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

本論文では定理証明支援系の一つである Coq を用いた，教育用コンパイラ MinCaml におけるコンパイルステージの一つ，K 正規化の形式的検証について述べる．コンパイラのバグは生成したコードに波及するため一層高品質なコードが求められるが，現実の処理系は複雑であり形式的に検証するのは困難である．MinCaml は OCaml で 2000 行程度の簡潔な実装であるが，その対象言語は ML の本質となる機能を多数有しており，現実の処理系を検証する上で良い指標になると考えられる．

それでもなお定理証明支援系を用いた検証としては大規模な事に変わりない上，コンパイラの検証には対象言語が拡張された場合に今までの証明全てが修正を迫られる等の困難がある．本研究では対象言語の意味論を Coinductive big-step operational semantics を用いて定義し，束縛を de Bruijn indices を用いて表現する．これによって停止しないプログラムに対しても正当性を証明できるのみならず，小ステップ意味論を用いるより証明が簡潔となるほか，PHOAS や Locally nameless といった別の束縛の表現と比較して理論的な簡単さと証明の簡潔さを両立できる．これと併せて Coq の証明自動化に関する機能を用い，対象言語が拡張された場合に極力証明を修正しなくても良いようにする．

謝辞

圧倒的感謝

目次

第 1 章

序論

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いるが，中でもコンパイラのバグは生成したコードに波及するためより深刻である．login をコンパイルする際にバックドアを仕込むようなコンパイラを生成するコンパイラが配布されたために，初期の UNIX にはソースコード上は全く形跡が無いバックドアが存在した．[1]

このような理由からコンパイラを形式的に検証する意義は大きい．同様の試みには C 言語を対象言語とした CompCert [2] などがあるが，実用に供されている言語は歴史的経緯から複雑になっている事が多く，理論的な難しさ以上に検証が困難である．そこで，本研究では教育用コンパイラである MinCaml [3] を検証することにした．MinCaml は OCaml で 2000 行程度と簡潔な実装であるが，定数畳み込みやデッドコード削除などの最適化も実装されており，camlopt や gcc に匹敵する高速なコードを生成できる実用的な処理系である．o その対象言語には以下のような特徴があり，現実の ML 処理系を検証する上で参考になると考えられる．

- 非純粋な関数型言語である
- 外部関数の呼び出しや外部の配列へのアクセスができる
- タプルや複数引数の関数といった可変個の要素を持つ構文がある

第 2 章

MinCaml

MinCaml の対象言語

2.1 K 正規形

第 3 章

束縛の表現

3.1 名前による表現

3.2 de Bruijn indices

3.3 Parametric higher-order abstract syntax(PHOAS)

3.4 Locally nameless representation

第 4 章

意味論の定義

4.1　小ステップ意味論

4.2　大ステップ意味論

4.2.1　Coinductive big-step operational semantics

第 5 章

Coq による形式的検証

5.1 小ステップ意味論による定式化

5.2 Coinductive big-step operational semantic による定式化

第 6 章

本論

6.1　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。

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

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。

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

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

6.2 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。

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

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

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

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

6.3 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 6.1 (定理のタイトル) 定理の内容

補題 6.2 (補題のタイトル) 補題の内容

系 6.3 (系のタイトル) 系の内容

命題 6.4 (命題のタイトル) 命題の内容

定義 6.5 (定義のタイトル) 定義の内容

例 6.6 (例のタイトル) 例の内容

仮定 6.7 (仮定のタイトル) 仮定の内容

公理 6.8 (公理のタイトル) 公理の内容

証明 6.9 (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） □

6.3.1 定理環境の使い方の例

補題 6.10 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 6.11 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 6.11 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 6.10 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

6.4 ソースコード

ソースコード 6.1 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 6.1　二分木のノードのリストアップ

```
type 'a bin_tree =
  | Leaf of 'a
  | Node of 'a bin_tree * 'a bin_tree

let rec listup_nodes = function
  | Leaf x -> [x]
  | 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 7 章

結論

参考文献

[1] Ken Thompson. Reflections on trusting trust. *Commun. ACM*, Vol. 27, No. 8, pp. 761–763, 1984.

[2] 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.

[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.