START SEPLOG + COQ 2

Keigo IMAI

ProofSummit
2011年9月25日
@豆蔵 セミナールーム

おまえ誰

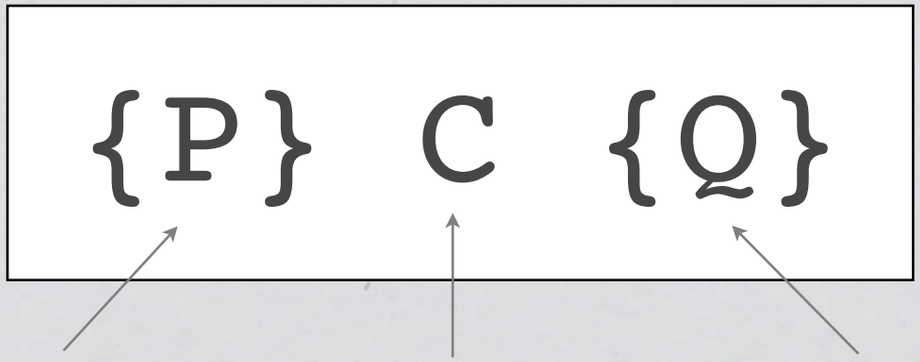
* @keigoi, 今井敬吾
* 名古屋の 有限会社 ITプランニングで勤務
* Haskell+OCaml好き

C言語(手続き型)プログラムの証明

* C言語プログラムの証明は、ホーア論理という枠組みで行う

* 最近10年で、ポインタを扱えるホーア論理の拡張Separation Logic（分離論理）が知られるようになった

ホーア論理による仕様の書き方



事前条件　プログラム　事後条件

例：

```
{y==9} x=y+1; {y==9&&x==10}
```

実際の証明では、ループ不変条件(loop invariant)と呼ばれる条件式をforループやwhileループに埋め込む必要がある。

論理式

* P /\ Q
* P -> Q
* P \/ Q
* a=b
* ...

分離論理(separation logic)の論理式

* P /\ Q
* P -> Q
* P \/ Q
* a=b
* ...

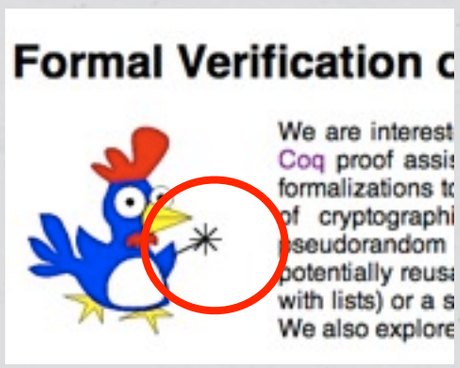
* x |-> v
* P ** Q
* P -* Q

論理結合子の追加

分離論理
ポインタとヒープの論理

* x |-> e
  * ヒープ領域にはオブジェクトeだけが確保されている
  * ポインタxがそのeを指している

* P ** Q
  * P かつ Q 、かつ PとQは異なるメモリ領域を参照している

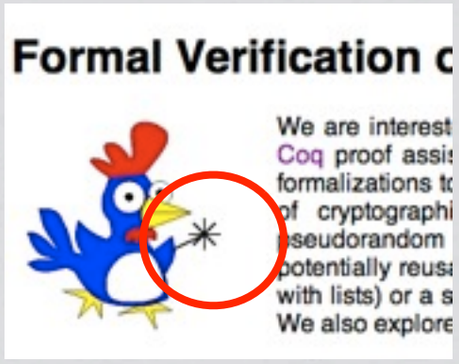


←これ

http://staff.aist.go.jp/reynald.affeldt/seplog/

分離論理

* P ** Q
  * P かつ Q 、かつ PとQは異なるメモリ領域を参照している
* P -* Q (魔法の杖?)
  * P ならば Q、
    かつ Pで参照しているメモリ領域は Q でも参照しなければならない (後述)



←これ

http://staff.aist.go.jp/reynald.affeldt/seplog/

Coq で seplog

（他にもある）

* Formal Verification of Memory Properties
  Reynald Affeldt (産総研)
  * Coqで教育用OSのメモリマネージャ検証
  * アセンブラによる暗号アルゴリズム記述の検証
* Ynot - A. Chlipala 他多数, (Harvard Univ.)
  * 副作用つきのMLプログラムを導出できる
* Practical Tactics for Separation Logic
  Andrew McCreight (Portland State Univ.)

