RustHorn: CHC-based Verification for Rust Programs

ESOP 2020

Yusuke Matsushita, Takeshi Tsukada, Naoki Kobayashi

The University of Tokyo, Japan

Verification with CHCs

program safety
problem

```c
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);
}
```

CHC
satisfiability
problem

$$Mc91(n, r) \Longleftarrow n > 100 \land r = n - 10$$

$$Mc91(n, r) \Longleftarrow n \leq 100 \land Mc91(n + 11, r') \land Mc91(r', r)$$

$$r = 91 \Longleftarrow n \leq 101 \land Mc91(n, r)$$

clean first-order logic
→ good for automated verification!

$Mc91(n, r)$: `mc91(n)` returns r if it terminates

Existing method for pointers

```
void mc91p(int n, int* r) {
  if (n > 100) *r = n - 10;
  else { int s; mc91(n + 11, &s); mc91(s, r); }
}
void test(int n) {
  if (n <= 101) { int l; mc91(n, &l); assert(l == 91); }
}
```

existing method

$$
\begin{aligned}
Mc91p(n, r, h, h') &\Longleftarrow n > 100 \land h' = h\{r \leftarrow n - 10\} \\
Mc91p(n, r, h, h') &\Longleftarrow n \leq 100 \land Mc91p(n + 11, ms, h, h'') \\
&\qquad\quad \land Mc91p(h''[ms], r, h'', h') \\
h'[r] = 91 &\Longleftarrow n \leq 101 \land Mc91p(n, r, h, h')
\end{aligned}
$$

simply pass around the global memory state

h, h': the **memory state** (address ↦ value) before/after the function call

Pointers are hard

quite trivial?

```
void just_rec(int* ma) {
  if (rand() >= 0) return;
  int old_a = *ma; int b = rand(); just_rec(&b);
  assert (old_a == *ma);
}
```

important property:
&b is a fresh address

existing method

$$JustRec(ma, h, sp, h', sp', r) \impliedby h' = h \;\wedge\; sp' = sp \;\wedge\; r = \text{true}$$

$$JustRec(ma, h, sp, h', sp', r) \impliedby mb = sp'' = sp + 1 \;\wedge\; h'' = h\{mb \leftarrow b\} \;\wedge\; JustRec(mb, h'', sp'', h', sp', r) \;\wedge\; r = (h[ma] == h'[ma])$$

$$r = \text{true} \impliedby JustRec(ma, h, sp, h', sp', r) \;\wedge\; ma \le sp$$

the solver has to find a **quantified invariant** to verify the safety:

existing method is not very scalable!

$$JustRec(ma, h, sp, h', sp', r) \;:\Longleftrightarrow\; r = \text{true} \wedge\; ma \le sp \;\wedge\; sp \le sp' \;\wedge\; \forall i \le sp \,.\, h[i] = h'[i]$$

Our work

- Focus on programs in the **Rust** language
  - **pointer usage** is managed based on **borrows**
- **Novel translation** from Rust programs to CHCs
  - clears away pointers and heaps
    pointer `ma` → **pair of values** $\langle a, a_\circ \rangle$
  - applied to **automated verification**
- Proof of the **correctness** and
  experimental evaluation of the **effectiveness**

`just_rec` revisited

```
void just_rec(int* ma) {
  if (rand() >= 0) return;
  int old_a = *ma; int b = rand(); just_rec(&b);
  assert (old_a == *ma);
}
```

follows Rust's borrow discipline

our method

no representation of the global memory state!

$$JustRec(\langle a, a_\circ \rangle, r) \impliedby a_\circ = a \ \land \ r = \mathsf{true}$$

$$JustRec(\langle a, a_\circ \rangle, r) \impliedby mb = \langle b, b_\circ \rangle \ \land \ JustRec(mb, r') \ \land \ a_\circ = a \ \land \ r = (a == a_\circ)$$

$$r = \mathsf{true} \impliedby JustRec(\langle a, a_\circ \rangle, r)$$

piece of cake for solvers!

Outline

- **Overview of our method**
- Proof of the correctness
- Experiments

What is **borrow**?

