Département d'informatique et de recherche opérationnelle (DIRO)
IFT2035 — Concepts des langages de programmation (automne 2025)
Auteur: *Mattéo Delabre* · Dernière mise à jour: *2025-12-12*

Devoir 3 — *Vérificateur de types*

Instructions Ce devoir doit être remis sur Studium **en équipe de deux** avant *le vendredi 19 décembre à 23 h 59*. Vous devez inscrire votre équipe sur Studium avant de pouvoir remettre votre devoir.

Intégrité Il n'est pas autorisé de copier du contenu de sources extérieures, y compris des devoirs d'autres équipes, sans les mettre entre guillemets ou sans en citer l'origine. L'utilisation de modèles de langue tel ChatGPT est autorisée seulement pour l'écriture d'une partie du code. Le rapport doit être entièrement écrit par votre équipe. L'utilisation de ces modèles est toutefois autorisée pour effectuer une révision linguistique du rapport. Au moins 50% de votre code doit avoir été écrit par votre équipe.

Pointage Le devoir vaut pour 5% de la note du cours.

Votre objectif pour ce devoir est d'utiliser le langage **Prolog** pour réaliser un **vérificateur de types** d'un langage fonctionnel à typage polymorphe nommé *Girard*, inventé pour ce devoir. Ce vérificateur s'intègrera dans un interprète pour le langage en question. Ce langage est proche du **Système F**, que nous avons étudié en classe.

TÂCHE Téléchargez le fichier `girard.pl` joint à ce devoir sur Studium. Une fois le devoir complété, vous devrez remettre:

- le fichier `girard.pl` complété en suivant les instructions ci-dessous;
- un rapport au format PDF d'un maximum de 3 pages expliquant votre démarche.

Ne changez pas les noms ni l'arité des prédicats déjà définis dans le fichier. Vous pouvez en renommer les paramètres ou ajouter de nouveaux prédicats au besoin.

Pour tester votre travail au fur et à mesure que vous progressez, vous pouvez lancer le toplevel Prolog `scryer-prolog` et utiliser la commande `['girard'].` pour charger les prédicats que vous avez définis dans le fichier du même nom.

Types

Dans le langage Girard, les **types** peuvent avoir l'une des formes suivantes:

- une **variable** de type, par exemple `x`, `var` ou `β`;
- une **flèche** entre deux expressions de types, par exemple `α -> (α -> α) -> α`;
- un **quantificateur** universel suivi d'une expression de type, comme `forall α . α`.

Un type est fermé s'il ne contient aucune variable de type libre. Il est ouvert dans le cas contraire. Ainsi, `forall α . α -> α` est fermé et pas `forall α . α -> β`.

Dans le programme Prolog, les types sont représentées par leur arbre de syntaxe abstraite, sous forme de termes composés:

- la variable `x` correspond à **`var("x")`**;
- la flèche `α -> α` correspond à **`arrow(var("α"), var("α"))`**;
- le quantificateur universel `forall α . α` correspond à **`forall("α", var("α"))`**.

Le prédicat `type_ast(Ast, Expr, [])`, déjà fourni dans la donnée de ce devoir, permet de convertir un type entre sa forme d'arbre de syntaxe abstraite (`Ast`) et sa forme de chaîne de caractères (`Expr`).

Sous forme de règles, les constructions permises pour les types sont:

$$\frac{\Delta \vdash \alpha : \text{type} \qquad \Delta \vdash \beta : \text{type}}{\Delta \vdash \text{arrow}(\alpha, \beta) : \text{type}}\,\text{Arrow} \qquad \frac{\Delta, \alpha : \text{type} \vdash \beta : \text{type}}{\Delta \vdash \text{forall}(\alpha, \beta) : \text{type}}\,\text{Forall}$$

$$\frac{\tau \in \Delta}{\Delta \vdash \text{var}(\tau) : \text{type}}\,\text{TVar}$$

