

Traduction d'OCaml vers une variante de Système F

Jonathan Protzenko
sous la direction de François Pottier

June 14, 2010

Plan

Introduction

Aperçu du problème

Contributions

Décoration d'ASTs

Traduction(s)

Système F plus coercions

Plan

Introduction

Aperçu du problème

Contributions

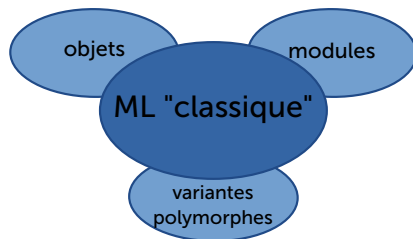
Décoration d'ASTs

Traduction(s)

Système F plus coercions

Pourquoi traduire?

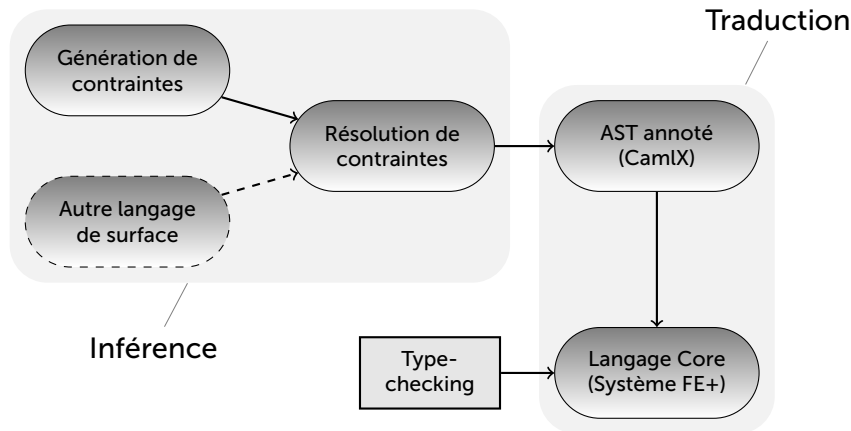
- On veut augmenter la confiance dans la chaîne de compilation
- "Well-typed programs can't go wrong" (Milner)
- Le système de types d'OCaml est trop complexe
 - Traduire le programme dans un langage de base
 - Vérifier le typage *a posteriori*



Objectifs à long terme

- Fournir un langage intermédiaire pour *effectuer des analyses* et compiler plus en avant: expressions *simples*, informations de type *riches*.
- Augmenter la confiance dans la chaîne de compilation: à défaut de prouver la correction du typeur, prouver la cohérence de ses résultats.
- Clarifier la sémantique du langage original: quelles sont les constructions qui s'expriment bien dans FE+?

Dans les grandes lignes...



Le processus se découpe en deux parties : génération/résolution de contraintes, et traductions jusqu'à Système FE+.

Plan

Introduction

Aperçu du problème

Contributions

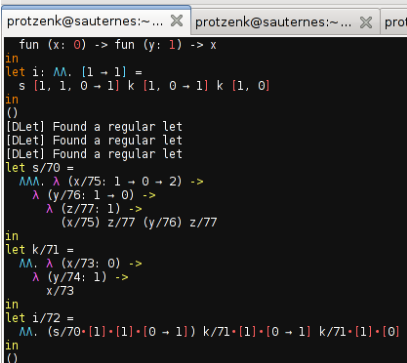
Décoration d'ASTs

Traduction(s)

