

Formal Verification of Borrow-Checking by Local Commutation Diagrams

Alban Reynaud Michez



The Rust programming language

- Between 65-75% of software vulnerabilities are memory vulnerabilities.
 - use-after-free, out-of-bound accesses, buffer overflow, etc.
- Goal of Rust: be a safe system language.
- **If it compiles, then it's safe**

How Rust enforces safety

Values are not copiable by default (ownership-based type system).

How Rust enforces safety

Values are not copiable by default (ownership-based type system).

Instead of pointers we are using **references** that are constrained by rules:

How Rust enforces safety

Values are not copiable by default (ownership-based type system).

Instead of pointers we are using **references** that are constrained by rules:

1. At any time, a place can be borrowed by at most one *mutable reference* `&mut T`. If so, it cannot be used.

How Rust enforces safety

Values are not copiable by default (ownership-based type system).

Instead of pointers we are using **references** that are constrained by rules:

1. At any time, a place can be borrowed by at most one *mutable reference* `&mut T`. If so, it cannot be used.
2. At any time, a place can be borrowed by as many *immutable references* `&T`. If so, it can be read but not written.

How Rust enforces safety

Values are not copiable by default (ownership-based type system).

Instead of pointers we are using **references** that are constrained by rules:

1. At any time, a place can be borrowed by at most one *mutable reference* `&mut T`. If so, it cannot be used.
2. At any time, a place can be borrowed by as many *immutable references* `&T`. If so, it can be read but not written.
3. A place cannot be mutably and immutably borrowed.

Example

```
fn f(x: i32, y: i32, b: bool) {  
  let z;  
  if b { z = &mut x; }  
  else { z = &mut y; }  
  *z += 1;  
  x = 4;  
  assert(*z == 4);  
}
```

- The reference z may borrow the variable x.

Example

```
fn f(x: i32, y: i32, b: bool) {  
    let z;  
    if b { z = &mut x; }  
    else { z = &mut y; }  
    *z += 1;  
    x = 4;  
    assert(*z == 4);  
}
```

- The reference z may borrow the variable x.
- We can use z in write mode.

Example

```
fn f(x: i32, y: i32, b: bool) {  
  let z;  
  if b { z = &mut x; }  
  else { z = &mut y; }  
  *z += 1;  
  x = 4;  
  assert(*z == 4);  
}
```

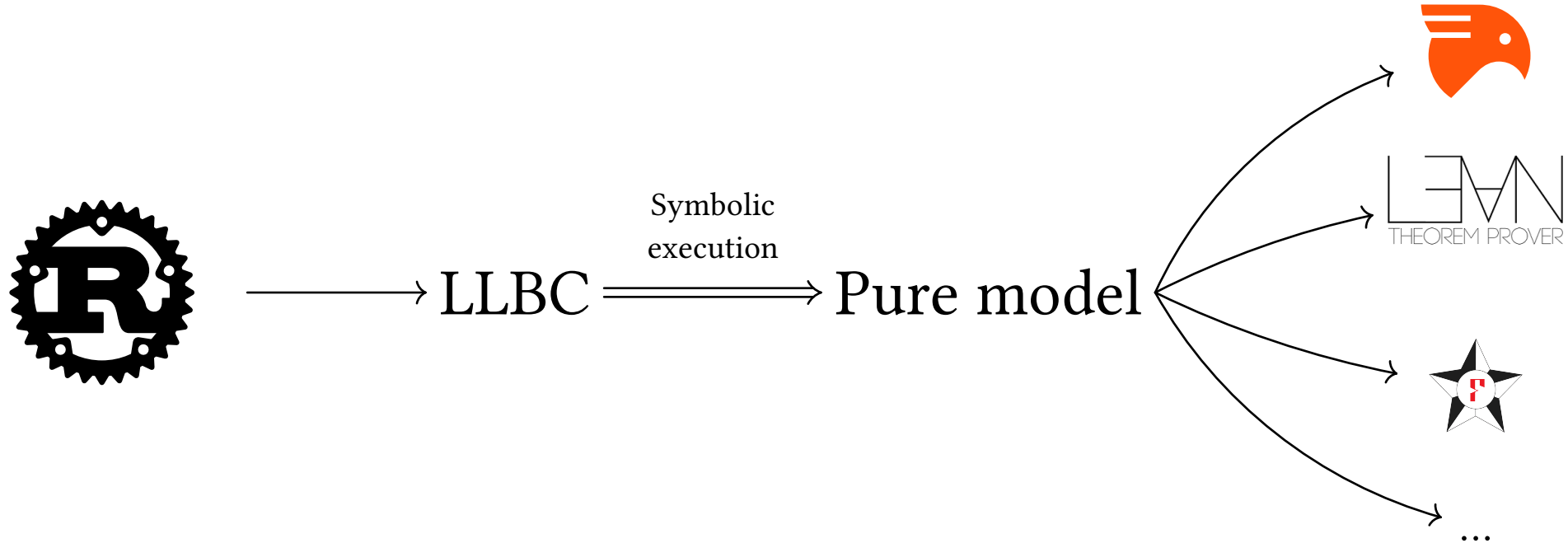
- The reference z may borrow the variable x.
- We can use z in write mode.
- The borrow is temporary, we can reuse the variable x...

