Inférence de flots d'information pour ML

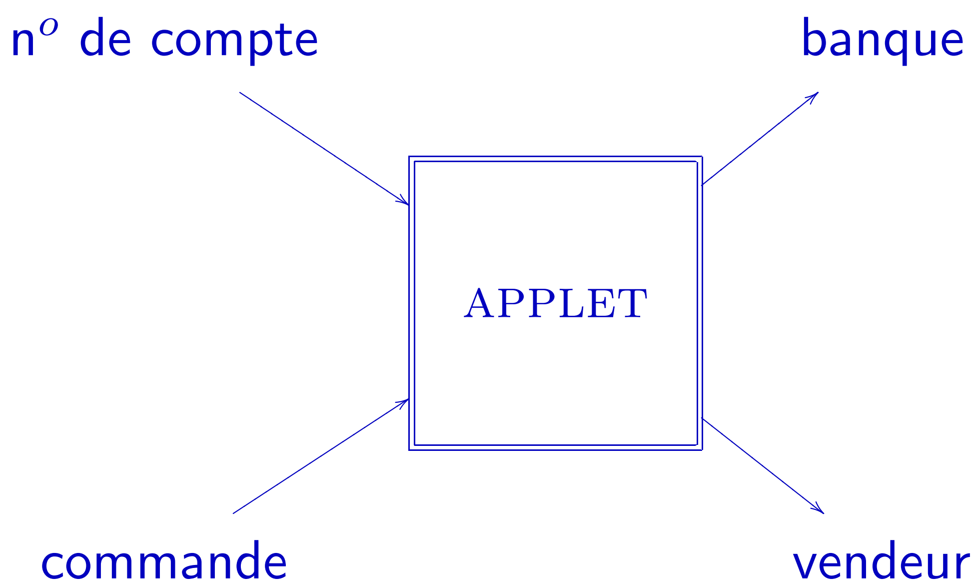
Vincent Simonet
INRIA Rocquencourt – Projet Cristal

Séminaire LANDE

Vendredi 26 octobre 2001

Vincent.Simonet@inria.fr http://cristal.inria.fr/~simonet/

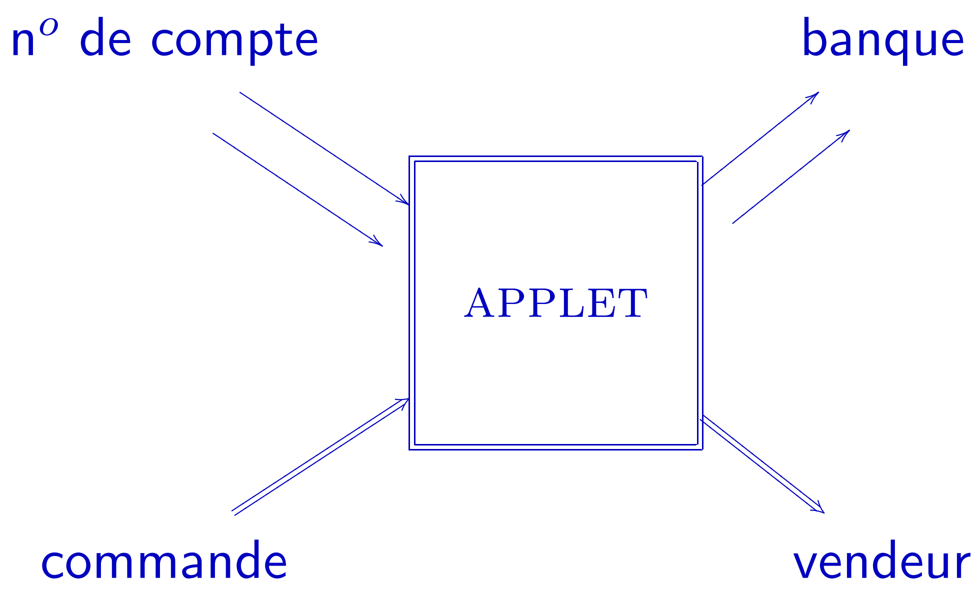
Flots d'information



$$\text{compte}^H \times \text{commande}^L \rightarrow \text{banque}^H \times \text{vendeur}^L$$

