INF 544 Luc Maranget

Les types (que l'on dit simples)

Luc.Maranget@inria.fr

http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/TLP/

A Qu'est-ce qu'un type ?

B PCF simplement typé.

C Sémantique dénotationelle.

Quelques problèmes à l'exécution

- Par exemple, `(`**`Fun`** `x -> x) + 1`
- Ou bien, `1 2` (un appliqué à deux)
- Et même, **`Ifz`** `(`**`Fun`** `x -> x+1)` **`Then`** `0` **`Else`** `1`

Et en effet,

- additionner des fonctions (impossible),
- appliquer des entiers (impossible),
- tester si une fonction vaut 0 (possible, mais bizarre, ne présage rien de bon pour la suite).

Ce qui se passe à l'exécution

Soit le cas `(`**Fun** `x -> x)+1`.

Dans par exemple la machine ASEC.

```
let rec run a s e c = match a,s,c with
...
| Int n2,Val (Int n1)::s,IOp ADD::c ->
    run (Int (n1+n2)) s e c
...
|  _,_,_ -> raise Error
```

Donc : lancer une exception (erreur détectée).

Ou dans le code compilé :

```
#IOp (Add)
        addl %eax,0(%esp)
        popl %eax
```

Donc : calculer n'importe quoi (erreur non-détectée).

Ce que dit la sémantique

De par exemple « `1 2` ».

- À grands pas : rien.
- À petits pas : terme bloqué (forme normale qui n'est pas une valeur).
- Dénotationelle : c'est quoi déjà ?

L'erreur peut être peu évidente

```
Let fact =
    Fix fact -> Fun n ->
      Ifz n Then 1 Else n * fact (n-1) In
Let r = Ifz 100-fact 10 Then 0 Else (Fun x -> x) In
r 1
```

L'exécution échoue (application de `0` à `1`).

On doit pouvoir détecter cette erreur à l'usine (compilation), avant qu'elle ne se produise chez le client (exécution).

Sémantique dénotationnelle

Associer à un terme `t` (syntaxe) un "vrai" truc $t = [\![\texttt{t}]\!]$.

Vrai veut dire disons,

- Avec les moyens reconnus légaux (cf. Marx).
- Vrais (cf. Platon).

Simplifions : il existe $\mathbb{N}$ et une "vraie" fonction est une fonction de A dans B où A et B sont des ensembles.

On peut donc essayer d'associer :

- À `0`, l'entier ($\in \mathbb{N}$) 0.
- À **Fun** `x -> x+1`, la fonction de $\mathbb{N}$ dans $\mathbb{N}$, qui à x associe $x + 1$.
- À **Fun** `x -> x`, la fonction de ? dans ?, qui à x associe x.
- À **Fun** `x -> x x` ? (Là, ça va pas être possible.)

Deux problèmes, une solution

Les problèmes :

- Ne livrer que des programmes sans erreurs (sans erreurs du style « `1 2` »).
- Définir la sémantique dénotationelle de PCF.
- (Éviter les paradoxes du genre $X = \{x \mid x \notin x\}$.)

La solution : obliger le programmeur à spécifier le domaine de définition des fonctions.

Une solution radicale, mais qui va fonctionner.

Un nouveau PCF

$$
\begin{array}{rcl}
t & ::== & \texttt{n} \\
& | & x \\
& | & t_1 \ \texttt{op} \ t_2 \\
& | & t_1 \ t_2 \\
& | & \textbf{Fun} \ (x{:}T) \ \texttt{->} \ t \\
& | & \textbf{Fix} \ (x{:}T) \ \texttt{->} \ t \\
& | & \textbf{Let} \ x \ \texttt{=} \ t_1 \ \textbf{In} \ t_2
\end{array}
$$

Cf. définition de méthode en Java:

```
int f(int x) { ... }
```

La même chose en Caml

Syntaxe abstraite du nouveau PCF (dans `T.Ast`).

```
(*
     Var.t    : type des variables (string)
    T.Type.t : type des types (???)
*)
type t =
  | Num of int
  | Var of Var.t
  | Op  of Op.t * t * t
  | Ifz of t * t * t
  | Let of Var.t * t * t
  | App of t * t
  | Fun of Var.t * T.Type.t * t
  | Fix of Var.t * T.Type.t * t
```

Mais qu'est-ce que `T.Type.t` (T) ?

