平成 26 年度 卒業論文

MinCaml の K 正規化の形式的検証

東北大学 工学部
情報知能システム総合学科

B2TB2512 水野雅之

指導教員：住井 英二郎 教授
論文指導教員：松田 一孝 准教授

平成 27 年 3 月 11 日　13:30–14:00
電気通信研究所 本館 5 階ゼミ室 (M531)

要旨

コンパイラのバグは生成したコードに波及するため影響が大きく，正当性の証明が盛んに試みられてきた．一方そのような先行研究では，対象言語に存在する IO，要素数が可変の構文といった機構が正当性の証明を困難にしていた．

MinCaml は非純粋な関数型言語のコンパイラであるが，IO，要素数が可変の構文を含んでおり，もし MinCaml を検証できた場合に得られる知見は対象言語に同様の機構を持つ処理系を検証する際にも有用であると考えられる．将来的に MinCaml における K 正規化処理の正当性を形式的に検証する事を目標とし，本研究では型なし λ 計算に算術演算，if，let といったプリミティブを追加した，MinCaml のサブセットにあたる言語における K 正規化処理の正当性を，定理証明支援系の一つである Coq を用いて検証する．

また，本研究ではコンパイラの検証をする上での負担を和らげるべく，対象言語の意味論を余帰納的大ステップ意味論を用いて定義し，束縛を de Bruijn インデックスを用いて表現する．これにより，停止しないかもしれないプログラムの正当性の定理証明においても，理論的な簡単さと証明の簡潔さの両立が可能となる．

コンパイラの正当性の証明は対象言語に対するスケーラビリティが低くなりがちであるが，本研究では Coq の証明自動化に関する機能を用いる事でこの問題の解決を図る．これにより，型なし λ 計算における K 正規化の正当性の証明に 110 行のスクリプトを要するのに対し，対象言語に算術演算，if，let といったプリミティブを追加した場合にも正当性の証明に要するスクリプトは 141 行に留められる．

謝辞

圧倒的感謝

目次

第 1 章

序論

1.1 研究背景

プログラムのバグが計り知れない社会的損失をもたらす事は良く知られており，ソフトウェア開発においてデバッグは不可欠な存在となっている．中でもコンパイラのバグは深刻であり，コンパイラにバグがあれば与えられたソースコードにバグが無くてもバグを含んだコードが出力されてしまう．このようにして発生したバグを発見するにはソースコードを精査するだけではなく，コンパイラによって生成されたコードを確認する必要があるため，ソースコード由来のバグを発見するより困難である．

そのため定理証明支援系を用いたコンパイラの正当性の証明が盛んに試みられてきた．その中でも成功した研究として，Leroy らによる CompCert [1] が挙げられる．これは C 言語のほとんどをカバーする機能を持ったサブセットのコンパイラを Coq によって検証する試みである．C 言語には IO の機能が含まれているため，CompCert の対象言語にも IO の機能が含まれている．その一方，C 言語の関数は複数の引数を取れるが，CompCert の対象言語は部分式に関数呼び出しを許さないようにして要素数が可変の構文に制限を設けている．

本研究と類似した，純粋でない関数型言語を対象言語としたコンパイラの検証には Chlipala の研究 [2] が挙げられる．その対象言語は高階関数，参照，二つ組といった機能を有している一方，組を二つ組に限定して要素数が可変の構文を含まないようになっている他，IO を扱う機能も存在しない．

検証を難しくするために要素数が可変な構文や IO といった対象言語の機構は忌避されがちであるが，住井による教育用コンパイラ MinCaml [3] は非純粋な関数型言語を対象とするほか，これらの機構を備えている．もし MinCaml を検証できれば，そこで得られる知見は対象言語に同様の機構を持つ処理系を検証する際にも有用であると考えられる．

1.2 本研究の貢献

将来的に MinCaml における K 正規化処理の正当性を形式的に検証する事を目標とし，本研究では型なし λ 計算に算術演算，if，let といったプリミティブを追加した，MinCaml のサブセットにあたる言語における K 正規化処理の正当性を，定理証明支援系の一つである Coq を用いて検証する．その際，Coq の証明自動化に関する機能を用いる事で対象言語に対する証明のスケーラビリティを確保する．

また，証明のスケーラビリティを評価するため，型なし λ 計算を対象言語とした K 正規化処理の正当性も検証し，対象言語に算術演算，if，let といったプリミティブを追加した場合と証明の行数を比較する．

第 2 章

MinCaml

住井による教育用コンパイラ [3]