$$(\forall \alpha\beta\gamma\delta)\ [\alpha \sqcup \beta \leq \gamma, \beta \leq \delta]\ \text{compte}^\alpha \times \text{commande}^\beta \rightarrow \text{banque}^\gamma \times \text{vendeur}^\delta$$

Non-interférence



Systèmes existants

Dennis Volpano et Geoffrey Smith (1997)
Système de type sur un langage impératif simple. Limité au premier ordre et à un nombre fini de références globales.

Nevin Heintze et Jon G. Riecke SLam Calculus (1997)
Lambda-calcul avec références et processus. Le typage des références n'est pas fin. Il n'est pas énoncé ni prouvé de propriété de sûreté.

Andrew C. Myers JFlow (1999)
Analyse de flots d'information sur Java. Le système est complexe et n'est pas prouvé.

Steve Zdancewic et Andrew C. Myers (2001)
Analyse sur un langage de bas niveau avec continuations linéaires.

Le langage ML

λ-calcul en appel par valeur avec let-polymorphe

$$x \qquad k \qquad \mathsf{fun}\ x \rightarrow e$$

$$e_1\, e_2 \qquad \mathsf{let}\ x = v\ \mathsf{in}\ e \qquad \mathsf{bind}\ x = e_1\ \mathsf{in}\ e_2$$

avec références

$$\mathsf{ref}\ e \qquad e_1 := e_2 \qquad !\, e$$

et exceptions

$$\varepsilon\, e \qquad \mathsf{raise}\ e \qquad e_1\ \mathsf{handle}\ \varepsilon\, x \succ e_2 \qquad e_1\ \mathsf{handle}\ x \succ e_2$$

Formes v-normales

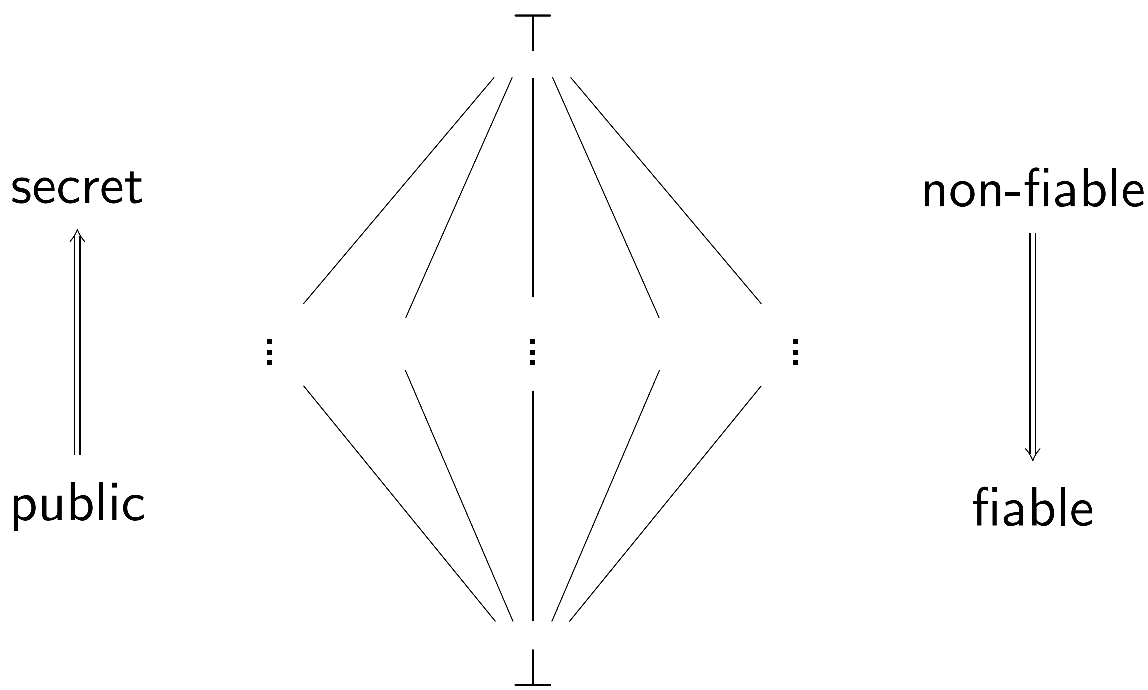
$$
\begin{array}{lcl}
v & ::= & x \mid k \mid \mathsf{fun}\ x \to e \mid \varepsilon\, v \\
e & ::= & v\, v \mid \mathsf{ref}\ v \mid v := v \mid !\, v \mid \mathsf{raise}\ v \mid \mathsf{let}\ x = v\ \mathsf{in}\ e \mid E[v] \\
E & ::= & \mathsf{bind}\ x = [\,]\ \mathsf{in}\ e \mid [\,]\ \mathsf{handle}\ \varepsilon\, x \succ e \mid [\,]\ \mathsf{handle}\ x \succ e
\end{array}
$$

