平成 27 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

平成 28 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

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の K 正規化の正当性を，定理証明支援系の一つである Coq を用いて検証する．

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り，IO，要素数が可変の構文を含んでいる．もし将来 MinCaml を検証できた場合，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な知見が得られると考えられる．本研究では MinCaml のコンパイルフェーズの中でも，束縛の付け替えが伴う非自明なプログラム変換である K 正規化処理に着目する．

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．これによって，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．

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．

そのためコンパイラの正当性の形式的検証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] [4] 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

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

MinCaml は住井による教育用コンパイラ [3] [4] である．これは値呼び戦略で，純粋でない言語機能を含む関数型言語を対象言語としている．また，OCaml で 2000 行程度の簡潔な実装ではあるが，型推論を行い，中間言語に変換した後に定数畳み込み等の最適化を行うといった現実の関数型言語処理系と共通した点を多数持っている．

2.1 対象言語

図 2.1 に MinCaml の抽象構文を示す．MinCaml は一般的な関数型言語と同様に，整数，浮動小数点数，真偽値，関数等のプリミティブを持ち，関数は第一級である．

本論文では扱わないが，配列のような純粋でない言語機能，組や複数引数の関数といった要素数が可変の構文や，外部関数呼び出しのような IO をサポートする機能が存在する．これらは定理証明支援系を用いて正当性を検証する際に特に問題となると考えられる．

e	::=	式
	c	定数
	$op(e_1, \cdots, e_n)$	算術演算
	if e_1 **then** e_2 **else** e_3	条件分岐
	let $x = e_1$ **in** e_2	変数定義
	x	変数の読み出し
	let rec $x\ y_1\ \cdots\ y_n = e_1$ **in** e_2	再帰関数定義
	$e\ e_1\ \cdots\ e_n$	関数呼び出し
	$(e_1, \cdots, e_n)$	組の作成
	let $(x_1, \cdots, x_n) = e_1$ **in** e_2	組からの読み出し
	Array.create $e_1\ e_2$	配列の作成
	$e_1.(e_2)$	配列からの読み出し
	$e_1.(e_2) \leftarrow e_3$	配列への書き込み

図 2.1　MinCaml の抽象構文と型

2.2 内部設計

MinCaml の対象言語とコンパイル先のアセンブリには多くの隔たりが存在するが，様々な中間言語を設定し，

1. 型推論
2. K 正規化
3. クロージャ変換
4. 仮想機械語生成
5. レジスタ割り当て

という処理を順番に行うことにより，ギャップを一つずつ埋める設計になっている．これらの処理は疎結合になっているため，個別に検証した後に組み合わせる事も容易である．

2.3 K 正規化

本研究では MinCaml のコンパイルフェーズのうち，束縛の付け替えが伴う非自明なプログラム変換である K 正規化処理に注目する．K 正規化とは，型推論を経て明示的に型が付いた言語を中間表現の一つである K 正規形に変換する操作を指し，直観的には全ての部分式を変数に束縛して名前を付けた形式に相当する．図 2.2 に MinCaml の K 正規形の抽象構文を示す．

例えば，e_1 の K 正規形は $K(e_1)$，e_2 の K 正規形は $K(e_2)$ とすると，$e_1 + e_2$ の K 正規形は $\textbf{let}\ x_1 = K(e_1)\ \textbf{in let}\ x_2 = K(e_2)\ \textbf{in}\ x_1 + x_2$ と表される．

$$
\begin{array}{rl}
e & ::= \\
 & c \\
 & op(x_1, \cdots, x_n) \\
 & \textbf{if}\ x = y\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \\
 & \textbf{if}\ x \leq y\ \textbf{then}\ e_1\ \textbf{else}\ e_2 \\
 & \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \\
 & x \\
 & \textbf{let rec}\ x\ y_1\ \cdots\ y_n = e_1\ \textbf{in}\ e_2 \\
 & x\ y_1\ \cdots\ y_n \\
 & (x_1, \cdots, y_n) \\
 & \textbf{let}\ (x_1, \cdots, x_n) = y\ \textbf{in}\ e \\
 & x.(y) \\
 & x.(y) \leftarrow z
\end{array}
$$

