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?
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 \textit{overlap}?

- $i = 1$
  
  \begin{center}
  \begin{tabular}{|c|c|c|c|c|c|}
  \hline
  1 & 2 & 3 & \cdots & k \\
  \hline
  \end{tabular}
  \end{center}

- $i = 2$
  
  \begin{center}
  \begin{tabular}{|c|c|c|c|c|c|}
  \hline
  1 & 1 & 3 & \cdots & k \\
  \hline
  \end{tabular}
  \end{center}

- $i = 3$
  
  \begin{center}
  \begin{tabular}{|c|c|c|c|c|c|}
  \hline
  1 & 1 & 1 & \cdots & k \\
  \hline
  \end{tabular}
  \end{center}
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

\[ a \triangleq \text{mut}_\alpha a \]

\( a \) is frozen until the end of \( \alpha \), even if \( b \) is freed early.
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.
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.

\[ b \triangleq \&\text{mut}_\alpha a \]

During \( \alpha \), \( a \) is frozen and inaccessible. Intuitively, the final value stored in \( b \) is the value of \( a \) after \( \alpha \).
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 $\alpha$, $a$ is frozen and inaccessible. Intuitively, the final value stored in $b$ is the value of $a$ after $\alpha$. 
Example: Mutating a reference

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

    *y = 15;

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

```plaintext
x := 10;
y := &mut_\alpha x;
t_1 := 15;
t_2 := &mut_\alpha t_1;
swap(y, t_2);
drop(t_2);
drop(y);
thaw_\alpha;
t_3 := x = 15;
assert _3;
t_4 := ()
return _4;
```
Example: Mutating a reference

\[
x := 10;
y := \&\text{mut}_\alpha x;
t_1 := 15;
t_2 := \&\text{mut}_\alpha t_1;
\text{swap}(y, t_2);
\text{drop}(t_2);
\text{drop}(y);
\text{thaw} \alpha;
t_3 := x = 15;
\text{assert } _3;
t_4 := ()
\text{return } _4;
\]
Example: Mutating a reference

let rec main () =

x := 10;
let x = 10 in
y := &mut_\alpha x;
let y = {* = x, ^ = any} in
t_1 := 15;
let x = ^ y in
t_2 := &mut_\alpha t_1;
let t_1 = 15 in
swap(y, t_2);
let t_2 = {* = s, ^ = any} in
drop(t_2);
let t_1 = ^ t_2 in
drop(y);
let t = ^ t_2 in
thaw \alpha;
let t_2 = {t_2 with * = * y} in
t_3 := x = 15;
let y = {y with * = t} in
assert _3;
assume { * t_2 = ^ t_2 };
t_4 := ();
assume { * y = ^ y };
return _4;
assert { x = 15 }
Example: Mutating a reference

```ocaml
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
```
Example: Mutating a reference

```ocaml
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, $v_1$)
Example: Mutating a reference

```ml
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 = v1
y = (10, v1)
```
Example: Mutating a reference

```ml
let rec main () =
  let x = 10 in
  let y = { * = x, ^ = any } in
  let x = ^ y in
  let t_1 = 15 in
  let t_2 = { * = s, ^ = any } in
  let t_1 = ^ t_2 in
  let t = * t_2 in
  let t_2 = { t_2 with * = * y } in
  let y = { y with * = t } in
  assume { * t_2 = ^ t_2 };
  assume { * y = ^ y };
  assert { x = 15 }
```

**Environment**

\[
\begin{align*}
  x &= v_1 \\
  y &= (10, v_1) \\
  t_1 &= 15
\end{align*}
\]
Example: Mutating a reference

```ocaml
let rec main () =
  let x = 10 in
  let y = {* = x, ^ = any} in
  let x = ^ y in
  let t1 = 15 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 = v1
y = (10, v1)
t1 = v2
t2 = (15, v2)
Example: Mutating a reference

```ml
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

\[
\begin{align*}
x &= v₁ \\
y &= (10, v₁) \\
t₁ &= v₂ \\
t₂ &= (15, v₂) \\
t &= 15
\end{align*}
\]
Example: Mutating a reference

