Evaluateur-Typeur de Lambda-Calcul

MU5IN555 TAS - Session 2024

1 Préliminaires

L'objectif de ce projet est d'écrire, dans un langage libre, un typeur et un évaluateur pour un λ-calcul contenant différentes fonctionnalités.

1.1 Langage

Le choix du langage est libre, mais:

- les langages **fonctionnels** avec *pattern-matching* (comme `Haskell` ou `OCaml`) sont fortement conseillés. Leur utilisation facilite grandement la rédaction de fonction "par cas" et le travail sur l'arbre syntaxique des termes ou des types. **Attention :** L'implémentation du polymorphisme faible demande l'utilisation d'états (par exemple, de référence) pendant l'inférence, ce qui nécessite un niveau de maitrîse de Haskell "correct" pour prendre en compte cette fonctionnalité.
- les langages avec des fonctionnalités **objet** (comme `Python` ou `Java`) sont plus compliqués à utiliser dans ce cadre, mais l'héritage et les tests d'instance peuvent être utilisés pour faciliter l'écriture de fonction récursive sur les arbres syntaxiques.
- les langages de **bas-niveau** (comme `C` ou `Rust`) sont assez fastidieux à utiliser pour cette tâche, car ils nécessitent de manipuler des structures et des enchainements pointeurs pour parcourir les arbres syntaxiques.
- `Prolog` est interdit. Ce langage contient un algorithme de résolution qui rend l'écriture d'une fonction d'unification superflue.

1.2 Consignes

- Le projet est à rendre individuellement, sous forme d'un lien Gitlab donné comme réponse au Devoir Moodle correspondant.
- Ajouter `@RomainDemangeon` comme *Maintainer* au projet Gitlab. (cette tâche et la précédente sont obligatoires toutes les deux)
- La *deadline* est indiquée dans Moodle.
- Le plagiat sera sévèrement sanctionné : indiquer les sources du code si nécessaire ; si une partie du code a été travaillée en groupe, indiquer les autres membres du groupe ; indiquer quand une I.A. générative a été utilisée (donner des exemples de *prompts* dans le rapport) ; dans tous les cas, la rédaction doit être personnelle.

1.3 Rapport

Un rapport (1 ou 2 pages) doit apparaître à la racine du repositoire. Il doit contenir un résumé des parties qui ont été traitées, la mise en évidence de points forts et de points faibles du projet et une description des difficultés rencontrées.

2 Evaluateur pour un λ calcul pur

On rappelle la syntaxe de λ:

$$M, N ::= x \ \mid \ \lambda x.M \ \mid \ M \ N$$

1. Définir un type somme ou une classe permettant de représenter les λ-termes sous forme d'AST (avec 3 noeuds différents, correspondant aux 3 constructeurs différents).

   Par exemple :

   ```
   type pterm = Var of string
              | App of pterm * pterm
              | Abs of string * pterm
   ```

2. Ecrire un *pretty printer* de termes[1] permettant de convertir les termes en chaînes de caractères lisibles.

   Par exemple :

   ```
   let rec print_term (t : pterm) : string =
     match t with
       Var x -> x
       | App (t1, t2) -> "(" ^ (print_term t1) ^" "^ (print_term t2) ^ ")"
       | Abs (x, t) -> "(fun "^ x ^" -> " ^ (print_term t) ^")"
   ```

3. Ecrire une fonction *d'alpha-conversion* qui renomme toutes les variables liées d'un terme par des nouvelles variables (ce qui nécessite probablement d'avoir une fonction pour générer des nouvelles variables, et un compteur global)

   Par exemple :

   ```
   let compteur_var : int ref = ref 0

   let nouvelle_var () : string = compteur_var := !compteur_var + 1;
     "X"^(string_of_int !compteur_var)

   let rec alphaconv (t : pterm) : pterm =
       ...
   ```

4. Ecrire une fonction *de substitution* qui remplace toutes les occurences libres d'une variable donnée par un terme.

   Par exemple :

   ```
   let rec substitution (x : string) (n: pterm) (t : pterm) : pterm =
       ...
   ```

5. Ecrire une fonction *de réduction Call-by-Value* qui implémente la stratégie LtR-CbV vue en cours/TD, en utilisant la fonction précédente.

   La fonction prend n'importe quel terme du λ-calcul pur et renvoie le terme obtenu après une étape de réduction LtR-CbV. On doit implémenter un moyen d'exprimer le fait que le terme ne peut pas se réduire.

   Par exemple :

   ```
   let rec ltr-ctb-step (t : pterm) : pterm option =
       ...
   ```

[1] ou une méthode *toString* des termes, dans le cas d'un langage objet

6. Ecrire une fonction *de normalisation* qui prend un terme, et le réduit jusqu'à obtenir sa forme normale en utilisant la stratégie LtT-CbV, ou boucle infiniment si la stratégie LtR-CbV diverge pour ce terme.

   Par exemple :

   ```
   let rec ltr-cbv-norm (t : pterm) : pterm =
       ...
   ```

   Ecrire ensuite une version de cette fonction qui prend en compte la divergence (par exemple avec un *timeout*).