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

2.1 対象言語

MinCaml の対象言語

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

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

2.2 内部設計

- 様々なコンパイルフェーズ
- 疎結合

2.3 K 正規化

全ての部分式に名前を付ける

$$a + b * c + d$$

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

束縛に関する操作
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

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

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

定理 2.1 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 2.2 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

第 3 章

束縛の表現

3.1 名前による表現

α 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

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

3.2 de Bruijn 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

$$\lambda x.\lambda y.\lambda z.\ xz(yz)$$

$$\lambda.\lambda.\lambda.\ 2\ 0\ (1\ 0)$$

α 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

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

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

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

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

K 正規化の実装

```
1 Fixpoint knormal e :=
2   match e with
3   | Exp.Var x => K.Var x
4   | Exp.Abs e => K.Abs (knormal e)
5   | Exp.App e1 e2 =>
6       K.Let (knormal e1)
7         (K.Let (shift 1 (knormal e2))
8           (App 1 0))
9   end.
```

3.3 その他の表現方法

PHOAS

Locally nameless representation

第 4 章

意味論

4.1 小ステップ意味論

4.2 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

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

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

意味論

$$\frac{}{b \Downarrow b}$$

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

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

余帰納的大ステップ意味論 (1/2)

余帰納的に定義 [Leroy 及び Grall 2009]

$$\frac{t_1 \Uparrow}{\overline{t_1\ t_2 \Uparrow}}$$

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

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

エラー

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

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

第 5 章

Coq による形式的検証

5.1 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

```
1  Inductive t :=
2    | Var : nat -> t
3  ⋮
4  Proof.
5    intros t.
6    induction t.
7    Case "Var".
```

```
1  Inductive t :=
2    @r{| Int : Z -> t}@
3    | Var : nat -> t
4  ⋮
5  Proof.
6    intros t.
7    induction t.
8    @r{Case "Int".}@
9      ⋮
10   Case "Var".
```