Ensembles de définition des fonctions

Pour éviter « `1 2` » et « `(`**`Fun`** `x -> x+1)+2` »), il doit suffire de séparer entiers et fonctions.

Pour autoriser

`Let` `f =` **`Fun`** `x -> x+1` **`In`** `f 1 + 2`

Et interdire

`Let` `f =` **`Fun`** `x -> x 1` **`In`** `f 1 + 2`

Il faut être un peu plus détaillé, les deux types de `f`

`Nat -> Nat` **`(Nat -> Nat) -> Nat`**

Algèbre des types

Les types sont décrits ainsi :

- **Nat** est un type.
- Si A et B sont des types, alors A `->` B est un type.

Autrement dit :

$$\mathbb{N} \in \mathcal{T} \qquad \frac{A \in \mathcal{T} \qquad B \in \mathcal{T}}{A \texttt{ -> } B \in \mathcal{T}}$$

Ou encore :

```
type t = Nat | Arrow of t * t
```

C'est le language des types (simples), il n'a pour le moment aucune sémantique.

Types et ensembles

Idée évidente pour la sémantique : des ensembles.

Selon notre programme de vérifier les types à la compilation, les types ne peuvent pas être des ensembles quelconques.

Ex. Pourquoi pas un type $\mathbb{N}^*$? (**Nat***) Ça serait bien commode :

Let `div` = **Fun** (`x`:**Nat**) -> **Fun** (`y`:**Nat***) -> `x / y`

Et bien non,

- Typer `div 100 (fact 10-100)` demande de prouver que la valeur de `fact 10` est > 100, c'est beaucoup demander à un compilateur.
- Quel type donner à la soustraction ?

Tant pis : l'usine (compilateur) peut livrer des programmes qui divisent par zéro.

Contrôle à l'usine

Écrire **Fun** $(x : A)$ `->` t n'empêche pas les erreurs.

```
(Fun (x:Nat) -> x 1) 2
```

Mais cela permet de définir facilement un sous-ensemble des termes : les termes bien typés.

L'ensemble des termes bien typés est défini par une relation

$$E \vdash t : A$$

(Le terme t possède le type A dans l'environnement E.)

Dans le cas d'un environnement vide on abrège

$$t : A$$

(Le terme t a le type A.)

Exemples

```
Fun (x:Nat) -> x 1
```

Mal typé, car `x` de type **Nat** est appliqué (à `2`).

```
Fix (x:Nat) -> x + 1
```

Bien typé.

```
Fix (f:Nat) ->
  Fun (n:Nat) ->
    Ifz n Then 1 Else 2 * f (n-1)
```

Mal typé, car `f` de type **Nat** est appliqué (à `n-1`). On aurait pu écrire:

```
Fix (f:Nat -> Nat) ->
  Fun (n:Nat) ->
    Ifz n Then 1 Else 2 * f (n-1)
```

Définition inductive de $E \vdash t : A$

$$E \vdash \mathtt{n} : \mathbf{Nat} \qquad \frac{E(x) = A}{E \vdash x : A} \qquad \frac{E \vdash t_1 : \mathbf{Nat} \qquad E \vdash t_2 : \mathbf{Nat}}{E \vdash t_1 \ \ op \ \ t_2 : \mathbf{Nat}}$$

$$\frac{E \vdash t_1 : \mathbf{Nat} \qquad E \vdash t_2 : A \qquad E \vdash t_3 : A}{E \vdash \mathbf{Ifz}\ t_1\ \mathbf{Then}\ t_2\ \mathbf{Else}\ t_3 : A}$$

$$\frac{E \vdash t_1 : A \texttt{ -> } B \qquad E \vdash t_2 : A}{E \vdash t_1\ t_2 : B} \qquad \frac{E \oplus [x : A] \vdash t : B}{E \vdash (\mathbf{Fun}\ (x : A) \texttt{ -> } t) : A \texttt{ -> } B}$$

$$\frac{E \oplus [x : A] \vdash t : A}{E \vdash (\mathbf{Fix}\ (x : A) \texttt{ -> } t) : A}$$

$$\frac{E \vdash t_1 : A \qquad E \oplus [x : A] \vdash t_2 : B}{E \vdash \mathbf{Let}\ x \texttt{ = } t_1\ \mathbf{In}\ t_2 : B}$$

