

Coupled Borrows

Expanding Core Proof Inference in Prusti

July 2023

Rust, Prusti, and Memory Safety

Rust compiler guarantees **memory safety**.

- Affine Types
- “Aliasing XOR Mutability”
- Ownership, Borrowing

Rust, Prusti, and Memory Safety

Rust compiler guarantees **memory safety**.

- Affine Types
- “Aliasing XOR Mutability”
- Ownership, Borrowing

Rust programmers accept compromises for decidable checking.

Rust, Prusti, and Memory Safety

Rust compiler guarantees **memory safety**.

- Affine Types
- “Aliasing XOR Mutability”
- Ownership, Borrowing

Rust programmers accept compromises for decidable checking.

Prusti

Decidable memory safety analysis automated **core proof**

Rust, Prusti, and Memory Safety

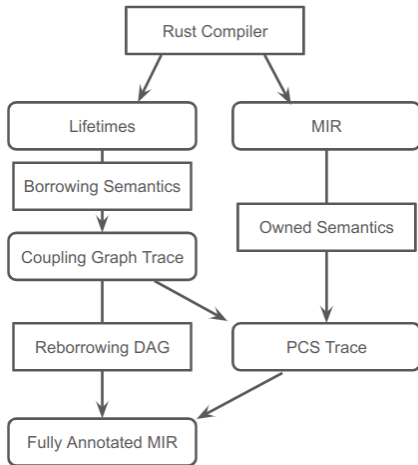
Released version of Prusti has limited support for

- Reborrowing inside loops
- Shared borrows
- Reborrowing at function boundaries
- Borrows inside structs

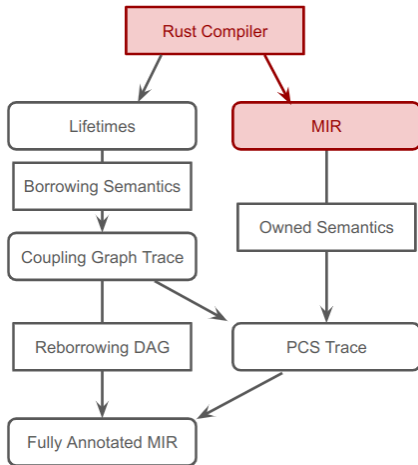
Main Problem

How can we extend Prusti's *model of Rust* to automatically support these features going forward?

Overview



Overview



Background: MIR

MIR: Mid-level Intermediate Representation

Place: Represent accessible memory as (local, [projection]).

- **locals:** arguments, locally declared variables: `_1`, `_2`, ...
- **projection:**
 - dereference (`*_1`)
 - field access (`_.0`)
 - enum variant downcast (`_3 as Some`)
 - more including indexing, slices, type casts ...

1st notion of memory safety



Memory Safety in Rust

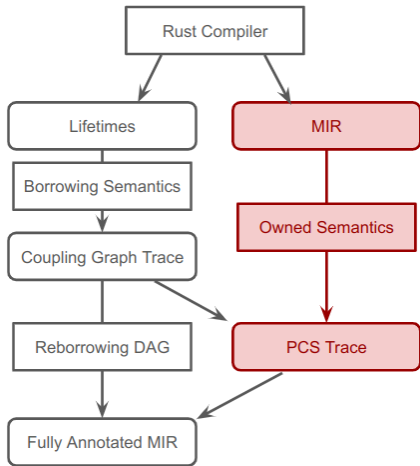
Fixed point analyses over MIR.

Ensures definite accessibility before use.

) No **use after free** errors.

Result of definite accessibility analysis.
Traces accessibility through the MIR.

Overview



Setting: Ownership

```
struct T {}  
fn main() {  
    let t1: T;  
    let mut t2 = T {};  
    t1 = t2;  
}
```

Setting: Ownership

```
struct T {}  
fn main() {  
    let t1: T;  
    let mut t2 = T {};  
    t1 = t2;  
}
```

Prusti's Approach to Explaining Owned Data

- 1 Model program state as *capabilities* to memory.
- 2 Describe *rules* in terms of that state.
- 3 Exploit *approximations* from the type system.

Capabilities

Things we can *do* with places: **Capabilities**.

Program state (all places): **Place Capability Summary (PCS)**.

Capability		Read?	Write?	Nonaliasing?	Example
<i>exclusive</i>	E p	yes	yes	yes	let mut p = ();
<i>immutable</i>	R p	yes	no	yes	let p = ();
<i>shared</i>	S p	yes	no	no	*p where p: &()
<i>write</i>	e p	no	yes	yes	let mut p: ();
<i>allocated</i>	r p	no	no	yes	drop(p); p

Capabilities

Things we can *do* with places: **Capabilities**.

Program state (all places): **Place Capability Summary (PCS)**.

