ML 勉強会

信用できる言語

Standard ML

@fetburner

2016 年 7 月 9 日

自己紹介

- **ろんだ** (@fetburner)
- **青葉山に篭って** Coq **を書く** M1
- ML **が好き**

今日の発表内容

処理系の形式的検証についてのサーベイ

CakeML **の紹介**

動機

バグの無いプログラムが書きたい

$\Rightarrow$ テストを書こう!

テスト

”Program testing can be used to show the presence of bugs, but never to show their absence!” (E.W. Dijkstra)

テストだけでは力不足...

⇒ **形式手法の出番**

形式手法

プログラムの仕様を形式的に書き下して
証明

プログラムにバグがない事が数学的に保障

現実

プログラムの正しさは処理系の正しさに依存

証明付きのプログラムも間違ったコンパイラでコンパイルしては正しく動かない...

⇒ 処理系も検証しよう！

先行研究

CompCert[Leroy ら 2006]

- 言語処理系検証のマイルストーン
- 実用的な C コンパイラの検証
  - ANSI C の大部分をサポート
  - PowerPC、ARM、x86 のネイティブコードを出力
  - GCC に匹敵するパフォーマンス
- 大部分を Coq で検証

でも C 言語なんて書きたくないよね...

関数型言語の検証

ML 処理系を検証する試みも多数存在

- LambdaTamer[Clipala 2010]
- Pilsner[Neis ら 2015]
- CakeML[Kumar ら 2014]

今回は特に CakeML に注目

実用的な Standard ML 処理系の検証

- ブートストラップにも成功
- ARMv6、ARMv8、MIPS-64、RISC-V のネイティブコードを出力
- Poly/ML の数倍速い

HOL4 による執拗なまでの証明

対象言語

Standard ML のサブセット

- 高階関数
- 副作用
  - 参照
  - 例外
- ヴァリアント
- パターンマッチ
- モジュール
- 入出力

対象言語の文法

$$
\begin{array}{lcl}
id & ::= & x \mid Mn\,.\,x \\
cid & ::= & Cn \mid M\,.\,Cn \\
t & ::= & \texttt{int} \mid \texttt{bool} \mid \texttt{unit} \mid \alpha \mid id \mid t\ id \mid \texttt{(}t(\texttt{,}t)^*\texttt{)}id \\
 & \mid & t\ \texttt{*}\ t \mid t\ \texttt{->}\ t \mid t\ \texttt{ref} \mid \texttt{(}t\texttt{)} \\
l & ::= & i \mid \texttt{true} \mid \texttt{false} \mid \texttt{()} \mid \texttt{[]} \\
p & ::= & x \mid l \mid cid \mid cid\ p \mid \texttt{ref}\ p \mid \texttt{\_} \mid \texttt{(}p(\texttt{,}p)^*\texttt{)} \mid \texttt{[}p\,(\texttt{,}p)^*\texttt{]} \\
 & \mid & p\ \texttt{::}\ p \\
e & ::= & l \mid id \mid cid \mid cid\ e \mid \texttt{(}e\texttt{,}e\,(\texttt{,}e)^*\texttt{)} \mid \texttt{[}e\,(\texttt{,}e)^*\texttt{]} \\
 & \mid & \texttt{raise}\ e \mid e\ \texttt{handle}\ p\ \texttt{=>}\ e\ (\texttt{|}\ p\ \texttt{=>}\ e)^* \\
 & \mid & \texttt{fn}\ x\ \texttt{=>}\ e \mid e\ e \mid \texttt{((}e\ \texttt{;)}^*\ e\texttt{)} \mid uop\ e \mid e\ op\ e \\
 & \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{case}\ e\ \texttt{of}\ p\ \texttt{=>}\ e\ (\texttt{|}\ p\ \texttt{=>}\ e)^* \\
 & \mid & \texttt{let}\ (ld|\texttt{;})^*\ \texttt{in}\ (e\ \texttt{;})^*\ e\ \texttt{end} \\
ld & ::= & \texttt{val}\ x\ \texttt{=}\ e \mid \texttt{fun}\ x\ y^+\ \texttt{=}\ e\ (\texttt{and}\ x\ y^+\ \texttt{=}\ e)^* \\
uop & ::= & \texttt{ref} \mid \texttt{!} \mid \texttt{\~{}} \\
op & ::= & \texttt{=} \mid \texttt{:=} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{div} \mid \texttt{mod} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{<>} \mid \texttt{::} \\
 & \mid & \texttt{before} \mid \texttt{andalso} \mid \texttt{orelse} \\
c & ::= & Cn \mid Cn\ \texttt{of}\ t \\
tyd & ::= & tyn\ \texttt{=}\ c\ (\texttt{|}\ c)^* \\
tyn & ::= & (\alpha(\texttt{,}\alpha)^*)\ x \mid \alpha\ x \mid x \\
d & ::= & \texttt{datatype}\ tyd\ (\texttt{and}\ tyd)^* \mid \texttt{val}\ p\ \texttt{=}\ e \\
 & \mid & \texttt{fun}\ x\ y^+\ \texttt{=}\ e\ (\texttt{and}\ x\ y^+\ \texttt{=}\ e)^* \\
 & \mid & \texttt{exception}\ c \\
sig & ::= & \texttt{:>}\ \texttt{sig}\ (sl\,|\texttt{;})^*\ \texttt{end} \\
sl & ::= & \texttt{val}\ x\ \texttt{:}\ t \mid \texttt{type}\ tyn \mid \texttt{datatype}\ tyd\ (\texttt{and}\ tyd)^* \\
top & ::= & \texttt{structure}\ Mn\ sig^?\ \texttt{=}\ \texttt{struct}\ (d\,|\texttt{;})^*\ \texttt{end;} \mid d\texttt{;} \mid e\texttt{;}
\end{array}
$$

