Deductive verification of programs with Rust-style typing

Xavier Denis

Université Paris-Saclay, CNRS, Inria,
Laboratoire de Recherche en Informatique, 91405, Orsay, France.

November 23, 2020

Motivation

- We need to use *pointers*, and also *reason* about them.
- C-style pointers are *too powerful*.
- Introduce issues: uninitialized memory, aliasing
- Makes reasoning *highly complex*.

Overwriting memcpy

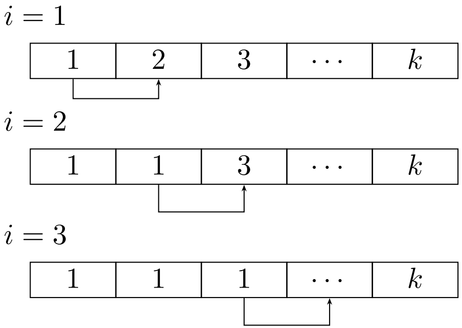
```
void memcpy(char * src, char * dest, int len) {
  for(int i = 0; i < len; i++) dest[i] = src[i]
}
```

What happens if `src` and `dest` *overlap*?

Overwriting memcpy

```
void memcpy(char * src, char * dest, int len) {
  for(int i = 0; i < len; i++) dest[i] = src[i]
}
```

What happens if `src` and `dest` *overlap*?



Ownership in Rust

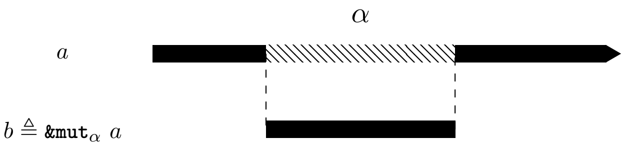
- In Rust, every cell of memory has a *unique owner*.
- This turns the heap into a *forest*.
- Rust adds *borrows*, a form of pointers with a static *lifetime*.
- Safety of borrows is checked statically by compiler.
- This typing discipline gives Rust *(manual) memory safety*

Borrows & Lifetimes

Mutability XOR Sharing

- Mutable borrows are *exclusive*, but can be turned into *shareable* immutable borrows.
- Borrows are implemented as pointers.
- A borrow must be released by the end of its *lifetime*.

Borrows & Lifetimes



a is frozen until the end of α, *even if b is freed early*.

Borrows & Lifetimes

```
fn memcpy(src: &mut [u8], dst: &mut [u8]) {
  for (s, d) in src.iter_mut().zip(dst.iter()) {
      *s = *d
  }
}

fn main () {
    let mut x = vec![1,2,3,4,5];
    let y = &mut x[0..3];
    let z = &mut x[1..4];
    memcpy(y, z)
}

error[E0499]: cannot borrow `x` as mutable more
than once at a time
```

Contributions

- Based on work of RustHorn (ESOP 2020)
- Deductive verification by translation to *functional language* for Rust-style languages.
- Proof of *safety* using original simulation approach between traces and configurations.
- Implemented this translation as a proof-of-concept extension to the Rust compiler targeting *Why3*.

Starting Point

Source: MiniMir, a kernel for languages with borrows
Target: Functional language with *any/assume non-determinism* and assertions.

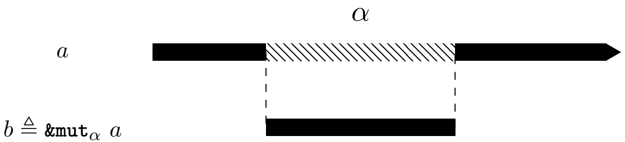
any/assume non-determinism

```
let x = any in
let y = x + 1 in
assume { 1 <= y };
let z = y + x + 2 in
assert { z >= 3 }
```

Translating

Translating borrows

Mutable borrows are translated to a *pair* of values: the current and *final* value that we *divine* at the creation of a borrow.



During α, a is *frozen* and *inaccessible*.
Intuitively, the final value stored in b is the value of a after α.

Translating

Translating borrows

Mutable borrows are translated to a *pair* of values: the current and *final* value that we *divine* at the creation of a borrow.

```
let b = { * = a, ^ = any } in
let a = ^ b in
....
let b = { b with * = .. } in
assume { * b = ^ b }
```

During α, a is *frozen* and *inaccessible*.
Intuitively, the final value stored in b is the value of a after α.