Capability		Read?	Write?	Nonaliasing?	Example
<i>exclusive</i>	E p	yes	yes	yes	let mut p = ();
<i>immutable</i>	R p	yes	no	yes	let p = ();
<i>shared</i>	S p	yes	no	no	*p where p: &()
<i>write</i>	e p	no	yes	yes	let mut p: ();
<i>allocated</i>	r p	no	no	yes	drop(p); p

Missing rows:

- No permissions.
- Write to aliased memory.

Example: Owned Data

```
StorageLive(1);  
StorageLive(2);
```

```
2 = T{};
```

```
1 = move 2;
```

```
0 = ();
```

```
StorageDead(2);
```

```
StorageDead(1);
```

```
return;
```

```
struct T {}  
fn main() {  
    let t1: T;  
    let mut t2 = T {};  
    t1 = t2;  
}
```


Example: Owned Data

```
//           { e 0 }  
StorageLive(1);  
StorageLive(2);  
  
2 = T{};  
  
1 = move 2;  
  
0 = ();  
  
StorageDead(2);  
  
  
StorageDead(1);  
  
return;
```

```
struct T {}  
fn main() {  
    let t1: T;  
    let mut t2 = T {};  
    t1 = t2;  
}
```

Rule:
allocate return place as e

Example: Owned Data

```
//           { e 0 }  
StorageLive(1);  
StorageLive(2);  
//           { e 0, e 1, e 2 }  
2 = T{};  
  
1 = move 2;  
  
0 = ();  
  
StorageDead(2);  
  
  
StorageDead(1);  
  
return;
```

```
struct T {}  
fn main() {  
    let t1: T;  
    let mut t2 = T {};  
    t1 = t2;  
}
```

Rule:

```
{ } StorageLive(1) { e 1 }  
{ } StorageLive(2) { e 2 }
```

Example: Owned Data

```

//           { e 0 }
StorageLive(1);
StorageLive(2);
//           { e 0, e 1, e 2 }
2 = T{};
//           { e 0, e 1, E 2 }
1 = move 2;

0 = ();

StorageDead(2);

StorageDead(1);

return;

```

```

struct T {}
fn main() {
    let t1: T;
    let mut t2 = T {};
    t1 = t2;
}

```

Rule:

```

{ e 2 }
  2 = (const)
{ E 2 }

```

Example: Owned Data

```

//           { e 0 }
StorageLive(1);
StorageLive(2);
//           { e 0, e 1, e 2 }
2 = T{};
//           { e 0, e 1, E 2 }
1 = move 2;
//           { e 0, R 1, e 2 }
0 = ();

StorageDead(2);

StorageDead(1);

return;

```

```

struct T {}
fn main() {
    let t1: T;
    let mut t2 = T {};
    t1 = t2;
}

```

Rule:

```

{E 2, e 1}
  1 = move 2
{e 2, R 1}

```

Example: Owned Data

```

//           { e 0 }
StorageLive(1);
StorageLive(2);
//           { e 0, e 1, e 2 }
2 = T{};
//           { e 0, e 1, E 2 }
1 = move 2;
//           { e 0, R 1, e 2 }
0 = ();
//           { E 0, R 1, e 2 }
StorageDead(2);

StorageDead(1);

return;

```

```

struct T {}
fn main() {
    let t1: T;
    let mut t2 = T {};
    t1 = t2;
}

```

Rule:

```

{ e 0 }
  0 = (const)
{ E 0 }

```

Example: Owned Data

```

//           { e 0 }
StorageLive(1);
StorageLive(2);
//           { e 0, e 1, e 2 }
2 = T{};
//           { e 0, e 1, E 2 }
1 = move 2;
//           { e 0, R 1, e 2 }
0 = ();
//           { E 0, R 1, e 2 }
StorageDead(2);
//           { E 0, R 1 }

StorageDead(1);

return;

```

```

struct T {}
fn main() {
    let t1: T;
    let mut t2 = T {};
    t1 = t2;
}

```

Rule:

```

{ e 2 }
  StorageDead(2)
{ }

```

Example: Owned Data

```

//           { e 0 }
StorageLive(1);
StorageLive(2);
//           { e 0, e 1, e 2 }
2 = T{};
//           { e 0, e 1, E 2 }
1 = move 2;
//           { e 0, R 1, e 2 }
0 = ();
//           { E 0, R 1, e 2 }
StorageDead(2);
//           { E 0, R 1 }
//           drop(1)
//           { E 0, r 1 }
StorageDead(1);

return;

```

```

struct T {}
fn main() {
    let t1: T;
    let mut t2 = T {};
    t1 = t2;
}

