

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

## MinCaml の K 正規化の正当性を検証

- 証明を簡潔に
  - 余帰納的意味論
  - ド・ブラン インデックス
- スケーラビリティの確保
  - 半自動証明

# アウトライン

- ① 研究背景
- ② MinCaml
- ③ 意味論の定義
- ④ 束縛の表現
- ⑤ 正当性の検証
- ⑥ 結論

# アウトライン

- ① 研究背景
- ② MinCaml
- ③ 意味論の定義
- ④ 束縛の表現
- ⑤ 正当性の検証
- ⑥ 結論

## CompCert(Leroy et al.)

- Cコンパイラの正当性の検証

## Chlipala POPL 2010

- 非純粋関数型言語処理系の正当性の検証

|   | CompCert     | Chlipala 2010 |
|---|--------------|---------------|
|   | 副作用 入出力      | 高階関数 副作用      |
| × | N 引数の構文 高階関数 | N 引数の構文 入出力   |

# N 引数の構文の難しさ

```
Inductive t :=  
  :  
  | App : list t -> t.
```

list の中が positive でないかもしれない

```
Definition list t := t -> t.  
Definition oops t :=  
  match t with  
  | App f => f t  
  | _ => t  
  end.  
(* 矛盾 ! *)  
Check (oops (App oops) : False).
```

# 入出力の難しさ

- 文面から評価結果が定まらない
  - e.g. `read_line ()`
- 出力結果を区別
  - 以下のプログラムは等価か?

|                 |                             |
|-----------------|-----------------------------|
| <code>()</code> | <code>println "hoge"</code> |
|-----------------|-----------------------------|

|                            |                             |
|----------------------------|-----------------------------|
| <code>while true do</code> | <code>while true do</code>  |
| <code>()</code>            | <code>println "hoge"</code> |
| <code>done</code>          | <code>done</code>           |

# アウトライン

- ① 研究背景
- ② MinCaml
- ③ 意味論の定義
- ④ 束縛の表現
- ⑤ 正当性の検証
- ⑥ 結論



## 住井による教育用コンパイラ (FDPE 2005)

- OCaml で 2000 行程度
- 本格的な処理系
  - 非純粋な関数型言語
  - 型推論
  - 定数畳み込み等の最適化

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

$M, N ::=$

$\vdots$

**let rec**  $x \ y_1 \ \cdots \ y_n = M \ \mathbf{in} \ N$

$M \ N_1 \ \cdots \ N_n$

$(M_1, \ \cdots, M_n)$

**let**  $(M_1, \ \cdots, M_n) = M \ \mathbf{in} \ N$

**Array.create**  $M \ N$

$M_1.(M_2)$

$M_1.(M_2) \leftarrow M_3$

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

$M, N ::=$

$\vdots$

**let rec**  $x \ y_1 \ \cdots \ y_n = M$  **in**  $N$

$M \ N_1 \ \cdots \ N_n$

$(M_1, \ \cdots, M_n)$

**let**  $(M_1, \ \cdots, M_n) = M$  **in**  $N$

**Array.create**  $M \ N$

$M_1.(M_2)$

$M_1.(M_2) \leftarrow M_3$

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

$M, N ::=$

$\vdots$

$\text{let rec } x \ y_1 \ \cdots \ y_n = M \ \text{in } N$

$M \ N_1 \ \cdots \ N_n$

$(M_1, \ \cdots, M_n)$

$\text{let } (M_1, \ \cdots, M_n) = M \ \text{in } N$

$\text{Array.create } M \ N$

$M_1.(M_2)$

$M_1.(M_2) \leftarrow M_3$

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

$M, N ::=$

$\vdots$

**let rec**  $x \ y_1 \ \cdots \ y_n = M$  **in**  $N$

$M \ N_1 \ \cdots \ N_n$

$(M_1, \ \cdots, M_n)$

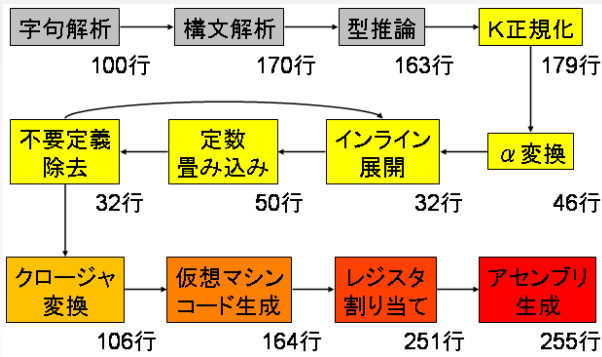
**let**  $(M_1, \ \cdots, M_n) = M$  **in**  $N$

