Rapport de TER:
Formalisation d'un système de types en Coq

Félix SASSUS BOURDA

29/04/2024

Table des matières

1. Présentation du résultat final 3
1.1. Syntaxe 3
1.2. Typage 4
1.3. Formes canoniques: 6
1.4. Réduction 6
1.4.1. Termes clos 6
1.4.2. Substitution 6
1.4.3. Réduction 6
1.5. Préservation 6
1.6. Progrès 7
1.7. Transformations 7
2. Difficultés rencontrées 8
2.1. Capture des variables dans la substitution 8
2.2. Principe d'induction sur les listes d'expressions 8
3. Pistes pour continuer 9

1. Présentation du résultat final

Le langage d'étude est une forme enrichie du λ-calcul simplement typé. Cette partie donne un aperçu du résultat final.

1.1. Syntaxe

Dans le fichiers `Syntax.v`, on trouve la définition de la syntaxe abstraite des types et des expressions du langage. Ainsi qu'une fonction permettant de manipuler les listes d'expressions et sa compatibilité avec les valeurs. Enfin, on peut trouver une extension du parser de Coq pour pouvoir écrire le langage de façon plus naturelle, dans une syntaxe de style ML.

```
Inductive type : Set :=
  (* Primitive types *)
  | Type_Unit : type
  | Type_Num : type
  | Type_Bool : type

  (* Functions *)
  | Type_Fun (t₁ t₂ : type) : type

  (* Product types *)
  | Type_Prod (t₁ t₂ : type) : type
  | Type_Recordt (l : lstype) : type

  (* Sum types *)
  | Type_Disjoint_Union (t₁ t₂ : type) : type
  | Type_Sum (name : string) : type
with lstype :=
  | LST_Nil : lstype
  | LST_Cons (x : string) (t : type) (tail : lstype) : lstype
.

Inductive expr : Set :=
  (* Lambda calculus *)
  | E_Var (x : string)
  | E_App (e₁ e₂ : expr)
  | E_Fun (x : string) (t : type) (e : expr)

  (* Booleans and conditions *)
  | E_True
  | E_False
  | E_If (e₁ e₂ e₃ : expr)
```

```
  (* Let expressions *)
  | E_Let (x : string) (e₁ e₂ : expr)

  (* Arithmetic *)
  | E_Num (z : Z)
  | E_Minus (e₁ e₂ : expr)
  | E_Eq (e₁ e₂ : expr)

  (* Pairs *)
  | E_Pair (e₁ e₂ : expr)
  | E_First (e : expr)
  | E_Second (e : expr)

  (* Records *)
  | E_Rec (bindings : lsexpr)
  | E_Rec_Access (e : expr) (x : string)

  (* Recursion *)
  | E_Fix (e : expr)

  (* Sum types *)
  | E_In_Left (t₁ t₂ : type) (e : expr)
  | E_In_Right (t₁ t₂ : type) (e : expr)
  | E_Match (e case_left case_right : expr)

  | E_Unit
  | E_Sum_Constr (constr : string) (e : expr)
  | E_Sum_Match (e default: expr) (branches : lsexpr)

with lsexpr : Set :=
  | LSE_Nil : lsexpr
  | LSE_Cons : string → expr → lsexpr → lsexpr
.
```

1.2. Typage

Dans le fichier Types.v, on trouve tout d'abord la définition du type des contextes de typage, qui sont des listes associants un indentifiant à un type. On peut trouver dans Maps.v la définition de ces listes ainsi que des théorèmes utiles vis-à-vis du comportement des valeurs stockées à l'ajout d'une nouvelle entrée.

Ensuite, viennent les règles de dérivation de typage, qui sont sous la forme $\Sigma/\Gamma \vdash e : t$ avec Σ la liste des types sommes et leurs constructeurs, Γ le contexte, e un expression et t un type. Elles sont définies comme 3

définitions mutuellement inductives, afin de pouvoir traiter les expressions, les listes associatives `<nom> : <expression>` utilisées pour les records et les mêmes listes, utilisées pour les `match` sur les types sommes.
À ces définitions s'ajoutent un théorème:

```
Theorem weakening :
    ∀ e Σ Γ Γ' t ,
    Maps.includedin Γ Γ' →
    Σ / Γ ⊢ e : t →
    Σ / Γ'⊢ e : t.
```

qui permet de déduire deux Corollaires, qui sont utilisés dans divers démonstrations:

```
Corollary weakening_empty : ∀ Γ Σ e t,
  has_type Σ empty e t →
  has_type Σ Γ e t.

Corollary weakening_eq :
    ∀ Γ₁ Γ₂ Σ e t,
    Maps.eq Γ₁ Γ₂ →
    has_type Σ Γ₁  e  t →
    has_type Σ Γ₂ e t.
```

Enfin, un lemme permet d'associer le typage d'un champ dans un `record` et celui de son expression associée:

```
Lemma lookup_has_type :
  ∀ Σ Γ x lse e lst t,
  Σ / Γ ⊢ᵣ lse : lst →
  lookup_lsexpr x lse = Some e →
  lookup_lstype x lst = Some t →
  Σ / Γ ⊢ e : t.
```

et un dernier lemme permet d'assurer que les branches d'un `match` sur un type sommme générique sont des fonctions:

```
Lemma lookup_branches_type_fun :
  ∀ Σ Γ name_sum constr branches t b tₐ,
  (Σ) / Γ ⊢ₛ name_sum ~> branches : (t) →
  lookup_lsexpr constr branches = Some b →
  lookup_type_sum constr Σ = Some (name_sum, tₐ) →
  Σ / Γ ⊢ b : {{ tₐ → t }}.
```