7. Définir une série d'exemples de termes contenant, entre autres, I, δ, Ω, S, $S\ K\ K$, $S\ I\ I$, les encodages de 0, 1, 2, 3 et les encodages des opérations arithmétiques usuelles.

   Tester la fonction de normalisation sur les exemples.

3 Types simples pour le λ-calcul

On rappelle que la syntaxe des types simples est donnée par:

$$T ::= v \ \mid \ T \to T$$

1. Définir un type somme ou une classe permettant de représenter les types simples sous forme d'AST.
   Par exemple :

   ```
   type ptype = Var of string | Arr of ptype * ptype | Nat
   ```

2. Ecrire un *prettyprinter* pour les types
   Par exemple :

   ```
   let rec print_type (t : ptype) : string =
     match t with
       Var x -> x
     | Arr (t1, t2) -> "(" ^ (print_type t1) ^" -> "^ (print_type t2) ^")"
   ```

3. Donner un type (une classe) pour les équations de typage, puis écrire une fonction qui génère des équations de typage à partir d'un terme. Cette fonction parcourt récursivement un terme, munie d'un environnement et d'un type cible T, en procédant ainsi:
   - Si le terme est une variable, elle trouve son type Tv dans l'environnement et génère l'équation $Tv = T$.
   - Si le terme est une abstraction, elle prend deux variables de type fraiches T_a et T_r, génère l'équation $T = T_a \to T_r$ puis génère récursivement les équations du corps de la fonction avec comme cible le type T_r et en rajoutant dans l'environnement que la variable liée par l'abstraction est de type T_a.
   - Si le terme est une application, elle prend une variable de type fraîche T_a, puis génère récursivement les équations du terme en position de fonction, avec le type cible $T_a \to T$, et les équations du terme en position d'argument avec le type cible T_a, en gardant le même environnement dans les deux cas.

   On aura besoin d'un type (une classe) pour les environnement, d'une fonction de recherche dans l'environnement et d'un générateur de variables de types (comme pour les termes).
   Par exemple,

   ```
   let compteur_var_t : int ref = ref 0

   let nouvelle_var_t () : string = compteur_var := !compteur_var + 1;
     "T"^(string_of_int !compteur_var)

   type equa = (ptype * ptype) list
   ```

```
type env = (string * ptype) list

let rec cherche_type (v : string) (e : env) : ptype =
    ...

let rec genere_equa (te : pterm) (ty : ptype) (e : env) : equa =
  match te with
    Var v -> ...
  | App (t1, t2) -> ...
  | Abs (x, t) -> ...
```

4. Ecrire une fonction d'*occur check* qui vérifie si une variable appartient à un type.
5. Ecrire une fonction qui substitue une variable de type par un type à l'intérieur d'un autre type, puis une fonction qui substitue une variable de type par un type partout dans un système d'équation.
6. Ecrire une fonction pour réaliser une étape d'unification dans les systèmes d'équations de typage selon un algo d'unification "simple", par exemple:
   - Si les deux types de l'équation sont égaux, on supprime l'équation
   - Si un des deux types est une variable X, qu'elle n'apparait pas dans l'autre type T_d, on supprime l'équation $X = T_d$ et on remplace X par T_d dans toutes les autres équations.
   - Si les deux types sont des types flèche $T_{ga} \to T_{gr} = T_{da} \to T_{dr}$, on supprime l'équation et on ajoute les équations $T_{ga} = T_{da}$ et $T_{gr} = T_{dr}$
   - Sinon on échoue.
7. Ecrire une fonction qui résout un système d'équation (avec un *timeout*) puis une fonction qui infère le type (ou la non-typabilité) d'un terme.
8. Tester extensivement le processus de typage.

4 Un λ-calcul enrichi et polymorphe

4.1 Evaluateur

On ajoute à la syntaxe:
- des entiers natifs (pas les entiers de Church) avec les opérateurs addition et soustraction.
- des listes natives (i.e. des séquences ordonnées d'éléments) avec les opérateurs tête, queue et cons.
- deux opérateurs *if zero then else* et *if empty then else* permettant de faire des branchements et testant respectivement les entiers et les listes.
- un opérateur de point fixe *fix*, permettant de définir récursivement des fonctions.
- un `let x = e1 in e2` natif.

1. Mettre à jour la syntaxe des termes, le *prettyprinter* et les constructeurs pour inclure les nouvelles fonctionnalités.
2. Mettre à jour l'évaluateur :
   - pour évaluer `let x = e1 in e2` on évalue d'abord `e1` en `v`, puis on remplace `x` par `v` dans `e2` et on évalue `e2`
   - pour évaluer `fix (phi -> M)`, on remplace `phi` par `fix (phi -> M)` partout dans `M`.
   - on ne peut pas traiter `izte` et `iete` (les branchements) comme des opérateurs "normaux" (contrairement aux autres), parce qu'on veut bloquer l'exécution du conséquent et de l'alternant tant que le branchement n'est pas réduit (ce qui n'est pas possible en *ltr-CbV* si ce sont des opérateurs "normaux"), il faut donc les représenter par des constructeur de termes.
3. Tester extensivement le processus d'évaluation, entre autres avec des fonctions polymorphes, et la fonction factorielle (sur les entiers natifs).

