Décurryfication certifiée

Zaynah Dargaye
INRIA Rocquencourt, projet Gallium

Plan

Introduction

Frontend miniML

Décurryfication

Éléments de preuve

Un petit exemple

```
let rec insert x l =
  match l with
   | [] -> [x]
   | h :: t ->
      if x < h then x :: l else h :: insert x t

let rec sort l =
  match l with
   | [] -> []
   | h :: t -> insert h (sort t)
```

Propriétés :

```
forall l,  is_sorted (sort l)
forall l, forall x,  nb_occ x (sort l) = nb_occ x l
```

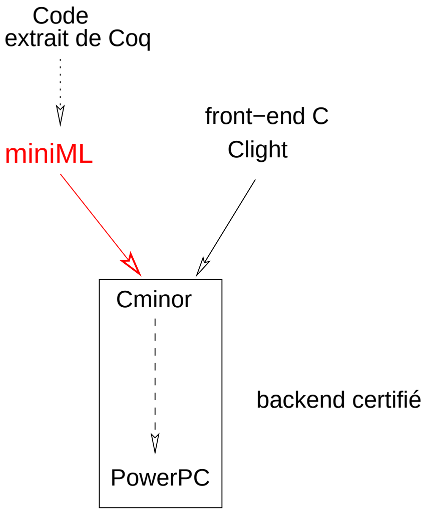
Peut-on raisonner sur l'assembleur produit ?

```
 camlExemple__insert_57:
subl $12, %esp
.L102:
movl %eax, %ecx
cmpl $1, %ebx
je .L100
movl %ebx, 8(%esp)
movl %ecx, 4(%esp)
movl (%ebx), %eax
movl %eax, 0(%esp)
pushl %eax
pushl %ecx
movl $caml_lessthan, %eax
call caml_c_call
.L103:
addl $8, %esp
cmpl $1, %eax
je .L101
.L104: movl caml_young_ptr, %eax
subl $12, %eax
movl %eax, caml_young_ptr
```

non franchement ?

```
.L107:
movl %eax, %ecx
.L108: movl caml_young_ptr, %eax
subl $12, %eax
movl %eax, caml_young_ptr
cmpl caml_young_limit, %eax
jb .L109
leal 4(%eax), %eax
movl $2048, -4(%eax)
movl 0(%esp), %ebx
movl %ebx, (%eax)
movl %ecx, 4(%eax)
addl $12, %esp
ret
.align 16
.L100:
.L111: movl caml_young_ptr, %eax
subl $12, %eax
movl %eax, caml_young_ptr
cmpl caml_young_limit, %eax
jb .L112
```

Compilateur certifié Compcert



Langage source miniML purement fonctionnel

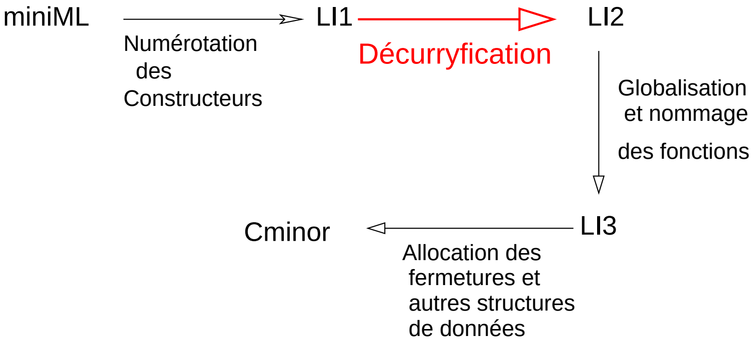
$$
\begin{array}{rcll}
t & ::= & i \mid f & \textit{(Constantes)} \\
& \mid & x & \textit{(Variable)} \\
& \mid & \lambda x.t & \textit{(Abstraction)} \\
& \mid & t_1 \ t_2 & \textit{(Application)} \\
& \mid & op\ [t_0; \ldots; t_n] & \textit{(Opérations arithmétiques)} \\
& \mid & \texttt{let}\ x = t_1\ \texttt{in}\ t_2 & \textit{(Liaison locale)} \\
& \mid & \texttt{letrec}\ f\ x = t_1\ \texttt{in}\ t_2 & \textit{(Définition récursive)} \\
& \mid & \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 & \textit{(Conditionnel)} \\
& \mid & C(t_1, \ldots, t_n) & \textit{(Constructeur)} \\
& \mid & \texttt{match}\ t\ \texttt{with} & \textit{(Filtrage)} \\
& & \mid C_1(x_{11}, \ldots, x_{1n}) \rightarrow t_1 \ \mid \ldots & \\
& & \mid C_n(x_{n1}, \ldots, x_{nk}) \rightarrow t_n &
\end{array}
$$