where x and y range over identifiers (must not start with a capital letter), α over SML-style type variables (*e.g.*, `'a`), Cn over constructor names (must start with a capital letter), Mn over module names, and i over integers.

なぜStandard MLか？

厳密な言語仕様

OCaml

```
(print_string"Mt.␣";1)
+(print_string"Aoba";2)
```

未定義動作

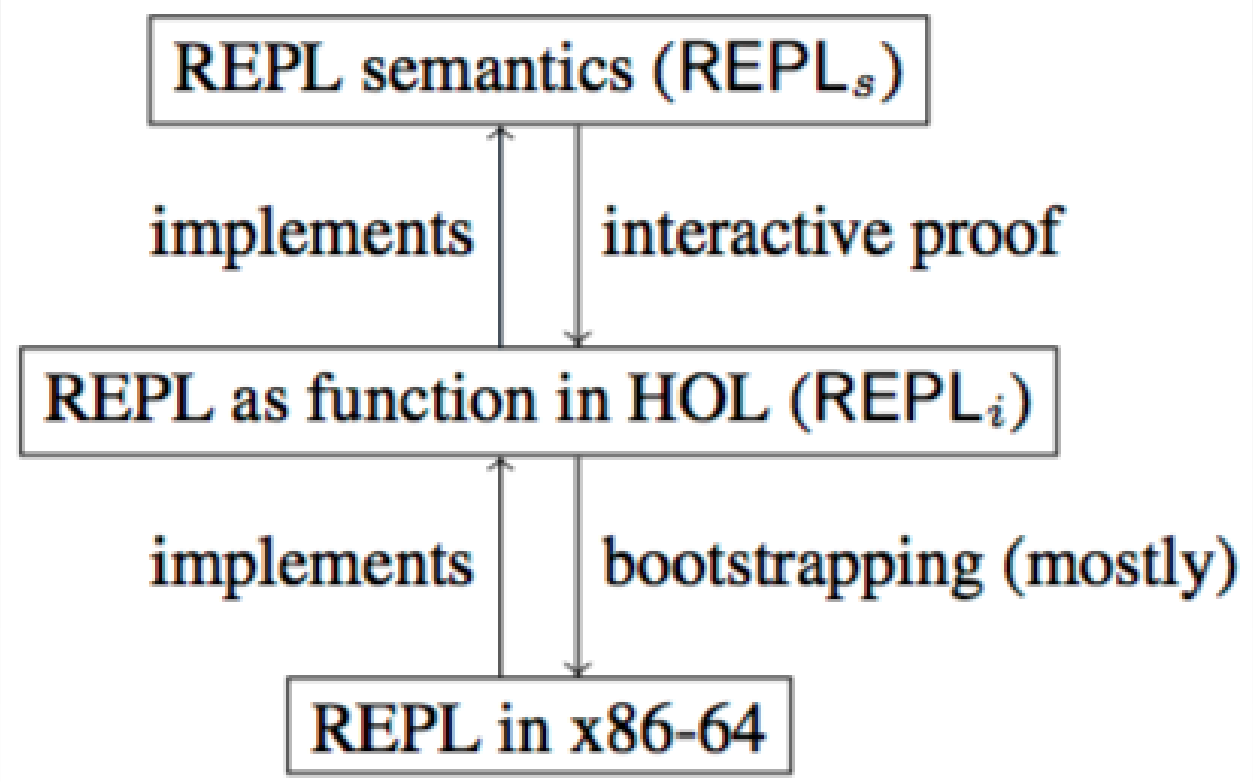
Standard ML

```
(print"Mt.␣";1)
+(print"Aoba";2)
```

