平成 27 年度 卒業研究発表会

# MinCamlのK正規化の形式的検証

B2TB2512 水野雅之

工学部　情報知能システム総合学科
住井・松田研究室

2016 年 3 月 11 日

# 研究目的

## MinCamlのK正規化の正当性を検証

- 証明を簡潔に
  - 余帰納的大ステップ意味論
  - de Buijnインデックス
- スケーラビリティの確保
  - 証明自動化機能を活用

# アウトライン

# アウトライン

# コンパイラの形式的検証の意義

コンパイラのバグは伝播

$\Rightarrow$ コンパイラ由来のバグはソースコード上に現れない

$\Rightarrow$ デバッグが困難

# 関連研究

CompCert [Leroy ら 2005]

- C コンパイラの正当性の検証

[Chlipala 2010]

- 非純粋関数型言語処理系の正当性の検証

# 本研究の目標と現状

**目標**:MinCamlのK正規化の正当性を検証

**現状**:サブセットのK正規化の正当性を検証

- **実装済**　1引数関数、算術演算、let、if
- **未実装**　タプル、複数引数関数、配列、外部関数呼び出し

# アウトライン

# MinCamlの特徴

**住井による教育用コンパイラ** [FDPE 2005]

- OCamlで2000**行程度の実装**
  - **非純粋な関数型言語**
  - **型推論**
  - **定数畳み込み等の最適化**

# 対象言語

- **高階関数**
- N引数の構文
- 副作用
- 外部関数呼び出し(入出力)

$$
\begin{aligned}
M, N, f ::= & \\
& \vdots \\
& \mathbf{let\ rec}\ x\ y_1\ \cdots\ y_n = M\ \mathbf{in}\ N \\
& f\ N_1\ \cdots\ N_n \\
& (M_1,\ \cdots\ , M_n) \\
& \mathbf{let}\ (M_1,\ \cdots\ , M_n) = M\ \mathbf{in}\ N \\
& \mathbf{Array.create}\ M\ N \\
& M_1.(M_2) \\
& M_1.(M_2) \leftarrow M_3
\end{aligned}
$$

# 対象言語

- 高階関数
- N 引数の構文
- 副作用
- 外部関数呼び出し (入出力)

$$
\begin{array}{l}
M, N, f ::= \\
\qquad \vdots \\
\qquad \mathbf{let\ rec}\ x\ y_1\ \cdots\ y_n = M\ \mathbf{in}\ N \\
\qquad f\ N_1\ \cdots\ N_n \\
\qquad (M_1,\ \cdots\ , M_n) \\
\qquad \mathbf{let}\ (M_1,\ \cdots\ , M_n) = M\ \mathbf{in}\ N \\
\qquad \mathbf{Array.create}\ M\ N \\
\qquad M_1.(M_2) \\
\qquad M_1.(M_2) \leftarrow M_3
\end{array}
$$

# 対象言語

- 高階関数
- N 引数の構文
- 副作用
- 外部関数呼び出し (入出力)

$$
\begin{aligned}
M, N, f ::= & \\
& \vdots \\
& \mathbf{let\ rec}\ x\ y_1\ \cdots\ y_n = M\ \mathbf{in}\ N \\
& f\ N_1\ \cdots\ N_n \\
& (M_1,\ \cdots\ , M_n) \\
& \mathbf{let}\ (M_1,\ \cdots\ , M_n) = M\ \mathbf{in}\ N \\
& \mathbf{Array.create}\ M\ N \\
& M_1.(M_2) \\
& M_1.(M_2) \leftarrow M_3
\end{aligned}
$$