Langage source miniML purement fonctionnel

Non traité :

- Les aspects impératifs tels que exceptions, références. (les codes extraits de `Coq` sont purement fonctionnels.)
- Les modules.

front-end miniML



Décurryfication

- Transformer une abstraction curryfiée $\lambda x_0 \dots \lambda x_n.\ t$ en abstraction n-aire $\lambda[x_0; \dots; x_n].t$.

Décurryfication

- Transformer une abstraction curryfiée $\lambda x_0 \ldots \lambda x_n.\ t$ en abstraction n-aire $\lambda[x_0; \ldots; x_n].t$.

- $\texttt{let}\ f = \lambda x.\lambda y.\ x + y\ \texttt{in}\ (f\ 3)\ 4$ devient
  $\texttt{let}\ f = \lambda[x; y].\ x + y\ \texttt{in}\ f\ [3; 4]$

Décurryfication

- Transformer une abstraction curryfiée $\lambda x_0 \ldots \lambda x_n.\ t$ en abstraction n-aire $\lambda[x_0; \ldots; x_n].t$.

- $\texttt{let}\ f = \lambda x.\lambda y.\ x + y\ \texttt{in}\ (f\ 3)\ 4$ devient
  $\texttt{let}\ f = \lambda[x; y].\ x + y\ \texttt{in}\ f\ [3; 4]$

- Par contre, $\texttt{let}\ f = \lambda x.\lambda y.\ x + y\ \texttt{in}\ f\ 3$ ne devient pas
  $\texttt{let}\ f = \lambda[x; y].\ x + y\ \texttt{in}\ f\ [3]$

Décurryfication

- Transformer une abstraction curryfiée $\lambda x_0 \ldots \lambda x_n.\ t$ en abstraction n-aire $\lambda[x_0; \ldots; x_n].t$.
- $\texttt{let}\ f = \lambda x.\lambda y.\ x + y\ \texttt{in}\ (f\ 3)\ 4$ devient
  $\texttt{let}\ f = \lambda[x; y].\ x + y\ \texttt{in}\ f\ [3; 4]$
- Par contre, $\texttt{let}\ f = \lambda x.\lambda y.\ x + y\ \texttt{in}\ f\ 3$ ne devient pas
  $\texttt{let}\ f = \lambda[x; y].\ x + y\ \texttt{in}\ f\ [3]$
- Traduction des applications partielles : utilisation de combinateur de curryfication :
  $\texttt{NCurry}_2\ =\ \lambda[f].\lambda[x].\lambda[y].\ f[x; y]$.
  $\texttt{let}\ f = \lambda x.\lambda y.\ x + y\ \texttt{in}\ f\ 3$ devient
  $\texttt{let}\ f = \lambda[x; y].\ x + y\ \texttt{in}\ (\texttt{NCurry}_2\ [f])\ [3]$

Le langage source $\mu\texttt{Ml}$:

- Syntaxe : $t \;::=\; i \mid n! \mid \lambda.t \mid t\ t \mid \texttt{let}\ t\ \texttt{in}\ t$
- Valeurs sémantiques : $v \;::=\; (i) \mid (t, e)$
- Sémantique opérationnelle à grand pas avec environnement :

$$e \vdash i \rightarrow (i) \qquad \frac{e(n) = v}{e \vdash n! \rightarrow v} \qquad e \vdash \lambda.\, t \rightarrow (t, e)$$

$$\frac{e \vdash t_1 \rightarrow (t, e_1) \quad e \vdash t_2 \rightarrow v_2 \quad v_2 :: e_1 \vdash t \rightarrow v}{e \vdash t_1\ t_2 \rightarrow v}$$

$$\frac{e \vdash t_1 \rightarrow v_1 \quad v_1 :: e \vdash t_2 \rightarrow v}{e \vdash \texttt{let}\ t_1\ \texttt{in}\ t_2 \rightarrow v}$$

Le langage cible `nMl` :

