RustHorn: CHC-based Verification for Rust Programs

ESOP 2020

Yusuke Matsushita Takeshi Tsukada* Naoki Kobayashi

University of Tokyo

* Now in Chiba University

Our Work: **RustHorn**

- A novel reduction from **Rust** programs to CHCs for automated verification
  - Removes pointers by leveraging Rust's **ownership** guarantees:
    a pointer **pa** → the pair of its current & **final** target values $\langle a, a_\circ \rangle$
    Prophecy!
  - Supports various features: recursive data types, reborrowing, etc.
  - Proof of soundness and completeness
  - Evaluation with benchmarks

CHC-based Automated Verification

[Grebenshchikov+ 2012]
[Bjørner+ 2015] …

Constrained Horn Clause

Verification Problem

```
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);
}
```

Functionally Correct?

⬇

CHC Satisfiability Problem

$$mc91(n,r) \Longleftarrow n > 100 \ \land \ r = n - 10$$

$$mc91(n,r) \Longleftarrow n \le 100 \ \land \ mc91(n+11,r') \ \land \ mc91(r',r)$$

$$r = 91 \Longleftarrow n \le 101 \ \land \ mc91(n,r)$$

Satisfiable! $mc91(n,r) \ \equiv \ r = 91 \ \lor \ (n > 100 \ \land \ r = n - 10)$

Automated CHC Solvers: Spacer [Komuravelli+ 2018], HoIce [Champion+ 2018], …

Challenge in Pointers

- Verification is often hard when the program has pointers
- Naive approach: Model the memory as an array h
  → Easily fails in the presence of dynamic memory allocation

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

$$jrec(pa,h,s,r,h',s') \Longleftarrow r = \text{true} \land h' = h \land s' = s$$

$$jrec(pa,h,s,r,h',s') \Longleftarrow jrec(s,h\{s \leftarrow b\},s+1,r',h',s') \land r = r' \mathbin{\&\&} h'[pa] == h[pa]$$

$$r = \text{true} \Longleftarrow jrec(pa,h,s,r,h',s') \land pa < s$$

Satisfiable with universal quantification → Very hard!

$$jrec(pa,h,s,r,h',s') \equiv (\forall i < s \,.\, h'[i] = h[i]) \land \cdots$$

Quantified invariant: a memory region is unchanged

RustHorn removes pointers for smooth verification!

Table of Contents

Rust in a Nutshell

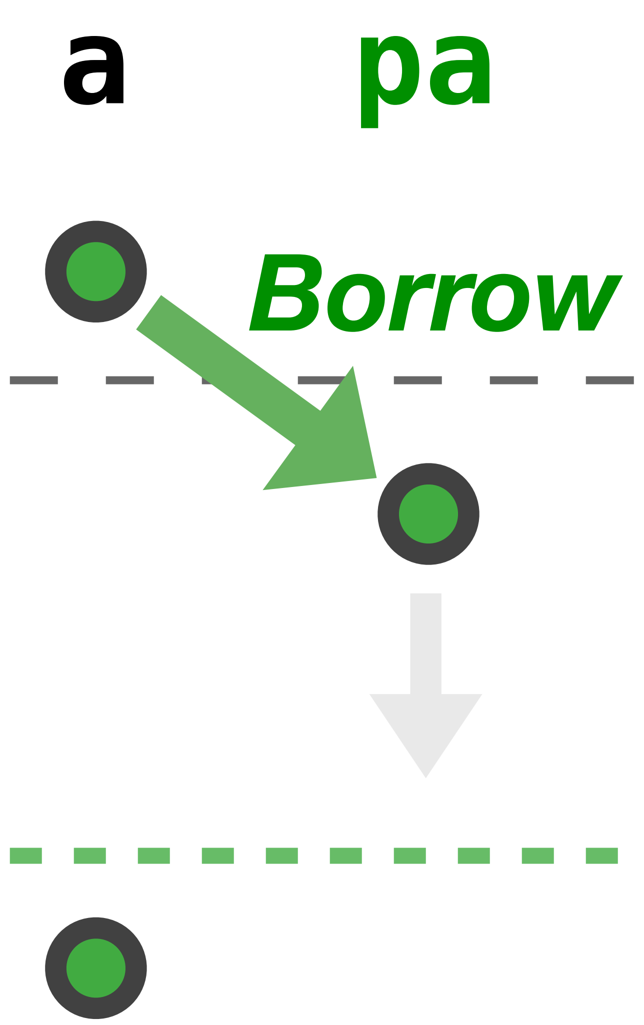
https://www.rust-lang.org

- Low-level memory operations & Safety guarantees by the type system
- **Ownership**: Necessary for object update, not sharable
- **Borrow**: Temporary transfer of ownership to a new reference

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

Reference

Deadline



Example of Borrows

(Partly omitted)

```
fn max(pa: &mut int, pb: &mut int) -> &mut int {
  if *pa >= *pb { pa } else { pb }
}

fn test(mut a: int, mut b: int) {
- - - - - - - - - - - - - - - - - - - - - - - - -
  let pc = max(&mut a, &mut b);
  *pc += 1;
- - - - - - - - - - - - - - - - - - - - - - - - -  Deadline of the two borrows
  assert!(a != b);
}
```