On peut mettre toute expression « usuelle » sous cette forme, à condition de choisir un ordre d'évaluation. Par exemple :

$$
e_1\, e_2 \Rightarrow \begin{cases} \mathsf{bind}\ x_1 = e_1\ \mathsf{in}\ (\mathsf{bind}\ x_2 = e_2\ \mathsf{in}\ x_1\, x_2) & \textit{éval. de gauche à droite} \\ \mathsf{bind}\ x_2 = e_2\ \mathsf{in}\ (\mathsf{bind}\ x_1 = e_1\ \mathsf{in}\ x_1\, x_2) & \textit{éval. de droite à gauche} \end{cases}
$$

Niveaux d'information

On associe à chaque donnée un niveau d'information, appartenant à un treillis $\mathcal{L}$. Ces niveaux peuvent représenter différentes propriétés : sécurité, intégrité...



Pour la suite, on fixe $\mathcal{L} = \{L \leq H\}$.

Flots directs et indirects

Flots directs

$$x := \text{not } y$$

$$x := (\text{if } y \text{ then } \mathit{false} \text{ else } \mathit{true})$$

Flots indirects

$$\text{if } y \text{ then } x := \mathit{false} \text{ else } x := \mathit{true}$$

$$x := \mathit{true};\ \text{if } y \text{ then } x := \mathit{false} \text{ else } ()$$

$$x := \mathit{true};\ (\text{if } y \text{ then raise } A \text{ else } ()) \text{ handle } \_ \succ x := \mathit{false}$$

À chaque point du programme correspond un niveau pc qui représente les informations que l'on peut apprendre en sachant que ce point a été exécuté.

Système de types

Approche semi-syntaxique

(exemples dans le cas du typage ML)

Système logique	Système syntaxique
Types bruts e.g. int, int $\rightarrow$ int...	Expressions de type e.g. int, α, $\alpha \rightarrow \alpha$...
Polytypes e.g. $\{t \rightarrow t \mid t$ type brut$\}$	Schémas de type e.g. $\forall \alpha . \alpha \rightarrow \alpha$

On raisonne sur le système logique. Le système syntaxique est interprété dans le système logique.

Algèbre de types

Les niveaux d'information ℓ, pc appartiennent à un treillis $\mathcal{L}$.