Coq の証明自動化機能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.
```

5.2 正当性の検証

5.3 言語の拡張

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

第 6 章

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

第 7 章

議論・関連研究

7.1 BNF の書き方の例

本節では、BNF によるプログラミング言語の構文の書き方を紹介する。構文木の書き方は一つというわけではないので、幾つかのバリエーションを紹介する。どの方法が良いと思うかは、個人の好みに依るところなので、好きなものを使えば良いと思う。

まず、次の方法では、array 環境を使って、BNF を書いている。array 環境は数式環境中で表のようなものを書くときに使う。基本的に、table 環境と使い方は同じである。

$t ::=$		terms:
	x	variables
	$\lambda x.\ t$	lambda abstraction
	$t_1\ t_2$	application
	true	true
	false	false
	if t_1 **then** t_2 **else** t_3	if statement

他にも、次のように、align 環境を使っても、似たようなものを書くことができる。

$$
\begin{aligned}
t ::= \quad & && \text{terms:} \\
& x && \text{variables} \\
& \lambda x.\ t && \text{lambda abstraction} \\
& t_1\ t_2 && \text{application} \\
& \textbf{true} && \text{true} \\
& \textbf{false} && \text{false} \\
& \textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3 && \text{if statement}
\end{aligned}
$$

array 環境を愚直に使う場合と比べて、式が中央揃えになるという点と、“variables” とかの説明が右端に来ている点が違う。説明は tag* マクロで出しており、これはもともと式番号を指定するためのものなので、若干使い方がおかしい気もするが、まぁ、いいだろう。自分の好みの方を使うと良いだろう。

BNF 全体を左揃えにしたいならば、次のように、flalign 環境を使うと良い。align 環境と違って、`&` を余分に 1 つ付ける必要がある、ということに注意して欲しい（詳しくはソースコードを見よ）。

$t ::=$	terms:
x	variables
$\lambda x.\ t$	lambda abstraction
$t_1\ t_2$	application
true	true
false	false
if t_1 **then** t_2 **else** t_3	if statement

7.2 導出木の書き方の例

導出木の書き方も色々あるが、ここでは、bussproofs.sty を使った方法を紹介する。導出木は、手書きでも書きにくいが、LaTeX だから書きやすいというわけでもなく、（使うパッケージにも依るが）そこそこの苦労は必要である。bussproofs.sty を除く多くの方法では、frac などをベースに「分数」で導出木を書く。bussproofs.sty はこれらとは全く異なるインタフェースであり、慣れれば比較的解りやすい。bussproofs.sty の動作は、（導出木を要素とする）スタックをイメージすると解りやすい。よく使うマクロは次の通り。

- `\AxiomC{...}`：Axiom を push する（導出木では葉に相当）
- `\UnaryInfC{...}`：スタックから部分導出木（仮定）を 1 つ pop して、それを新たに作ったノード（結論）の子供にすることで、新たな部分導出木を作成し、push する。
- `\BinaryInfC{...}`：スタックから部分導出木（仮定）を 2 つ pop して、`\UnaryInfC` と同様の動作を行う。
- `\TrinaryInfC{...}`：スタックから部分導出木（仮定）を 3 つ pop して、`\UnaryInfC` と同様の動作を行う。

実際の使い方は以下の通り。

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{T-Var}$$

$$\frac{\Gamma, x : T \vdash t : U}{\Gamma \vdash \lambda x.\ t : T \to U}\ \text{T-Abs}$$

$$\frac{\Gamma \vdash t_1 : T \to U \qquad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1\ t_2 : U}\ \text{T-App}$$

$$\dfrac{\dfrac{\dfrac{}{x : \mathbf{Bool} \to \mathbf{Bool} \vdash \mathbf{true} : \mathbf{Bool}}\ \text{T-True}}{\vdash \lambda x.\ \mathbf{true} : (\mathbf{Bool} \to \mathbf{Bool}) \to \mathbf{Bool}}\ \text{T-Abs} \qquad \dfrac{\dfrac{y : \mathbf{Bool} \in y : \mathbf{Bool}}{y : \mathbf{Bool} \vdash y : \mathbf{Bool}}\ \text{T-Var}}{\vdash \lambda y.\ y : \mathbf{Bool} \to \mathbf{Bool}}\ \text{T-Abs}}{\vdash (\lambda x.\ \mathbf{true})\ (\lambda y.\ y) : \mathbf{Bool}}\ \text{T-App}$$

7.3 定理環境

この論文クラスファイルでは、デフォルトで以下の定理環境を提供している。

定理 7.1 (定理のタイトル) 定理の内容

補題 7.2 (補題のタイトル) 補題の内容

系 7.3 (系のタイトル) 系の内容

命題 7.4 (命題のタイトル) 命題の内容

定義 7.5 (定義のタイトル) 定義の内容

例 7.6 (例のタイトル) 例の内容

仮定 7.7 (仮定のタイトル) 仮定の内容

公理 7.8 (公理のタイトル) 公理の内容

証明 7.9 (証明のタイトル) 証明の内容 □

証明 (証明のタイトル) 証明の内容（番号なしの証明環境。証明を `\ref` で参照する必要がないなら、こっちを使うほうが自然かも） □

7.3.1 定理環境の使い方の例

補題 7.10 論文の中で最重要とは言えないような性質・命題は補題 (lamma) にする。補題や定理から直ちに導けるような軽い命題は系 (corollary) にする（細かい使い分けは人による）。

証明 `proof*` のように、アスタリスク付きの環境では、番号が付かない。

定理 7.11 提案手法の最も重要な性質や命題は、定理 (theorem) として書く。読者の心をくすぐる興味深いステートメントを書こう。

証明 定理 7.11 の華麗な証明。その美しい証明に、読者の目は釘付けだ！

Case 1. 自明

Case 2. 補題 7.10 から直ちに導ける。

Case 3. 言うまでもない。目を瞑れば証明が見えてくる。

Case 4. あんまり自明じゃない

(i) 自明じゃないと思ったけど、やっぱり自明だった

(ii) ほらね、こんなに簡単

7.4 ソースコード

ソースコード 7.1 は二分木を深さ優先探索して、ノードを列挙する関数である。

ソースコード 7.1　二分木のノードのリストアップ

```
1 type 'a bin_tree =
2   | Leaf of 'a
3   | Node of 'a bin_tree * 'a bin_tree
4
5 let rec listup_nodes = function
6   | Leaf x -> [x]
7   | Node (r, l) -> (listup_nodes r) @ (listup_nodes l)
```

ソースコードの書き方等については slide ブランチの slide.tex を参照されたし。

第 8 章

結論

参考文献

[1] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006: Int. Symp. on Formal Methods*, Vol. 4085 of *Lecture Notes in Computer Science*, pp. 460–475. Springer, 2006.

[2] Adam Chlipala. A verified compiler for an impure functional language. In *POPL'10: Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2010.

[3] Eijiro Sumii. Mincaml: a simple and efficient compiler for a minimal functional language. In *Proceedings of the 2005 workshop on Functional and declarative programming in education, Tallinn, Estonia, September 25 - 25, 2005*, pp. 27–38, 2005.