←D論ではGCの証明を
Coqでやっている

seplog @ AIST

- 比較的簡潔なため、これを読んでみましょう：
  - Formal Verification of Memory Properties
    by Reynald Affeldt (産総研)
  - ソースも簡潔な ICFEM06版を使います
    (他のでも良かったかも)
- 合計： 2万行
  - プログラムのソースコード2万行なら楽勝ですよね？
  - さらに：例題以外の(基本部分の)証明はひとまず追わないことにしよう

まずはビルド

* http://proofcafe.co.cc/~keigoi/ から icfem06-8.3.tar.bz2 をダウンロード
* 展開して ./build でビルド (要Coq8.3)

行数

seplog本体

```
 834 util.v
2388 heap.v
1221 bipl.v
1080 axiomatic.v
1081 contrib.v
 196 vc.v
  46 seplog_header.v
```

タクティック

```
 152 util_tactic.v
 373 heap_tactic.v
 222 contrib_tactics.v
```

例題

```
 547 examples.v
 385 example_reverse_list.v
2216 frag.v
 276 frag_examples.v
1489 omega_test.v
 130 hmAlloc_example.v
1306 topsy_hm.v
2762 topsy_hmAlloc.v
3163 topsy_hmAlloc2.v
 536 topsy_hmFree.v
 313 topsy_hmInit.v
20716 total
```

seplog内訳

多くは証明や補題が占めているので、基本的な定義のみを追えば一通りわかるはず

基本的な補題など	834	util.v
データ構造(ヒープ・ストア用)の定義	2388	heap.v
手続き型言語の式の定義、分離論理の定義	1221	**bipl.v**
手続き型言語の文の定義、ホーア論理の定義	1080	**axiomatic.v**
いろいろ	1081	contrib.v
最弱事前条件(WP)・検証条件(VC)の計算など	196	**vc.v**
インポート用	46	seplog_header.v

seplogの論理式 in Coq

seplogで使う論理式をアサーションと呼ぶ

```
Definition assert := store.s -> heap.h -> Prop.
```

(bipl.v, 747行目付近)

「ストア」と「ヒープ」に依存する

（ストア…ローカル変数領域（スタック））

ストア、ヒープ

* ヒープ：
  * モジュール mapを使って定義されている (@heap.v)

```
Module Type HEAP.        ← mapモジュールは HEAP型をもつ
  Parameter emp : h.
  Parameter singleton : l -> v -> h.
  Parameter lookup : l -> h -> option v.
  Parameter lookup_singleton : forall n w, lookup n (singleton n w) = Some w.
  Parameter extensionality: forall h1 h2, (forall x, lookup x h1 = lookup x h2) -> h1 = h2.
  Parameter update : l -> v -> h -> h.
  Parameter update_singleton : forall n w w', update n w' (singleton n w) = singleton n w'.
  Parameter union : h -> h -> h.
  Notation "k '+++' m" := (union k m) (at level 69) : heap_scope.
…（関連する補題がつづく）…
```

* ストア：ローカル変数領域

  mapを使って定義されているが、無効参照がないような定義になっていた（空だった場合は0が帰る）

seplogで扱う手続き型言語(1)：式

bipl.v で定義されている

整数値 (274行目)

```
Inductive expr : Set :=
  var_e : var.v -> expr
| int_e : val -> expr
| add_e : expr -> expr -> expr
| min_e : expr -> expr -> expr
| mul_e : expr -> expr -> expr
| div_e : expr -> expr -> expr.
```