Example

```
fn f(x: i32, y: i32, b: bool) {  
    let z;  
    if b { z = &mut x; }  
    else { z = &mut y; }  
    *z += 1;  
    x = 4;  
    assert(*z == 4);  
}
```

- The reference z may borrow the variable x.
- We can use z in write mode.
- The borrow is temporary, we can reuse the variable x...
- ... but this must end the reference z \Rightarrow compilation error.

The Aeneas project: proof by functional translation



The Aeneas project: proof by functional translation

Supported features:

- Basic type constructions (borrows, tuples, enumerations, vectors, boxes);
- Loops;
- Traits.

Unsupported features:

- Nested borrows;
- Unsafe Rust;
- Concurrency.

The LLBC semantics

```
fn f(x: i32, y: i32, b: bool) {  
  let z;  
  if b { z = &mut x; }  
  else { z = &mut y; }  
  *z += 1;  
  x = 4;  
  assert(*z == 4);  
}
```

$b \mapsto \text{true},$
 $x \mapsto 8,$
 $y \mapsto 3,$
 $z \mapsto \perp$

The LLBC semantics

```
fn f(x: i32, y: i32, b: bool) {  
  let z;  
  if b { z = &mut x; }  
  else { z = &mut y; }  
  *z += 1;  
  x = 4;  
  assert(*z == 4);  
}
```

$b \mapsto \text{true},$

$x \mapsto \text{loan}^m \ell,$

$y \mapsto 3,$

$z \mapsto \text{borrow}^m \ell \delta$

The LLBC semantics

```
fn f(x: i32, y: i32, b: bool) {  
  let z;  
  if b { z = &mut x; }  
  else { z = &mut y; }  
  *z += 1;  
  x = 4;  
  assert(*z == 4);  
}
```

$b \mapsto \text{true},$
 $x \mapsto \text{loan}^m \ell,$
 $y \mapsto 3,$
 $z \mapsto \text{borrow}^m \ell 9$

The LLBC semantics

```
fn f(x: i32, y: i32, b: bool) {  
  let z;  
  if b { z = &mut x; }  
  else { z = &mut y; }  
  *z += 1;  
  x = 4;  
  assert(*z == 4);  
}
```

$b \mapsto \text{true},$
 $x \mapsto \text{loan}^m \ell,$
 $y \mapsto 3,$
 $z \mapsto \text{borrow}^m \ell 9$

The LLBC semantics

```
fn f(x: i32, y: i32, b: bool) {  
  let z;  
  if b { z = &mut x; }  
  else { z = &mut y; }  
  *z += 1;  
  x = 4;  
  assert(*z == 4);  
}
```

$$\begin{aligned} b &\mapsto \text{true}, \\ x &\mapsto 9, \\ y &\mapsto 3, \\ z &\mapsto \perp \end{aligned}$$

Reorganization step: ending the borrow ℓ

The LLBC semantics

```
fn f(x: i32, y: i32, b: bool) {  
  let z;  
  if b { z = &mut x; }  
  else { z = &mut y; }  
  *z += 1;  
  x = 4;  
  assert(*z == 4);  
}
```

$b \mapsto \text{true},$
 $x \mapsto 4,$
 $y \mapsto 3,$
 $z \mapsto \perp$

The LLBC semantics

```
fn f(x: i32, y: i32, b: bool) {  
  let z;  
  if b { z = &mut x; }  
  else { z = &mut y; }  
  *z += 1;  
  x = 4;  
  assert(*z == 4);  
}
```

$$\begin{aligned} b &\mapsto \text{true}, \\ x &\mapsto 4, \\ y &\mapsto 3, \\ z &\mapsto \perp \end{aligned}$$

Error: impossible to read the reference z.

The LLBC model

- LLBC (low-level borrow calculus) is a model of (safe) Rust.

