Compilation de la réécriture vers la ZAM

Mathieu Boespflug

21 Novembre 2007

1. Introduction
   - La réécriture dans le $\lambda\Pi$-calcul modulo
   - Applications à la compilation de langages fonctionnels
   - Préliminaires
2. Le calcul symbolique pour une compilation vers la ZAM
   - Le calcul symbolique de Benjamin Grégoire pour la normalisation du λ-calcul
   - Présentation du calcul symbolique
3. Perspectives
   - Extensions
   - Autres directions

1. Introduction
   - La réécriture dans le $\lambda\Pi$-calcul modulo
   - Applications à la compilation de langages fonctionnels
   - Préliminaires

2. Le calcul symbolique pour une compilation vers la ZAM
   - Le calcul symbolique de Benjamin Grégoire pour la normalisation du λ-calcul
   - Présentation du calcul symbolique

3. Perspectives
   - Extensions
   - Autres directions

Le $\lambda\Pi$-calcul modulo

- Le $\lambda\Pi$-calcul étend les types du λ-calcul simplement typé vers des types dépendents.
- Il est possible d'exprimer dans ce langage toutes les preuves en déduction naturelle minimale de la logique de prédicats minimale :
  - Pour toute théorie $\mathcal{T}$, étant donné un ensemble de noms de variables V,
  $$Q \in \mathcal{T} \longmapsto v \in V$$
  - Les variables libres d'un terme en $\lambda\Pi$-calcul correspondent aux axiomes de la théorie $\mathcal{T}$.
  - Problème : perte de la garantie de l'existence de types vides et autres propriétés d'une théorie fortement normalisante.
  - Solution :
    - Passer de la déduction naturelle à la déduction modulo,
    - et passer d'un ensemble d'axiomes dans $\mathcal{T}$ à des règles de réécriture.

Le $\lambda\Pi$-calcul modulo

- Le $\lambda\Pi$-calcul étend les types du λ-calcul simplement typé vers des types dépendents.
- Il est possible d'exprimer dans ce langage toutes les preuves en déduction naturelle minimale de la logique de prédicats minimale :
    - Pour toute théorie $\mathcal{T}$, étant donné un ensemble de noms de variables V,

$$Q \in \mathcal{T} \longmapsto v \in V$$

    - Les variables libres d'un terme en $\lambda\Pi$-calcul correspondent aux axiomes de la théorie $\mathcal{T}$.
    - Problème : perte de la garantie de l'existence de types vides et autres propriétés d'une théorie fortement normalisante.
    - Solution :
        - Passer de la déduction naturelle à la déduction modulo,
        - et passer d'un ensemble d'axiomes dans $\mathcal{T}$ à des règles de réécriture.

définition

Une règle de réécriture est un quadruple (Γ, T, l, r), écrit

$$l \longrightarrow^{\Gamma, T} r$$

où Γ est un contexte, T un type, l, r deux termes du λΠ-calcul en forme β-normale.

- Obéit à certaines contraintes de type :

  $l \longrightarrow^{\Gamma, T} r$ est bien typée dans le contexte Σ si $\Sigma\Gamma$ est bien formé et $l, r : T$.

définition

Une règle de réécriture est un quadruple (Γ, T, l, r), écrit

$$l \longrightarrow^{\Gamma, T} r$$

où Γ est un contexte, T un type, l, r deux termes du $\lambda\Pi$-calcul en forme β-normale.

- Obéit à certaines contraintes de type :

  $l \longrightarrow^{\Gamma, T} r$ est bien typée dans le contexte Σ si $\Sigma\Gamma$ est bien formé et $l, r : T$.

Un exemple de règles dans $\lambda\Pi$

Il est possible d'encoder tous Pure Type System (PTS) fonctionnel dans le $\lambda\Pi$-calcul[1].

types d'univers

$U_s : Type$

axiome

$\langle s_1, s_2 \rangle$

régle de réécriture

$\dot{s_1} : U_{s_2}$

fonctions de décodage

$\epsilon_s : U_s \Rightarrow Type$

variable

$\dot{s_1} : U_{s_2}$

variable

$\dot{\Pi}_{\langle s_1, s_2, s_3 \rangle} : \Pi X : U_{s_1} : (((\epsilon_1\ X) \Rightarrow U_{s_2}) \Rightarrow U_{s_3})$