```ocaml
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

```ocaml
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

- \( x = v_1 \)
- \( y = (15, v_1) \)
- \( t_1 = v_2 \)
- \( t_2 = (10, v_2) \)
- \( t = 15 \)
Example: Mutating a reference

```ocaml
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**
- \( 10 = v_2 \)
Example: Mutating a reference

```ocaml
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**

- \( 10 = v_2 \)
- \( 15 = v_1 \)
Example: Mutating a reference

```
let rec main () =
    let x = 10 in
    let y = {* = x, ^ = any} in
    let x = ^ y in
    let t_1 = 15 in
    let t_2 = {* = s, ^ = any} in
    let t_1 = ^ t_2 in
    let t = * t_2 in
    let t_2 = {t_2 with * = * y} in
    let y = {y with * = t} in
    assume { * t_2 = ^ t_2 };
    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 $\llbracket \mathcal{P} \rrbracket$ 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 $\Theta = C \xrightarrow{\mathcal{P}} C'$ 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 $\Theta = C \xrightarrow{\mathcal{P}} C'$ and a anyML configuration $K$ such that $\Theta \sim_{\mathcal{P}} K$, if $C \xrightarrow{\mathcal{P}} C''$, there exists a $K'$ such that $K \xrightarrow{\mathcal{P}} K'$ and $C'' \xrightarrow{\mathcal{P}} C' \sim_{\mathcal{P}} K'$. 
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 $\alpha$ will have at the end of $\alpha$. 
Prophecy Maps

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

\[
\begin{align*}
C & \rightarrow^* C' = \text{thaw } \alpha & \rightarrow^* C'' = \text{thaw } \beta & \rightarrow^* \cdots \\
\text{Proph}(C') & \leftarrow \text{Proph}(C'') & \leftarrow \text{Proph}(C''') & \leftarrow \cdots
\end{align*}
\]
Proving preservation: &mut

```
x := 10;
y := &mut_\alpha 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**

```
x \leftrightarrow a, y \leftrightarrow b \mid a \leftrightarrow 10, b \leftrightarrow a
```

**anyML Environment**

```
x \leftrightarrow 10, y \leftrightarrow (10, ?)
```

...
Proving preservation: \&mut

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

**MiniMir Frame / Heap**

\[
\text{let } x = 10 \text{ in } \\
\text{let } y = \{ * = x, \hat = any \} \text{ in } \\
\text{let } x = \hat y \text{ in } \\
\text{drop}(y); \\
\text{thaw } \alpha
\]

**anyML Environment**

\[
\text{let } x = 10 \text{ in } \\
\text{let } y = (10, ?) \text{ in } \\
\text{assume } \{ * y = \hat y \};
\]

\[
\text{let } x = 10 \text{ in } \\
\text{anyML Environment}
\]

\[
\begin{align*}
\text{MiniMir Frame / Heap} & \quad \text{anyML Environment} \\
x \mapsto a, y \mapsto b | a \mapsto 10, b \mapsto a & \quad x \mapsto 10, y \mapsto (10, ?) \\
\text{...} & \\
x \mapsto a | a \mapsto 15
\end{align*}
\]
Proving preservation: \texttt{\&mut}

\begin{align*}
\texttt{x} & := 10; \\
\texttt{y} & := \texttt{\&mut}_\alpha \texttt{x}; \\
\texttt{...} & \\
\texttt{...} & \\
\texttt{drop(y);} & \\
\texttt{...} & \\
\texttt{thaw} & \alpha
\end{align*}

\textit{MiniMir Frame / Heap}

\begin{align*}
x & \mapsto a, y \mapsto b \mid a \mapsto 10, b \mapsto a \\
\texttt{...} & \\
x & \mapsto a \mid a \mapsto 15
\end{align*}

\textit{anyML Environment}

\begin{align*}
\texttt{let rec main () =} & \\
\texttt{let x = 10 in} & \\
\texttt{let y = \{* = x, ^ = 15\} in} & \\
\texttt{let x = ^ y in} & \\
\texttt{...} & \\
\texttt{assume \{ * y = ^ y \};} & \\
\texttt{...} & \\
x & \mapsto 10, y \mapsto (10, 15)
\end{align*}
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.