The LLBC model

- LLBC (low-level borrow calculus) is a model of (safe) Rust.
- The borrow rules are enforced dynamically.

The LLBC model

- LLBC (low-level borrow calculus) is a model of (safe) Rust.
- The borrow rules are enforced dynamically.
- Reorganization steps \hookrightarrow : end borrows and transfer back resources.

$$\frac{\Omega \hookrightarrow^* \Omega' \quad \Omega' \vdash s \rightsquigarrow (r, \Omega'')}{\Omega \vdash s \rightsquigarrow \Omega''}$$

with:

- Ω, Ω' and Ω'' : program states.
- s : program to execute.
- r : program result.
- $\Omega \vdash s \rightsquigarrow (r, \Omega')$: in state Ω , the program s executes with final state Ω' .

What are the properties of LLBC?

In 2024, Ho, Fromherz and Protzenko show that:

- LLBC can be equipped with a heap-based semantics (*à la* CompCert).

What are the properties of LLBC?

In 2024, Ho, Fromherz and Protzenko show that:

- LLBC can be equipped with a heap-based semantics (*à la* CompCert).
- The symbolic execution acts as a borrow-checker.

What are the properties of LLBC?

In 2024, Ho, Fromherz and Protzenko show that:

- LLBC can be equipped with a heap-based semantics (*à la* CompCert).
- The symbolic execution acts as a borrow-checker.

Source: *Sound Borrow-Checking for Rust via Symbolic Semantics*, ICFP 2024

What are the properties of LLBC?

In 2024, Ho, Fromherz and Protzenko show that:

- LLBC can be equipped with a heap-based semantics (*à la* CompCert).
- The symbolic execution acts as a borrow-checker.

Source: *Sound Borrow-Checking for Rust via Symbolic Semantics*, ICFP 2024

Goal: formalize this work in the Rocq prover and validate the methodology.

Symbolic execution of LLBC

```
fn f(x: i32, y: i32, b: bool) {  
  let z;  
  if b { z = &mut x; }  
  else { z = &mut y; }  
  *z += 1;  
  x = 4;  
  // assert(*z == 4);  
}
```

$$b \mapsto \sigma_{\text{bool}}, \quad x \mapsto \sigma_{\text{int}}, \quad y \mapsto \sigma_{\text{int}},$$
$$z \mapsto \perp$$

Symbolic execution of LLBC

```
fn f(x: i32, y: i32, b: bool) {
  let z;
  if b { z = &mut x; }
  else { z = &mut y; }
  *z += 1;
  x = 4;
  // assert(*z == 4);
}
```

$$\begin{aligned}
 b &\mapsto \sigma_{\text{bool}}, & x &\mapsto \sigma_{\text{int}}, & y &\mapsto \sigma_{\text{int}}, \\
 z &\mapsto \perp \\
 b &\mapsto \text{true}, & x &\mapsto \text{loan}^m \ell, & y &\mapsto \sigma_{\text{int}}, \\
 z &\mapsto \text{borrow}^m \ell \sigma_{\text{int}}
 \end{aligned}$$

Symbolic execution of LLBC

```
fn f(x: i32, y: i32, b: bool) {
  let z;
  if b { z = &mut x; }
  else { z = &mut y; }
  *z += 1;
  x = 4;
  // assert(*z == 4);
}
```

$$\begin{aligned}
 b &\mapsto \sigma_{\text{bool}}, & x &\mapsto \sigma_{\text{int}}, & y &\mapsto \sigma_{\text{int}}, \\
 z &\mapsto \perp \\
 b &\mapsto \text{true}, & x &\mapsto \text{loan}^m \ell, & y &\mapsto \sigma_{\text{int}}, \\
 z &\mapsto \text{borrow}^m \ell \sigma_{\text{int}} \\
 b &\mapsto \text{false}, & x &\mapsto \sigma_{\text{int}}, & y &\mapsto \text{loan}^m \ell, \\
 z &\mapsto \text{borrow}^m \ell \sigma_{\text{int}}
 \end{aligned}$$

Joining two states

The states to join are:

$$b \mapsto \text{true}, x \mapsto \text{loan}^m \ell, y \mapsto \sigma_{\text{int}}, \quad z \mapsto \text{borrow}^m \ell \sigma_{\text{int}}$$

$$b \mapsto \text{false}, x \mapsto \sigma_{\text{int}}, \quad y \mapsto \text{loan}^m \ell, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}}$$