Les exceptions sont décrites par des rangées d'alternatives r :

$$\begin{array}{rcl} a & ::= & \mathsf{Abs} \mid \mathsf{Pre}\ pc \\ r & ::= & \{\varepsilon \mapsto a\}_{\varepsilon \in \mathcal{E}} \end{array}$$

Les types sont annotées par des niveaux et des rangées :

$$t \quad ::= \quad \mathsf{int}^{\ell} \mid \mathsf{unit} \mid (t \xrightarrow{pc\ [r]} t)^{\ell} \mid t\ \mathsf{ref}^{\ell} \mid r\ \mathsf{exn}^{\ell}$$

Jugements

On distingue deux formes de jugements.

Jugements sur des valeurs

$$\Gamma \vdash v : t$$

Jugements sur des expressions

$$pc, \Gamma \vdash e : t \;\; [r]$$

Contraintes

Contraintes de sous-typage $t_1 \leq t_2$

La relation de sous-typage étend l'ordre sur les niveaux aux types. Par exemple :

$$\mathsf{int}^{\ell_1} \leq \mathsf{int}^{\ell_2} \Leftrightarrow \ell_1 \leq \ell_2 \qquad\qquad \mathsf{Abs} \leq \mathsf{Pre}\ pc$$

Gardes $\ell \lhd t$

Les gardes permettent de « marquer » un type par un niveau d'information :

$$pc \lhd \mathsf{int}^{\ell} \Leftrightarrow pc \leq \ell \qquad\qquad pc \lhd t\ \mathsf{ref}^{\ell} \Leftrightarrow pc \leq \ell$$

Conditionnelles $pc \leq_{\mathsf{Pre}} a$

$pc \leq_{\mathsf{Pre}} a$ est une abbréviation pour $a \neq \mathsf{Abs} \Rightarrow \mathsf{Pre}\ pc \leq a$.

Sous-typage et polymorphisme

Sous-typage et polymorphisme interviennent de manière orthogonale :

Sous-typage Permet d'augmenter librement le niveau d'une donnée (e.g. considérer *secrète* une donnée *publique* ou *non-fiable* une donnée *fiable*) :

$$\frac{\Gamma \vdash v : t \qquad t \leq t'}{\Gamma \vdash v : t'}$$

Polymorphisme Nécessaire pour pouvoir utiliser la même fonction avec des entrées de niveaux différents :

$$\mathsf{let}\ succ \ = \ \mathsf{fun}\ x \rightarrow (x+1)$$

Références

$$\textsc{Ref}\quad \frac{\Gamma \vdash v : t \qquad pc \lhd t}{pc, \Gamma \vdash \textsf{ref}\ v : t\ \textsf{ref}^{\ell}\ \ [r]}$$

$$\textsc{Deref}\quad \frac{\Gamma \vdash v : t'\ \textsf{ref}^{\ell} \qquad t' \leq t \qquad \ell \lhd t}{pc, \Gamma \vdash !\,e : t\ \ [r]}$$

$$\textsc{Assign}\quad \frac{\Gamma \vdash e_1 : t\ \textsf{ref}^{\ell} \qquad \Gamma \vdash e_2 : t \qquad pc \lhd t \qquad \ell \lhd t}{pc, \Gamma \vdash e_1 := e_2 : \textsf{unit}\ \ [r]}$$

Le contenu d'une référence a un niveau supérieur à égal
– au pc du point où la référence est créée,
– au pc de chaque point du programme où son contenu est susceptible d'être modifié.

Exceptions

$$\textsc{Raise}\quad \frac{\Gamma \vdash v : \mathit{typexn}(\varepsilon)}{pc, \Gamma \vdash \mathsf{raise}\ (\varepsilon\, v) : *\ \left[\varepsilon : \mathsf{Pre}\ pc; \partial\mathsf{Abs}\right]}$$

$$\textsc{Handle}\quad \frac{\begin{array}{c} pc, \Gamma \vdash e_1 : t\ \left[\varepsilon : \mathsf{Pre}\ pc'; r_1\right] \\ pc \sqcup pc', \Gamma[x \mapsto \mathit{typexn}(\varepsilon)] \vdash e_2 : t\ \left[\varepsilon : a_2; r_2\right] \qquad pc' \lhd t \end{array}}{pc, \Gamma \vdash e_1\ \mathsf{handle}\ \varepsilon\, x \succ e_2 : t\ \left[\varepsilon : a_2; r_1 \sqcup r_2\right]}$$