図 2.2　MinCaml の K 正規形

実際の MinCaml では K 正規化と同時に様々なプログラム変換が行われているが，本論文では簡単化のために扱わない．

第 3 章

構文の定義

Coq を用いて K 正規化の正当性を証明するためには，まず対象言語の構文を Coq 上で定義しなくてはならない．対象言語の構文は意味論の定義，ひいては K 正規化の正当性の証明にも関わるため，構文を適切に定義する事によって証明を簡潔にできる．

3.1 名前による束縛の表現の問題点

定理証明支援系をを用いてコンパイラの検証をする際，構文の定義の中でもとりわけ検証の難易度を左右するのが束縛の表現方法である．紙の上で定式化を行う場合，束縛の表現方法として最も良く用いられるのは名前による表現と考えられる．

しかし，定理証明支援系を用いて検証を行う際，名前による束縛の表現には問題点が多い．名前による表現では構文的に等価な式以外にも，$\lambda x.\lambda y.\ x$ と $\lambda a.\lambda b.\ a$ のように変数名を付け替えた式同士も等価な式とみなして扱う必要がある．このような α 等価性を定式化し，関連する補題を証明する負担が無視できない．

また，$[x \mapsto z](\lambda z.\ x)$ のような代入を単純に計算すると $\lambda z.\ z$ となるが，これでは代入された z に対応している束縛が変わってしまう．このような事が起こらないよう，$\lambda z'.\ z$ のように適宜変数名を付け替える必要があるが，これも定理証明支援系を用いて定式化する場合に煩雑となる．

3.2 de Bruijn インデックス

このような名前による表現の欠点を改善した束縛の表現方法として，de Bruijn インデックスが挙げられる．これは変数の位置から数えて何番目の束縛に対応するかで束縛を表現するものである．例えば，名前によって束縛を表現した式 $\lambda x.\lambda y.\lambda z.\ x\ z\ (y\ z)$ を de Bruijn インデックスで表現すると $\lambda.\lambda.\lambda.\ 2\ 0\ (1\ 0)$ となる．

名前による表現では構文的に等価な式以外にも，α 等価な式も等価として扱わなければならなかった．一方 de Bruijn インデックスでは α 等価な式は全て構文的に等価となるため，定理証明支援系での扱いに適する．例えば，$\lambda x.\lambda y.\ x$ と $\lambda a.\lambda b.\ a$ のように α 等価な式は，de Bruijn イン

デックスで表現するとどちらも $\lambda.\lambda.\ 1$ となる．

また，束縛の表現に変数名を用いないため，変数名の選び方によって束縛の対応関係が変わる事も無い．

de Bruijn インデックスで束縛を表現した場合に用いられる操作の一つに，シフトが存在する．これは直観的には全ての自由変数のインデックスを指定されただけずらす事に相当する．

シフトが必要になる場面として，束縛に関する操作が挙げられる．例えば，名前による表現では $(\lambda x.\ x)\ (\lambda x.\ y)$ に相当する de Bruijn インデックスで束縛を表現した式を K 正規化する事を考える．名前による表現では，変数名を慎重に選べば $\textbf{let}\ a = \lambda x.\ x\ \textbf{in let}\ b = \lambda x.\ y\ \textbf{in}\ a\ b$ のように単純に let を挿入するだけで K 正規形が得られる．一方，de Bruijn インデックスによる表現では let が挿入されると束縛の対応関係がずれてしまうため，これを修正するためにシフトが必要となる．式 t を d だけシフトする事を $\uparrow^d t$ と書くと，de Bruijn インデックスで束縛を表現した式 $(\lambda.\ 0)\ (\lambda.\ 1)$ の K 正規形は

$$\begin{aligned}&\textbf{let}\ \_ = \lambda.\ 0\ \textbf{in let}\ \_ = \uparrow^1 (\lambda.\ 1)\ \textbf{in}\ 1\ 0\\ &= (\textbf{let}\ \_ = \lambda.\ 0\ \textbf{in let}\ \_ = \lambda.\ 2\ \textbf{in}\ 1\ 0)\end{aligned}$$

