Un plongement efficace de règles de réécriture dans OCaml

Mathieu Boespflug

6 Mai 2008

L'histoire d'une convergence :

- Calcul symbolique de Gregoire et Leroy, 2002,
- Compilation vers la ZAM - Olivier Hermant, 2005,
- Normalisation par evaluation - Berger et Schwichtenberg, 1991, Danvy, 1996, ...
- Moca - Blanqui, Hardin, Weis, 2007.

$$\text{règles :} \quad r \quad ::= \quad k\ p_1 \cdots\ p_2 \longrightarrow t$$

- Règle commence toujours par une constante de tête
- Filtrage de premier ordre

Exemple :

$$+\ 0\ x \longrightarrow x$$
$$+\ x\ 0 \longrightarrow x$$
$$+\ (Sx)\ y \longrightarrow S\ (+\ x\ y)$$

1 Introduction

2 Traduction

3 Exemples

4 Vers la normalisation forte

5 Conclusion

1 Introduction

2 Traduction

3 Exemples

4 Vers la normalisation forte

5 Conclusion

1 Introduction

2 Traduction

3 Exemples

4 Vers la normalisation forte

5 Conclusion

1 Introduction

2 Traduction

3 Exemples

4 Vers la normalisation forte

5 Conclusion

1 Introduction

2 Traduction

3 Exemples

4 Vers la normalisation forte

5 Conclusion

machines abstraites

Résoudre son problème passe d'abord par le choix de son langage

$$\text{règles :} \quad r \quad ::= \quad k\ p_1 \cdots\ p_2 \longrightarrow t$$

Soit un ensemble $\mathcal{R}$ de règles.
Objectifs :

1. Optimiser la recherche de redex (sous-termes correspondant au coté gauche d'une règle $r \in \mathcal{R}$).
2. Représenter les termes de manière à permettre une implémentation efficace de la subsitution.

Exemple :

$$\text{map f [+ 4 2 ; + 8 12]} \vdash \text{map f [6 ; 20]} \vdash \ldots$$

- ELAN
- Maude
- ASF+SDF
- Moca
- etc, etc...

- simplifier la compilation des règles de réécriture
- porter la vitesse d'exécution de ces machines de β-réduction à notre problème.
  - Reprise du raisonnement de (Grégoire et Leroy, 2002) pour la réécriture.
    - Réduire la réécriture à de la β-réduction + X.

Problème : Il est clair qu'une machine de β-réduction est un outil efficace pour 2. mais que doit-on faire pour effectuer 1. efficacement ?
Solution : n'exprimons pas le problème en λ-calcul. Exprimons le dans le λ-calcul + types algébriques et construction de match.

- simplifier la compilation des règles de réécriture
- porter la vitesse d'exécution de ces machines de β-réduction à notre problème.
  - Reprise du raisonnement de (Grégoire et Leroy, 2002) pour la réécriture.
    - Réduire la réécriture à de la β-réduction + X.

Problème : Il est clair qu'une machine de β-réduction est un outil efficace pour 2. mais que doit-on faire pour effectuer 1. efficacement ?
Solution : n'exprimons pas le problème en λ-calcul. Exprimons le dans le λ-calcul + types algébriques et construction de match.

- simplifier la compilation des règles de réécriture
- porter la vitesse d'exécution de ces machines de β-réduction à notre problème.
  - Reprise du raisonnement de (Grégoire et Leroy, 2002) pour la réécriture.
    - Réduire la réécriture à de la β-réduction + X.

Problème : Il est clair qu'une machine de β-réduction est un outil efficace pour 2. mais que doit-on faire pour effectuer 1. efficacement ?
Solution : n'exprimons pas le problème en λ-calcul. Exprimons le dans le λ-calcul + types algébriques et construction de match.

Comment OCaml fait des match efficaces

```
let merge xs ys = match xs, ys with
| [], _ -> ...
| _, [] -> ...
| x::xs, y::ys -> ...
```

Comment OCaml fait des match efficaces

```
catch
  (catch
    (switch xs with
      case []: ...
      case (::):
        (switch ys with
          case (::): ...
          default: exit))
   with
    (switch ys with
      case []: ...
      default: exit))
   with (failwith "Partial match")
```

Traduction formelle

- Syntaxe des termes t non-spécifiée.
- patterns : $p ::= \_ \mid x \mid c(p_1, \ldots, p_n)$
- valeurs : $v ::= c(v_1, \ldots, v_n)$
- règles : $r ::= k\ p_1 \cdots\ p_2 \longrightarrow t$
- Ensemble de constantes $\mathcal{C}$.
- Ensemble de constructeurs $\mathcal{K}$.
- $\mathcal{C}$ et $\mathcal{K}$ disjoints. Supposons une bijection $(\hat{\cdot})$

$$\forall c \in \mathcal{C}, \forall k \in \mathcal{K}, \hat{c} = k \Leftrightarrow \hat{k} = c$$

Traduction formelle

- patterns : $p ::= \_ \mid x \mid c(p_1, \ldots, p_n)$
- valeurs : $v ::= c(v_1, \ldots, v_n)$

$$[\![e]\!] = \texttt{e} \qquad [\![\_]\!] = \texttt{\_}$$

$$[\![x]\!] = \texttt{x} \qquad [\![c(p_1, \ldots, p_n)]\!] = \hat{c}([\![p_1]\!], \ldots, [\![p_n]\!])$$

$$\left[\!\!\left[\begin{array}{ccc} k\ p_{11} \cdots p_{1n} & \longrightarrow & e_1 \\ \vdots & & \vdots \\ k\ p_{m1} \cdots p_{mn} & \longrightarrow & e_m \end{array} \right]\!\!\right] =$$

```
let rec c x1 ... xn =
  match (x1, ..., xn) with
  | ([[p1]], ..., [[pn]]) -> [[e]]
  | default -> ĉ(x1, ..., xn)
```