- **Borrow**: temporary **transfer** of **update** permission
  - while data is borrowed, the lender cannot even read it

```
int a = 5;
{ int* ma = &a; *ma += 3; }
print(a); // 8
```

borrow

update

release ma

end of borrow

the new value is passed from ma to a

Our method

Rust's **mutable reference** `&mut T`

- **pointer** `ma` → **pair of values** $\langle a, a_\circ \rangle$
  - a: the **current value** of data
    - can be freely **updated** by pointer `ma`
  - $a_\circ$: the **new value** of data **at the end of borrow**
    - **constrained** to a at the time `ma` is **released**
    - the **original owner** must know **only** $a_\circ$

taking **information** from the **future!**

related to **prophecy variables** [Abadi & Lamport 1991] [Jung+ 2020]

Example: take_max

```
int* take_max(int* ma, int* mb) {
  if (*ma >= *mb) return ma; else return mb;
}
void test(int a, int b) {
  { int* mc = take_max(&a, &b); *mc += 1; }
  assert (a != b);
}
```

release mb

borrow a&b

update

end of borrow of a&b

release mc

read a&b

sample execution

call test(5,3)

→ **borrow** a&b to ma&mb; **call** take_max(ma,mb)

→ **release** mb; **move** ma to mc (∵ 5≧3)

→ mc **updates** the data 5→6; **release** mc (=ma)

→ **borrow** of a&b **ends**; **assert** a≠b (i.e. 6≠3)

$a_\circ$ $b_\circ$

Example: `take_max`

```c
int* take_max(int* ma, int* mb) {
  if (*ma >= *mb) return ma; else return mb;
}
void test(int a, int b) {
  { int* mc = take_max(&a, &b); *mc += 1; }
  assert (a != b);
}
```

our method

release mb

$$\mathit{TakeMax}(\langle a, a_\circ \rangle, \langle b, b_\circ \rangle, r) \iff a \geq b \ \land\ \boxed{b_\circ = b} \ \land\ r = \langle a, a_\circ \rangle$$

$$\mathit{TakeMax}(\langle a, a_\circ \rangle, \langle b, b_\circ \rangle, r) \iff a < b \ \land\ a_\circ = a \ \land\ r = \langle b, b_\circ \rangle$$

$$\mathit{Test}(a, b, r) \iff \mathit{TakeMax}(\langle a, \boxed{a_\circ} \rangle, \langle b, \boxed{b_\circ} \rangle, \langle c, c_\circ \rangle) \ \land\ \boxed{c' = c + 1}$$

borrow a&b

$$\land\ \boxed{c_\circ = c'} \ \land\ r = \boxed{(a_\circ \mathrel{!=} b_\circ)}$$

update

$$r = \text{true} \iff \mathit{Test}(a, b, r)$$

release mc **read** a&b

What features are supported?

- Our method supports:
  - **recursive data types** (e.g. **lists** and **trees**)
  - **recursions** and **loops**
  - various **borrow patterns** *(under non-lexical lifetimes)*, including **reborrows**

Outline

- Overview of our method
- **Proof of the correctness**
- Experiments

Proof of the correctness

- **Correctness** is proved via operational semantics based on **prophecy variables** ($a_\circ$)
  - The *execution sequence* corresponds to the *resolution sequence* on the output CHCs
  - **Guarantee of Rust's type system**:
    a *prophecy variable* is *rightly resolved* into the value at the end of borrow *before* the original owner accesses it

Operational semantics with prophecy variables

```
int* take_max(int* ma, int* mb) {
  if (*ma >= *mb) return ma; else return mb;
}
void test(int a, int b) {
  { int* mc = take_max(&a, &b); *mc += 1; }
  assert (a != b);
}
```

call `test`(5,3) [a=5,b=3]

→ **borrow** & **call** `take_max` [ma=$\langle 5,a_\circ \rangle$,mb=$\langle 3,b_\circ \rangle$] [a=$a_\circ$,b=$b_\circ$]

→ **release** mb [ma=$\langle 5,a_\circ \rangle$] [a=$a_\circ$,b=3]

→ [mc=$\langle 5,a_\circ \rangle$,a=$a_\circ$,b=3]

→ **update** [mc=$\langle 6,a_\circ \rangle$,a=$a_\circ$,b=3]

→ **release** mc [a=6,b=3]

→ **assert** $6 \neq 3$

Outline

- Overview of Our Method
- Proof of the Correctness
- **Experiments**

Experiments

- Implemented **RustHorn**, a prototype CHC-based verifier based on our method
  - CHC solver: **Spacer** [Komuravelli+ 2014] or **HoIce** [Champion+ 2018]
- Evaluated **RustHorn** in comparison to **SeaHorn**
  - **SeaHorn** [Gurfinkel+ 2015]: CHC-based verifier for C based on the existing method for pointers
  - Benchmarks:
    1. **SeaHorn's tests** that suit the core of Rust
    2. Ones featuring **various pointer usages in Rust**