> TÂCHE Implantez les prédicats **`type_freevars(Type, Vars)`**, qui relie un type `Type` à sa liste de variables libres `Vars`, et **`env_type(TypeVarEnv, Type)`**, qui vérifie si un type `Type` peut être construit en suivant les règles ci-dessus, où `TypeVarEnv` correspond au contexte Δ. *(Vous pourriez avoir besoin des prédicats `member/2` de la librairie `lists` et `ord_union/3` et `ord_subtract/3` de la librairie `ordsets`.)*

Les types polymorphes du langage se comportent comme des λ-expressions à l'échelle des types. Ainsi, il est possible de spécialiser un type comme `forall α . α -> α` en retirant le quantificateur `forall α .` et en remplaçant dans `α -> α` toutes les occurrences de `α` par un autre type.

Lors de ce remplacement, il faut toutefois prendre garde à ne pas capturer des variables; ainsi la substitution de `β` par `α` dans `forall α . α -> β` devrait donner `forall α' . α' -> α` et non pas `forall α . α -> α`.

> TÂCHE Implantez le prédicat **`type_subst(Type, Search, Replace, Subst)`**, qui substitue dans le type `Type` les occurrences de la variable de type nommée `Search` par le type `Replace` pour donner le résultat `Subst`. *(Vous pouvez utiliser le prédicat fourni* `name_fresh_vars/3` *pour générer des noms de variables frais.)*

Comme les λ-expressions, la variable de type dans un type quantifié universellement peut changer de nom par α-renommage sans changer la signification du type; ainsi, le type `forall β . β -> β` est équivalent au type `forall α . α -> α`. Formellement, le fait que deux types τ et τ' soient équivalents est noté par $\tau \sim \tau'$.

> TÂCHE Implantez le prédicat **`type_equiv(Type1, Type2)`**, qui vérifie si deux types sont équivalents.

Valeurs

Dans le langage Girard, les **valeurs** peuvent avoir l'une des formes suivantes:

- une **variable** de valeur, par exemple `x`, `var` ou `β`;
- une **λ-expression**, par exemple `lambda x : α . e`, où `α` est un type et `e` une valeur;
- une **application**, par exemple `l r`, où `l` et `r` sont des valeurs;
- un expression **polymorphe**, par exemple `poly a . e`, où `a` est une variable de type qui peut être utilisée à l'intérieur de la valeur `e`;
- une **spécialisation**, par exemple `f [t]`, où `f` est une valeur et `t` un type.

Il faut faire attention à ne pas mélanger les types avec les valeurs, qui ont deux syntaxes différentes et dont les variables sont indépendantes. Par exemple, dans la valeur `poly α . lambda α : α . α`, le nom `α` est utilisé à la fois pour une variable de type et pour une variable de valeur qui a pour type `α`.

Dans le programme Prolog, les valeurs sont représentées par leur arbre de syntaxe abstraite, sous forme de termes composés:

- la variable `x` correspond à **`var("x")`**;
- la λ-expression `lambda x : α . e` correspond à **`lambda("x", var("α"), var("e"))`**;
- l'application `l r` correspond à **`apply(var("l"), var("r"))`**;

- l'expression polymorphe `poly a . e` correspond à **`poly("a", var("e"))`**;
- la spécialisation `f [t]` correspond à **`spec(var("f"), var("t"))`**.

Le prédicat `expr_ast(Ast, Expr, [])`, déjà fourni dans la donnée de ce devoir, permet de convertir une valeur entre sa forme d'arbre de syntaxe abstraite (`Ast`) et sa forme de chaîne de caractères (`Expr`).