1.3. Formes canoniques:

Le fichier `Canonical_form.v` énonce des lemmes assurant la forme de certaines expressions quand ce sont des valeurs (par exemples, si v est une valeur de type `Bool` alors `v = false` ou `v = true`)

1.4. Réduction

Pour déinir la réduction des expression, il faut d'abord définir quelques concepts:

1.4.1. Termes clos

Dans `Closed.v`, on trouve une fonction récursive qui décide si une variable est libre dans une expression. On peut avec cette définition énoncer la définition d'un terme clos:

```
Definition closed e := ∀ x, ¬ is_free_in x e.
```

Ainsi que quelques énoncés, dont en particulier

```
Theorem typed_empty :
    ∀ e, ∀ Σ t,
    has_type Σ empty e t →
    closed e.
```

qui assure qu'un terme typable dans un contexte vide est clos, ainsi que différent lemmes d'inversions.

1.4.2. Substitution

On définit de manière simple la substitution, avec la condition supplémentaire que dans les cas où on lie une variable, le terme substitué doit être clos. On prouve ensuite qu'il existe toujours une substitution de x par s dans e, qu'elle est déterministe et qu'elle préserve le typage.

1.4.3. Réduction

On peut maintenant définir la réduction. Elle est en call-by-value, et on démontre qu'elle est déterministe.

1.5. Préservation

On a maintenant tous les outils pour prouver la préservation du typage lors de la réduction:

```
Theorem preservation :
  ∀ e Σ e' t,
  has_type Σ empty e t  →
  e --> e'  →
  has_type Σ empty e' t.
```

La preuve se fait par induction structurelle sur `è`, et découle assez directement des différents lemmes et théorèmes précédents.

1.6. Progrès

De même, on peut prouver la propriété de progrès: un terme clos bien typé est une valeur, ou il peut de réduire

```
Theorem expr_progress : ∀ e Σ t,
  has_type Σ empty e t →
  value e ∨ ∃ e', e --> e'.
```

La preuve se fait également par récurrence structurelle sur `e`, et découle en particulier des lemmes sur la forme canonique des valeurs.

1.7. Transformations

Dans `Transformations/Pair_Records.v`, on définit une traduction qui transforme les paires de forme (e_1, e_2) en `record` de forme $\{\text{first} := e_1; \text{second} := e_2\}$. On ajoute ensuite différentes définitions qui caractérisent l'absence de pairs dans une expression, un contexte, un type…On prouve ensuite différents résultats, pour arriver au résultat final:

```
Theorem pair_free_type :
  ∀ e Σ Γ t,
  Σ / Γ ⊢ e : t →
  (to_pair_free_sum_env Σ) / (to_pair_free_env Γ)
  ⊢ `to_pair_free e` : (to_pair_free_type t).
```

2. Difficultés rencontrées

2.1. Capture des variables dans la substitution

Dans ce travail, les variables sont nommées. Ainsi, fun $x : t \to x \neq$ fun $y : t \to y$. Cette décision impacte en particulier 2 résultats:

- Tout d'abord, la substitution demande que le terme substitué soit clos si on lie une variable, sinon on pourrait obtenir que

  $$(\text{fun } y : t \to x)[x := y] = \text{fun } y : t \to y$$

  Il est donc nécessaire de rajouter cette condition afin que cette situation ne puisse pas avoir lieu, ce qui a comme conséquence le deuxième point.

- Dans l'énoncé de la préservation du typage à la réduction, on demande que le terme soit clos – car typable dans un contexte vide – car il est nécessaire d'avoir, dans le cas (fun $x : t \to e_1)e_2$, e_2 clos. Ce résultat pourrait être plus général en travaillant sur un contexte quelconque sans cette contrainte à la substitution.

2.2. Principe d'induction sur les listes d'expressions

Par défaut, le principe d'induction pour les définitions mutuellement inductives n'est pas assez puissant. Ce problème s'est présenté à deux occasions:
- Lors de la définition des `records`, j'avais décidé au début de les encoder par des listes associatives, mais le principe d'induction généré n'était pas assez général. J'ai donc créé les listes directement comme des termes. L'approche fonctionnait mais ressemblait plus à une astuce pour dépanner.
- Lors de la généralisation des `match`, une nouvelle liste associative s'est révélée nécessaire. La méthode utilisée pour les `record` ne marchait pas dans ce cas. La solution a alors été de définir les listes associatives d'expressions avec une définition mutuellement inductive, puis de générer avec la commande `Scheme` un principe d'induction plus fort. J'ai donc réutilisé cette définition pour redéfinir plus proprement les `record`.

3. Pistes pour continuer

Il y a plusieurs pistes possibles pour continuer ce travail:

- Régler le problème de la capture des variables, soit en utilisant l'α-équivalence sur les termes, soit en utilisant une approche sans noms, avec les indices de Bruijn par exemple
- Implémenter les types sommes récursifs
- Vérifier l'exhaustivité des `match`
- Continuer sur les transformations:
  - Prouver la réciproque des résultats déjà prouvés (ou une version proche quand la réciproque n'est pas exacte).
  - Transformer la première version des types sommes, avec les constructeurs `inl` et `inr`, en un type somme générique
  - Transformer les booléens en un type somme générique