**Array.create**  $M \ N$

$M_1.(M_2)$

$M_1.(M_2) \leftarrow M_3$

- 様々なコンパイルフェーズ
- 疎結合



全ての部分式に名前を付ける

$$a + b * c + d$$

let  $x = b * c$  in  
let  $y = a + x$  in  
 $y + d$

束縛に関する操作

全ての部分式に名前を付ける

$$a + b * c + d$$

**let**  $x = b * c$  **in**  
**let**  $y = a + x$  **in**  
 $y + d$

束縛に関する操作



# 期待される正当性

## K 正規化後の項を評価してみる

$\lambda x. \lambda y. \lambda z. x + y + z$

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

**let**  $a = x + y$  **in**

$a + z$

### 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

# アウトライン

- ① 研究背景
- ② MinCaml
- ③ 意味論の定義**
- ④ 束縛の表現
- ⑤ 正当性の検証
- ⑥ 結論

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

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

## 意味論

$$\frac{\overline{b \Downarrow b} \quad \overline{\lambda x. t \Downarrow \lambda x. t} \quad \frac{t_1 \Downarrow \lambda x. t_0 \quad t_2 \Downarrow v_2}{[x \mapsto v_2]t_0 \Downarrow v}}{t_1 t_2 \Downarrow v}$$

## エラー

`true 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

## 余帰納的定義 (Leroy 2006)

$$\frac{t_1 \uparrow}{t_1 \ t_2 \uparrow}$$

$$\frac{t_1 \Downarrow v_1 \quad t_2 \uparrow}{t_1 \ t_2 \uparrow}$$

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

エラー

無限ループ

`true true`  $\nrightarrow$

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

適用できる規則がない

無限回の規則適用を許す

区別できる

どのような入出力を行ったかを表す  
ラベルを付与

**read\_line ()**  $\Downarrow$  "hoge" / **read\_line ()** = "hoge"

|   |                                |
|---|--------------------------------|
|   | (while true do                 |
|   | print_endline "hoge"           |
| while true do () done $\uparrow$ / $\epsilon$ | done) $\uparrow$               |
|   | / print_endline "hoge" $\dots$ |

## 区別できる



# アウトライン

- ① 研究背景
- ② MinCaml
- ③ 意味論の定義
- ④ 束縛の表現**
- ⑤ 正当性の検証
- ⑥ 結論

## $\alpha$ 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

$$\begin{aligned} [x \mapsto z](\lambda z. x) &\simeq \lambda z'. z \\ &\neq \lambda z. z \end{aligned}$$

## 何番目の束縛かで変数を表現

- 内側から外側へ数える

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

## $\alpha$ 等価な式は構文的に等価

- 名前の freshness から解放

$\lambda x. \lambda y. x$

$\lambda a. \lambda b. a$

$\lambda. \lambda. 1$

## 自由変数のインデックスをずらす

$$\uparrow^d t$$

## 束縛の付け替え

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

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

```
let a = λx. x in
let b = λx. y in
a b
```

```
let _ = λ. 0 in
let _ = ↑1 (λ. 1) in
1 0
```

# 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

- ① 研究背景
- ② MinCaml
- ③ 意味論の定義
- ④ 束縛の表現
- ⑤ 正当性の検証**
- ⑥ 結論

## 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

$tactic_1; tactic_2$

`f_equal`

`solve[tactic1 | ... | tacticn]`

$tactic_1$  で生じたサブゴール全てに  $tactic_2$  を適用

$t_1 \ t_2 \cdots t_n = t'_1 \ t'_2 \cdots t'_n$  から  $t_1 = t'_1, t_2 = t'_2 \cdots$

どれかでサブゴールを閉じられるなら閉じる

```
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.
```



## 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   |

高いスケーラビリティを示した

# アウトライン

- ① 研究背景
- ② MinCaml
- ③ 意味論の定義
- ④ 束縛の表現
- ⑤ 正当性の検証
- ⑥ 結論

K 正規化を Coq で検証できた

- ド・ブラン 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

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