ブール値 (470行目)

```
Inductive expr_b : Set :=
  true_b: expr_b
| eq_b : expr -> expr -> expr_b
| neq_b : expr -> expr -> expr_b
| ge_b : expr -> expr -> expr_b
| gt_b : expr -> expr -> expr_b
| neg_b : expr_b -> expr_b
| and_b : expr_b -> expr_b -> expr_b
| or_b : expr_b -> expr_b -> expr_b.
```

seplogで扱う手続き型言語(2)：文

axiomatic.v, 40行目

```
Inductive cmd : Set :=
  skip : cmd
  | assign : var.v -> expr -> cmd
  | lookup : var.v -> expr -> cmd
  | mutation : expr -> expr -> cmd
  | malloc : var.v -> expr -> cmd
  | free : expr -> cmd
  | while : expr_b -> cmd -> cmd
  | seq : cmd -> cmd -> cmd
  | ifte : expr_b -> cmd -> cmd -> cmd.
```

動作意味

* 手続き型プログラムのインタプリタをCoq上に作る
    * ただしCoqでは「止まらない関数」を書けない。そこで…
* 三項関係 exec axiomatic.v, 73行目〜

```
Inductive exec : option state -> cmd -> option state -> Prop :=
```

この初期状態から　　このプログラムで　　最後の状態はこうなる

exec s c s’ を s -- c --> s’ と書く(Notationが定義されている)

Big step semanticsという。

プログラムの動作意味(1)

* exec_skip : Some (s,h) -- skip --> Some (s,h)
  * skip はストア・ヒープの状態を変えない

* exec_assign :
  Some (s,h) -- x <- e -->
  Some (store.update x (eval e s) s, h)
  * ストアへの代入

プログラムの動作意味 (2)
連接・分岐 (抜粋)

* exec_seq :

  (s -- c --> s') -> (s' -- d --> s'') -> (s -- c; d --> s'')

* exec_ifte_true :

  eval_b b s = true ->

  (Some (s,h)-- c --> s') ->

  (Some (s,h) -- ifte b thendo c elsedo d --> s')

プログラムの動作意味 (3)
ループ

* exec_while_true :

  eval_b b s = true ->

  (Some (s,h) -- c --> s') ->

  (s' -- while b c --> s'') ->

  (Some (s,h) -- while b c --> s'')

* exec_while_false :

  eval_b b s = false ->

  (Some (s,h) -- while b c --> Some (s,h))

プログラムの動作意味 (4)
メモリ確保

新しいヒープは既存の
ヒープから分離されている

xに新しいポインタnを代入

* exec_malloc :

```
h # heap.singleton n (eval e s) ->
(Some (s, h) -- (x <- malloc e) -->
  Some (store.update x (Z_of_nat n) s,
          h +++ (heap.singleton n (eval e s))))
```

「nがeの評価結果を指す」という新しいのヒープを結合

ホーア論理(ホーアの3つ組)

* 3項関係 semax axiomatic.v, 172行目〜

```
Inductive semax : assert -> cmd -> assert -> Prop :=
```

事前条件　　プログラム　　事後条件

semax P c Q を {{P}} c {{Q}} と書く

ホーア論理の公理
あるいはプログラムの公理的意味論(1)

* semax_skip: {{P}} skip {{P}}

* semax_assign: {{update_store2 x e P}} x <- e {{P}}

  * update_store2 x e P は

    P{e/x} （論理式P中のローカル変数をeに置換）

    だと思って下さい

Hoare Logicの推論規則に対応していることを確認しよう（参考文献）

ホーア論理の公理
あるいはプログラムの公理的意味論(2)

* seplog!

  * semax_malloc :

    {{ $n_{|->}e$ -* P[n/x] }} x <-malloc e {{P}}

  * semax_free :

    {{exists l, val2loc (eval e s) = l /\
      exists v, heap.lookup l h = Some v /\ P s (h --- l)}}
    free e {{P}}

ホーア論理の例：階乗計算の仕様

examples.v, 234行目から

事前条件

```
{{ fun s (_:heap.h) =>
  store.lookup m s = n /\
  store.lookup x s = 1 }}
```

事前条件：ローカル変数 m = n かつ x = 1

プログラム

```
(while' ((var_e m) =/= int_e 0)
 (fun s (_:heap.h) =>
   store.lookup x s *
  facto (store.lookup m s) = facto n /\
   store.lookup m s >= 0)
  (x <- var_e x *e var_e m;
   m <- var_e m -e int_e 1))
```

ループ不変条件

プログラム本体：

```
while (m!=0) { (ループ)
```

ループ不変条件： x * m! = n!