4.2 Types

On ajoute aux types :

- un type **N** pour les entiers,
- un constructeur de type $[T]$ pour les listes.
- un constructeur de type $\forall X.T$ pour gérer le let-polymorphisme.

1. Mettre à jour la syntaxe des types, le *prettyprinter* et les constructeurs.
2. Mettre à jour la génération d'équations :
   - pour traiter un opérateur (par exemple +, ou *hd*) dans la génération d'équation pour le type cible T, on égalise T et le type de l'opérateur (respectivement $T = \mathbf{N} \to \mathbf{N} \to \mathbf{N}$, ou $T = \forall X.[X] \to X$)
   - pour traiter un entier dans la génération d'équation pour le type cible T, on génère $T = \mathbf{N}$
   - pour traiter un branchement dans la génération d'équation pour le type cible T, on génère les équations de la condition (avec pour type cible, le type **N** ou $\forall X.[X]$ en fonction du type de branchement), les équations du conséquent avec la cible T et les équations de l'alternant avec la cible T.
   - pour traiter un `let x = e1 in e2` dans la génération d'équation pour le type cible T, on type (il faut ici utiliser la fonction de typage, ce qui induit une récusion mutuelle entre la génération d'équation, l'unification et le typage) `e1`, on récupère son type `T0` (si le typage n'échoue pas) et on génère les équations pour `e2` en ajoutant à l'environnement que `x` a le type $\forall \texttt{X1}, ..., \texttt{X}_\texttt{k}.\texttt{T0}$ (généralisation du type `T0`).
3. Ecrire une fonction qui généralise un type à partir d'un environnement e, c'est-à-dire qui ajoute un $\forall X.$ autour du type pour chacune de ses variables libres qui n'est pas dans l'environnement e.
4. Les modifications à apporter à l'unifications sont les suivantes:
   - quand un des deux type d'une équation est un $\forall$, on lui applique une "barendregtisation" (on renomme ses variables de type liées) et on "l'ouvre" (on garde la même équation, mais sans le $\forall$)
   - quand les deux termes sont des listes, on égalise le type des éléments.
   - si les constructeurs de types (à l'exception du $\forall$) ne sont pas les mêmes à gauche et à droite d'une équation, on échoue.
5. Tester extensivement le processus de typage, entre autres avec des fonctions polymorphes, et la fonction factorielle (sur les entiers natifs).

5 Traits impératifs

5.1 Evaluateur

On ajoute au langage des traits impératifs, concrétisés par l'apparition d'une valeur *unit* `()`, de *régions* (des cases mémoires) dans la syntaxe et par trois nouveaux opérateurs:

- `!e` pour le déréférencement,
- `ref e` pour la création de région,
- `e1 := e2` pour l'assignement.

1. Mettre à jour la syntaxe, le *pretty printer* et les constructeurs.
2. Mettre à jour l'évaluateur :
   - pour réduire `!e` on réduit d'abord `e`. Si `e` est une région `rho`, le terme `!rho` se réduit en le terme associé à la région `rho` dans l'état.
   - pour réduire `ref e` on réduit d'abord `e`. Si `e` est une valeur `v`, on crée une nouvelle région `rho` dans l'état, à laquelle on associe `v`, et `ref v` se réduit en `rho`.
   - pour réduire `e1 := e2` on réduit d'abord `e1`, puis `e2`, puis si `e1` est une région `rho`, alors on associe la valeur de `e2` à `rho` dans l'état, et `e1 := e2` se réduit en `()`.

5.2 Types

On ajoute aux types:

- le type `unit`,
- le constructeur de type `Ref T`, qui représente une région dans laquelle on stocke des valeurs de type `T`,

1. Mettre à jour les types.
2. Mettre à jour la génération d'équations.
3. Mettre à jour l'unification pour traiter les nouveaux constructeurs de types.

5.3 Polymorphisme faible

Sans surprise, le typeur accepte `let l = ref [] in let _ = l := [(^x.x)] in (hd !l) + 2` alors que le terme est "moralement mal-typé" (il ne peut pas se réduire). Pour pouvoir rejeter un tel terme, il faut ajouter la notion de polymorphisme faible, comme vue en cours.

1. Introduire la notion de non-expansivité pour les termes, de polymorphismes faibles pour les types, et modifier le typeur en conséquence pour proposer un système de type correct pour le langage

6 Aller plus loin

Suggestion d'extensions :

- une implémentation d'autres stratégies et/ou d'une évaluation non-déterministe (ou "totale").
- un lexer/parser de λ-calcul,
- une gestion des types produits (enregistrements) et/ou des types sommes (ou option),
- une gestion des objets (définition de classe, création d'objets, appels de méthodes) et implémenter les différentes caractéristiques (polymorphisme de rangée, sous-typage, surcharge) vue en cours.
- une gestion des exceptions,