となる．

3.3 対象言語の構文の定義

本研究における対象言語は MinCaml のサブセットに相当するが，MinCaml のサブセットのままでは意味論の定義に適さない部分がある．次章で定義するような環境を用いない大ステップ意味論では，値は式のサブセットである必要が生じる．しかし，MinCaml には再帰関数の定義を行う構文はあるものの，関数の値に相当する構文が存在せずこれを満たさない．

従って，本研究で扱う対象言語では再帰関数の定義を行う構文に代わって，匿名関数を表す構文を導入する．図 3.1 に本研究の対象言語の抽象構文を示す．

e	::=	式
	c	定数
	$op(e_1, \cdots, e_n)$	算術演算
	$\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3$	条件分岐
	$\textbf{let}\ \_ = e_1\ \textbf{in}\ e_2$	変数定義
	k	変数 (インデックスで表現)
	$\lambda.\ e$	匿名関数
	$e_1\ e_2$	関数呼び出し

v	::=	値
	c	定数
	$\lambda.\ e$	関数

図 3.1　本研究で扱う対象言語の抽象構文

また，本研究の対象言語におけるシフト $\uparrow^d e$ の定義を図 3.2，並列代入 $[k \mapsto \overline{e}]e$ の定義を図 3.3

に示す.

$$
\begin{array}{lcl}
\uparrow^d_{\mathcal{C}} c & = & c \\
\uparrow^d_{\mathcal{C}} (op(e_1, \cdots, e_n)) & = & op(\uparrow^d_{\mathcal{C}} e_1, \cdots, \uparrow^d_{\mathcal{C}} e_n) \\
\uparrow^d_{\mathcal{C}} (\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) & = & \textbf{if } \uparrow^d_{\mathcal{C}} e_1 \textbf{ then } \uparrow^d_{\mathcal{C}} e_2 \textbf{ else } \uparrow^d_{\mathcal{C}} e_3 \\
\uparrow^d_{\mathcal{C}} (\textbf{let } \_ = e_1 \textbf{ in } e_2) & = & \textbf{let } \_ = \uparrow^d_{\mathcal{C}} e_1 \textbf{ in } \uparrow^d_{\mathcal{C}+1} e_2 \\
\uparrow^d_{\mathcal{C}} k & = & \begin{cases} k & k < \mathcal{C} \\ k + d & \text{otherwise} \end{cases} \\
\uparrow^d_{\mathcal{C}} (\lambda.\ e) & = & \lambda.\ \uparrow^d_{\mathcal{C}+1} e \\
\uparrow^d_{\mathcal{C}} (e_1\ e_2) & = & (\uparrow^d_{\mathcal{C}} e_1)\ (\uparrow^d_{\mathcal{C}} e_2)
\end{array}
$$

図 3.2　シフトの定義

$$
\begin{array}{lcl}
[k \mapsto \overline{e}]c & = & c \\
[k \mapsto \overline{e}]\,(op(e_1, \cdots, e_n)) & = & op([k \mapsto \overline{e}]e_1, \cdots, [k \mapsto \overline{e}]e_n) \\
[k \mapsto \overline{e}]\,(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) & = & \textbf{if } [k \mapsto \overline{e}]e_1 \textbf{ then } [k \mapsto \overline{e}]e_2 \textbf{ else } [k \mapsto \overline{e}]e_3 \\
[k \mapsto \overline{e}]\,(\textbf{let } \_ = e_1 \textbf{ in } e_2) & = & \textbf{let } \_ = [k \mapsto \overline{e}]e_1 \textbf{ in } [k+1 \mapsto \overline{e}]e_2 \\
[k \mapsto \overline{e}]\,k' & = & \begin{cases} k' & k' < k \\ \uparrow^k \overline{e}_{k'-k} & k \le k' \wedge k' < k + |\overline{e}| \\ k' - |\overline{e}| & \text{otherwise} \end{cases} \\
[k \mapsto \overline{e}]\,(\lambda.\ e) & = & \lambda.\ [k+1 \mapsto \overline{e}]e \\
[k \mapsto \overline{e}]\,(e_1\ e_2) & = & ([k \mapsto \overline{e}]e_1)\ ([k \mapsto \overline{e}]e_2)
\end{array}
$$

図 3.3　代入の定義

前にも見たように，シフトを用いる事で K 正規化を実装する事が出来る．式 e の K 正規化 $K(e)$ の実装を図 3.4 に示す.

