RustHorn: CHC-based Verification for Rust Programs

ESOP 2020

2020/9/10 JSSST 2020 トップカンファレンス特別講演

松下 祐介, 塚田 武志, 小林 直樹 (東京大学)

目次

CHCに基づく自動プログラム検証 [Bjørner+ 2015]

Constrained **H**orn **C**lause

プログラム検証問題

“assert は常に成功する?”

CHC充足可能性問題

“条件を満たす $mc91$ は存在する?”

```
int mc91(int n) {
  if (n > 100) return n - 10;
  else return mc91(mc91(n + 11));
}
void test(int n) {
  if (n <= 101) assert(mc91(n) == 91);
}
```

$$
\begin{aligned}
&mc91(n, r) \impliedby n > 100 \land r = n - 10 \\
&mc91(n, r) \impliedby n \leq 100 \land mc91(n + 11, r') \land mc91(r', r) \\
&r = 91 \impliedby n \leq 101 \land mc91(n, r)
\end{aligned}
$$

解 (≒不変条件) $mc91(n, r) :\Longleftrightarrow r = 91 \lor (n > 100 \land r = n - 10)$

既存のCHCソルバ (Spacer [Komuravelli+ 2018], Holce [Champion+ 2018] etc.) で自動的に解ける

ポインタ操作の絡む検証の困難

- ポインタ操作のあるプログラムの自動検証は一般に困難
  - 既存手法 [Gurfinkel+ 2015] では CHC でメモリ状態を配列により表現し対処
    → 一般にはスケールしづらい

単純な例でも量化子のある解が必要になり難しい

```
bool jrec(int *ma) {
  if (rand()) return true;
  int a0 = *ma; int b = rand();
  return jrec(&b) && *ma == a0;
}
void test(int a) { assert(jrec(&a)); }
```

$$
\begin{aligned}
&jrec(ma,h,s,r,h',s') \impliedby r = \text{true} \land h' = h \land s' = s \\
&jrec(ma,h,s,r,h',s') \impliedby h'' = h\{s \leftarrow b\} \land s'' > s \land \\
&\quad jrec(s,h'',s'',r',h',s') \land r = (r' \,\&\&\, (h'[ma] == h[ma])) \\
&r = \text{true} \impliedby jrec(ma,h,s,r,h',s') \land ma < s
\end{aligned}
$$

h, h': メモリ状態、s, s': スタックポインタ

解　$jrec(ma,h,s,r,h',s') :\Longleftrightarrow (\forall i < s \,.\, h'[i] = h[i]) \land \cdots$

本研究の概要

- **Rust** プログラムから CHC への新しい帰着手法による自動検証
  - ポインタを扱うプログラムの検証において、
    メモリの表現を無くし、効率の良い検証を目指す
    - Rust の型システムによるポインタに関する保証を利用
      ポインタ `ma` → 現在&借用終了時の値の組 $\langle a, a_\circ \rangle$
  - 幅広い機能 (再帰的データ型, 再借用 etc.) を扱える
  - 正当性を証明、実験で有効性を確認

提案手法の概観

```
bool jrec(int *ma) {
  if (rand()) return true;
  int a0 = *ma; int b = rand();
  return jrec(&b) && *ma == a0;
}
void test(int a) { assert(jrec(&a)); }
```

既存手法

$$
\begin{aligned}
&jrec(ma, h, s, r, h', s') \Longleftarrow r = \text{true} \wedge h' = h \wedge s' = s \\
&jrec(ma, h, s, r, h', s') \Longleftarrow h'' = h\{s \leftarrow b\} \wedge s'' > s \wedge \\
&\quad jrec(s, h'', s'', r', h', s') \wedge r = (r' \,\&\&\, (h'[ma] == h[ma])) \\
&r = \text{true} \Longleftarrow jrec(ma, h, s, r, h', s') \wedge ma < s
\end{aligned}
$$

提案手法

$$
\begin{aligned}
&jrec(\langle a, a_\circ \rangle, r) \Longleftarrow r = \text{true} \wedge a_\circ = a \\
&jrec(\langle a, a_\circ \rangle, r) \Longleftarrow a_\circ = a \wedge \\
&\quad jrec(\langle b, b_\circ \rangle, r') \wedge r = (r' \,\&\&\, (a_\circ == a)) \\
&r = \text{true} \Longleftarrow jrec(\langle a, a_\circ \rangle, r)
\end{aligned}
$$

メモリの表現が不要!