The join state is:

$$b \mapsto \sigma_{\text{bool}}, x \mapsto \quad, y \mapsto \quad, z \mapsto \quad,$$

Joining two states

The states to join are:

$$\begin{aligned}
 & b \mapsto \text{true}, x \mapsto \text{loan}^m \ell, y \mapsto \sigma_{\text{int}}, \quad z \mapsto \text{borrow}^m \ell \sigma_{\text{int}} \\
 & b \mapsto \text{false}, x \mapsto \sigma_{\text{int}}, \quad y \mapsto \text{loan}^m \ell, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}}
 \end{aligned}$$

The join state is:

$$\begin{aligned}
 & b \mapsto \sigma_{\text{bool}}, x \mapsto \quad, y \mapsto \quad, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}}, \\
 & A\{ \quad, \quad \text{loan}^m \ell \}
 \end{aligned}$$

Joining two states

The states to join are:

$$\begin{aligned}
 & b \mapsto \text{true}, x \mapsto \text{loan}^m \ell, y \mapsto \sigma_{\text{int}}, \quad z \mapsto \text{borrow}^m \ell \sigma_{\text{int}} \\
 & b \mapsto \text{false}, x \mapsto \sigma_{\text{int}}, \quad y \mapsto \text{loan}^m \ell, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}}
 \end{aligned}$$

The join state is:

$$\begin{aligned}
 & b \mapsto \sigma_{\text{bool}}, x \mapsto \text{loan}^m \ell_x, y \mapsto \quad, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}}, \\
 & A\{\text{borrow}^m \ell_x \sigma_{\text{int}}, \quad \text{loan}^m \ell\}
 \end{aligned}$$

Joining two states

The states to join are:

$$\begin{aligned}
 & b \mapsto \text{true}, x \mapsto \text{loan}^m \ell, y \mapsto \sigma_{\text{int}}, \quad z \mapsto \text{borrow}^m \ell \sigma_{\text{int}} \\
 & b \mapsto \text{false}, x \mapsto \sigma_{\text{int}}, \quad y \mapsto \text{loan}^m \ell, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}}
 \end{aligned}$$

The join state is:

$$\begin{aligned}
 & b \mapsto \sigma_{\text{bool}}, x \mapsto \text{loan}^m \ell_x, y \mapsto \text{loan}^m \ell_y, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}}, \\
 & A\{\text{borrow}^m \ell_x \sigma_{\text{int}}, \text{borrow}^m \ell_y \sigma_{\text{int}}, \text{loan}^m \ell\}
 \end{aligned}$$

Symbolic execution of LLBC

```
fn f(x: i32, y: i32, b: bool) {
  let z;
  if b { z = &mut x; }
  else { z = &mut y; }
  *z += 1;
  x = 4;
  // assert(*z == 4);
}
```

$$b \mapsto \sigma_{\text{bool}}, x \mapsto \text{loan}^m \ell_x, y \mapsto \text{loan}^m \ell_y, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}},$$

$$A\{\text{borrow}^m \ell_x \sigma_{\text{int}}, \text{borrow}^m \ell_y \sigma_{\text{int}}, \text{loan}^m \ell\}$$

Symbolic execution of LLBC

```
fn f(x: i32, y: i32, b: bool) {
  let z;
  if b { z = &mut x; }
  else { z = &mut y; }
  *z += 1;
  x = 4;
  // assert(*z == 4);
}
```

$$b \mapsto \sigma_{\text{bool}}, x \mapsto \text{loan}^m \ell_x, y \mapsto \text{loan}^m \ell_y, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}},$$

$$A\{\text{borrow}^m \ell_x \sigma_{\text{int}}, \text{borrow}^m \ell_y \sigma_{\text{int}}, \text{loan}^m \ell\}$$

Reorganization steps:

Symbolic execution of LLBC

```
fn f(x: i32, y: i32, b: bool) {
  let z;
  if b { z = &mut x; }
  else { z = &mut y; }
  *z += 1;
  x = 4;
  // assert(*z == 4);
}
```

$$b \mapsto \sigma_{\text{bool}}, x \mapsto \text{loan}^m \ell_x, y \mapsto \text{loan}^m \ell_y, z \mapsto \perp$$

$$A\{\text{borrow}^m \ell_x \sigma_{\text{int}}, \text{borrow}^m \ell_y \sigma_{\text{int}}\}$$

Reorganization steps:

- ending the borrow ℓ_z .