Vérification des types

Les règles de typage du langage Girard sont identiques à celles du Système F:

$$\frac{\Delta \mid \Gamma \vdash e_1 : \mathrm{arrow}(\alpha, \beta) \qquad \Delta \mid \Gamma \vdash e_2 : \alpha' \qquad \alpha \sim \alpha'}{\Delta \mid \Gamma \vdash \mathrm{apply}(e_1, e_2) : \beta}\,\mathrm{App}$$

$$\frac{\Delta \mid \Gamma, x : \alpha \vdash e : \beta \qquad \Delta \vdash \alpha : \mathrm{type}}{\Delta \mid \Gamma \vdash \mathrm{lambda}(x, \alpha, e) : \mathrm{arrow}(\alpha, \beta)}\,\mathrm{Abs}$$

$$\frac{\Delta \mid \Gamma \vdash e : \mathrm{forall}(\alpha, \beta) \qquad \Delta \vdash \tau : \mathrm{type}}{\Delta \mid \Gamma \vdash \mathrm{spec}(e, \tau) : \beta[\alpha := \tau]}\,\mathrm{Spec}$$

$$\frac{\Delta, \alpha : \mathrm{type} \mid \Gamma \vdash e : \beta}{\Delta \mid \Gamma \vdash \mathrm{poly}(\alpha, e) : \mathrm{forall}(\alpha, \beta)}\,\mathrm{Poly} \qquad \frac{x : \tau \in \Gamma}{\Delta \mid \Gamma \vdash \mathrm{var}(x) : \tau}\,\mathrm{EVar}$$

> TÂCHE
> Implantez le prédicat **`env_expr_type(TypeVarEnv, TypeEnv, Expr, Type)`** qui relie l'expression de valeur `Expr` à son type `Type`, dans l'environnement de types `TypeVarEnv` (Δ) et l'environnement de valeurs `TypeEnv` (Γ).

Évaluation

Les règles d'évaluation du langage Girard sont similaires à celles du λ-calcul. L'évaluation se fait en ignorant les indications de types, puisque les types sont supposés avoir été vérifiés avant de commencer l'évaluation.

$$\frac{\Theta \vdash e_1 \Rightarrow \mathrm{lambda}(x, \tau, f) \qquad \Theta \vdash f[x := e_2] \Rightarrow e_3}{\Theta \vdash \mathrm{apply}(e_1, e_2) \Rightarrow e_3}\,\mathrm{VApp}_1$$

$$\frac{\Theta \vdash e_1 \Rightarrow f_1 \qquad \Theta \vdash e_2 \Rightarrow f_2 \qquad f_1 \neq \mathrm{lambda}(x, \tau, f)}{\Theta \vdash \mathrm{apply}(e_1, e_2) \Rightarrow \mathrm{apply}(f_1, f_2)}\,\mathrm{VApp}_2$$

$$\frac{\Theta \vdash e \Rightarrow f}{\Theta \vdash \mathrm{lambda}(x, \tau, e) \Rightarrow \mathrm{lambda}(x, \tau, f)}\,\mathrm{VAbs} \qquad \frac{}{\Theta \vdash \mathrm{spec}(e, \tau) \Rightarrow e}\,\mathrm{VSpec}$$

$$\frac{}{\Theta \vdash \mathrm{poly}(\alpha, e) \Rightarrow e}\,\mathrm{VPoly}$$

$$\frac{(x = e) \in \Theta}{\Theta \vdash \mathrm{var}(x) \Rightarrow e}\,\mathrm{VVar}_1 \qquad \frac{(x = e) \notin \Theta}{\Theta \vdash \mathrm{var}(x) \Rightarrow \mathrm{var}(x)}\,\mathrm{VVar}_1$$

TÂCHE Implantez les prédicats **`expr_freevars(Expr, Vars)`** et **`expr_subst(Expr, Search, Replace, Subst)`**, similaires aux prédicats `type_freevars/2` et `type_subst/4` mais fonctionnant sur les valeurs au lieu des types. Le prédicat **`env_expr_reduce(ValueEnv, Expr, Value)`**, qui relie une valeur `Expr` à sa valeur réduite `Value` dans l'environnement de valeurs (Θ), est déjà fourni dans la donnée du devoir.

Toplevel

Une fois les tâches précédentes complétées, vous obtiendrez un **toplevel** pour le langage Girard en lançant la commande `scryer-prolog girard.pl -g main` (ou la commande `rlwrap scryer-prolog girard.pl -g main` pour pouvoir éditer l'entrée avec les raccourcis habituels).