$$
\begin{array}{lcl}
K(c) & = & c \\
K(op(e_1, \cdots, e_n)) & = & \\
 & & \textbf{let } \_ = K(e_1) \textbf{ in} \\
 & & \textbf{let } \_ = \uparrow^1 K(e_2) \textbf{ in} \\
 & & \vdots \\
 & & \textbf{let } \_ = \uparrow^{n-1} K(n) \textbf{ in} \\
 & & op(n-1, \cdots, 0) \\
K(\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3) & = & \textbf{let } \_ = K(e_1) \textbf{ in if } 0 \textbf{ then } K(e_2) \textbf{ else } K(e_3) \\
K(\textbf{let } \_ = e_1 \textbf{ in } e_2) & = & \textbf{let } \_ = K(e_1) \textbf{ in } K(e_2) \\
K(k) & = & k \\
K(\lambda.\ e) & = & \lambda.\ K(e) \\
K(e_1\ e_2) & = & \textbf{let } \_ = K(e_1) \textbf{ in let } \_ = K(e_2) \textbf{ in } 1\ 0
\end{array}
$$

図 3.4　K 正規化の定義

第 4 章

意味論

コンパイラの正当性を検証する際には対象言語の意味論を定義する必要がある．MinCaml の意味論は形式的に定義されていないため，本研究では MinCaml の非形式的な意味論を基に対象言語の意味論を定義する．

4.1 大ステップ操作的意味論の問題点

本研究の対象言語の意味論を定義するにあたって，大ステップ操作的意味論を採用する事にする．大ステップ操作的意味論の評価規則は構文とほぼ一対一に対応しており，比較的単純なプログラム変換の検証には小ステップ操作的意味論より適している．本研究の対象言語の評価規則を図 4.1 に示す．

$$\frac{}{c \Downarrow c}\ \text{E-Const}$$

$$\frac{e_1 \Downarrow v_1\ \cdots\ e_n \Downarrow v_n \qquad op(v_1, \cdots, v_n) = v}{op(e_1, \cdots, e_n) \Downarrow v}\ \text{E-Op}$$

$$\frac{e_1 \Downarrow \textbf{true} \qquad e_2 \Downarrow v}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \Downarrow v}\ \text{E-IfTrue}$$

$$\frac{e_1 \Downarrow \textbf{false} \qquad e_3 \Downarrow v}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \Downarrow v}\ \text{E-IfFalse}$$

$$\frac{e_1 \Downarrow v_1 \qquad [0 \mapsto v_1]e_2 \Downarrow v}{\textbf{let}\ \_ = e_1\ \textbf{in}\ e_2 \Downarrow v}\ \text{E-Let}$$

$$\frac{}{\lambda.\ e \Downarrow \lambda.\ e}\ \text{E-Abs}$$

$$\frac{e_1 \Downarrow \lambda.\ e \qquad e_2 \Downarrow v_2 \qquad [0 \mapsto v_2]e \Downarrow v}{e_1\ e_2 \Downarrow v}\ \text{E-App}$$

図 4.1　本研究の対象言語の評価規則

その一方で，停止しない評価と行き詰まり状態の区別が困難である問題がある．例えば，図 4.1 で定義した意味論では，適用できる規則が存在しないため，$\forall v.\ \mathbf{true}\ \mathbf{true} \not\Downarrow v$ が成り立つ．一方，帰納的定義では無限に導出規則が続くような命題は導出できないため，$\forall v.\ (\lambda x.\ x)\ (\lambda x.\ y) \not\Downarrow v$ が成り立つ．このように，関係 $e \Downarrow v$ だけでは停止しない評価と行き詰まり状態を区別できない．

4.2 余帰納的大ステップ意味論

大ステップ操作的意味論においても停止しない評価と行き詰まり状態を区別する試みとして，Leroy と Grall によって余帰納的大ステップ意味論が提案されている．[5] これは，評価が停止しない事を表す述語を余帰納的に定義する事で，従来の評価規則を補うものである．