Non-interférence

On considère une expression e de type int^L avec un « trou » x marqué H :

$$(x \mapsto t) \vdash e : \mathsf{int}^L \qquad\qquad H \lhd t$$

Théorème de non-interférence

$$\text{Si } \begin{cases} \vdash v_1 : t \\ \vdash v_2 : t \end{cases} \text{ et } \begin{cases} e[x \Leftarrow v_1] \longrightarrow^* v_1' \\ e[x \Leftarrow v_2] \longrightarrow^* v_2' \end{cases} \text{ alors } v_1' = v_2'$$

Le résultat de l'évaluation de e ne dépend pas de la valeur d'entrée placée dans le trou.

Preuve de non-interférence

1. On définit une extension *ad hoc* du langage qui permet de raisonner sur les points communs et les différences entre deux programmes.
2. On vérifie que le système de types pour le langage étendu possède *subject reduction*.
3. On montre que la non-interférence pour le langage initial est une conséquence de *subject reduction*.

Calcul partagé

Le calcul partagé permet de considérer simultanément deux expressions et deux réductions en gérant le partage.

Syntaxe

$$v ::= \ldots \mid \langle v \mid v \rangle \qquad\qquad e ::= \ldots \mid \langle e \mid e \rangle$$

On ne considère que les expressions où les $\langle \ldots \mid \ldots \rangle$ ne sont pas imbriqués.

Codage

Une expression du calcul partagé permet de représenter deux expressions du calcul simple en gérant le partage :

$$\text{if } \mathit{true} \text{ then } 0 \text{ else } 1 \qquad\qquad \text{if } \mathit{false} \text{ then } 0 \text{ else } 1$$

$$[\![\cdot \mid \cdot]\!]$$

$$\Downarrow$$

$$\text{if } \langle \mathit{true} \mid \mathit{false} \rangle \text{ then } 0 \text{ else } 1$$

Les projections $\lfloor \cdot \rfloor_1$ et $\lfloor \cdot \rfloor_2$ permettent de retrouver les expressions originelles :

$$\text{if } \langle \mathit{true} \mid \mathit{false} \rangle \text{ then } 0 \text{ else } 1$$

$$\lfloor \cdot \rfloor_1 \qquad\qquad \lfloor \cdot \rfloor_2$$

$$\text{if } \mathit{true} \text{ then } 0 \text{ else } 1 \qquad\qquad \text{if } \mathit{false} \text{ then } 0 \text{ else } 1$$

Réduction du calcul partagé

À partir des règles de réduction du calcul de base, on obtient les règles pour le calcul partagé. À chaque fois que des crochets bloquent la réduction, il faut les déplacer.

Exemple :

$$(\mathsf{fun}\ x \rightarrow e)\, v \rightarrow e[x \Leftarrow v] \qquad (\beta)$$

$$\langle v_1 \mid v_2 \rangle\, v \rightarrow \langle v_1\ \lfloor v \rfloor_1 \mid v_2\ \lfloor v \rfloor_2 \rangle \qquad (\text{lift-app})$$

Simulation

Correction

Si $e \rightarrow e'$ alors $\begin{cases} \lfloor e \rfloor_1 \longrightarrow^{=} \lfloor e' \rfloor_1 \\ \lfloor e \rfloor_2 \longrightarrow^{=} \lfloor e' \rfloor_2 \end{cases}$

(calcul partagé) (calcul simple)

Complétude

Si $\begin{cases} e_1 \longrightarrow^{*} v_1 \\ e_2 \longrightarrow^{*} v_2 \end{cases}$ alors $\llbracket e_1 \mid e_2 \rrbracket \longrightarrow^{*} \llbracket v_1 \mid v_2 \rrbracket$