[1] Embedding Pure Type Systems in the $\lambda\Pi$-modulo, Cousineau, Dowek, 2007

Un exemple de règles dans $\lambda\Pi$

Il est possible d'encoder tous Pure Type System (PTS) fonctionnel dans le $\lambda\Pi$-calcul[1].

types d'univers

$U_s : Type$

axiome

$\langle s_1, s_2 \rangle$

régle de réécriture

$\dot{s_1} : U_{s_2}$

fonctions de décodage

$\epsilon_s : U_s \Rightarrow Type$

variable

$\dot{s_1} : U_{s_2}$

variable

$\dot{\Pi}_{\langle s_1, s_2, s_3 \rangle} : \Pi X : U_{s_1} : (((\epsilon_1\ X) \Rightarrow U_{s_2}) \Rightarrow U_{s_3})$

[1] Embedding Pure Type Systems in the $\lambda\Pi$-modulo, Cousineau, Dowek, 2007

Encodage du Calcul des Constructions

$$
\begin{array}{rcl}
\epsilon_{Kind}\ (\dot{Type}) & \longrightarrow & U_{Type} \\
\epsilon_{Kind}\ (\dot{\Pi}_{\langle Type, Type, Type\rangle}\ X\ Y) & \longrightarrow & \Pi x : (\epsilon_{Type}\ X)\ (\epsilon_{Type}\ (Y\ x)) \\
\epsilon_{Kind}\ (\dot{\Pi}_{\langle Type, Kind, Kind\rangle}\ X\ Y) & \longrightarrow & \Pi x : (\epsilon_{Type}\ X)\ (\epsilon_{Kind}\ (Y\ x)) \\
\epsilon_{Kind}\ (\dot{\Pi}_{\langle Kind, Type, Kind\rangle}\ X\ Y) & \longrightarrow & \Pi x : (\epsilon_{Kind}\ X)\ (\epsilon_{Type}\ (Y\ x)) \\
\epsilon_{Kind}\ (\dot{\Pi}_{\langle Kind, Kind, Kind\rangle}\ X\ Y) & \longrightarrow & \Pi x : (\epsilon_{Kind}\ X)\ (\epsilon_{Kind}\ (Y\ x))
\end{array}
$$

Fusion de listes par manipulation syntaxique

- La performance de programmes fonctionnels repose beaucoup aujourd'hui sur les optimisations du compilateur.

  $\longrightarrow$ *Éviter les calculs inutiles.*

- Dans GHC, il est possible pour le programmeur de spécifier ses propres optimisations, sous formes de règles de manipulation syntaxique des termes[2].
- Exemple :

```
{-# RULES
    "map/map"  forall f g xs. map f (map g xs) =
                         map (f.g) xs
#-}
```

[2] Playing by the rules, Peyton Jones, Tolmach, Hoare, 2001.

Importance dans les langages fonctionnels

- Ces règles sont utilisées pour les optimisations de fusion de listes (ex : foldr/build).
- Essentielles pour la bonne performance et la viabilité de certaines bibliothèques bas niveau en Haskell telles que ByteString[3].
- Cependent,
  - Règles syntaxique
  - dont l'application est limitée aux premières phases de la compilation.
- Prochaines étapes :
  - exécuter ces règles de manière efficace,
  - intercaler leur exécution avec l'évaluation faible des λ-termes.

[3] Rewriting Haskell strings, Coutts, Stuart, Leshchinskiy, 2006

Compilation

L'idée : transformation préalable de l'arbre syntaxique du programme afin d'accélérer son exécution.

Traduction vers

$$\left\{\begin{array}{l}\text{un calcul équivalent}\\\text{un modèle d'exécution équivalent}\end{array}\right.$$

Les machines abstraites

"The discussion [generally focusses] around the design of a so-called "abstract machine", which distills the key aspects of the compilation technique without becoming swamped in the details of source language or code generation."

— *Simon Peyton Jones* [4]

[4] Implementing lazy functional languages - the spineless tagless G-Machine, Peyton Jones, 1992.

Points essentiels :

- Traduction du parcours d'un arbre de termes vers une suite d'instructions bas niveau.
- Moins d'indirections et de points de choix.
- Plus efficace à interpréter.
- Plus facile à optimiser et à traduire vers du code machine.

