\
\mid & \mathsf{fix}\ (r_1, \ldots, r_n) = (v_1, \ldots, v_n) & \textit{cycles}
\end{array}
$$

Propriétés de l'algorithme

Formalisation

- un langage de valeurs : sous-ensemble de ML **sans calcul**, avec partage (**let**) et cycles (**let rec**) ;
- le langage de valeurs-mémoire v précédent ;
- une fonction de « sérialisation » : $[\![e]\!] = v$;
- un algorithme de vérification $\mathit{Check}(\Gamma, \sigma, v)$ produisant un échec ou un programme e et un environnement Γ'.

Correction

Soient une valeur v, et un schéma de type σ. Si $\mathit{Check}(\emptyset, \sigma, v)$ réussit en calculant un programme e, alors $[\![e]\!] = v$ et $\forall \tau \leq \sigma,\ \emptyset \vdash^{ML^{rec}} e : \tau$.

Complétude

Soient une expression close e, et un type τ. Si $\emptyset \vdash^{ML} e : \tau$, alors $\mathit{Check}(\emptyset, \mathit{gen}(\emptyset, \tau), [\![e]\!])$ réussit.

Implémentation

Types mutables et types cycliques

- pris en compte

Restrictions actuelles

- on ne gère pas les variants polymorphes ;
- on ne gère pas les fermetures, c'est-à-dire ni les fonctions, ni les objets.

Actuellement tout est codé en OCaml

En attendant de voir si les tests de performance confirment ce choix.

Conclusion

Une méthode de vérification efficace qui garantit l'intégrité des valeurs désérialisées.

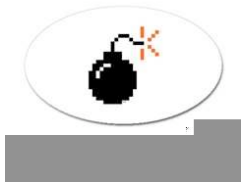
Méthode de passage de types à l'exécution et algorithme de vérification sont indépendants l'un de l'autre.

Une alternative sûre au mécanisme actuel d'OCaml.

Implémentation

Les valeurs *mutables*

?



?

```
let r = ref [ ]
let str = Marshal.to_string (r,r)
let (i,f) = Marshal.from_string
    « (int list) ref * (float list) ref » str

i := [0]
List.map print_float !f
```

Implémentation

Les valeurs *mutables*

```
let r = ref [ ]
let str = Marshal.to_string (r,r)
let (i,f) = Marshal.from_string
    « (int list) ref * (float list) ref » str

i := [0]
List.map print_float !f
```

Solution

On vérifie que le type attendu pour un *champ mutable* d'un enregistrement ne contient pas de variable quantifiée universellement introduite par anti-unification.

Gestion des valeurs de type

Sérialisation

- écriture d'un graphe mémoire ;

Désérialisation

- polymorphisme :
  - accepter la valeur «$\forall\alpha\ .\ \alpha$ `list`» au type `(int list) tyRepr` ;
  - la refuser au type $\forall\alpha\ .\ \alpha$ `tyRepr`.
- $\Rightarrow$ Autoriser la lecture de la représentation d'un type, comme étant du type de la représentation d'une de ses instances

Solution : extension de la vérification par une analyse explicite des différents cas de la structure de `tyRepr`.

Complétude (bis)

Avec récursion polymorphe :

$\mathbf{type}\ \alpha\ \mathtt{nest} = \ldots \mid \mathtt{B}\ \mathbf{of}\ (\alpha * \alpha)\ \mathtt{nest} \mid \ldots$
$\mathbf{let}\ \mathbf{rec}\ \mathtt{r} : (\forall\alpha\ .\ \alpha\ \mathtt{nest}) = \mathtt{B}\ \mathtt{r}$
$\mathbf{let}\ \mathtt{str} = \mathtt{Marshal.to\_string}\ (\mathtt{r} : \mathtt{int\ nest})$

Échec :

$\mathtt{Marshal.from\_string}\ «\ \mathtt{int\ nest}\ »\ \mathtt{str}$

Car :

$$(\mathtt{int} * \mathtt{int})\ \mathtt{nest} \not\simeq \mathtt{int\ nest}.$$