Table of Contents

Verifying Rust Programs

- Motivation: Remove pointers in Rust programs for smooth verification
- Naive approach: Model each reference just as its target value

  → We can't model the lender after the borrow's deadline

```
fn max(pa: &mut int, pb: &mut int) -> &mut int {
  if *pa >= *pb { pa } else { pb }
}
fn test(mut a: int, mut b: int) {
  let pc = max(&mut a, &mut b); *pc += 1; assert!(a != b);
}
```

$$
\begin{aligned}
max(a,b,r) &\Longleftarrow a \geq b \ \land \ r = a \\
max(a,b,r) &\Longleftarrow a < b \ \land \ r = b \\
? \neq ? &\Longleftarrow max(a,b,c) \ \land \ c' = c + 1
\end{aligned}
$$

Our Method

Key idea: Take the **final target value** $a_\circ$ for each borrow

- When borrow **a** (a) to **pa**, **prophesy** $a_\circ$ and model **pa** as $\langle a, a_\circ \rangle$
- When release **pa** $(\langle a, a_\circ \rangle)$, set $a_\circ = a$

```
fn max(pa: &mut int, pb: &mut int) -> &mut int {
  if *pa >= *pb { pa } else { pb }
}
fn test(mut a: int, mut b: int) {
  let pc = max(&mut a, &mut b); *pc += 1; assert!(a != b);
}
```

$$max(\langle a, a_\circ \rangle, \langle b, b_\circ \rangle, r) \Longleftarrow a \geq b \land b_\circ = b \land r = \langle a, a_\circ \rangle$$

$$max(\langle a, a_\circ \rangle, \langle b, b_\circ \rangle, r) \Longleftarrow a < b \land a_\circ = a \land r = \langle b, b_\circ \rangle$$

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

Advanced Example — with a recursive data type

(Partly omitted)

```
enum list { Cons(int, Box<list>), Nil }
fn pick(pla: &mut list) -> &mut int { match pla {
  Cons(pa, pla2) => if rand() { pa } else { pick(pla2) }
} }
fn test(mut la: list) {
  let s = sum(&la);  let pa = pick(&mut la);  *pa += 1;
  assert!(sum(&la) == s + 1);
}
```

$$pick(\langle a :: la', a_\circ :: la'_\circ \rangle, r) \iff la'_\circ = la' \ \wedge \ r = \langle a, a_\circ \rangle$$

$$pick(\langle a :: la', a_\circ :: la'_\circ \rangle, r) \iff a_\circ = a \ \wedge \ pick(\langle la', la'_\circ \rangle, r)$$

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

Simple solution! $pick(\langle la, la_\circ \rangle, \langle a, a_\circ \rangle) \equiv \mathsf{sum}(la_\circ) - \mathsf{sum}(la) = a_\circ - a$

We successfully verified this automatically in our experiment

Correctness of Our Reduction

- We formalized (a core of) Rust and our reduction from Rust to CHCs and proved soundness and completeness of our reduction

Soundness and Completeness Theorem

For any Rust function f (that does not input references),

the input-output relation of f $\Leftrightarrow$ the least solution to f in CHCs

Proof: By constructing a bisimulation between Rust and CHC resolution, modeling each prophecy $a_\circ$ as a logic variable

Table of Contents

Implementation & Experiment `https://github.com/hopv/rust-horn`

- Implemented a prototype Rust verifier that uses our method (**RustHorn**)
  - Analyzes Rust's mid-level IR, supports various features
  - Backend CHC solvers: Spacer [Komuravelli+ 2018] & HoIce [Champion+ 2018]
- Evaluated RustHorn in comparison with SeaHorn [Gurfinkel+ 2015]
  - SeaHorn: CHC-based C verifier, uses the array-based reduction
  - 58 Benchmarks: Both in Rust and C, (a) 16 from SeaHorn’s tests, (b) 42 featuring various use cases of borrowing

Overview of Experimental Results

- **RustHorn** succeeded in various benchmarks

From SeaHorn's tests

Benchmark (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

Featuring various use cases of borrowing

Benchmark (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

Recursive data types

Benchmark (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

Table of Contents

- Rust in a Nutshell
- Our Method
- Evaluation
- **Related Work**

Related Work

- CHC-based automated verification of pointer programs
  - SeaHorn [Gurfinkel+ 2015] (C/C++), JayHorn [Kahsai+ 2016] (Java): do not use ownership, easily raise false alarms
  - ConSORT [Toman+ 2020] (Java): uses a fractional ownership model, requires extra annotations on ownership
- Verification of Rust programs leveraging Rust's ownership guarantees
  - Prusti [Astrauskas+ 2018] (separation logic), Electrolysis [Ulrich 2016] (purely functional language): do not support some reference operations (e.g., split of references)

Summary

- **RustHorn**: CHC-based automated verification of **Rust** programs
  - Leverages Rust's ownership guarantees:
    a reference **pa** → the pair of its current & final target values $\langle a, a_\circ \rangle$
  - Supports various features, including recursive data types
  - Correctness proof & Experimental evaluation
- Ongoing work: Prove in Coq, support unsafe code (**RustHornBelt**)