Points essentiels :

- Traduction du parcours d'un arbre de termes vers une suite d'instructions bas niveau.
- Moins d'indirections et de points de choix.
- Plus efficace à interpréter.
- Plus facile à optimiser et à traduire vers du code machine.

Points essentiels :

- Traduction du parcours d'un arbre de termes vers une suite d'instructions bas niveau.
- Moins d'indirections et de points de choix.
- Plus efficace à interpréter.
- Plus facile à optimiser et à traduire vers du code machine.

Points essentiels :

- Traduction du parcours d'un arbre de termes vers une suite d'instructions bas niveau.
- Moins d'indirections et de points de choix.
- Plus efficace à interpréter.
- Plus facile à optimiser et à traduire vers du code machine.

Le calcul symbolique de Benjamin Grégoire

- Normalisation forte du λ-calcul
- Essentielle pour l'exécution efficace de termes de preuves dans Coq.
- L'idée : réutiliser la ZAM de OCaml, héritant ainsi d'une implémentation efficace de la β-réduction.
- Traduction vers un calcul symbolique, qui sera ensuite compilé vers la ZAM.

- Normalisation forte du λ-calcul
- Essentielle pour l'exécution efficace de termes de preuves dans Coq.
- L'idée : réutiliser la ZAM de OCaml, héritant ainsi d'une implémentation efficace de la β-réduction.
- Traduction vers un calcul symbolique, qui sera ensuite compilé vers la ZAM.

- Normalisation forte du λ-calcul
- Essentielle pour l'exécution efficace de termes de preuves dans Coq.
- L'idée : réutiliser la ZAM de OCaml, héritant ainsi d'une implémentation efficace de la β-réduction.
- Traduction vers un calcul symbolique, qui sera ensuite compilé vers la ZAM.

- Normalisation forte du λ-calcul
- Essentielle pour l'exécution efficace de termes de preuves dans Coq.
- L'idée : réutiliser la ZAM de OCaml, héritant ainsi d'une implémentation efficace de la β-réduction.
- Traduction vers un calcul symbolique, qui sera ensuite compilé vers la ZAM.

Termes du calcul symbolique :

$$\begin{array}{lcll} b\text{-termes :} & b & ::= & x \mid \lambda x.b \mid b_1\ b_2 \mid [\tilde{x}\ v_1\ \ldots\ v_n] \\ \text{valeurs :} & v & ::= & \lambda x.b \mid [\tilde{x}\ v_1\ \ldots\ v_n] \end{array}$$

Rajout d'une règle de réduction :

$$[\tilde{x}\ v_1\ \ldots\ v_n]v \mapsto [\tilde{x}\ v_1\ \ldots\ v_n\ v]$$

Réduction forte par itération de la réduction symbolique faible et relecture

$$\mathcal{N}(b) = \mathcal{R}(\mathcal{V}_s(b)) \tag{1}$$

$$\mathcal{R}(\lambda x.b) = \lambda y.\mathcal{N}((\lambda x.b)\ [\tilde{y}]) \quad y \text{ une nouvelle variable} \tag{2}$$

$$\mathcal{R}([\tilde{x}\ v_1\ \ldots\ v_n]) = x\,\mathcal{R}(v_1)\ \ldots\ \mathcal{R}(v_n) \tag{3}$$

Exemple

$$a = (\lambda f.\lambda x.f\ (x\ (f\ x)))\ \lambda z.z$$

évaluation faible $\mathcal{V}_s$

$$v = (\lambda x.(\lambda z.z)(x((\lambda z.z)x)))$$

évaluation après relecture de $v[\tilde{u}]$

$$v' = [\tilde{u}[\tilde{u}]]$$

relecture

$$v'' = u\ u$$

fin.

Exemple

$$a = (\lambda f.\lambda x.f\ (x\ (f\ x)))\ \lambda z.z$$

évaluation faible $\mathcal{V}_s$

$$v = (\lambda x.(\lambda z.z)(x((\lambda z.z)x)))$$

évaluation après relecture de $v[\tilde{u}]$

$$v' = [\tilde{u}[\tilde{u}]]$$

relecture

$$v'' = u\ u$$

fin.