簡単な解 $jrec(\langle a, a_\circ \rangle, r) :\Longleftrightarrow r = \text{true}$

より非自明な例を後に紹介

目次

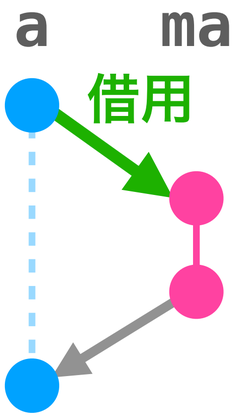
- 背景と概要
- **Rust の型システム**
- 提案手法
- 実装と評価
- 関連研究

Rust の型システム

- ポインタの**権限**を制限、安全性を保証
  - 原則: 任意の資源 x に対し、あるポインタが x への更新権限を持つとき、
    他のポインタは x への権限を一切持たない (読取も更新も不可)
- **借用**: あるポインタの x への更新権限を、x を指す新しいポインタに貸す
  - 貸し手はその貸出期間中は資源 x へのすべての権限を失う

可変参照

```
let mut a: int = 1;
let ma: &mut int = &mut a;
*ma += 10;
print(a); // 11
```



目次

提案手法

- 借用 & 可変参照による更新 を以下の方法で表現

1. 借用時に 変数 $a_\circ$ (借用終了時の値) を貸し手と可変参照で共有
2. 可変参照は a (現在の値) と $a_\circ$ との組 $\langle a, a_\circ \rangle$ として表現 (a は逐次更新)
3. 後で可変参照 $\langle a, a_\circ \rangle$ を解放するときに $a_\circ = a$ と束縛

借用終了時の値にのみ着目すれば良い！

```
let mut a: int = 1;
let ma: &mut int = &mut a;
*ma += 5; *ma += 5;
print(a); // 11
```

≈

```
let mut a: int = 1;
let ma: &mut int = &mut a;
*ma += 10;
print(a); // 11
```

基本的な変換例 (可変参照の参照先の動的決定)

```
fn max<'a>(ma: &'a mut int, mb: &'a mut int) -> &'a mut int {
  if *ma >= *mb { ma } else { mb }
}
fn test(mut a: int, mut b: int) {
  let mc = max(&mut a, &mut b); *mc += 1; assert!(a != b);
}
```

max: 2つの可変参照のうち (現在の) 値が大きいほうを返す

$$max(\langle a, a_\circ \rangle, \langle b, b_\circ \rangle, r) \Longleftarrow a \geq b \wedge b_\circ = b \wedge r = \langle a, a_\circ \rangle$$
$$max(\langle a, a_\circ \rangle, \langle b, b_\circ \rangle, r) \Longleftarrow a < b \wedge a_\circ = a \wedge r = \langle b, b_\circ \rangle$$
$$a_\circ \neq b_\circ \Longleftarrow max(\langle a, a_\circ \rangle, \langle b, b_\circ \rangle, \langle c, c_\circ \rangle) \wedge c_\circ = c + 1$$

可変参照を解放するタイミングで借用終了時の値を束縛

再帰的データ型を扱う検証例(リストの破壊的更新)

(一部省略)

```
enum List { Cons(int, Box<List>), Nil }
fn pick(mla: &mut List) -> &mut int { match mla {
  Cons(ma, mla2) => if rand() { ma } else { pick(mla2) }
} }
fn test(mut la: List) {
  let s0 = sum(&la); let ma = pick(&mut la); *ma += 1; assert!(sum(&la) == s0+1);
}
```

pick: リストへの可変参照からいずれかの要素への可変参照を取る

sum はリスト上で帰納的に定義された関数

$$pick(\langle [a \mid la'], [a_\circ \mid la'] \rangle, r) \Longleftarrow r = \langle a, a_\circ \rangle$$

$$pick(\langle [a \mid la'], [a \mid la'_\circ] \rangle, r) \Longleftarrow pick(\langle la', la'_\circ \rangle, r)$$

$$\mathrm{sum}(la_\circ) = \mathrm{sum}(la) + 1 \Longleftarrow pick(\langle la, la_\circ \rangle, \langle a, a_\circ \rangle) \wedge a_\circ = a + 1$$

単純な解! $pick(\langle la, la_\circ \rangle, \langle a, a_\circ \rangle) :\Longleftrightarrow a_\circ - a = \mathrm{sum}(la_\circ) - \mathrm{sum}(la)$

後述の実験においても完全な自動検証に成功

変換の形式化と正当性証明

- Rust の形式化 COR を作り、(提案手法の) 変換を形式化
  - 幅広い機能 (再帰的データ型, 再借用 etc.) をサポート

- 変換の正当性を証明
  - 「COR の任意の (可変参照を入出力に取らない) 関数 f について、
    (f の CHC 表現の最小解) ⇔ (f の入出力関係)」
  - CHC での導出 (resolution) と COR プログラムの実行との間の双模倣による証明
    - 借用終了時の値 $a_\circ$ を特別な変数として扱う