Extension OCaml en Camlp4

- Imbrication de règles de réécriture dans des programmes OCaml.
- Permet de définir dans le toplevel d'un module un ensemble de règles, groupées par constante de tête.

```
let f x = 42

let rule r p11 ... p1n --> t1
     and r p21 ... p2n --> t2
     ...
     and r pm1 ... pmn --> tm

let rule s ...
     and s ...
```

Fonction factorielle

(demo)

Extension de la transformation

Problème : L'interpréteur construit via les règles précédentes ne trouve que la forme normale de tête faible.

$\longrightarrow$ *la stratégie de normalisation de l'interpréteur n'est pas complète.*

Solution : Utiliser la normalisation par évaluation pour réduire tous les redex.

Pourquoi la normalisation par évaluation ?

- Permet encore une fois de traiter un évaluateur existant comme une boite noire.

Extension de la transformation

Problème : L'interpréteur construit via les règles précédentes ne trouve que la forme normale de tête faible.

> $\longrightarrow$ *la stratégie de normalisation de l'interpréteur n'est pas complète.*

Solution : Utiliser la normalisation par évaluation pour réduire tous les redex.
Pourquoi la normalisation par évaluation ?

- Permet encore une fois de traiter un évaluateur existant comme une boite noire.

Calcul à deux niveaux

- Construction d'un modèle dénotationel de la syntaxe des termes.

  $\longrightarrow$ *Calcul à deux niveaux.*

- En OCaml, représentons la syntaxe par un type algébrique

```
type t = Var of string
       | Lam of string * t
       | App of t * t
```

- Représentons la dénotation par des valeurs OCaml :

  `x`, `fun x -> t`, `t t'`, etc.

Relations de réification et de réflection

Réification :

$$\downarrow^{\alpha} e = e$$

$$\downarrow^{t_1 \to t_2} e = \texttt{Lam}(x_1,\ \downarrow^{t_2} \overline{e\ \texttt{@}\ \uparrow_{t_1} x_1}) \qquad \text{avec } x_1 \text{ frais.}$$

Réflection :

$$\uparrow_{\alpha} e = e$$

$$\uparrow_{t_1 \to t_2} e = \overline{\texttt{fun}\ x_1\ \texttt{->}\ \uparrow_{t_2} \texttt{App}(e,\ \downarrow^{t_1} x_1)}$$

Relations de réification et de réflection

Réification :

$$\downarrow^{\alpha} e = e$$

$$\downarrow^{t_1 \to t_2} e = \underline{\lambda x_1.\ \downarrow^{t_2} \overline{(e @ (\uparrow_{t_1} x_1))}} \qquad \text{avec } x_1 \text{ frais.}$$

Réflection :

$$\uparrow_{\alpha} e = e$$

$$\uparrow_{t_1 \to t_2} e = \overline{\lambda x_1.\ \uparrow_{t_2} \underline{(e @ (\downarrow^{t_1} x_1))}}$$

Construction de l'interprétation d'un terme

$$[\![x]\!]\ \rho = \ \rho(x) \tag{1}$$

$$[\![\underline{\lambda x.s}]\!]\ \rho = (\overline{\lambda S.[\![s]\!]})\ \rho[x \mapsto S] \tag{2}$$

$$[\![\underline{s@t}]\!]\ \rho = ([\![s]\!]\ \rho)\ ([\![t]\!]\ \rho) \tag{3}$$

Normalisation par évaluation

Normalisation d'un terme clos :

$$nbe_\tau t = \downarrow^\tau (\llbracket t \rrbracket\ \emptyset)$$

Exemple : Normalisation de l'identité combinatoire

```
let k = Lam("x", Lam ("y", Var "x"))
let s = Lam("x", Lam ("y", Lam("z",
                              App(App(Var "x", Var "z"),
                                  App(Var "y", Var "z")))))
let skk = App (App (s, k), k)
```

```
nbe (b --> b) skk = Lam("x1", Var("x1"))
```

Exemple : Normalisation en présence de réécriture

```
let t = fun x -> fact 5
```

```
nbe (b --> nat) t = Lam("x1", Nat(S(S(S(...O)...))))
```

Conclusions

- Compilation de règles arbitraires en code OCaml, par simple transformation syntaxique, l'ensemble du code compilé constituant un interpréteur efficace pour ces règles.
- Règles non-linéaires et d'arité variable une simple extension.
- La stratégie d'évaluation de l'interpréteur obtenu est l'appel par valeur.
- Avons implémenté la normalisation forte, mais encore incomplète, puisque la stratégie d'évaluation est l'appel par valeur, et parce que la réification ne marche pas toujours en présence de `match ... with`.

Suite des travaux

- Étudier les aspects opérationels de la normalisation par évaluation, notamment en comparaison avec (Grégoire et Leroy, 2002).
- Changer de primitives de match – match lazy par exemple.
- Intégration de la normalisation des règles associatives-commutatives – extension de l'approche implémentée dans Moca ?