Symbolic execution of LLBC

```
fn f(x: i32, y: i32, b: bool) {
  let z;
  if b { z = &mut x; }
  else { z = &mut y; }
  *z += 1;
  x = 4;
  // assert(*z == 4);
}
```

$$b \mapsto \sigma_{\text{bool}}, x \mapsto \text{loan}^m \ell_x, y \mapsto \text{loan}^m \ell_y, z \mapsto \perp$$

$$_ \mapsto \text{borrow}^m \ell_x \sigma_{\text{int}}, _ \mapsto \text{borrow}^m \ell_y \sigma_{\text{int}}$$

Reorganization steps:

- ending the borrow ℓ_z .
- ending the region A .

Symbolic execution of LLBC

```
fn f(x: i32, y: i32, b: bool) {
  let z;
  if b { z = &mut x; }
  else { z = &mut y; }
  *z += 1;
  x = 4;
  // assert(*z == 4);
}
```

$$b \mapsto \sigma_{\text{bool}}, x \mapsto \sigma_{\text{int}}, y \mapsto \text{loan}^m \ell_y, z \mapsto \perp$$

$$_ \mapsto \perp, _ \mapsto \text{borrow}^m \ell_y \sigma_{\text{int}}$$

Reorganization steps:

- ending the borrow ℓ_z .
- ending the region A .
- ending the borrow ℓ_x .

Symbolic execution of LLBC

```
fn f(x: i32, y: i32, b: bool) {
  let z;
  if b { z = &mut x; }
  else { z = &mut y; }
  *z += 1;
  x = 4;
  // assert(*z == 4);
}
```

$$b \mapsto \sigma_{\text{bool}}, x \mapsto 4, y \mapsto \text{loan}^m \ell_y, z \mapsto \perp$$

$$_ \mapsto \perp, _ \mapsto \text{borrow}^m \ell_y \sigma_{\text{int}}$$

Reorganization steps:

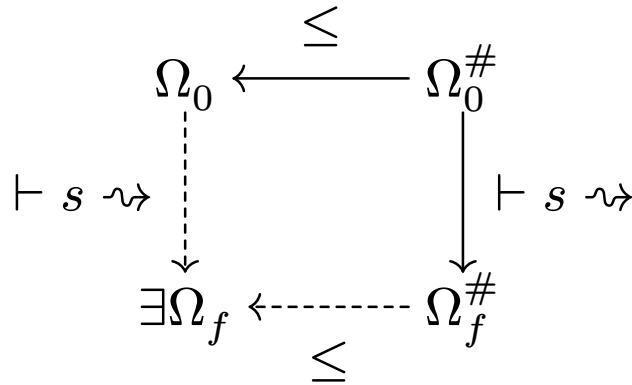
- ending the borrow ℓ_z .
- ending the region A .
- ending the borrow ℓ_x .

Symbolic execution as a borrow-checker

- The program s borrow-checks if $\Omega_0^\# \vdash s \rightsquigarrow (r, \Omega_f^\#)$
($\Omega_0^\#$ symbolic initial state, $\Omega_f^\#$ symbolic final state, r result.)

Symbolic execution as a borrow-checker

- The program s borrow-checks if $\Omega_0^\# \vdash s \rightsquigarrow (r, \Omega_f^\#)$
($\Omega_0^\#$ symbolic initial state, $\Omega_f^\#$ symbolic final state, r result.)
- Forward simulation diagram:



The relation \leq

$b \mapsto \text{true}, x \mapsto \text{loan}^m \ell, y \mapsto 3, z \mapsto \text{borrow}^m \ell 8$

The relation \leq

$$\begin{aligned} & b \mapsto \text{true}, x \mapsto \text{loan}^m \ell, y \mapsto 3, z \mapsto \text{borrow}^m \ell 8 \\ \leq & b \mapsto \sigma_{\text{bool}}, x \mapsto \text{loan}^m \ell, y \mapsto \sigma_{\text{int}}, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}} \end{aligned}$$

- Turning constants into symbolic values.

The relation \leq

$$\begin{aligned}
 & b \mapsto \text{true}, x \mapsto \text{loan}^m \ell, y \mapsto 3, z \mapsto \text{borrow}^m \ell \delta \\
 \leq & b \mapsto \sigma_{\text{bool}}, x \mapsto \text{loan}^m \ell, y \mapsto \sigma_{\text{int}}, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}} \\
 \leq & b \mapsto \sigma_{\text{bool}}, x \mapsto \text{loan}^m \ell_x, y \mapsto \sigma_{\text{int}}, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}} \\
 & A\{\text{borrow}^m \ell_x \sigma_{\text{int}}, \text{loan}^m \ell\}
 \end{aligned}$$

- Turning constants into symbolic values.
- The borrow of x is placed into region A .