目次

実装と実験 https://github.com/hopv/rust-horn

- 提案手法に基づき、Rust プログラムの自動検証器 (RustHorn) を実装
  - Rust コンパイラの中間表現を利用、再帰的データ型等の機能もサポート
  - バックエンドCHCソルバ: Spacer [Komuravelli+ 2018]・Hoice [Champion+ 2018]

- 58個のベンチマークで既存手法 SeaHorn [Gurfinkel+ 2015] と性能を比較
  - SeaHorn: C言語のためのCHCベース自動検証器、前述のメモリ表現を利用
  - ベンチマーク: (a) SeaHorn由来の16個、(b) 可変参照の様々な使用例を扱う42個
    - 各々 Rust 版と C 言語版を作成、(a) は safe Rust で表現可能なものを選出

実験結果 概略

- SeaHorn 由来の課題では
  両検証器が同等の性能

Task a	RustHorn	SeaHorn
simple-01	<0.1	<0.1
simple-04	0.5	0.8
simple-05	<0.1	<0.1
simple-06	0.1	timeout
hhk2008	40.5	<0.1
unique-scalar	<0.1	<0.1
bmc-1-safe	<0.1	<0.1
bmc-1-unsafe	<0.1	<0.1
bmc-2-safe	0.1	<0.1
bmc-2-unsafe	<0.1	<0.1
bmc-3-safe	<0.1	<0.1
bmc-3-unsafe	<0.1	<0.1
diamond-1-safe	<0.1	<0.1
diamond-1-unsafe	<0.1	<0.1
diamond-2-safe	<0.1	<0.1
diamond-2-unsafe	<0.1	<0.1

- 可変参照による更新の絡む課題の
  多くで RustHorn のみが検証に成功

Task b	RustHorn	SeaHorn
imax-base-safe	<0.1	false alarm
imax-base-unsafe	<0.1	<0.1
imax-base3-safe	<0.1	false alarm
imax-base3-unsafe	<0.1	<0.1
imax-repeat-safe	0.1	false alarm
imax-repeat-unsafe	<0.1	<0.1
imax-repeat3-safe	0.2	false alarm
imax-repeat3-unsafe	<0.1	<0.1
ldec-base-safe	<0.1	false alarm
ldec-base-unsafe	<0.1	<0.1
ldec-base3-safe	<0.1	false alarm
ldec-base3-unsafe	<0.1	<0.1
ldec-exact-safe	<0.1	false alarm
ldec-exact-unsafe	<0.1	<0.1
ldec-exact3-safe	<0.1	false alarm
ldec-exact3-unsafe	<0.1	<0.1

再帰的データ型 (リスト・二分木)

Task b	RustHorn	SeaHorn
append-safe	<0.1	false alarm
append-unsafe	0.2	0.1
inc-all-safe	<0.1	false alarm
inc-all-unsafe	0.3	<0.1
inc-some-safe	<0.1	false alarm
inc-some-unsafe	0.3	0.1
inc-some2-safe	timeout	false alarm
inc-some2-unsafe	0.3	0.4
append-t-safe	<0.1	timeout
append-t-unsafe	0.3	0.1
inc-all-t-safe	timeout	timeout
inc-all-t-unsafe	0.1	<0.1
inc-some-t-safe	timeout	timeout
inc-some-t-unsafe	0.3	0.1
inc-some2-t-safe	timeout	false alarm
inc-some2-t-unsafe	0.4	0.1

目次

- 背景と概要
- Rust の型システム
- 提案手法
- 実装と評価
- **関連研究**

関連研究

- ポインタを扱うプログラムの CHC に基づく自動検証
  - SeaHorn [Gurfinkel+ 2015] (C/C++): 配列に基づくメモリ表現、前述の問題点
  - JayHorn [Kahsai+ 2016] (Java): 不変条件に基づくメモリ表現、柔軟性に問題
- Rust プログラムの型情報を利用した検証
  - Prusti [Astrauskas+ 2018]: 半自動検証、借用終了時の値に関する仕様記述を許す
  - Electrolysis [Ullrich 2016]: 定理証明支援系、可変参照をレンズで表現

結論

- Rust プログラムから CHC への新しい帰着手法による自動検証
  - Rust の型システムの性質を利用、メモリ表現が不要
    ポインタ (可変参照) `ma` → 現在&借用終了時の値の組 $\langle a, a_\circ \rangle$
  - 幅広い機能 (再帰的データ型, 再借用 etc.) を扱える
  - 正当性を証明、実験で有効性を確認

- 今後の課題: unsafe コードを含む Rust プログラムの検証への拡張