(calcul simple) (calcul partagé)

Typage de $\langle \ldots \mid \ldots \rangle$

$$\textsc{Bracket}\quad \frac{\Gamma \vdash v_1 : t \qquad \Gamma \vdash v_2 : t \qquad H \lhd t}{\Gamma \vdash \langle v_1 \mid v_2 \rangle : t}$$

Une valeur de type int^H peut être en entier k ou un crochet $\langle k_1 \mid k_2 \rangle$.

Une valeur de type int^L ne peut être qu'un entier simple k.

Subject reduction et non-interférence

Soit $(x \mapsto t) \vdash e : \mathsf{int}^L$ avec $H \triangleleft t$.

Subject-reduction

Si $\vdash e' : \mathsf{int}^L$ et $e' \rightarrow^* v'$ alors $\vdash v' : \mathsf{int}^L$

$$\begin{array}{ccc} \uparrow & \qquad\qquad & | \\ e' = e[x \Leftarrow v] & & v' = k \\ | & & \downarrow \end{array}$$

Non-interférence pour le calcul partagé

Si $\vdash v : t$ et $e[x \Leftarrow v] \rightarrow^* v'$ alors $\lfloor v' \rfloor_1 = \lfloor v' \rfloor_2$

Non-interférence

Soit $(x \mapsto t) \vdash e : \mathsf{int}^L$ avec $H \lhd t$.

Non-interférence pour le calcul partagé

$$\text{Si} \quad \vdash v : t \quad \text{et} \quad e[x \Leftarrow v] \longrightarrow^* v' \quad \text{alors} \quad \lfloor v' \rfloor_1 = \lfloor v' \rfloor_2$$

$$v = \langle v_1 \mid v_2 \rangle \qquad\qquad v' = [\![v_1 \mid v_2]\!]$$

Non-interférence pour le calcul simple

$$\text{Si} \quad \begin{cases} \vdash v_1 : t \\ \vdash v_2 : t \end{cases} \quad \text{et} \quad \begin{cases} e[x \Leftarrow v_1] \longrightarrow^* v_1' \\ e[x \Leftarrow v_2] \longrightarrow^* v_2' \end{cases} \quad \text{alors} \quad v_1' = v_2'$$

Extension du langage

On souhaite généralement étendre le langage étudié :

Pour accroître la richesse du langage Ajout de sommes, de paires. Étude d'un cas général pour des « primitives » des langages réels (opérations arithmétiques, de comparaison).

Pour mieux typer certaines constructions usuelles

$$e_1 \text{ finally } e_2 \hookrightarrow \text{bind } x = (e_1 \text{ handle } y \succ e_2;\ \text{raise } y) \text{ in } e_2;\ x$$

$$e_1 \text{ handle } x \succ e_2 \text{ reraise} \hookrightarrow e_1 \text{ handle } x \succ (e_2;\ \text{raise } x)$$

Notre approche se prête particulièrement bien à ce type d'extension : il suffit d'étendre la preuve de *subject reduction* aux nouvelles règles introduites.

Extension du langage

Primitives

$$\frac{\Gamma \vdash v_1 : \mathsf{int}^{\ell} \qquad \Gamma \vdash v_2 : \mathsf{int}^{\ell}}{pc, \Gamma \vdash v_1 + v_2 : \mathsf{int}^{\ell} \quad [\partial\mathsf{Abs}]} \qquad \frac{\Gamma \vdash v_1 : t \qquad \Gamma \vdash v_2 : t \qquad t \blacktriangleleft \ell}{pc, \Gamma \vdash v_1 = v_2 : \mathsf{bool}^{\ell} \quad [\partial\mathsf{Abs}]}$$

$$\frac{\Gamma \vdash v : t \qquad t \blacktriangleleft \ell}{pc, \Gamma \vdash \mathsf{hash}\ v : \mathsf{int}^{\ell} \quad [\partial\mathsf{Abs}]}$$

Une nouvelle forme de contraintes $t \blacktriangleleft \ell$

$t \blacktriangleleft \ell$ contraint *tous* les niveaux d'information apparaissant dans t et ses sous termes à être inférieurs à ℓ.