The relation \leq

$$\begin{aligned}
& b \mapsto \text{true}, x \mapsto \text{loan}^m \ell, y \mapsto 3, z \mapsto \text{borrow}^m \ell \delta \\
\leq & b \mapsto \sigma_{\text{bool}}, x \mapsto \text{loan}^m \ell, y \mapsto \sigma_{\text{int}}, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}} \\
\leq & b \mapsto \sigma_{\text{bool}}, x \mapsto \text{loan}^m \ell_x, y \mapsto \sigma_{\text{int}}, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}} \\
& A\{\text{borrow}^m \ell_x \sigma_{\text{int}}, \text{loan}^m \ell\} \\
\leq & b \mapsto \sigma_{\text{bool}}, x \mapsto \text{loan}^m \ell_x, y \mapsto \text{loan}^m \ell_y, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}} \\
& A\{\text{borrow}^m \ell_x \sigma_{\text{int}}, \text{loan}^m \ell\}, B\{\text{borrow}^m \ell_y \sigma_{\text{int}}\}
\end{aligned}$$

- Turning constants into symbolic values.
- The borrow of x is placed into region A .
- The variable y is declared as borrowed in a region B .

The relation \leq

$$\begin{aligned}
& b \mapsto \text{true}, x \mapsto \text{loan}^m \ell, y \mapsto 3, z \mapsto \text{borrow}^m \ell 8 \\
\leq & b \mapsto \sigma_{\text{bool}}, x \mapsto \text{loan}^m \ell, y \mapsto \sigma_{\text{int}}, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}} \\
\leq & b \mapsto \sigma_{\text{bool}}, x \mapsto \text{loan}^m \ell_x, y \mapsto \sigma_{\text{int}}, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}} \\
& A\{\text{borrow}^m \ell_x \sigma_{\text{int}}, \text{loan}^m \ell\} \\
\leq & b \mapsto \sigma_{\text{bool}}, x \mapsto \text{loan}^m \ell_x, y \mapsto \text{loan}^m \ell_y, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}} \\
& A\{\text{borrow}^m \ell_x \sigma_{\text{int}}, \text{loan}^m \ell\}, B\{\text{borrow}^m \ell_y \sigma_{\text{int}}\} \\
\leq & b \mapsto \sigma_{\text{bool}}, x \mapsto \text{loan}^m \ell_x, y \mapsto \text{loan}^m \ell_y, z \mapsto \text{borrow}^m \ell \sigma_{\text{int}} \\
& A\{\text{borrow}^m \ell_x \sigma_{\text{int}}, \text{loan}^m \ell, \text{borrow}^m \ell_y \sigma_{\text{int}}\}
\end{aligned}$$

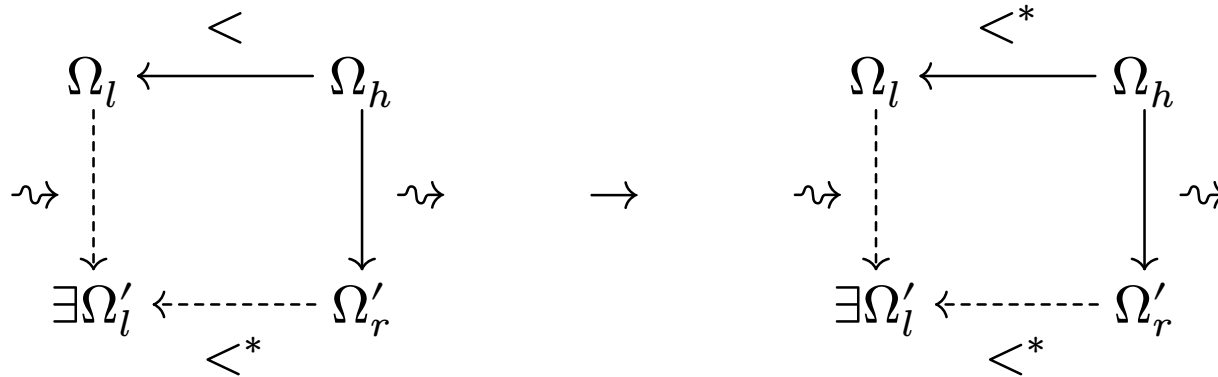
- Turning constants into symbolic values.
- The borrow of x is placed into region A .
- The variable y is declared as borrowed in a region B .
- Merging the regions A and B .