余帰納的大ステップ意味論を用いた本研究の対象言語の評価規則を図 4.2 に示す．

$$\frac{e_1 \Downarrow v_1 \ \cdots\ e_m \Uparrow}{op(e_1, \cdots, e_n) \Uparrow}\ \text{D-Op}$$

$$\frac{e_1 \Uparrow}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Uparrow}\ \text{D-If}$$

$$\frac{e_1 \Downarrow \mathbf{true} \qquad e_2 \Uparrow}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Uparrow}\ \text{D-IfTrue}$$

$$\frac{e_1 \Downarrow \mathbf{false} \qquad e_3 \Uparrow}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \Uparrow}\ \text{D-IfFalse}$$

$$\frac{e_1 \Uparrow}{\mathbf{let}\ \_ = e_1\ \mathbf{in}\ e_2 \Uparrow}\ \text{D-LetL}$$

$$\frac{e_1 \Downarrow v_1 \qquad [0 \mapsto v_1]e_2 \Uparrow}{\mathbf{let}\ \_ = e_1\ \mathbf{in}\ e_2 \Uparrow}\ \text{D-LetR}$$

$$\frac{e_1 \Uparrow}{e_1\ e_2 \Uparrow}\ \text{D-AppL}$$

$$\frac{e_1 \Downarrow \lambda.\ e \qquad e_2 \Uparrow}{e_1\ e_2 \Uparrow}\ \text{D-AppR}$$

$$\frac{e_1 \Downarrow \lambda.\ e \qquad e_2 \Downarrow v_2 \qquad [0 \mapsto v_2]e \Uparrow}{e_1\ e_2 \Uparrow}\ \text{D-App}$$

図 4.2　余帰納的大ステップ意味論による本研究の対象言語の評価規則

これらの規則を用いると，余帰納的定義では無限回の規則適用が可能なため，$(\lambda x.xx)(\lambda x.xx) \Uparrow$ が成り立つ．一方，適用できる規則が存在しないため，$\mathbf{true}\ \mathbf{true} \not\Uparrow$ が成り立つ．このように，停止しない評価と行き詰まり状態を区別できる．

第 5 章

Coq による形式的検証

本研究で扱う対象言語の構文及び意味論が定義できたため，Coq を用いて K 正規化の正当性を形式的に検証する．その際，Coq の証明自動化機能を活用することによって証明が対象言語に対してスケールするよう図る．また，型無し λ 計算における K 正規化の正当性も証明し，証明に要したスクリプトの行数を比較する事により，実際のスケーラビリティの評価を行う．

5.1 対象言語や K 正規化におけるメタ定理

Coq によって正当性を検証するにあたって，K 正規化処理で期待される正当性はどのような命題で表されるか，その命題はどのようにして証明すれば良いかについて考える．

5.1.1 シフト及び代入についてのメタ定理

シフト及び代入について，以下のようなメタ定理が成り立つ．

- $\forall c\ e.\ \uparrow_c^0 e = e$
- $\forall c\ c'\ d\ d'\ e.\ c \leq c' \leq c + d \Longrightarrow \uparrow_{c'}^{d'} \uparrow_c^d e = \uparrow_c^{d+d'} e$
- $\forall c\ c'\ d\ d'\ e.\ c' \leq c \Longrightarrow \uparrow_{c'}^{d'} \uparrow_c^d e = \uparrow_{d'+c}^{d} \uparrow_{c'}^{d'} e$
- $\forall c\ d\ e\ k\ \overline{e}.\ c \leq k \Longrightarrow \uparrow_c^d [k \mapsto \overline{e}] e = [d + k \mapsto \overline{e}] \uparrow_c^d e$
- $\forall c\ d\ e\ k.\ c \leq k \wedge |\overline{e}| + k \leq d + c \Longrightarrow [x \mapsto |\overline{e}|] \uparrow_c^d e = \uparrow_c^{d-|\overline{e}|} e$

これらは式 e についての帰納法と大小比較についての場合分けで証明できる．

大ステップ操作的意味論においては代入の代わりに環境を用いて意味論を定義する事もできるが，その場合クロージャが存在するために同様の補題を定義するのが難しい．従って，本研究では代入を用いた大ステップ操作的意味論によって対象言語の意味論を定義する．