Typage de la puissance de deux

$$\dfrac{\dfrac{\dfrac{[\cdots;\texttt{n} : \mathbf{Nat}] \vdash \texttt{n} : \mathbf{Nat} \qquad E \vdash \texttt{1} : \mathbf{Nat} \qquad E \vdash t_2 : \mathbf{Nat}}{[\texttt{f} : \mathbf{Nat} \texttt{ -> } \mathbf{Nat}, \texttt{n} : \mathbf{Nat}] \vdash \mathbf{Ifz}\ \texttt{n}\ \mathbf{Then}\ \texttt{1}\ \mathbf{Else}\ t_2 : \mathbf{Nat}}}{[\texttt{f} : \mathbf{Nat} \texttt{ -> } \mathbf{Nat}] \vdash (\mathbf{Fun}\ (\texttt{x} : \mathbf{Nat}) \texttt{ -> } \mathbf{Ifz} \cdots) : \mathbf{Nat} \texttt{ -> } \mathbf{Nat}}}{\dfrac{\emptyset \vdash (\mathbf{Fix}\ (\texttt{f} : \mathbf{Nat} \texttt{ -> } \mathbf{Nat}) \texttt{ -> } t) : \mathbf{Nat} \texttt{ -> } \mathbf{Nat} \qquad \emptyset \vdash \texttt{100} : \mathbf{Nat}}{\emptyset \vdash (\mathbf{Fix}\ (\texttt{f} : \mathbf{Nat} \texttt{ -> } \mathbf{Nat}) \texttt{ -> } t)\ \texttt{100} : \mathbf{Nat}}}$$

Avec $E \vdash t_2 : \mathbf{Nat}$ prouvé comme suit :

$$\dfrac{E \vdash \texttt{2} : \mathbf{Nat} \qquad \dfrac{E \vdash \texttt{f} : \mathbf{Nat} \texttt{ -> } \mathbf{Nat} \qquad \dfrac{E \vdash \texttt{n} : \mathbf{Nat} \qquad E \vdash \texttt{1} : \mathbf{Nat}}{E \vdash \texttt{n-1} : \mathbf{Nat}}}{E \vdash \texttt{f (n-1)} : \mathbf{Nat}}}{[\texttt{f} : \mathbf{Nat} \texttt{ -> } \mathbf{Nat}, \texttt{n} : \mathbf{Nat}] \vdash \texttt{2 * f (n-1)} : \mathbf{Nat}}$$

N.B. Inutile de calculer 2^{100}.

La relation $E \vdash t : A$

Ressemble très fortement à une sémantique « à grand pas ».

Pour cela, on l'appelle :

- Sémantique *statique* (car valable au moment de la compilation).
- Interprétation *abstraite*, car elle est une *abstraction* de la sémantique, qui calcule des ensembles de valeurs et non plus des valeurs.

Calculer A tel que $t : A$ est surtout *vérifier* le type de t, car on ne découvre (infère) pas franchement A.

Décidabilité

Avec des règles aussi précises,

- On écrit facilement un programme qui étant donnés E et t trouve A tel que $E \vdash t : A$ ou échoue.
- Et de plus A est unique.
- On dit que la relation « $\emptyset \vdash t : A$ », notée « $t : A$ », est décidable et non-ambigüe.

C'est ce que nous ferons en TP.

```
open T
exception Error

val check : (Var.t * Type.t) list -> Ast.t -> Type.t
(* L'appel 'check E t' envoie A tel que E ⊢ t : A,
   ou lance Error si il n'existe pas de tel A *)
```

Correction du typage statique

On veut exprimer que si un terme est bien typé (à la compilation), alors il ne se produit pas d'erreurs (de type) lors de l'exécution.

- Sémantique à petit pas. Deux gros lemmes :
  - *subject reduction*, c-à-d, si $t : A$ et $t \to t'$, alors $t' : A$.
    Cela se montre par induction sur la dérivation $t : A$.
  - *progress*, c'-à-d, si $t : A$ et t n'est pas une valeur, alors t n'est pas en forme normale.
    Cela se montre en détaillant les termes bloqués et en montrant qu'ils ne sont pas typables.
  - Il vient ce **Théorème** *(Correction du typage statique)*, $t : A$ entraîne l'absence d'erreurs (de type) à l'exécution.