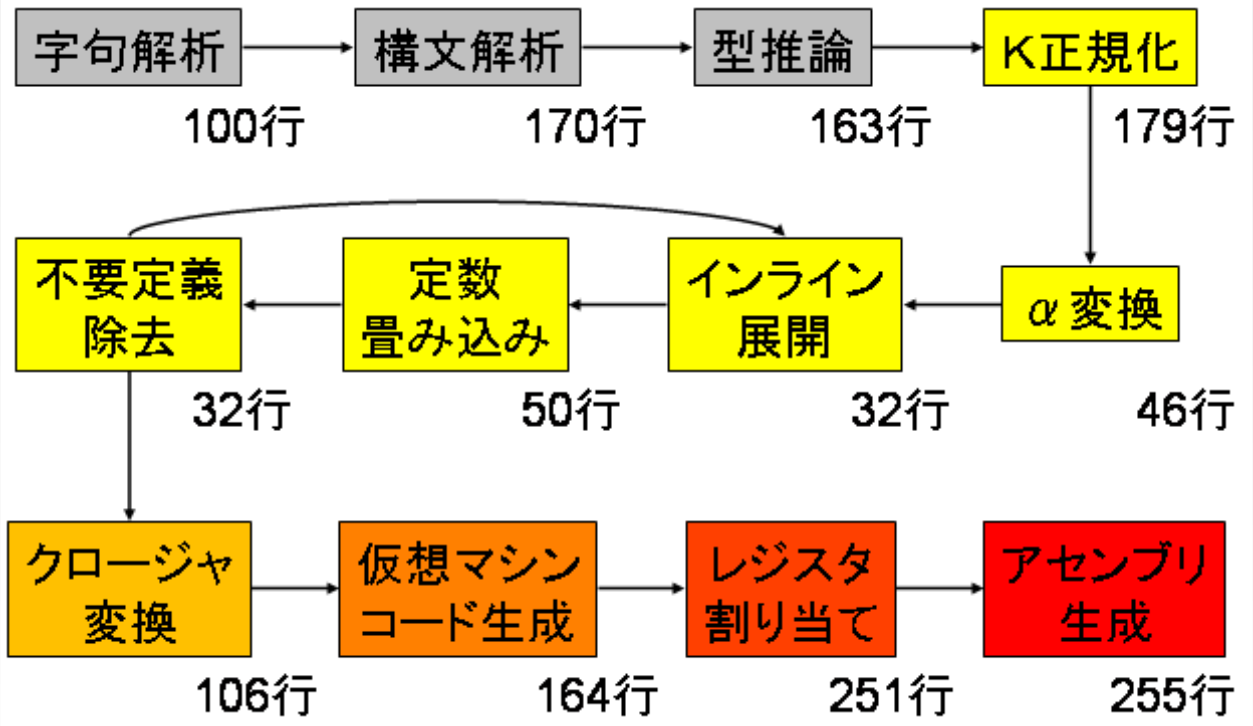
# 対象言語

- 高階関数
- N 引数の構文
- 副作用
- 外部関数呼び出し (入出力)

$$
\begin{array}{l}
M, N, f ::= \\
\qquad \vdots \\
\qquad \mathbf{let\ rec}\ x\ y_1\ \cdots\ y_n = M\ \mathbf{in}\ N \\
\qquad f\ N_1\ \cdots\ N_n \\
\qquad (M_1, \ \cdots\ , M_n) \\
\qquad \mathbf{let}\ (M_1, \ \cdots\ , M_n) = M\ \mathbf{in}\ N \\
\qquad \mathbf{Array.create}\ M\ N \\
\qquad M_1.(M_2) \\
\qquad M_1.(M_2) \leftarrow M_3
\end{array}
$$

# 内部設計

- 様々なコンパイルフェーズ
- 疎結合




http://esumii.github.io/min-caml/index8.html

## 全ての部分式に名前を付ける

$$a + b * c + d$$

$$\begin{aligned}&\textbf{let } x = b * c \textbf{ in}\\&\textbf{let } y = a + x \textbf{ in}\\&y + d\end{aligned}$$

## 束縛に関する操作

## 全ての部分式に名前を付ける

$$a + b * c + d$$

$$\begin{aligned}&\textbf{let } x = b * c \textbf{ in}\\&\textbf{let } y = a + x \textbf{ in}\\&y + d\end{aligned}$$