Example: Mutating a reference

```
fn main () {
    let mut x = 10;
    let y = &mut x

    * y = 15;

    assert_eq!(x, 15);
}
```

```
x := 10;
y := &mut_α x;
t_1 := 15;
t_2 := &mut_α t_1;
swap(y, t_2);
drop(t_2);
drop(y);
thaw α;
t_3 := x = 15;
assert _3;
t_4 := ();
return _4;
```

Example: Mutating a reference

```
x := 10;
y := &mutα x;
t1 := 15;
t2 := &mutα t1;
swap(y, t2);
drop(t2);
drop(y);
thaw α;
t3 := x = 15;
assert _3;
t4 := ();
return _4;
```

Example: Mutating a reference

```
x := 10;                let rec main () =
y := &mut_α x;            let x = 10 in
t_1 := 15;                let y = {* = x, ^ = any} in
t_2 := &mut_α t_1;        let x = ^ y in
swap(y, t_2);             let t_1 = 15 in
drop(t_2);                let t_2 = {* = s, ^ = any} in
drop(y);                  let t_1 = ^ t_2 in
thaw α;                   let t = * t_2 in
t_3 := x = 15;            let t_2 = {t_2 with * = * y} in
assert _3;                let y = {y with * = t} in
t_4 := ();                assume { * t_2 = ^ t_2 };
return _4;                assume { * y = ^ y };
                          assert { x = 15 }
```

Example: Mutating a reference

```
let rec main () =
  let x = 10 in
  let y = {* = x, ˆ = any} in
  let x = ˆ y in
  let t1 = 15 in
  let t2 = {* = s, ˆ = any} in
  let t1 = ˆ t2 in
  let t = * t2 in
  let t2 = {t2 with * = * y} in
  let y = {y with * = t} in
  assume { * t2 = ˆ t2 };
  assume { * y = ˆ y };
  assert { x = 15 }
```

Environment

```
x = 10
```

Example: Mutating a reference

```
let rec main () =
  let x = 10 in
  let y = {* = x, ^ = any} in
  let x = ^ y in
  let t1 = 15 in
  let t2 = {* = s, ^ = any} in
  let t1 = ^ t2 in
  let t = * t2 in
  let t2 = {t2 with * = * y} in
  let y = {y with * = t} in
  assume { * t2 = ^ t2 };
  assume { * y = ^ y };
  assert { x = 15 }
```

Environment

x = 10

y = $\{10, \boldsymbol{v_1}\}$

Example: Mutating a reference

```
let rec main () =
  let x = 10 in
  let y = {* = x, ^ = any} in
  let x = ^ y in
  let t1 = 15 in
  let t2 = {* = s, ^ = any} in
  let t1 = ^ t2 in
  let t = * t2 in
  let t2 = {t2 with * = * y} in
  let y = {y with * = t} in
  assume { * t2 = ^ t2 };
  assume { * y = ^ y };
  assert { x = 15 }
```

Environment

x = v_1

y = $\{10,\ v_1\}$

Example: Mutating a reference

```
let rec main () =
  let x = 10 in
  let y = {* = x, ^ = any} in
  let x = ^ y in
  let t₁ = 15 in
  let t₂ = {* = s, ^ = any} in
  let t₁ = ^ t₂ in
  let t = * t₂ in
  let t₂ = {t₂ with * = * y} in
  let y = {y with * = t} in
  assume { * t₂ = ^ t₂ };
  assume { * y = ^ y };
  assert { x = 15 }
```

Environment

x = v_1

y = {10, v_1}

t_1 = 15

Example: Mutating a reference

```
let rec main () =
  let x = 10 in
  let y = {* = x, ^ = any} in
  let x = ^ y in
  let t₁ = 15 in
  let t₂ = {* = s, ^ = any} in
  let t₁ = ^ t₂ in
  let t = * t₂ in
  let t₂ = {t₂ with * = * y} in
  let y = {y with * = t} in
  assume { * t₂ = ^ t₂ };
  assume { * y = ^ y };
  assert { x = 15 }
```

Environment

x = v_1

y = $\{10,\ v_1\}$

t_1 = 15

t_2 = $\{15,\ v_2\}$

Example: Mutating a reference

```
let rec main () =
  let x = 10 in
  let y = {* = x, ^ = any} in
  let x = ^ y in
  let t1 = 15 in
  let t2 = {* = s, ^ = any} in
  let t1 = ^ t2 in
  let t = * t2 in
  let t2 = {t2 with * = * y} in
  let y = {y with * = t} in
  assume { * t2 = ^ t2 };
  assume { * y = ^ y };
  assert { x = 15 }
```

Environment

x = v_1

y = $\{10,\ v_1\}$

$\boldsymbol{t_1 = v_2}$

$t_2 = \{15,\ v_2\}$

Example: Mutating a reference

```
let rec main () =
  let x = 10 in
  let y = {* = x, ^ = any} in
  let x = ^ y in
  let t1 = 15 in
  let t2 = {* = s, ^ = any} in
  let t1 = ^ t2 in
  let t = * t2 in
  let t2 = {t2 with * = * y} in
  let y = {y with * = t} in
  assume { * t2 = ^ t2 };
  assume { * y = ^ y };
  assert { x = 15 }
```