Correction, à grand pas

Une méthode pas satisfaisante, prouver:

$$(t : A \wedge t \hookrightarrow v) \Rightarrow v : A$$

Un peu court, car termes qui bouclent et erreurs de type sont confondus.

On ajoute une « valeur » wrong qui exprime une erreur de type, et des règles de propagation.

$$\frac{t_1 \hookrightarrow n}{t_1\ t_2 \hookrightarrow \mathsf{wrong}} \qquad \frac{t_1 \hookrightarrow \mathsf{wrong}}{t_1\ t_2 \hookrightarrow \mathsf{wrong}} \qquad \frac{t_2 \hookrightarrow \mathsf{wrong}}{t_1\ t_2 \hookrightarrow \mathsf{wrong}} \qquad \ldots$$

On obtient alors une formulation plus précise, qui entraîne:

$$(t : A \wedge t \hookrightarrow v) \Rightarrow v \neq \mathsf{wrong}$$

Une dernière réflexion

Le typage $A : t$ garantit l'absence d'erreurs de type à l'exécution.

La réciproque est bien entendu fausse.

```
(Ifz 0 Then 0 Else (Fun x -> x))+1
```

Cette *incomplétude* est un prix à payer pour le le typage statique.

Pourquoi ? À cause de l'indécidabilité de l'arrêt : il n'existe pas d'algorithme qui permet de décider si P (bien typé) termine.

Soit P bien typé (et donc exempt d'erreur) et :

```
(Ifz P Then 0 Else (Fun x -> x)) + 1
```

Un erreur se produit à l'exécution entraîne P termine (et P s'évalue n différent de zéro).

Un dernier thérorème

Si $t : A$ et que t ne contient pas **Fix**, alors toutes les réductions issues de t terminent (Tait).

Corollaire :

- Le terme `(`**Fun** `x -> x x) (`**Fun** `x -> x x)` n'est pas typable.
- Aucun codage du point fixe n'est typable. Disons qu'un codage du pt. fixe est un terme Y avec

$$Y\ t \rightarrow^* t\ (Y\ t)$$

On peut alors coder **Fix** f `->` t comme Y (**Fun** f `->` t).

Pour un exemple, cf poly (Exercice 2.10).

Sémantique dénotationelle

Maintenant que les termes contiennent des types, il devient possible d'associer une fonction des mathématiques à toute fonction PCF.

Commençons par associer (fonction $[\![\cdot]\!]$) un ensembles à chaque type de PCF.

$$[\![\mathbf{Nat}]\!] = \mathbb{N} \qquad\qquad [\![A \texttt{ -> } B]\!] = [\![A]\!] \to [\![B]\!]$$

(où $E_1 \to E_2$ est l'ensemble des fonctions de E_1 vers E_2).

On a alors six règles évidentes.

$$[\![\texttt{n}]\!] = n \qquad\qquad [\![t_1 \texttt{ op } t_2]\!] = [\![t_1]\!]\, op [\![t_2]\!]$$

$$[\![\textbf{Ifz } t_1 \textbf{ Then } t_2 \textbf{ Else } t_3]\!] =$$

$[\![t_2]\!]$ si $[\![t_1]\!]$ est 0, ou $[\![t_3]\!]$ si $[\![t_1]\!]$ est $n \neq 0$

Sémantique dénotationelle triviale

À $t : A$ on associe $[\![t]\!]$ dans $[\![A]\!]$. ($[\![t]\!]$ est une *valeur sémantique*, un elément de $\bigcup_{A \in \mathcal{T}} [\![A]\!]$).

La sémantique des termes ouverts est relative à un *environnment sémantique* E des variables vers les valeurs sémantiques,

- $[\![x]\!]_E = E(x)$
- $[\![\textbf{Let } x \texttt{ = } t_1 \textbf{ In } t_2]\!]_E = [\![t_2]\!]_{E \oplus [x = [\![t_1]\!]_E]}$
- $[\![\textbf{Fun } (x : A) \texttt{ -> } t]\!]_E$, la fonction qui à tout v de $[\![A]\!]$, associe $[\![t]\!]_{E \oplus [x=v]}$.
- $[\![t_1\ t_2]\!]_E = [\![t_1]\!]_E([\![t_2]\!]_E)$

Avec une définition à peine modifiée (qui explicite le type des valeurs sémantiques de l'environnement), on verrait que si $t : A$, alors $[\![t]\!]$ existe et $\in [\![A]\!]$, mais bon.