5.1.2 K 正規化の正当性

K 正規化に期待される性質について考える．

K 正規化後の式を評価してみると，例えば $1+2$ のような評価後に整数値を取るような式は，K 正規化後の式 **let** $a=1$ **in let** $b=2$ **in** $a+b$ も共に 3 に評価される．この事から一見 K 正規化後の式を評価した結果は K 正規化前の式を評価した結果と一致するように思われるが，これには反例が存在する．

例えば，$\lambda x.\lambda y.\ x+y+z$ とその K 正規形 $\lambda x.\lambda y.\lambda z.$ **let** $a=x+y$ **in** $a+z$ を評価する事を考える．値評価では関数の中身は簡約されないため，前者は $\lambda x.\lambda y.\ x+y+z$ に評価される一方，後者は $\lambda x.\lambda y.\lambda z.$ **let** $a=x+y$ **in** $a+z$ となって一致しない．しかし，両者の評価結果を比較すると，定理 5.1 が成り立つ事が予想される．

定理 5.1 式 e が値 v に評価される場合，式 e を K 正規化した結果 K(e) は値 v を K 正規化した結果 K(v) に評価される

また，定理 5.2 も成り立つ事が予想される．

定理 5.2 式 e の評価が停止しない場合，式 t を K 正規化した結果 K(e) の評価は停止しない

これらは評価の導出に関する帰納法で証明できる．その際，以下のような補題が必要となる．

- $\forall c\ d\ e.\ K(\uparrow_c^d e) = \uparrow_c^d K(e)$
- $\forall e\ k\ \bar{e}.\ K([k \mapsto \bar{e}]e) = [k \mapsto K(\bar{e})]K(e)$

5.2 素朴な検証の問題点

コンパイラの正当性の証明を証明する際，自動証明を用いないような素朴な検証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．例えば，$\forall e\ v\ v'.e \Downarrow v \wedge e \Downarrow v' \Longrightarrow v = v'$ のような命題を式 e の構造に関する帰納法で証明する事を考える．自動証明を用いない場合，対象言語が型無し λ 計算ならば 3 通りの場合分けが必要であるが，対象言語に真偽値の定数を追加すると 16 通りもの場合分けが必要となる．自動証明を用いない場合，全ての場合分けに対して個別に証明を書かなければならないため，素朴な検証ではすぐに破綻してしまう事が予想される．

5.3 Coq による証明の方針

本研究では，Coq の証明自動化機能を用いる事で構文の違いの吸収を図った．先ほどの例を自動証明を活用して証明するとソースコード 5.1 のようになる．

ソースコード 5.1　自動証明の活用例

```
1 Lemma evalto_decidable : forall e v1 v2,
2   evalto e v1 -> evalto e v2 -> v1 = v2.
3 Proof.
4   intros e v1 v2 Hevalto1.
```

```
5    generalize dependent v2.
6    induction Hevalto1;
7      intros v2_ Hevalto2;
8      inversion Hevalto2;
9      clear Hevalto2;
10     repeat (subst; match goal with
11     | [ H : Abs _ = Abs _ |- _ ] => inversion H; clear H
12     | [ Hevalto : evalto ?t ?v,
13         IH : forall v, evalto ?t v -> _ = v |- _ ] =>
14         generalize (IH _ Hevalto); intros; clear Hevalto
15     end); congruence.
16 Qed.
```

Coq において証明は，与えられた結論に対して仮定の導入や場合分け等の操作に相当するタクティックを適用する事で進行する．[6] $tactic_1; tactic_2$ と書くと二つのタクティックを合成する事もできる．この場合 $tactic_1$ が現在のサブゴールに適用され，その結果生じた全てのサブゴールに $tactic_2$ が適用される．

また，`match goal with` 構文を使う事によって，サブゴールの形によって場合分けを行う事もできる．

これらの様な機能によって，対象言語が変更された場合にも方針に従って証明を進める事ができ，自明な変更に対しては殆ど証明を変更せずに無く検証を行う事も可能である．

5.4 言語の拡張に対するスケーラビリティの評価

言語の拡張に対する証明のスケーラビリティを評価するため，型無し λ 計算における K 正規化の正当性を検証した後，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て証明の再利用性を評価する．