Proving the correctness of symbolic execution

- Abstraction is defined by local transformation rules.

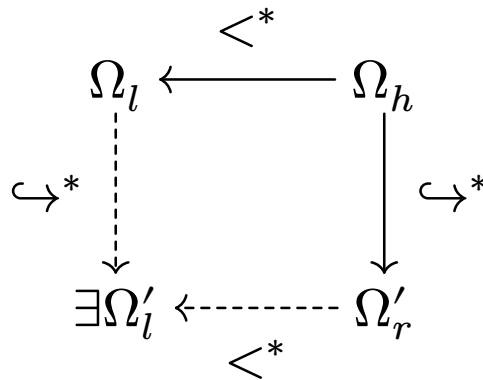
Ex: $\Omega[v] < \Omega[\sigma]$

- The relation \leq is the reflexive-closure of $<$.
- Proof-scheme:



But there's an error!

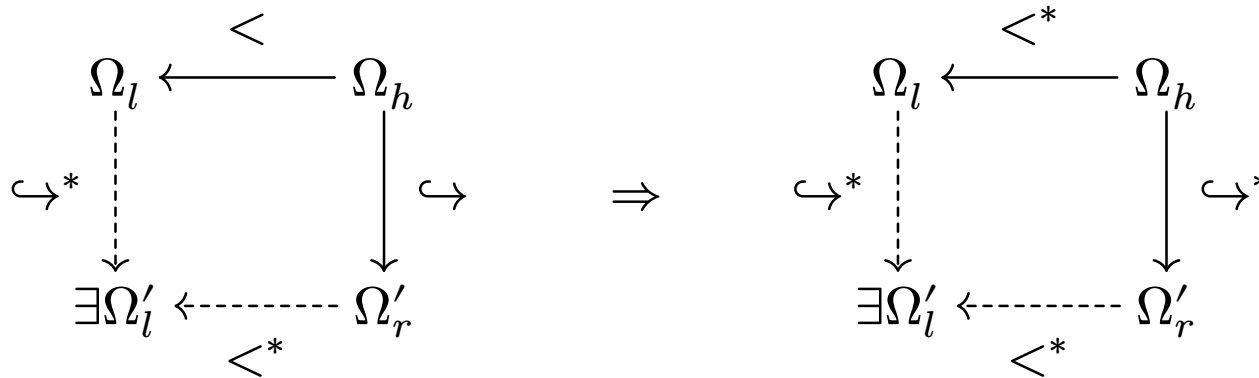
We need to prove that sequences of reorganizations preserve $<^*$.



But there's an error!

We need to prove that sequences of reorganizations preserve $<^*$.

The authors claim that the proof can be done by induction on $<^*$ and \hookrightarrow^* .

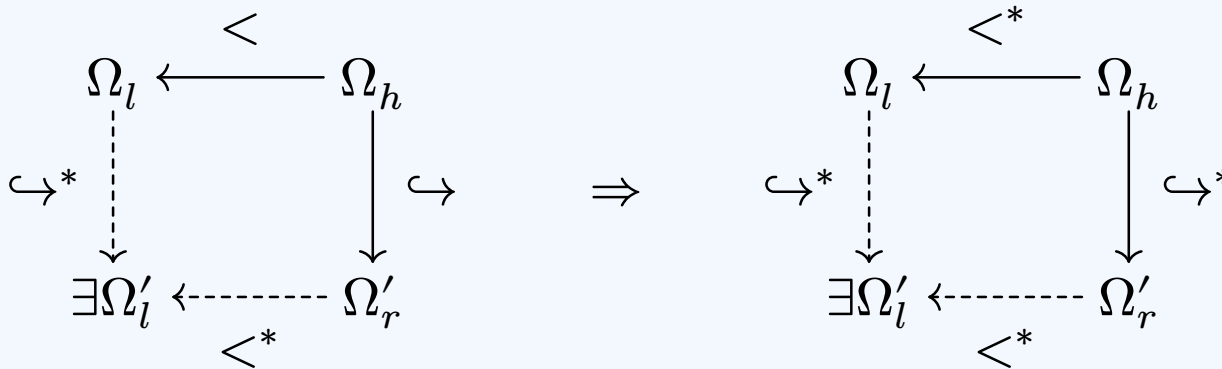


Newman's lemma

If there exists a measure $|\cdot| : \Omega \rightarrow \mathbb{N}$ such that:

- For all $\Omega_l < \Omega_h$, $|\Omega_l| < |\Omega_h|$.
- For all $\Omega \hookrightarrow \Omega'$, $|\Omega'| < |\Omega|$.