Système F plus coercions

Trois grands axes de travail

- Récrire un système d'inférence par contraintes, et l'adapter pour donner un *AST annoté*.
- Élaborer un processus de traduction d'un fragment d'OCaml vers un langage minimaliste
- Concevoir le système de types qui permet de justifier le comportement d'OCaml



```

protzenk@sauternes:~... X protzenk@sauternes:~... X prot
fun (x: 0) -> fun (y: 1) -> x
in
let i:  $\lambda\lambda. [1 \rightarrow 1] =$ 
  s [1, 1, 0  $\rightarrow$  1] k [1, 0  $\rightarrow$  1] k [1, 0]
in
()
[DLet] Found a regular let
[DLet] Found a regular let
[DLet] Found a regular let
let s/70 =
   $\lambda\lambda\lambda. \lambda (x/75: 1 \rightarrow 0 \rightarrow 2) \rightarrow$ 
     $\lambda (y/76: 1 \rightarrow 0) \rightarrow$ 
       $\lambda (z/77: 1) \rightarrow$ 
        (x/75) z/77 (y/76) z/77
in
let k/71 =
   $\lambda\lambda. \lambda (x/73: 0) \rightarrow$ 
     $\lambda (y/74: 1) \rightarrow$ 
      x/73
in
let i/72 =
   $\lambda\lambda. (s/70 \cdot [1] \cdot [1] \cdot [0 \rightarrow 1]) \cdot k/71 \cdot [1] \cdot [0 \rightarrow 1] \cdot k/71 \cdot [1] \cdot [0]$ 
in
()
```


Plan

Introduction

Décoration d'ASTs

Le cœur du problème

Garder les annotations à portée de main

Traduction(s)

Système F plus coercions

Plan

Introduction

Décoration d'ASTs

Le cœur du problème

Garder les annotations à portée de main

Traduction(s)

Système F plus coercions

L'inférence par contraintes

- Nouvelle présentation d'un algorithme « classique »
- Séparation claire et élégante entre génération et résolution
- Préférable à l'implémentation OCaml, performante mais difficile d'accès

Exemple:

$$\llbracket \lambda z. t : T \rrbracket = \exists X_1 X_2. (\text{let } z : X_1 \text{ in } \llbracket t : X_2 \rrbracket \wedge X_1 \rightarrow X_2 \leq T)$$

Que fait l'inférence par contraintes?

L'inférence par contraintes répond oui ou non.

Au mieux, affiche les types inférés des définitions top-level.

```
let (x, y) = (fun x -> x) (1, fun x -> x)
```

```
val x: int
```

```
val y:  $\forall \alpha. \alpha \rightarrow \alpha$ 
```

Comment l'adapter pour afficher un AST annoté?

(Pas de valeur restriction dans les exemples)

Plan

Introduction

Décoration d'ASTs

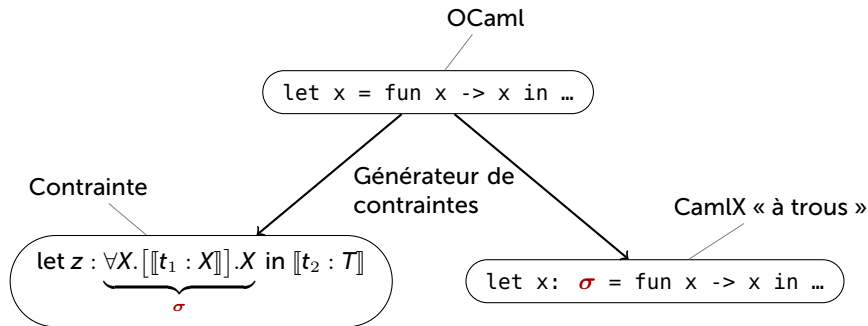
Le cœur du problème

Garder les annotations à portée de main

Traduction(s)

Système F plus coercions

Une méthode ad-hoc



Idée: le générateur de contraintes renvoie *deux* arbres qui *partagent* des structures σ décrivant les schémas de type.

Fonctionnement de cette méthode

- Le générateur de contraintes *pré-alloue* des « boîtes vides » correspondant aux futurs résultats du solveur de contraintes.
- Le solveur résout la contrainte, et remplit au passage les boîtes.
- Les boîtes sont partagées: après la résolution des contraintes, les trous sont remplies, et l'AST « CamlX » est désormais annoté.

Avec un peu de recul...

- Simple et efficace : il s'agit de « faire suivre » les informations nécessaires.
- Facile à implémenter : solution d'une remarquable flexibilité.
- Peu élégant : le contenu des boîtes expose les structures internes du solveur.
- Pas de scope: les schémas de type sont extrudés, sortis de leur contexte.
- Formalisation difficile.

Il faudrait arriver à une forme plus propre et plus propice aux transformations...