Experimental results

Group	Instance	Property	RustHorn w/Spacer	RustHorn w/HoIce	SeaHorn w/Spacer as is	SeaHorn w/Spacer modified
simple	01	safe	<0.1	<0.1	<0.1	
	04-recursive	safe	0.5	timeout	0.8	
	05-recursive	unsafe	<0.1	<0.1	<0.1	
	06-loop	safe	timeout	0.1	timeout	
	hhk2008	safe	timeout	40.5	<0.1	
	unique-scalar	unsafe	<0.1	<0.1	<0.1	
bmc	1	safe	0.2	<0.1	<0.1	
		unsafe	0.2	<0.1	<0.1	
	2	safe	timeout	0.1	<0.1	
		unsafe	<0.1	<0.1	<0.1	
	3	safe	<0.1	<0.1	<0.1	
		unsafe	<0.1	<0.1	<0.1	
	diamond-1	safe	0.1	<0.1	<0.1	
		unsafe	<0.1	<0.1	<0.1	
	diamond-2	safe	0.2	<0.1	<0.1	
		unsafe	<0.1	<0.1	<0.1	
inc-max	base	safe	<0.1	<0.1	false alarm	<0.1
		unsafe	<0.1	<0.1	<0.1	<0.1
	base/3	safe	<0.1	<0.1	false alarm	
		unsafe	0.1	<0.1	<0.1	
	repeat	safe	0.1	timeout	false alarm	0.1
		unsafe	<0.1	0.4	<0.1	<0.1
	repeat/3	safe	0.2	timeout	<0.1	
		unsafe	<0.1	1.3	<0.1	
swap-dec	base	safe	<0.1	<0.1	false alarm	<0.1
		unsafe	0.1	timeout	<0.1	<0.1
	base/3	safe	0.2	timeout	false alarm	<0.1
		unsafe	0.4	0.9	<0.1	0.1
	exact	safe	0.1	0.5	false alarm	timeout
		unsafe	<0.1	26.0	<0.1	<0.1
	exact/3	safe	timeout	timeout	false alarm	false alarm
		unsafe	<0.1	0.4	<0.1	<0.1

just-rec	base	safe	<0.1	<0.1	<0.1
		unsafe	<0.1	0.1	<0.1
linger-dec	base	safe	<0.1	<0.1	false alarm
		unsafe	<0.1	0.1	<0.1
	base/3	safe	<0.1	<0.1	false alarm
		unsafe	<0.1	7.0	<0.1
	exact	safe	<0.1	<0.1	false alarm
		unsafe	<0.1	0.2	<0.1
	exact/3	safe	<0.1	<0.1	false alarm
		unsafe	<0.1	0.6	<0.1
lists	append	safe	tool error	<0.1	false alarm
		unsafe	tool error	0.2	0.1
	inc-all	safe	tool error	<0.1	false alarm
		unsafe	tool error	0.3	<0.1
	inc-some	safe	tool error	<0.1	false alarm
		unsafe	tool error	0.3	0.1
	inc-some/2	safe	tool error	timeout	false alarm
		unsafe	tool error	0.3	0.4
trees	append-t	safe	tool error	<0.1	timeout
		unsafe	tool error	0.3	0.1
	inc-all-t	safe	tool error	timeout	timeout
		unsafe	tool error	0.1	<0.1
	inc-some-t	safe	tool error	timeout	timeout
		unsafe	tool error	0.3	0.1
	inc-some/2-t	safe	tool error	timeout	false alarm
		unsafe	tool error	0.4	0.1

- RustHorn+HoIce: **good** at recursive data types
- **RustHorn** is largely **comparable** to **SeaHorn**

Related work

- **CHC-based verification of programs with pointers**
  - [Gurfinkel+ 2015] CHC-based verifier for C
  - [Kahsai+ 2016] CHC-based verifier for Java
- **Verification for Rust**
  - [Jung+ 2018] Verify Rust libraries with Coq
  - [Ullrich 2016] Translate some Rust programs into functional programs
  - [Hahn 2016] [Müller+ 2018] [Erdin 2019] Verify Rust programs on Viper

Related work

- **Verification using ownership/permission**
  - [Bornat+ 2005] [Müller+ 2016] [Jung+ 2015] Separation logic with ownership
  - [Cohen+ 2009] [Barnett+ 2011] Verification platform with ownership
- **Prophecy variables** — Information of the future
  - [Abadi & Lamport 1991] Idea of prophecy variables
  - [Jung+ 2020] Separation logic with prophecy variables

Conclusion

- **Novel translation** from **Rust** programs to CHCs
  - **pointer** `ma` → **pair of values** $\langle a, a_\circ \rangle$
  - supports recursive data types, reborrows, etc.
  - applied to **automated verification**
- Proof of the **correctness** and experimental evaluation of the **effectiveness**