対象言語の拡張前後で証明の行数を比較すると，型無し λ 計算における検証では 110 行を要したのに対し，対象言語を拡張した後では 141 行に留まった．

証明の内容について比較すると，シフトや代入についての補題のように，式の構造に関する帰納法で証明できるような単純な補題については変更する事無く再利用できた．K 正規化の正当性のように複雑な証明については変更を強いられたが，サブゴールについての場合分けを追加する事で対処が可能であった．

第 6 章

議論・関連研究

本研究では束縛の表現に de Bruijn インデックスを用いたが，名前による束縛の表現の欠点を改善した他の手法として，パラメトリック高階抽象構文 (PHOAS) [7] や局所名前無し表現 [8] などが挙げられる．

ホスト言語の機能を用いた束縛の表現方法として，高階抽象構文 (HOAS) が知られている．例えば，HOAS によって束縛を表現した場合，$\lambda x.\ x$ を表す抽象構文木は `Abs (fun x => x)` となる．PHOAS は HOAS を発展させたものであり，Coq のような定理証明支援系に適用できなかった問題が修正されている．

PHOAS を束縛の表現に用いた場合，名前による束縛の表現で問題となった名前の付け替えをホスト言語に任せられる利点がある．加えて，代入もホスト言語の機能を用いて簡単に表せる．

一方，PHOAS によって束縛を表現した言語の性質を証明しようとすると，ホスト言語の機能を用いているために必要以上に一般的な命題を証明しなければならない．また，PHOAS の理論も複雑であるため，証明を書くのが困難となる．

局所名前無し表現は名前による束縛の表現と de Bruijn インデックスの折衷案であり，自由変数は名前によって表現し，束縛変数は de Bruijn インデックスによって表現する．束縛変数を de Bruijn インデックスで表したため，α 同値性を考えなくて良い，代入を行う際に変数名の付け替えを行わなくて良い等の利点が存在する．また，自由変数は名前によって表したため，シフトが不要になる，束縛の付け替えが伴う操作の実装が de Bruijn インデックスより自然となる等の利点も存在する．

ただし，自由変数と束縛変数で構文を分けて扱う必要があり，それに伴って名前によって表現された変数に対する代入とインデックスによって表現された変数に対する代入に相当する操作もそれぞれ実装する必要がある．それに伴ってそれぞれに対して補題を証明しなければならないため，非常に煩雑である．また，インデックスによって表現された変数が本当に束縛変数になっているかも検証しなければならない．

これらの束縛の表現も比較検討したところ，de Bruijn インデックスが最も理論的な簡単さと証明の簡潔さのバランスに優れていると判断した．

第 7 章

結論

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の K 正規化の正当性を Coq で証明できた．その際 de Buijn インデックス，余帰納的大ステップ意味論を採用した事により証明を簡潔にできた．加えて，Coq の証明自動化に関する機能を用いた事によって，対象言語の変更に対して証明がスケールするようにできた．

本研究では対象言語を MinCaml のサブセットに限定しているが，今後は組や複数引数の関数といった要素数が可変の構文，配列や外部関数呼び出しのような副作用を伴う言語機能を含むように対象言語を拡張し，MinCaml と同等なものにする必要がある．

謝辞

本論文を作成するにあたり，指導教員の住井先生にはお忙しい中貴重な時間を割いて頂き，丁寧かつ熱心なご指導を賜りました．ここに感謝の意を表します．

また，要旨及び章立ての添削等，広範にわたってご指導頂いた松田先生に深く感謝いたします．

そして，多くの経験に裏打ちされた指摘を下さった Kiselyov 先生及び住井・松田研究室の先輩方，研究で悩んだ事があればすぐに相談に乗ってくれた同級生の皆様に感謝いたします．

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.

[4] 英二郎住井. MinCaml コンパイラ (ソフトウェア論文). コンピュータソフトウェア, Vol. 25, No. 2, pp. 28–38, apr 2008.

[5] Xavier Leroy and Hervé Grall. Coinductive big-step operational semantics. *Inf. Comput.*, Vol. 207, No. 2, pp. 284–304, 2009.

[6] The coq proof assistant reference manual, 2009.

[7] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pp. 143–156, New York, NY, USA, 2008. ACM.

[8] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning*, pp. 1–46, 2011. 10.1007/s10817-011-9225-2.