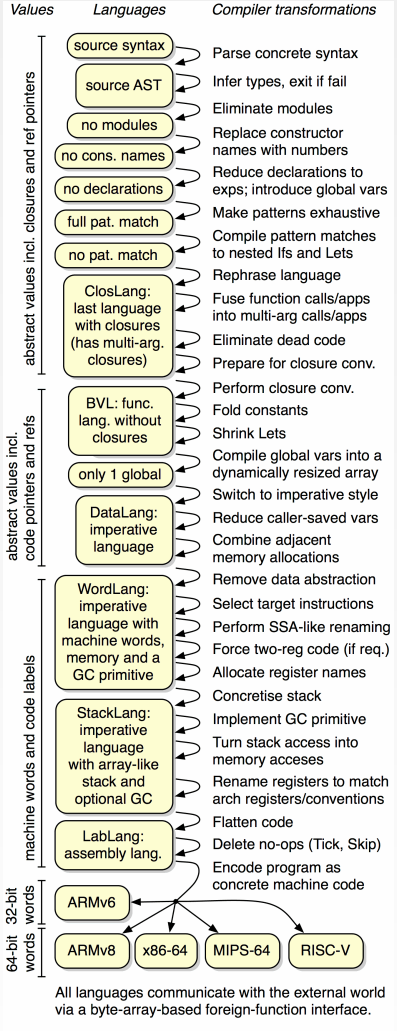
”Mt. Aoba”と**表示**

検証の流れ

REPL が使用を満たす事を証明して extract⟶
ブートストラップ
今は x86 バックエンドも証明済



CakeML の構成



コンパイルフェーズごとに証明して結合

パーサー

PEG で実装

定理 1 (パーサーの健全性)

PEG が非終端記号 N のパースに成功した場合、そのパース結果は文脈自由文法での N の導出木と対応するトークンの列となる

定理 2 (パーサーの完全性)

入力に対応する構文解析木が存在する場合、PEG は構文解析に成功する

ML 多相の型推論

定理 3 (型推論の健全性)

与えられたプログラムの型推論に成功した場合、そのプログラムには型を付けられる

定理 4 (型推論の完全性)

型を付けられるプログラムの型推論は成功する

コード生成、最適化

入力されたプログラムと出力されたプログラムが等価であると期待

変換前のプログラムの評価結果と変換後のプログラムの評価結果が対応している事を示す

結論

SML **にはこんなにも検証された処理系がある**...

結論

Standard ML **は信用できる**!