Dans ce toplevel, vous pouvez saisir une expression du langage. Celle-ci sera convertie en arbre de syntaxe abstraite (via le prédicat `expr_ast/3`), puis typée (via le prédicat `env_expr_type/4`) et enfin évaluée (via le prédicat `env_expr_reduce/3`). S'il n'y a pas d'erreur lors de ces trois étapes, le type et la valeur de l'expression sont affichées.

Vous pouvez également saisir une expression de la forme `Nom = Expr`. Si l'expresion `Expr` est syntaxiquement et sémantiquement valide, elle devient associée à la variable de valeur appelée `Nom` pour la suite de la session. Dans les expressions suivantes, `Nom` est remplacée automatiquement par sa définition.

Exemples

Voici un exemple de session dans le toplevel. Assurez-vous que tous les exemples suivants fonctionnent, et essayez d'en construire d'autres!

▷ Booléens

```
> true = poly α . lambda x : α . lambda y : α . x
  Type: forall α.α->α->α
Valeur: lambda x:α.lambda y:α.x
```

```
> false = poly α . lambda x : α . lambda y : α . y
  Type: forall α.α->α->α
Valeur: lambda x:α.lambda y:α.y

> not = lambda x : forall α . α->α->α . x [forall α.α->α->α] false true
  Type: (forall α.α->α->α)->forall α.α->α->α
Valeur: lambda x:forall α.α->α->α.x (lambda x:α.lambda y:α.y)
        (lambda x:α.lambda y:α.x)

> not true
  Type: forall α.α->α->α
Valeur: lambda x:α.lambda y:α.y

> not false
  Type: forall α.α->α->α
Valeur: lambda x:α.lambda y:α.x

> not not
  Type: [Erreur]
```

▷ Entiers

```
> 0 = poly α . lambda x : α . lambda f : α -> α . x
  Type: forall α.α->(α->α)->α
Valeur: lambda x:α.lambda f:α->α.x

> 1 = poly α . lambda x : α . lambda f : α -> α . f x
  Type: forall α.α->(α->α)->α
Valeur: lambda x:α.lambda f:α->α.f x

> succ = lambda n : forall α.α->(α->α)->α . poly α . lambda x : α .
         lambda f : α -> α . f (n [α] x f)
  Type: (forall α.α->(α->α)->α)->forall α.α->(α->α)->α
Valeur: lambda n:forall α.α->(α->α)->α.lambda x:α.lambda f:α->α.f (n x f)

> succ 0
  Type: forall α.α->(α->α)->α
Valeur: lambda x:α.lambda f:α->α.f x

> succ 1
  Type: forall α.α->(α->α)->α
Valeur: lambda x:α.lambda f:α->α.f (f x)

> succ true
  Type: [Erreur]

> 5 = succ (succ (succ (succ 0))))
  Type: forall α.α->(α->α)->α
Valeur: lambda x:α.lambda f:α->α.f (f (f (f (f x))))

> isZero = lambda n : forall α.α->(α->α)->α . n [forall α.α->α->α]
           true (lambda x : forall α.α->α->α . false)
```

```
  Type: (forall α.α->(α->α)->α)->forall α.α->α->α
Valeur: lambda n:forall α.α->(α->α)->α.n (lambda x:α.lambda y:α.x)
        (lambda x:forall α.α->α->α.lambda x:α.lambda y:α.y)

> isZero 0
  Type: forall α.α->α->α
Valeur: lambda x:α.lambda y:α.x

> isZero 1
  Type: forall α.α->α->α
Valeur: lambda x:α.lambda y:α.y

> isZero 5
  Type: forall α.α->α->α
Valeur: lambda x:α.lambda y:α.y

> isZero true
  Type: [Erreur]
```

