

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)
- 青葉山に籠って Coq を書く M1
- ML が好き



処理系の形式的検証についてのサーベイ

CakeML の紹介

バグの無いプログラムが書きたい

⇒ テストを書こう!

"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

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

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！

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...

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

Standard ML のサブセット

- 副作用
- ヴァリアント
- パターンマッチ

なぜ Standard ML か？

言語仕様が厳密に定義されている

OCaml

Standard ML

対象言語の文法

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

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.

フォント

こんにちは、世界。

こんにちは、世界。

こんにちは、世界。

こんにちは、世界。

こんにちは、世界。

こんにちは、世界。

こんにちは、世界。

こんにちは、世界。

こんにちは、世界。

こんにちは、世界。

番号なし箇条書き：

- 項目 1
- 項目 2
- 項目 3

番号つき箇条書き：

1. 項目 1
2. 項目 2
3. 項目 3

ブロックの使用例

ブロックのタイトル

ブロックの内容。

ブロックのタイトル

exampleblock は例のためのブロックです。

ブロックのタイトル

alertblock は強調のためのブロックです。alert のブロック版だと思えばいいでしょう。

定理環境の使用例

定義 1.1 (定義のタイトル)

定義の内容

補題 2.2 (補題のタイトル)

補題の内容

定理 3.4 (定理のタイトル)

定理の内容

証明のタイトル.

証明の内容



ブロック環境

次の環境が使えます。

- block
- exampleblock
- alertblock
- 定義 (definition)
- 公理 (axiom)
- 定理 (theorem)
- 補題 (lemma)
- 系 (corollary)
- 命題 (proposition)
- 証明 (proof) — 他の環境と少しだけ使い方が違うので注意

block, exampleblock, alertblock はただの色違い。それ以外は block 環境と同じ色。

オーバーレイ

オーバーレイ (overlay) とは、

オーバーレイ

オーバーレイ (overlay) とは、

- 単純なアニメーションみたいなもの

オーバーレイ

オーバーレイ (overlay) とは、

- 単純なアニメーションみたいなもの
- 最初のスライドでは隠していた文字や図形を、あとから表示させる

オーバーレイ

オーバーレイ (overlay) とは、

- 単純なアニメーションみたいなもの
- 最初のスライドでは隠していた文字や図形を、あとから表示させる
- よく使うのは pause (他にもいろいろある)

ソースコードの書き方

ソースコードは verbatim 環境でも書けるが、あまり綺麗ではない。

listings を使うのがおすすめ：

- listings.sty — LaTeX で綺麗なソースコードを書くためのスタイルファイル
- jlisting.sty — ソースコード中で日本語を使いたい時に必要 (listings.sty と併用)

ソースコードの書き方

- frame 環境のオプションに fragile を指定する
 - 指定の方法はソースコードを参照
 - 指定しないと、コンパイルできない
- listings はあまり高度な自動色付けができない
 - せいぜい、予約語の強調とか、文字列・コメントの色つけ程度
 - 細かい強調は手動で行ったほうが良い（後述）

ソースコードの例

- 長いソースコードには `lstlisting` 環境を使う
- 文中のソースコードには `lstinline` マクロを使う (用法は `verb` と同じ)

例 1) `lstlisting` 環境 :

```
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)
```

例 2) `lstinline` マクロ :

`listup_nodes` の型は `'a bin_tree -> 'a list` である。

一時的にスタイル or 言語を変更する

ソースコードの強調表示の設定：

- 共通の定義はプリアンブルの `lstset` で行う。
- 個別に変更するときは、`lstlisting`、`lstinline` のオプションで指定する。

例 1) フレームなし

```
let rec fact n =  
  if n = 0 then 1 else n * (fact (n - 1))
```

例 2) C 言語に変更

```
int fact (int n) {  
  if (n == 0) {  
    return 1;  
  } else {  
    return n * fact(n - 1);  
  }  
}
```

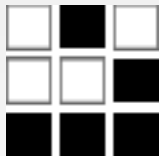
ソースコードの手動強調表示

以下の書式で強調表示ができるようになっている。
(使い方はソースコードを参照)

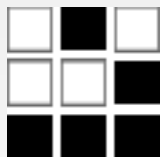
- @/.../@ — イタリック: *hoge*
- @r{...}@ — 赤: **hoge**
- @g{...}@ — 緑: **hoge**
- @b{...}@ — 青: **hoge**

例)

```
let fact n
  let rec fact' i acc =
    if i = 0 then acc else fact' (i - 1) (n * acc)
  in
  fact' n 1
```

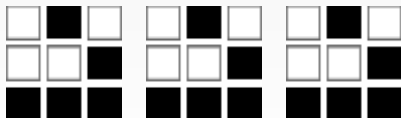


- ページを横に分割
- 図・表・文を横に並べて配置可能
- よく使うレイアウト
- minipage 環境でも同じ事ができる



- ページを横に分割
- 図・表・文を横に並べて配置可能
- よく使うレイアウト
- minipage 環境でも同じ事ができる

入れ子にしてみる



- 3 つ以上の分割も可能
- 入れ子も可能
- 柔軟に使えて便利！

SML にはこんなにも検証された処理系がある...

Standard MLは信用できる!

APPENDIX

予備のライド

予備ライドは `appendix` 環境の中に書きましょう。