Paires

$$t \quad ::= \quad \ldots \mid t_1 \times t_2$$

Il n'est pas nécessaire de munir le constructeur de types $\times$ d'un niveau d'information propre. Les niveaux présents dans les types des composantes suffisent :

$$\begin{array}{lcl} \ell \lhd t_1 \times t_2 & \Leftrightarrow & \ell \lhd t_1 \wedge \ell \lhd t_2 \\ t_1 \times t_2 \blacktriangleleft \ell & \Leftrightarrow & t_1 \blacktriangleleft \ell \wedge t_2 \blacktriangleleft \ell \end{array}$$

Vers une extension du compilateur Caml

Le langage étudié permet de prendre en compte la totalité du langage Caml (à l'exception de la bibliothèque `threads`).

L'implantation d'un prototype est en cours. Elle nécessite de résoudre plusieurs problèmes liés à l'utilisation d'un système de types doté de sous-typage :

- Efficacité de l'algorithme d'inférence
- Lisibilité des types inférés
- Clarté des messages d'erreur
- ...

Inférence de types

Un algorithme d'inférence comporte deux parties relativement distinctes.

Un jeu de règles d'inférence On peut dériver de manière quasi-systématique un jeu de règles d'inférence d'un système tel que le nôtre.

$$\textsc{Ref}\ \frac{\Gamma \vdash v : t \qquad pc \lhd t}{pc, \Gamma \vdash \mathsf{ref}\ v : t\ \mathsf{ref}^{\ell}\ \ [r]} \rightsquigarrow \textsc{Inf-Ref}\ \frac{\Gamma, C \vdash v : \alpha}{\pi, \Gamma, C \cup \{\beta = \alpha\ \mathsf{ref}^{\lambda}, \pi \lhd \alpha\} \vdash \mathsf{ref}\ v : \beta\ \ [\rho]}$$

Un solveur Les schémas de types comportent des ensembles de contraintes. Il faut tester leur satisfiabilité et les simplifier.

Exemple : listes

```
type ('a, 'b) list = <'b>
  | []
  | (::) of 'a * ('a, 'b) list
```

```
let rec length = function
  | []     -> 0
  | _ :: l -> 1 + length l
```

$$\forall[\alpha \leq \beta]. * \ \mathsf{list}^{\alpha} \rightarrow \mathsf{int}^{\beta}$$

Exemple : listes (2)

```
let rec iter f = function
  | []     -> ()
  | x :: l -> f x; iter f l
```

$$\forall[\delta \leq \partial\gamma].(\alpha \xrightarrow{\gamma\ [\delta]} *)^\gamma \rightarrow \alpha\ \mathsf{list}^\gamma \xrightarrow{\gamma\ [\delta]} \mathsf{unit}$$

```
let rec iter2 f = fun
  | []         []         -> ()
  | (x1 :: l1) (x2 :: l2) -> f x1 x2; iter2 f l1 l2
  | _          _          -> raise X
```

$$\forall[\epsilon \leq \zeta; \mathsf{Pre}\ \gamma \leq \zeta; \delta \leq \partial\gamma].$$

$$(\alpha \xrightarrow{\gamma\ [X:\epsilon;\delta]} \beta \xrightarrow{\gamma\ [X:\epsilon;\delta]} *)^\gamma \rightarrow \alpha\ \mathsf{list}^\gamma \rightarrow \beta\ \mathsf{list}^\gamma \xrightarrow{\gamma\ [X:\zeta;\delta]} \mathsf{unit}$$