Then:



Why Newman's lemma is insufficient

- Reorganizations make the size decrease.

Why Newman's lemma is insufficient

- Reorganizations make the size decrease.
- But there is no decrease of the size along the relation $<$.

Why Newman's lemma is insufficient

- Reorganizations make the size decrease.
- But there is no decrease of the size along the relation $<$.

$$\Omega[\text{Cons}(0, \text{Cons}(\dots, \text{Cons}(n, \text{Nil})))] < \Omega[\sigma_{\text{List}}]$$

Merge rule

Introduction of a symbolic value:

$$\frac{v : T \quad \text{no borrow, loan in } v}{\Omega[v] < \Omega[\sigma_T]}$$

Merge rule

Introduction of a symbolic value:

$$\frac{v : T \quad \text{no borrow, loan in } v}{\Omega[v] <_{|v|} \Omega[\sigma_T]}$$

Merge rule

Introduction of a symbolic value:

$$\frac{v : T \quad \text{no borrow, loan in } v}{\Omega[v] <_{|v|} \Omega[\sigma_T]}$$

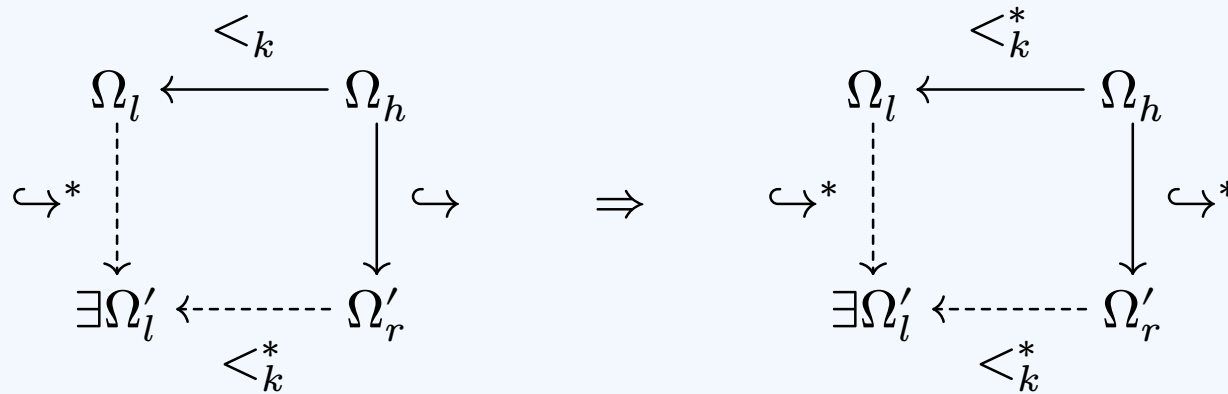
Goal: if $\Omega_l <_k \Omega_h$ then $|\Omega_l| < |\Omega_h| + k$.

Newman's lemma (variant)

If there exists a measure $|\cdot| : \Omega \rightarrow \mathbb{N}$ such that:

- For all $\Omega_l <_k \Omega_h$, $|\Omega_l| < |\Omega_h| + k$.
- For all $\Omega \hookrightarrow \Omega'$, $|\Omega'| < |\Omega|$.

Then:




$$\Omega_0 <^*_k \Omega_n \iff \Omega_0 <_{k_0} \Omega_1 <_{k_1} \dots <_{k_{n-1}} \Omega_n \text{ and } k_0 + \dots + k_{n-1} \leq k$$


The Rocq formalization

- Proof with a minimal fragment (mutable borrow): 
 - No if-then-else, shared borrows, loops, function calls, etc.

The Rocq formalization

- Proof with a minimal fragment (mutable borrow): 
 - No if-then-else, shared borrows, loops, function calls, etc.
- Goal: validating the methodology.

The Rocq formalization

- Proof with a minimal fragment (mutable borrow): 
 - No if-then-else, shared borrows, loops, function calls, etc.
- Goal: validating the methodology.
- Simulation proofs crucially rely on automation:
 - Automatic rewriting.
 - Automatic resolution (+ specialized tactics)

Next objectives and perspectives

- Extend the fragment to if-then-else (joins) and loops.

Next objectives and perspectives

- Extend the fragment to if-then-else (joins) and loops.
- Translate LLBC to CompCert.

Next objectives and perspectives

- Extend the fragment to if-then-else (joins) and loops.
- Translate LLBC to CompCert.
- Prove the functional translation of Aeneas.

Any questions?