Exemple

$$a = (\lambda f.\lambda x.f\ (x\ (f\ x)))\ \lambda z.z$$

évalution faible $\mathcal{V}_s$

$$v = (\lambda x.(\lambda z.z)(x((\lambda z.z)x)))$$

évaluation après relecture de $v[\tilde{u}]$

$$v' = [\tilde{u}[\tilde{u}]]$$

relecture

$$v'' = u\ u$$

fin.

Exemple

$$a = (\lambda f.\lambda x.f\ (x\ (f\ x)))\ \lambda z.z$$

évaluation faible $\mathcal{V}_s$

$$v = (\lambda x.(\lambda z.z)(x((\lambda z.z)x)))$$

évaluation après relecture de $v[\tilde{u}]$

$$v' = [\tilde{u}[\tilde{u}]]$$

relecture

$$v'' = u\ u$$

fin.

Exemple

$$a = (\lambda f.\lambda x.f\ (x\ (f\ x)))\ \lambda z.z$$

évaluation faible $\mathcal{V}_s$

$$v = (\lambda x.(\lambda z.z)(x((\lambda z.z)x)))$$

évaluation après relecture de $v[\tilde{u}]$

$$v' = [\tilde{u}[\tilde{u}]]$$

relecture

$$v'' = u\ u$$

fin.

Calcul symbolique pour des extensions du λ-calcul

$$
\begin{array}{llcl}
b\text{-termes} & a & ::= & x \mid \lambda x.a \mid a_1\ a_2 \mid C_i(\vec{a}) \\
 & & & \mid \texttt{match}(a)\ \texttt{with}(\vec{x}_i \to a_i)_{i \in I} \mid [k] \\
\text{accumulateurs :} & k & ::= & \tilde{x} \mid kv \mid \texttt{match}(k)\ \texttt{with}(\vec{x}_i \to b_i)_{i \in I} \\
\text{valeurs :} & v & ::= & C_i(\vec{v}) \mid [k] \mid \lambda x.b
\end{array}
$$

Réduction faible du calcul symbolique

$$
\begin{aligned}
(\lambda x.b)\ v &\mapsto b[x \leftarrow v] && (4)\\
[k]\ v &\mapsto [k\ v] && (5)\\
\mathtt{match}(C_j(\vec{v}))\ \mathtt{with}(\vec{x}_i \rightarrow b_i)_{i\in I} &\mapsto b_j[\vec{x}_j \leftarrow \vec{v}] && (6)\\
\mathtt{match}([k])\ \mathtt{with}(\vec{x}_i \rightarrow b_i)_{i\in I} &\mapsto [\mathtt{match}(k)\ \mathtt{with}(\vec{x}_i \rightarrow b_i)_{i\in I}] && (7)
\end{aligned}
$$

avec $\Gamma_v ::= []v \mid b[] \mid C_i(\vec{b}, [], \vec{v}) \mid \mathtt{match}([)\ \mathtt{with}](\vec{x}_i \rightarrow b_i)_{i\in I}$.

Travaux non publiés de Olivier Hermand basés sur le *calcul symbolique* de Benjamin Grégoire

- Traitement des règles de réécriture du premier ordre.
- Ajout de la règle de réduction *rewrite* en plus de la β-réduction :

$$c\ t'_1\ \ldots\ t'_n \longrightarrow \sigma t_{n+1}$$

si $c\ t_1\ \ldots\ t_n \longrightarrow t_{n+1} \in \mathcal{R}$ et $c\ t'_1\ \ldots\ t'_n = \sigma(c\ t_1\ \ldots\ t_n)$.

- Exemple :

$$\begin{array}{rcl} \mathit{Plus\ Zero} & \longrightarrow & \mathit{id} \\ \mathit{Plus}\ (S\ x)\ y & \longrightarrow & S\ (\mathit{Plus}\ x\ y) \end{array}$$

Passage à des indices de De Bruijn

$$\begin{array}{lcl} f\ x\ y & \longrightarrow & \lambda z.x \\ f\ x\ x & \longrightarrow & \lambda z.x \end{array}$$

$$\Updownarrow$$

$$\begin{array}{lcl} f\ \_\ \_ & \longrightarrow & \lambda.2 \\ f\ \_\ 0 & \longrightarrow & \lambda.1 \end{array}$$