- Syntaxe : $t \ ::= \ i \ | \ n! \ | \ \lambda^n.t \ | \ t\ [t_0; \ldots; t_n] \ | \ \texttt{let}\ t\ \texttt{in}\ t$
- Valeurs sémantiques : $v \ ::= \ (i) \ | \ (n, t, e)$
- Sémantique opérationnelle à grand pas avec environnement :

$$e \vdash i \rightarrow (i) \qquad \frac{e(n) = v}{e \vdash n! \rightarrow v} \qquad e \vdash \lambda^n.\, t \rightarrow (n, t, e)$$

$$\frac{\begin{array}{c} e \vdash t_1 \rightarrow (n, t, e_1) \quad e \vdash t_0 \rightarrow v_0 \ \ldots \ e \vdash t_n \rightarrow v_n \\ [v_n :: \ldots :: v_0]@e_1 \vdash t \rightarrow v \end{array}}{e \vdash t_1\ [t_0; \ldots; t_n] \rightarrow v}$$

$$\frac{e \vdash t_1 \rightarrow v_1 \quad v_1 :: e \vdash t_2 \rightarrow v}{e \vdash \texttt{let}\ t_1\ \texttt{in}\ t_2 \rightarrow v}$$

Transformation et Analyse statique

- Analyse de flot de contrôle très simplifiée, qui associe à chaque variable une information d'arité γ :

$$\begin{array}{lclll} \gamma & ::= & \texttt{Known}\ n & \text{fonction d'arité } n+1 \\ & | & \texttt{Unknown} & \text{entier ou fonction d'arité inconnue} \end{array}$$

- S'effectue en même temps que la transformation au travers d'un environnement $\Gamma ::= \gamma_0 :: \gamma_1 :: \ldots$
  Γ est une approximation statique de l'environnement d'évaluation.

Transformation du `let`

- Reconnaître les variables liées par `let` à des abstractions curryfiées : $\texttt{let}\ \underbrace{\lambda.\ \ldots\lambda}_{n+1}.\ t\ \texttt{in}\ \ldots$
- Leur associer l'information $\texttt{Known}\ n$

$$\llbracket \texttt{let}\ \underbrace{\lambda.\ldots.\lambda}_{n+1}.\ t\ \texttt{in}\ t_1 \rrbracket_\Gamma \quad = \quad \begin{array}{l} \texttt{let}\ \lambda^n.\ \llbracket t \rrbracket_{\underbrace{\texttt{Unknown}\ldots\texttt{Unknown}}_{n+1}::\Gamma} \\ \texttt{in}\ \llbracket t_1 \rrbracket_{\texttt{Known}\ n::\Gamma} \end{array}$$

- Dans les autres cas :

$$\llbracket \texttt{let}\ t_1\ \texttt{in}\ t_2 \rrbracket_\Gamma \; = \; \texttt{let}\ \llbracket t_1 \rrbracket_\Gamma\ \texttt{in}\ \llbracket t_2 \rrbracket_{\texttt{Unknown}::\Gamma}$$

Transformation des variables

Selon l'information associée dans l'environnement de compilation :

- Si $\Gamma(v) = \texttt{Unknown}$:

$$[\![v!]\!]_\Gamma = v!$$

- Si $\Gamma(v) = \texttt{Known}\ n$:

$$[\![v!]\!]_\Gamma = \texttt{NCurry}_{n+1}\ [v!]$$

avec

$$\texttt{NCurry}_n = \lambda^0.\underbrace{\lambda^0.\ldots.\lambda^0}_{n}.\ n!\ [(n-1)!;\ldots;0!]$$

Transformation de l'application

On décurryfie les applications de la forme $((\ldots(v!\ t_0)\ldots\ t_n))$ où $\Gamma(v) = \texttt{Known}\ n$.

Selon la forme du sous terme gauche :

- Si $t = (\ldots(v!\ t_0)\ldots t_{n-1})$ et $\Gamma(v) = \texttt{Known}\ n$:
  $[\![t\ t_n]\!]_\Gamma \ = \ v!\ [\ [\![t_0]\!]_\Gamma; \ldots; [\![t_n]\!]_\Gamma]$
- Sinon : $[\![t_a\ t_b]\!]_\Gamma \ = \ [\![t_a]\!]_\Gamma\ [\ [\![t_b]\!]_\Gamma\ ]$

Transformation des autres termes

$$
\begin{array}{rcl}
[\![i]\!]_\Gamma & = & i \\
[\![\lambda .\ t]\!]_\Gamma & = & \lambda^0 .\ [\![t]\!]_{\texttt{Unknown::}\Gamma}
\end{array}
$$

Preuve de préservation sémantique

- La preuve utilise une version relationnelle de la transformation `transl` $\Gamma\ t_\mu\ t_n$.
- La propriété à prouver :

```
Theorem p2n_correct:
  forall (ce:compilenv)
         (menv:list Mval)(m:Mterm)(vm:Mval)
         (nenv:list Nval)(n:Nterm),
  transl ce m n ->
  Meval menv m vm ->
  match_env ce menv nenv ->
  exists (vn: Nval), Neval nenv n vn /\ match_val vm vn.
```

- Elle repose sur une relation entre valeurs μ`ML` et `nML` appelée `match_val` et son extension aux environnements `match_env`.