Frustration

Toute cette dépense théorique pour des paraphrases en italiques !

Mais, la sémantique triviale laisse de côté deux questions.

- Que vaut $[\![\texttt{1/0}]\!]$? Faux problème, cela vaut error, on décide $\mathsf{error} \in [\![A]\!]$ pour tout A, et on modifie les règles pour propager cette erreur.
- Et le point fixe ? Vrai problème.

  La tentation est grande de définir $[\![\mathbf{Fix}\ (x : A)\ \texttt{->}\ t]\!]$ comme point fixe de

  $$[\![\mathbf{Fun}\ (x : A)\ \texttt{->}\ t]\!]$$

  Mais cela ne va pas se faire tout seul.

Point fixe

En PCF on peut écrire le terme **Fix** (x:A) -> t (du moment que l'on a $[\mathtt{x} : A] \vdash t : A$).

On se se préocupe pas de l'existence du point fixe de [[**Fun** (x:A) -> t]]. Par exemple :

$$T_1 = \textbf{Fix}\ (\mathtt{x}\!:\!\textbf{Nat})\ \texttt{->}\ \mathtt{x+1}$$

Or, il n'existe pas d'entier x tel que $x = x + 1$.

Si on revient à la sémantique opérationnelle, on avait :

$$(\textbf{Fix}\ (\mathtt{x}\!:\!A)\ \texttt{->}\ t) \to [\textbf{Fix}\ (\mathtt{x}\!:\!A)\ \texttt{->}\ t\backslash x]t$$

Et donc :

$$T_1 \to T_1\mathtt{+1} \to (T_1\mathtt{+1})\mathtt{+1} \to \cdots$$

Pour compléter les valeurs sémantiques on donne la valeur $\perp$ à ce terme (terme de type **Nat**, qui ne termine pas).

Une vérification

On a donc posé $[\![\mathbf{Nat}]\!] = \mathbb{N} \cup \{\bot\}$.

Si on complète les quatres règles des opérations

$$[\![t_1 \ \texttt{op} \ t_2]\!] = \bot, \text{ si } [\![t_1]\!] \text{ ou } [\![t_2]\!] \text{ est } \bot$$

Alors $\bot$ est l'unique point fixe de [[**Fun** (`x:`**Nat**) `-> x+1`]]

De même, [[**Fun** (`x:`**Nat**) `-> x`]] (function qui à x entier associe x) avait plusieurs point fixes $0, 1, 2, \ldots$

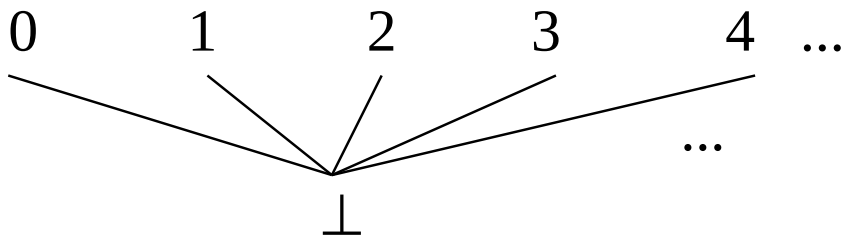
Maintenant, cette fonction (qui à $x \in [\![\mathbf{Nat}]\!]$ associe x) a un pt. fixe de plus $\bot$.

Et c'est celui là que l'on veut pour [[**Fix** (`x:`**Nat**) `-> x`]] (termine pas).

Points fixes de [[**Fun** (`x:`**Nat**) `-> x+x`]] ? $\bot$ et 0. Valeur de [[**Fix** (`x:`**Nat**) `-> x+x`]] ? $\bot$

Comment choisir son point fixe

Ordre de Scott sur le domaine [[**Nat**]].



Ordre faiblement complet ? Oui !

On veut le plus petit point fixe.

Il existe bien pour toutes les fonctions croissantes et continues de [[**Nat**]] dans [[**Nat**]].

Ce qui est le cas de [[**Fun** (`x:`**Nat**) `-> x+1`]] etc.