Plan

Introduction

Décoration d'ASTs

Traduction(s)

Quelles traductions ?

Le décodeur

Le désucreur

Système F plus coercions

Plan

Introduction

Décoration d'ASTs

Traduction(s)

Quelles traductions ?

Le décodeur

Le désucreur

Système F plus coercions

De CamlX « à trous »...

... vers CamlX « tout court »

- Se passer des champs mutables et des classes d'équivalence
- Types en indices de De Bruijn
- Ne pas changer les expressions, simplement les types

```
let (x, y):  $\Lambda$ . [int * ( $\emptyset \rightarrow \emptyset$ )] =  
  (fun (x: int * ( $\emptyset \rightarrow \emptyset$ )) -> x)  
  (1, (fun (x:  $\emptyset$ ) -> x))  
in  
( )
```

Désucrer CamlX vers...

... Système FE+

- Enlever les constructions redondantes d'OCaml
- Offrir de meilleures garanties (unicité des identifiants)
- Une vraie syntaxe des expressions Système F
- Et des coercions (*more on this later*)

match

```
Λ. (λ (x/41: int * (0 → 0)) -> x/41)
    (1, (λ (x/42: 0) -> x/42))
```

with

```
| (x/39, y/40) ▶ ∀x; x0[•[bottom]] ->
    ()
```

Plan

Introduction

Décoration d'ASTs

Traduction(s)

Quelles traductions ?

Le décodeur

Le désucreur

Système F plus coercions

Remplissage des boîtes...

Sont conservées les schémas de types et les variables d'instanciation.

Les schémas sont résolus au niveau des contraintes `let`.

`function`, `fun`, `let`, `match` → contrainte `let`. Donc :

- créer une boîte ;
- l'attacher à la contrainte `let` ;
- l'attacher au nœud CamlX correspondant.

... et nettoyage

les structures union-find sont mutables et contiennent du partage. On utilise des types avec des indices de De Bruijn.

- La généralisation se fait au niveau des `let` : pas de nœud Λ dans la syntaxe des expressions ;
- L'application de type se fait au niveau des instanciations : pas de nœud « application de type » ;
- les patterns sont présents dans les `let` et `function` ;
- les `let` sont multiples.

C'est une représentation avec des types clairs mais des expressions complexes.

Plan

Introduction

Décoration d'ASTs

Traduction(s)

Quelles traductions ?

Le décodeur

Le désucreur

Système F plus coercions

Rôle du désucreur

- Les patterns sont utilisés à de nombreux endroits en OCaml: on les restreint aux `match` uniquement.
- Les `let` and peuvent définir simultanément plusieurs motifs: on utilise des identifiants uniques pour s'en passer.
- Les Λ et les applications de types deviennent des nœuds normaux de la syntaxe des expressions.

... et surtout, sont ajoutées des coercions.

Plan

Introduction

Décoration d'ASTs

Traduction(s)

Système F plus coercions

Introduction

○○○○
○○

Décoration d'ASTs

○○○
○○○○

Traduction(s)

○○○
○○○
○○

Système F plus coercions

<++>

<++>

Introduction

○○○○
○○

Décoration d'ASTs

○○○
○○○○

Traduction(s)

○○○
○○○
○○

Système F plus coercions

<++>

<++>

Introduction

○○○○
○○

Décoration d'ASTs

○○○
○○○○

Traduction(s)

○○○
○○○
○○

Système F plus coercions

<++>

<++>

Introduction

○○○○
○○

Décoration d'ASTs

○○○
○○○○

Traduction(s)

○○○
○○○
○○

Système F plus coercions

<++>

<++>

Introduction

○○○○
○○

Décoration d'ASTs

○○○
○○○○

Traduction(s)

○○○
○○○
○○

Système F plus coercions

<++>

<++>

Introduction

○○○○
○○

Décoration d'ASTs

○○○
○○○○

Traduction(s)

○○○
○○○
○○

Système F plus coercions

<++>

<++>

Introduction

○○○○
○○

Décoration d'ASTs

○○○
○○○○

Traduction(s)

○○○
○○○
○○

Système F plus coercions

<++>

<++>