**束縛に関する操作**

# 期待される定理(1/2)

## K 正規化後の項を評価してみる

### 単純な値では一致

$$1 + 2 \Downarrow 3$$

$$\begin{array}{l}\textbf{let } a = 1 \textbf{ in} \\ \textbf{let } b = 2 \textbf{ in} \\ a + b \Downarrow 3\end{array}$$

### 元の値の K 正規形と一致

$$\begin{array}{l}\lambda x.\lambda y.\lambda z.\ x + y + z \\ \Downarrow \lambda x.\lambda y.\lambda z.\ x + y + z\end{array}$$

$$\begin{array}{l}\lambda x.\lambda y.\lambda z. \\ \textbf{let } a = x + y \textbf{ in} \\ a + z \\ \Downarrow \lambda x.\lambda y.\lambda z. \\ \textbf{let } a = x + y \textbf{ in} \\ a + z\end{array}$$

# 期待される定理(2/2)

**定理 1**

**項 t が値 v に評価される場合、項 t を K 正規化した結果 K(t) は値 v を K 正規化した結果 K(v) に評価される**

**定理 2**

**項 t の評価が停止しない場合、項 t を K 正規化した結果 K(t) の評価は停止しない**

**今回検証した言語の範囲では評価は決定的なため、逆も成り立つ**

# アウトライン

# 名前による表現

## $\alpha$ 等価性の議論が面倒

$$\lambda x.\lambda y.\ x \simeq \lambda a.\lambda b.\ a$$

## fresh な名前が必要になる

- 束縛の関係を乱さないよう変数名を選ぶ

$$\begin{aligned}[x \mapsto z](\lambda z.\ x) &\simeq \lambda z'.\ z \\ &\not\simeq \lambda z.\ z\end{aligned}$$

# de Buijn インデックス

## 何番目の束縛かで変数を表現

- 内側から外側へ数える

$\lambda x.\lambda y.\lambda z.\ xz(yz)$ $\quad$ $\lambda.\lambda.\lambda.\ 2\ 0\ (1\ 0)$

## $\alpha$ 等価な式は構文的に等価

- 名前の freshness を保証

$\lambda x.\lambda y.\ x$

$\lambda a.\lambda b.\ a$

$\lambda.\lambda.\ 1$

## 自由変数のインデックスをずらす

$$\uparrow^d t$$

## 束縛に関する操作

$(\lambda x.\ x)\ (\lambda x.\ y)$

$$\begin{array}{l}\textbf{let}\ a = \lambda x.\ x\ \textbf{in}\\ \textbf{let}\ b = \lambda x.\ y\ \textbf{in}\\ a\ b\end{array}$$

$(\lambda.\ 0)\ (\lambda.\ 1)$

$$\begin{array}{l}\textbf{let}\ \_ = \lambda.\ 0\ \textbf{in}\\ \textbf{let}\ \_ = \uparrow^1 (\lambda.\ 1)\ \textbf{in}\\ 1\ 0\end{array}$$

# シフト

## 自由変数のインデックスをずらす

$$\uparrow^d t$$

## 束縛に関する操作

$$(\lambda x.\ x)\ (\lambda x.\ y)$$

$$\begin{array}{l}\textbf{let}\ a = \lambda x.\ x\ \textbf{in}\\ \textbf{let}\ b = \lambda x.\ y\ \textbf{in}\\ a\ b\end{array}$$

$$(\lambda.\ 0)\ (\lambda.\ 1)$$

$$\begin{array}{l}\textbf{let}\ \_ = \lambda.\ 0\ \textbf{in}\\ \textbf{let}\ \_ = \lambda.\ 2\ \textbf{in}\\ 1\ 0\end{array}$$

# K 正規化の実装

```
Fixpoint knormal e :=
  match e with
  | Exp.Var x => K.Var x
  | Exp.Abs e => K.Abs (knormal e)
  | Exp.App e1 e2 =>
      K.Let (knormal e1)
        (K.Let (shift 1 (knormal e2))
          (App 1 0))
  end.
```