（facto は Coqでの階乗の定義）

```
x := x * m;  // ループ本体
m := m - 1;}
```

事後条件

```
{{ fun s (_:heap.h) =>
  store.lookup x s = facto n }}.
```

事後条件：ローカル変数 x = !n

証明してみよう:
ホーア論理 (分離論理なし)

* 階乗

* Lemma factorial : forall n, 0 <= n ->
  forall x m, var.set (x::m::nil) ->
  {{ fun s h => store.lookup m s = n /\ store.lookup x s = 1 }}
  while (var_e m =/= int_e 0%Z) (
    x <- var_e x *e var_e m;
    m <- var_e m -e int_e 1
)
  {{ fun s h => store.lookup x s = facto n }}.

証明してみよう:
ホーア論理 + 分離論理

* ポインタx,yが指すヒープの内容を入れ替える

* Lemma swap: forall x y z v a b, var.set (x::y::z::v::nil) ->
  {{ (var_e x |-> int_e a) ** (var_e y |-> int_e b) }}
  z <-* var_e x;
  v <-* var_e y;
  var_e x *<- var_e v;
  var_e y *<- var_e z
  {{ (var_e x |-> int_e b) ** (var_e y |-> int_e a) }}.

seplog:基本タクティック

* 仮定の H1 : P ** Q を P, Q の証明に分割
  （と同時に、ヒープも h1 h2 に分割）
  Decompose_sepcon H1 h1 h2

* 結論の P ** Q を サブゴール2つに分割
  （ヒープは h1, h2 に分割 (仮定にh=h1++h2が必要)）
  Compose_sepcon h1 h2.

例：メモリ操作を伴うプログラムの仕様

教育用OS Topsyのメモリ管理部分hmAllocの仕様（産総研 Reynald Affeldt氏らによる*）

事前条件

```
∀ adr x sizex size, adr > 0 → size > 0 →
  {∃ l, Heap_List l adr ∧
        ln_hl l (x,sizex,alloc) adr ∧
        (s |= var_e hmStart == nat_e adr) }
```

adrは既存ヒープlの先頭である
x は既存の確保済みブロックを指す
初期化済みである(hmStartがadrを指す)

プログラム

```
hmAlloc(result,size);
```

事後条件

```
{∃ l, ∃ y, y > 0 ∧
  (s |= var_e result == nat_e (y+2)) ∧
  ∃ size'', size'' ≥ size ∧
   (Heap_List l adr ** Array (y+2) size'')∧
   ln_hl l (x,sizex,alloc) adr ∧
   ln_hl l (y,size'',alloc) adr ∧
   x ≠ y }.
```

（ブロックのヘッダ2ワード分を除いている）

戻り値は新規ブロックへのポインタ(y+2)
新規ブロックのサイズは要求より大きい
新規ブロックはヒープから分離されている
xは既存の確保済みブロックを指す
yはサイズsize''のブロックを指す
yは新しいポインタである

（分離を表す演算子）

(*) Nicolas Marti and Reynald Affeldt and Akinori Yonezawa, Formal Verification of the Heap Manager of an Operating System using Separation Logic, 8th International Conference on Formal Engineering Methods (ICFEM 2006), Macao SAR, China, October 29–November 3, 2006

休題：自動証明ではこんな道具を使います

* 関数 wp (weakest precondition)
  * 事後条件とプログラムから「最弱」事前条件を計算
    * あとは 事前条件 ⇒ 最弱事前条件 を証明すればよい
* 関数 vc (verification condition)
  * 与えられたプログラムから、ループ不変条件が正しいことを検査する論理式を生成

Coqで見る自動証明の雰囲気：
定理 wp_vc_util

* Lemma wp_vc_util: forall c Q P ,
  (forall s h, vc c Q s h) ->
  (P ==> wp c Q) ->
  {{P}} proj_cmd c {{Q}}.