▷ Listes

```
> nil = poly τ . poly α . lambda x : α . lambda s : τ -> α -> α . x
  Type: forall τ.forall α.α->(τ->α->α)->α
Valeur: lambda x:α.lambda s:τ->α->α.x

> cons = poly τ . lambda t : τ . lambda l : forall α . α -> (τ -> α -> α) -> α .
         poly α . lambda x : α . lambda s : τ -> α -> α . s t (l [α] x s)
  Type: forall τ.τ->(forall α.α->(τ->α->α)->α)->forall α.α->(τ->α->α)->α
Valeur: lambda t:τ.lambda l:forall α.α->(τ->α->α)->α.lambda x:α.
        lambda s:τ->α->α.s t (l x s)

> ex1 = cons [forall α.α->(α->α)->α] 0 (nil [forall α.α->(α->α)->α])
  Type: forall α.α->((forall α.α->(α->α)->α)->α->α)->α
Valeur: lambda x:α.lambda s:τ->α->α.s (lambda x:α.lambda f:α->α.x) x

> ex2 = cons [forall α.α->(α->α)->α] 0
         (cons [forall α.α->(α->α)->α] 1 (nil [forall α.α->(α->α)->α]))
  Type: forall α.α->((forall α.α->(α->α)->α)->α->α)->α
Valeur: lambda x:α.lambda s:τ->α->α.s (lambda x:α.lambda f:α->α.x)
        (s (lambda x:α.lambda f:α->α.f x) x)

> ex3 = cons [forall α.α->(α->α)->α] 0
         (cons [forall α.α->(α->α)->α] 1
          (cons [forall α.α->(α->α)->α] 5 (nil [forall α.α->(α->α)->α])))
  Type: forall α.α->((forall α.α->(α->α)->α)->α->α)->α
Valeur: lambda x:α.lambda s:τ->α->α.s (lambda x:α.lambda f:α->α.x)
        (s (lambda x:α.lambda f:α->α.f x)
         (s (lambda x:α.lambda f:α->α.f (f (f (f (f x))))) x))

> len = poly τ . lambda l : forall α . α -> (τ -> α -> α) -> α .
        l [forall α.α->(α->α)->α] 0
        (lambda t : τ . lambda l : forall α . α -> (α -> α) -> α . succ l)
  Type: forall τ.(forall α.α->(τ->α->α)->α)->forall α.α->(α->α)->α
```

```
Valeur: lambda l:forall α.α->(τ->α->α)->α.l (lambda x:α.lambda f:α->α.x)
        (lambda t:τ.lambda l:forall α.α->(α->α)->α.lambda x:α.lambda f:α->α.f
         (l x f))

> len [forall α.α->(α->α)->α] ex1
  Type: forall α.α->(α->α)->α
Valeur: lambda x:α.lambda f:α->α.f x

> len [forall α.α->(α->α)->α] ex2
  Type: forall α.α->(α->α)->α
Valeur: lambda x:α.lambda f:α->α.f (f x)

> len [forall α.α->(α->α)->α] ex3
  Type: forall α.α->(α->α)->α
Valeur: lambda x:α.lambda f:α->α.f (f (f x))

> len 0
  Type: [Erreur]

> len true
  Type: [Erreur]
```

Critères d'évaluation

Votre travail est évalué sur les critères suivants:

- **3 points** sont attribués sur le code remis. Il est évalué à l'aide de tests automatiques. Le critère le plus important est que votre code fonctionne correctement. L'exécution de votre code ne devrait générer aucun avertissement ou erreur. La qualité du code est également évaluée : votre code devrait être concis et clair et inclure des commentaires au besoin. Sachez que le devoir peut être résolu en ajoutant ~100 lignes au fichier.
- **2 points** seront attribués sur le rapport remis. Votre rapport, d'une longueur maximale de 3 pages, devrait faire état de votre démarche dans la résolution du devoir. Expliquez-y les choix que vous avez faits lors de l'écriture du code. Si applicable, détaillez-y votre utilisation de modèles de langue. Montrez-y les tests que vous avez réalisés pour vérifier que votre code est correct.