Intuition de la relation entre fermetures (1/4)

Source		Transformé
$\lambda.a$		$\lambda^0.[\![a]\!]$
$\Downarrow$		$\Downarrow$
(a, e)	$\sim$	$(0, [\![a]\!], [\![e]\!])$

Intuition de la relation entre fermetures (2/4)

Source		Transformé
$\texttt{let}\ \lambda.\lambda.a\ \texttt{in}\ b$		$\texttt{let}\ \lambda^1.[\![a]\!]\ \texttt{in}\ [\![b]\!]$
$\Downarrow$		$\Downarrow$
$(\lambda.a, e)$	$\sim$	$(1, [\![a]\!], [\![e]\!])$

Intuition de la relation entre fermetures (3/4)

Source		Transformé
$\texttt{let}\ \lambda.\lambda.a\ \texttt{in}\ 0!\ 3$		$\texttt{let}\ \lambda^1.[\![a]\!]\ \texttt{in}\ (\lambda^0.\lambda^0.\lambda^0.\ 2!\ [1!, 0!])\ [0!]\ [3]$
$\Downarrow$		$\Downarrow$
$(\lambda.a, e)$	$\sim$	$(0,\ \lambda^0.\ 2!\ [1!, 0!],\ (1, [\![a]\!], [\![e]\!]) :: [\![e]\!])$

Intuition de la relation entre fermetures (4/4)

Source		Transformé
$\texttt{let}\ \lambda.\lambda.a\ \texttt{in}\ 0!\ 3$		$\texttt{let}\ \lambda^1.[\![a]\!]\ \texttt{in}\ (\lambda^0.\lambda^0.\lambda^0.\ 2!\ [1!, 0!])\ [0!]\ [3]$
$\Downarrow$		$\Downarrow$
$(a,\ 3 :: e)$	$\sim$	$(0, 2!\ [1!, 0!],\ 3 :: (1, [\![a]\!], [\![e]\!]) :: [\![e]\!])$

Relation de correspondance entre fermetures (1/3)

```
Inductive match_val: Mval -> Nval -> Prop :=
...
| match_val_clos: forall (m:Mterm)(menv:list Mval)
  (n:Nterm) (nenv:list Nval)(cenv:compilenv),

  transl (Unknown :: ce) m n ->
  match_env ce menv nenv ->

  match_val (PMClos menv m)
            (NClos 0 nenv n)
...
```

Relation de correspondance entre fermetures (2/3)

```
| match_val_curry: forall
(ar:nat) (margs:list Mval) (nargs:list Nval)
(m:Mterm) (menv:list Mval)(n:Nterm) (nenv ne:list Nval) ,

transl (Unknown_n (S ar) cenv) m n ->
match_env cenv menv nenv ->
(List.length margs)<= ar ->
match_env (Unknown_n (length margs)) margs nargs ->

match_val (pcurry_clos ar m nargs menv)
          (ncurry_clos ar (NClos ar nenv n) nargs ne)
...
```

Relation de correspondance entre fermetures (3/3)

```
Definition pcurry_clos (ar:nat)
(m:Mterm) (args menv:list Mval) :=
PMClos  (args++menv) (PMFun (ar-(length args))m).

Definition ncurry_clos (ar:nat) (clos:Nval)
 (args nenv:list Nval): Nval :=
 NClos 0 (args ++ clos ::nenv)
 (NLam_n (ar - length args) (ncurry_body n))
```

Pour aller plus loin

- Stratification en deux niveaux de la relation de correspondance entre fermetures. (valeurs de variable/valeurs d'exécution)
- Extension aux fonctions récursives et au pattern-matching.
- Implémentation de la transformation sous forme fonctionnelle f et preuve de sa correction : $\forall a\, \Gamma,\ \texttt{transl}\ \Gamma\ a\ (f\ \Gamma\ a)$.