3. かつVCが成立

2. 事前条件が最弱事前条件を含意

1. {P} c {Q} を証明するには

  * なぜ上記が成立するのか考えよう（vc, wpの定義などは参考文献）

* 結局、ホーア論理の自動証明は次の2ステップ：

  1. 全てのfor文のloop invariantを考える
  2. vcを証明する

新しい潮流：ICFP'11では

* 論文：
  Characteristic Formulae for the Verification of Imperative Programs

* プログラムから、プログラムと一意に対応する表すCF(特性論理式)を生成

  * これを証明すればよい

* ツール： CFML http://www.chargueraud.org/softs/cfml/

  * 副作用入りのCamlコードからCoqで読める特性論理式を生成

Characteristic Formulae?

* プロセス(プログラム)pの振る舞いを一意に特定する論理式のこと
* もとは**プロセス計算**と呼ばれる、並行プログラムを扱う分野の考え方
* プロセスpを表現する様相論理式 L(p) がある
* p と q が振る舞い等価ならば、 L(p) と L(q) は論理的に等価である

Characteristic Formulae?

* プロセス(プログラム)pの振る舞いを一意に特定する論理式のこと
* もとは**プロセス計算**と呼ばれる、並行プログラムを扱う分野の考え方
* プロセスpを表現する様相論理式 L(p) がある
* p と q が振る舞い等価ならば、 L(p) と L(q) は論理的に等価である

Algebraic laws for nondeterminism and concurrency

Full Text: Pdf Buy this Article

Authors: Matthew Hennessy Univ. of Edinburgh, Edinburgh, Scotland
Robin Milner Univ. of Edinburgh, Edinburgh, Scotland

Published in:
· Journal
Journal of the ACM (JACM) JACM Homepage archive
Volume 32 Issue 1, Jan. 1985
ACM New York, NY, USA
table of contents doi>10.1145/2455.2460

Characteristic Formulae?

* プロセス(プログラム)pの振る舞いを一意に特定する論理式のこと
* もとは**プロセス計算**と呼ばれる、並行プログラムを扱う分野の考え方
* プロセスpを表現する様相論理式 L(p) がある
* p と q が振る舞い等価ならば、 L(p) と L(q) は論理的に等価である

THEOREM 2.2. *If each R_i is image finite then*

$$p \sim q \quad \text{if and only if} \quad \mathscr{L}(p) = \mathscr{L}(q).$$

This characterization theorem (proved in Appendix A), together with our examples, which indicate that in $\mathscr{L}$ it is possible to discuss deadlocking properties of processes, encourages us to believe that our notion of observation equivalence is

Published in:
· Journal
Journal of the ACM (JACM) JACM Homepage archive
Volume 32 Issue 1, Jan. 1985
ACM New York, NY, USA
table of contents doi>10.1145/2455.2460

まとめ

* Reynolds のチュートリアルを（ある程度）読めば、
  Affeldt さんの seplog も使えるようになる
  （気がする）

* 他にも様々なseplogライブラリがある

* できるだけ自動化し、外部の自動定理証明器に投げ…
  難しいところだけCoqで証明できたらいいな

  * Ltacを使いこなしたいね！

  * Coqでメタ定理を色々作れると楽かも

* ICFP'11 ： プログラムの証明にはホーア論理だけじゃない！

ホーア論理 参考文献

* 林 晋, プログラム検証論 (情報数学講座), 共立出版, 1995.
* Ole-Johan Dahl, Verifiable Programming, Prentice-Hall, 1992.
* 大学の講義資料とか (wpとvcの定義が手元になかったので参考にした)
  * コロラド大学 プログラミング言語の基礎 講義資料
    * http://www.cs.colorado.edu/~bec/courses/csci5535-s09/schedule.html
  * ミンホ大学 形式手法 講義資料（ページはポルトガル語だが、スライドは英語）
    * http://www3.di.uminho.pt/~jno/html/mfes-0809.html の Hoare Logic, Verification Conditions など

Seplog 参考文献

* Separation Logic: A Logic for Shared Mutable Data Structures
  http://www.cs.cmu.edu/~jcr/seplogic.pdf
  * Reynolds によるチュートリアル。ここから始めると吉？
* Formal Verification of Memory Properties
  http://staff.aist.go.jp/reynald.affeldt/seplog/
  * 今回使用したAffeldtさんによるseplogのCoq実装
* Ynot http://ynot.cs.harvard.edu/
* Practical Tactics for Separation Logic
  http://web.cecs.pdx.edu/~mccreigh/ptsl/