Environment

x = v_1
y = $\{10,\ v_1\}$
t_1 = v_2
t_2 = $\{15,\ v_2\}$
t = 15

Example: Mutating a reference

```
let rec main () =
  let x = 10 in
  let y = {* = x, ^ = any} in
  let x = ^ y in
  let t1 = 15 in
  let t2 = {* = s, ^ = any} in
  let t1 = ^ t2 in
  let t = * t2 in
  let t2 = {t2 with * = * y} in
  let y = {y with * = t} in
  assume { * t2 = ^ t2 };
  assume { * y = ^ y };
  assert { x = 15 }
```

Environment

x = v_1
y = {10, v_1}
t_1 = v_2
t_2 = {10, v_2}
t = 15

Example: Mutating a reference

```
let rec main () =
  let x = 10 in
  let y = {* = x, ^ = any} in
  let x = ^ y in
  let t1 = 15 in
  let t2 = {* = s, ^ = any} in
  let t1 = ^ t2 in
  let t = * t2 in
  let t2 = {t2 with * = * y} in
  let y = {y with * = t} in
  assume { * t2 = ^ t2 };
  assume { * y = ^ y };
  assert { x = 15 }
```

Environment

x = v_1

y = {15, $\boldsymbol{v_1}$}

t_1 = v_2

t_2 = {10, v_2}

t = 15

Example: Mutating a reference

```
let rec main () =
  let x = 10 in
  let y = {* = x, ^ = any} in
  let x = ^ y in
  let t1 = 15 in
  let t2 = {* = s, ^ = any} in
  let t1 = ^ t2 in
  let t = * t2 in
  let t2 = {t2 with * = * y} in
  let y = {y with * = t} in
  assume { * t2 = ^ t2 };
  assume { * y = ^ y };
  assert { x = 15 }
```

Environment

x = v_1
y = {15, v_1}
t_1 = v_2
t_2 = {10, v_2}
t = 15

Equalities

10 = $\boldsymbol{v_2}$

Example: Mutating a reference

```
let rec main () =
  let x = 10 in
  let y = {* = x, ^ = any} in
  let x = ^ y in
  let t1 = 15 in
  let t2 = {* = s, ^ = any} in
  let t1 = ^ t2 in
  let t = * t2 in
  let t2 = {t2 with * = * y} in
  let y = {y with * = t} in
  assume { * t2 = ^ t2 };
  assume { * y = ^ y };
  assert { x = 15 }
```

Environment

x = v_1
y = $\{15,\ v_1\}$
t_1 = v_2
t_2 = $\{10,\ v_2\}$
t = 15

Equalities

10 = v_2
15 = $\boldsymbol{v_1}$

Example: Mutating a reference

```
let rec main () =
  let x = 10 in
  let y = {* = x, ^ = any} in
  let x = ^ y in
  let t1 = 15 in
  let t2 = {* = s, ^ = any} in
  let t1 = ^ t2 in
  let t = * t2 in
  let t2 = {t2 with * = * y} in
  let y = {y with * = t} in
  assume { * t2 = ^ t2 };
  assume { * y = ^ y };
  assert { x = 15 }
```

Environment

x = v_1
y = $\{15,\ v_1\}$
t_1 = v_2
t_2 = $\{10,\ v_2\}$
t = 15

Equalities

10 = v_2
15 = v_1

Safety

Theorem (Safety)

Given a well-typed MiniMir program $\vdash \mathcal{P}$, if $[\![\mathcal{P}]\!]$ is safe, then $\mathcal{P}$ is safe.

To prove this we establish a simulation between *MiniMir traces* and *anyML configurations*.

Preservation

Lemma (Progress)

Given a MiniMir trace $\Theta = C \rightarrow^{*}_{\mathcal{P}} C'$ *and a anyML configuration such that* $C \sim_{\mathcal{P}} K$*, if* K *is not stuck then* C *is not stuck.*

Lemma (Preservation of Simulation)

Given a MiniMir trace $\Theta = C \rightarrow^{*}_{\mathcal{P}} C'$ *and a anyML configuration* K *such that* $\Theta \sim_{\mathcal{P}} K$*, if* $C \rightarrow_{\mathcal{P}} C''$*, there exists a* K' *such that* $K \rightarrow K'$ *and* $C'' \rightarrow^{*}_{\mathcal{P}} C' \sim_{\mathcal{P}} K'$*.*

Simulation

- The simulation $\sim_{\mathcal{P}}$ gives a *readback* of MiniMir heap to anyML environments.
- How do we readback a mutable borrow? We *prophecise* its final value.
- A prophecy is the value an address a as type T borrowed for α will have at the end of α.