# アウトライン

# 大ステップ意味論

**比較的単純な**
**プログラム変換の検証に適する**

**が**

**無限ループとエラーの区別が困難**

# 例:型無しラムダ計算

## 構文

$$
\begin{array}{rcl}
t & ::= & b \\
& | & x \\
& | & \lambda x.\ t \\
& | & t\ t
\end{array}
\qquad\qquad
\begin{array}{rcl}
v & ::= & b \\
& | & \lambda x.\ t
\end{array}
$$

## 意味論

$$\frac{}{b \Downarrow b} \qquad \frac{}{\lambda x.\ t \Downarrow \lambda x.\ t}$$

$$\frac{t_1 \Downarrow \lambda x.\ t_0 \qquad t_2 \Downarrow v_2 \qquad [x \mapsto v_2]t_0 \Downarrow v}{t_1\ t_2 \Downarrow v}$$

# 大ステップ意味論の問題点

## エラー

$$\mathbf{true}\ \mathbf{true} \not\Downarrow v$$

適用できる規則が無い

## 無限ループ

$$(\lambda x.xx)(\lambda x.xx) \not\Downarrow v$$

有限回の規則適用で導出できない

## 区別できない

# 余帰納的大ステップ意味論 (1/2)

## 余帰納的に定義 [Leroy 2006]

$$\frac{t_1 \Uparrow}{t_1\ t_2 \Uparrow} \qquad \frac{t_1 \Downarrow v_1 \quad t_2 \Uparrow}{t_1\ t_2 \Uparrow}$$

$$\frac{t_1 \Downarrow \lambda x.\ t_0 \quad t_2 \Downarrow v_2 \quad [x \mapsto v_2]t_0 \Uparrow}{t_1\ t_2 \Uparrow}$$

# 余帰納的大ステップ意味論 (2/2)

## エラー

$\mathbf{true}\ \mathbf{true} \not\Uparrow$

適用できる規則がない

## 無限ループ

$(\lambda x.xx)(\lambda x.xx) \Uparrow$

無限回の規則適用を許す

## 区別できる

# 参考:入出力を含む言語への拡張

## どのような入出力を行ったかを表すラベルの列を付与

$$\mathbf{print\_endline}\ ”hoge”;\ \mathbf{read\_line}\ ()$$
$$\Downarrow ”fuga”\ /\ (\mathbf{print\_endline}\ ”hoge”,\ \mathbf{read\_line}\ () = ”fuga”)$$

## ストリームを用いれば無限回の入出力も

$$(\mathbf{while\ true\ do\ print\_endline}\ ”hoge”\mathbf{done})$$
$$\Uparrow / (\mathbf{print\_endline}\ ”hoge”, \mathbf{print\_endline}\ ”hoge”, \cdots)$$

# アウトライン

# コンパイラの検証

## 言語拡張のたび全ての証明の修正が必要

```
Inductive t :=
  | Var : nat -> t
.
Proof.
  intros t.
  induction t.
  Case "Var".
```

```
Inductive t :=
  | Int : Z -> t
  | Var : nat -> t
.
Proof.
  intros t.
  induction t.
  Case "Nat".
    .
  Case "Var".
```

# 構文の違いを自動証明で吸収

- $tactic_1; tactic_2$
- $\mathbf{solve}[tactic_1 | \cdots | tactic_n]$

```
Lemma shift_0 : forall e c,
  shift c 0 e = e.
Proof.
  intros e.
  induction e; intros ?; simpl;
    f_equal;
    solve [ apply shift_var_0 | eauto ].
Qed.
```

# 拡張性の評価

## プリミティブ、if、letを追加

| | 構文 | 証明の行数 |
|---|---|---|
| 拡張前 | 変数 匿名関数 関数適用 | 110 |
| 拡張後 | 20 種類 | 141 |

## 単純な変更に対してはスケール

# アウトライン

# 結論

K 正規化を Coq で検証できた

- de Buijn インデックス、
  余帰納的大ステップ意味論の採用で証明が簡潔に
- 証明自動化による再利用性の高い証明

# 今後の課題

さらに言語を拡張し、MinCaml と同等に

- 組や複数引数の関数
- 配列
- 外部関数呼び出し

K 正規化以外の検証も