Filtrage avec des indices de premier ordre

$$
\begin{array}{rcl}
\Phi(c = c)_{[0 \leftarrow \vec{x}]} & \triangleright & [0 \leftarrow \vec{x}] \\
\Phi(c = t)_{[0 \leftarrow \vec{x}]} & \triangleright & \mathit{fail} \\
\Phi(\_ = t)_{[0 \leftarrow \vec{x}]} & \triangleright & [0 \leftarrow t.\vec{x}] \\
\Phi(m = t)_{[0 \leftarrow \vec{x}]} & \triangleright & [0 \leftarrow \vec{x}] \text{ if } t = x_m \\
\Phi(m = t)_{[0 \leftarrow \vec{x}]} & \triangleright & \mathit{fail} \text{ if } t \neq x_m \\
\Phi(t_1 \ t_2 = p_1 \ p_2)_{[0 \leftarrow \vec{x}]} & \triangleright & \Phi(t_1 = t_2)_{\Phi(t_1 = p_1)_{[0 \leftarrow \vec{x}]}}
\end{array}
$$

Langage du calcul symbolique

$$
\begin{array}{rcl}
t & := & n \mid t\ t \mid c \mid v \\
k & := & \tilde{c} \mid k\ v \mid n \\
v & := & \lambda.t \mid [k] \mid \|\bar{c}_r\ v_1\ \ldots\ v_m\| \\
r & := & m \mid \tilde{c} \\
p & := & c_{/0} \mid \_{}_{/1} \mid n_{/0} \mid (p_{/n}\ p_{/m})_{/m+n} \\
m & := & \texttt{match}(n)\ \texttt{with}\ \begin{array}{lcl} & p_{11} \ldots p_{1n} & \mapsto & t_1 \\ \mid & p_{21} \ldots p_{2n} & \mapsto & t_2 \\ & \vdots & & \vdots \\ \mid & p_{k1} \ldots p_{kn} & \mapsto & t_k \\ \mid & \texttt{default} & \mapsto & r \end{array}
\end{array}
$$

Règles de réduction faible

$$
\begin{aligned}
[k]\ v &\mapsto [k\ v] && (8)\\
\lambda.t\ v &\mapsto t[0 \leftarrow v] && (9)\\
||c_{\mathtt{match}(n)\ldots}\ v_1\ \ldots\ v_m||\ v_{m+1} &\mapsto ||c_{\mathtt{match}(n)\ldots}\ v_1\ \ldots\ v_{m+1}|| && (10)\\
||c_{\mathtt{match}(n)\ldots}\ v_1\ \ldots\ v_{n-1}||\ v_n &\mapsto t_l[0 \leftarrow \vec{x}] \text{ if match} && (11)\\
||c_{\mathtt{match}(n)\ldots}\ v_1\ \ldots\ v_{n-1}||\ v_n &\mapsto ||\bar{c}_r\ v_1\ \ldots\ v_{n-1}||\ v_n && (12)\\
||\bar{c}_{\tilde{c}}\ v_1\ \ldots\ v_{n-1}||\ v_n &\mapsto [\tilde{c}\ v_1\ \ldots\ v_{n-1}\ v_n] && (13)\\
c &\mapsto ||\bar{c}_r|| && (14)\\
\Gamma(t) &\mapsto \Gamma(t') \text{ if } t \mapsto t' && (15)
\end{aligned}
$$

Extensions et autres directions

Extensions du calcul de Olivier Hermant :

- Filtrage d'ordre supérieur.

  $\longrightarrow$ *filtrage en HOR avec indices de De Bruijn*[5].

- Gestion des théories associatives-commutatives.

[5] A De Bruijn notation for Higher Order Rewriting, Bonelli, Kesner, Ríos, 2000

- Transposer les travaux de P.E. Moreau vers de systémes de types riches.
- L'évaluation partielle dirigée par les types de Danvy [6].
- La dérivation automatique de machines abstraites étant donné une sémantique dénotationelle[7].
- Compilation via des réseaux d'interaction tels que proposés dans la thèse de F.R. Sinot.

[6] Type directed partial evaluation, Danvy, 1996

[7] From interpreter to compiler and abstract machine : a functional derivation, Danvy, 2003