Prophecy Maps

For a MiniMir trace $\Theta = C \to^* C' \not\to$, we calculate a *prophecy map* by walking Θ *backwards*.
At each **`thaw`**, we record the values of all variables being *unfrozen*.

$$C \longrightarrow^* C' = \texttt{thaw}\ \alpha \longrightarrow^* C'' = \texttt{thaw}\ \beta \longrightarrow^* \cdots$$

$$\mathrm{Proph}(C) \longleftarrow \mathrm{Proph}(C') \longleftarrow \mathrm{Proph}(C'') \longleftarrow \cdots$$

Proving preservation: `&mut`

```
x := 10;
y := &mut_α x;
...
...
drop(y);
...
```

```
let rec main () =
  let x = 10 in
  let y = {* = x,^ = any} in
  let x = ^ y in
  ...
  assume { * y = ^ y };
  ...
```

MiniMir Frame / Heap

anyML Environment

$x \mapsto a, y \mapsto b \mid a \mapsto 10, b \mapsto a$

$x \mapsto 10, y \mapsto (10, ?)$

...

Proving preservation: `&mut`

```
x := 10;
y := &mut_α x;
...
...
drop(y);
...
thaw α
```

```
let rec main () =
 let x = 10 in
 let y = {* = x,^ = any} in
 let x = ^ y in
 ...
 assume { * y = ^ y };
 ...
```

MiniMir Frame / Heap

anyML Environment

$x \mapsto a, y \mapsto b \mid a \mapsto 10, b \mapsto a$

$x \mapsto 10, y \mapsto (10, ?)$

...

$x \mapsto a \mid a \mapsto 15$

Proving preservation: `&mut`

```
x := 10;
y := &mut_α x;
...
...
drop(y);
...
thaw α
```

```
let rec main () =
  let x = 10 in
  let y = {* = x, ^ = 15} in
  let x = ^ y in
  ...
  assume { * y = ^ y };
  ...
```

MiniMir Frame / Heap

anyML Environment

$x \mapsto a, y \mapsto b \mid a \mapsto 10, b \mapsto a$

$x \mapsto 10, y \mapsto (10, 15)$

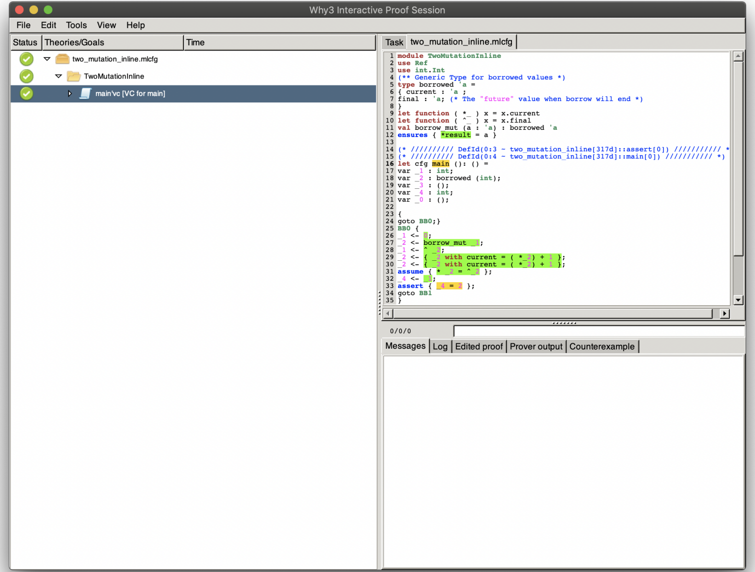
...

$x \mapsto a \mid a \mapsto 15$

Limitations and Difficulties

1. Complex syntactic proof with many cases
2. Proof does not cover function calls
3. Requires reasoning about future states

Current Work: Experimentation



Current Work: Experimentation

1. *Creusot*: a prototype implementation targeting Why3
2. Translates from *MIR* to *MLCFG*, a CFG front-end to *WhyML*
3. Extended with pre/post-conditions, invariants.

Conclusion

- Mutable borrows constrain pointers through non-aliasing.
- Leverage this to verify Rust-style programs by *translation* to functional language.
- Represent borrows as *pairs of current and final value*.
- Use original simulation between *traces and configurations* to prophecise final values.
- Implemented a PoC tool to experimentally validate approach.

Future Work

- Exploring a new proof based on *RustBelt*
- Specifications for Rust
- Extend with support for other Rust features: inner mutability, trait objects, closures.