Et dans le cas général...

Définition revue des domaines sémantiques :

- $[\![\mathbf{Nat}]\!] = \mathbb{N} \cup \{ \bot \}$
- $[\![A \texttt{ -> } B]\!]$ fonctions *croissantes et continues* de $[\![A]\!]$ dans $[\![B]\!]$.

L'ordre sur $[\![A \texttt{ -> } B]\!]$ est défini point à point, $f \leq g$ sisi
$\forall x \in [\![A]\!], f(x) \leq g(x)$.

Il y a un plus petit elt : la fonction qui à $x \in [\![A]\!]$ associe $\bot_B$
(notée $\bot_{A \texttt{ -> } B}$).

Cette relation est faiblement complète (permutation de limites).

Tous les termes de type A s'interprètent bien dans $[\![A]\!]$ (continuité).

La valeur de **Fix** $(x : A)$ -> t est *définie* comme le plus petit point fixe de $[\![\mathbf{Fun}\ (x : A) \texttt{ -> } t]\!]$.

Qu'est-ce que ça veut dire en pratique ?

Fonction puisssance de deux :

```
Let P =
  Fix (p:Nat -> Nat) ->
    Fun (n:Nat) ->
      Ifz n Then 1 Else 2 * p (n-1)
```

Noté **Fix** (p:**Nat** -> **Nat**) -> t.

Pour avoir la valeur du terme ci-dessus, on calcule le pppf de
$\Phi = [\![\textbf{Fun}\ (\texttt{p:}\textbf{Nat}) \texttt{ -> } t]\!]$.

Le plus pppf de Φ est la limite de la suite
$(a_k) = (\Phi^k(\bot_{\textbf{Nat -> Nat}}))$.

$$
\begin{array}{lllllllll}
a_0 & \bot & \bot & \bot & \bot & \bot & \bot & \bot & \bot \cdots \\
a_1 & \bot & 1 & \bot & \bot & \bot & \bot & \bot & \bot \cdots \\
a_2 & \bot & 1 & 2 & \bot & \bot & \bot & \bot & \bot \cdots \\
a_3 & \bot & 1 & 2 & 4 & \bot & \bot & \bot & \bot \cdots \\
a_4 & \bot & 1 & 2 & 4 & 8 & \bot & \bot & \bot \cdots \\
a_5 & \bot & 1 & 2 & 4 & 8 & 16 & \bot & \bot \cdots \\
a_6 & \bot & 1 & 2 & 4 & 8 & 16 & 32 & \bot \cdots
\end{array}
$$

Car par exemple : $a_6(5) = 2 * a_5(4)$ (en sautant quelques étapes).

Mise en relation des sémantiques

Il y a un théorème (pas facile) :

$$[\![t]\!] = n \Leftrightarrow t \hookrightarrow n$$

Ce théorème s'applique-t-il à l'évaluation en appel par nom ou par valeur ?

Considérer :

```
Let k = Fun (x:Nat) -> Fun (y:Nat) -> y In
Let loop = Fix (n:Nat) -> n In
k 0 loop
```

Cela vaut $[\![\mathtt{y}]\!]_{[\mathtt{loop}=\bot,\mathtt{y}=0]}$ qui vaut 0.

Donc c'est appel par nom.

À quoi ça sert ?

C'est beau de savoir que l'on programme de vraies fonctions.

À prouver des théorèmes. Par ex.

> On ne peut pas programmer en PCF typé, la fonction g qui prend f : **Nat -> Nat** en argument, et renvoie 0 si f 0 vaut zéro et 1 autrement.

Soit donc g de $[\![$**(Nat -> Nat) -> Nat**$]\!]$ telle que ci-dessus.

Considérer $f_0 = [\![$**Fun x:Nat -> 0**$]\!]$, $f_1 = [\![$**Fun x:Nat -> 1**$]\!]$, On $\bot_{\textbf{Nat -> Nat}} \leq f_0$ et $\bot_{\textbf{Nat -> Nat}} \leq f_1$. Soit ($g$ croissante) $g(\bot_{\textbf{Nat -> Nat}}) = \bot$ C.Q.F.D.

Bon on joue sur les mots, voir Exercice 5.11 pour un exemple plus sérieux.

- TP, vérification de type.
- La prochaine fois, inférence de type.