```

Rule:

```

{ R 1 }
    drop(1)
{ r 1 }

```

Example: Owned Data

```

//           { e 0 }
StorageLive(1);
StorageLive(2);
//           { e 0, e 1, e 2 }
2 = T{};
//           { e 0, e 1, E 2 }
1 = move 2;
//           { e 0, R 1, e 2 }
0 = ();
//           { E 0, R 1, e 2 }
StorageDead(2);
//           { E 0, R 1 }
//           drop(1)
//           { E 0, r 1 }
StorageDead(1);
//           { E 0 }
return;

```

```

struct T {
fn main() {
    let t1: T;
    let mut t2 = T {};
    t1 = t2;
}

```

Rule:

```

{ r 1 }
  StorageDead(1)
{ }

```


Example: Owned Data

```

//           { e 0 }
StorageLive(1);
StorageLive(2);
//           { e 0, e 1, e 2 }
2 = T{};
//           { e 0, e 1, E 2 }
1 = move 2;
//           { e 0, R 1, e 2 }
0 = ();
//           { E 0, R 1, e 2 }
StorageDead(2);
//           { E 0, R 1 }
//           drop(1)
//           { E 0, r 1 }
StorageDead(1);
//           { E 0 }
return;
//           { }

```

```

struct T {
fn main() {
    let t1: T;
    let mut t2 = T {};
    t1 = t2;
}

```

Rule:

```

{ E 0 }
return
{ }

```

Example: Packing and Unpacking

```
struct T {}  
struct S { f: T, g: T }
```

```
StorageLive(s);  
StorageLive(x);  
s = S {f: T{}, g: T{}};  
// { E s, e x }
```

```
x = move s.g;
```

Example: Packing and Unpacking

```
struct T {}  
struct S { f: T, g: T }  
  
StorageLive(s);  
StorageLive(x);  
s = S {f: T{}, g: T{}};  
// { E s, e x }  
//   unpack(E s)  
// { E s.f, E s.g, e x }  
x = move s.g;
```

Rule:

```
{ E s }  
  unpack(E s)  
{ E s.f, E s.g }
```

Example: Packing and Unpacking

```
struct T {}  
struct S { f: T, g: T }  
  
StorageLive(s);  
StorageLive(x);  
s = S {f: T{}, g: T{}};  
//           { E s, e x }  
//           unpack(E s)  
//           { E s.f, E s.g, e x }  
x = move s.g;  
//           { E s.f, e s.g, R x }
```

Example: Packing and Unpacking

```
struct T {}  
struct S { f: T, g: T }
```

```
StorageLive(s);  
StorageLive(x);  
s = S {f: T{}, g: T{}};  
//           { E s, e x }  
//           unpack(E s)  
//           { E s.f, E s.g, e x }  
x = move s.g;  
//           { E s.f, e s.g, R x }
```

```
StorageLive(y);  
y = s;
```

Example: Packing and Unpacking

```
struct T {}  
struct S { f: T, g: T }
```

```
StorageLive(s);  
StorageLive(x);  
s = S {f: T{}, g: T{}};  
// { E s, e x }  
// unpack(E s)  
// { E s.f, E s.g, e x }  
x = move s.g;  
// { E s.f, e s.g, R x }
```

```
StorageLive(y);  
y = S;  
// Error  
// can't obtain (E s) from  
// { E s.f, e s.g, R x, e y }
```

```
error[E0382]: use of partially moved value: `s`  
--> src/main.rs:7:13  
6 |     let x = s.g;  
  |           --- value partially moved here  
7 |     let y = s;  
  |           ^ value used here after partial move  
= note: partial move occurs because `s.g` has type `T`,  
       which does not implement the `Copy` trait
```

Example: Packing and Unpacking

```
struct T {}  
struct S { f: T, g: T }
```

```
StorageLive(s);  
StorageLive(x);  
s = S {f: T{}, g: T{}};  
//           { E s, e x }  
//           unpack(E s)  
//           { E s.f, E s.g, e x }  
x = move s.g;  
//           { E s.f, e s.g, R x }
```

```
s.g = T{};
```

```
y = s;
```

Example: Packing and Unpacking

```
struct T {}
struct S { f: T, g: T }
```

```
StorageLive(s);
StorageLive(x);
s = S {f: T{}, g: T{}};
//           { E s, e x }
//           unpack(E s)
//           { E s.f, E s.g, e x }
x = move s.g;
//           { E s.f, e s.g, R x }
```

```
s.g = T{};
//           { E s.f, E s.g, R x, e y }
//           pack(E s)
//           { E s, R x, e y }
y = s;
//           { e s, R x, R y }
```

Rule:

```
{ E s.f, E s.g }
  pack(E s)
  { E s }
  { E s }
```


Example: Packing and Unpacking

```
struct T {}
struct S { f: T, g: T }

StorageLive(s);
StorageLive(x);
s = S {f: T{}, g: T{}};
//           { E s, e x }
//           unpack(E s)
//           { E s.f, E s.g, e x }
x = move s.g;
//           { E s.f, e s.g, R x }

s.g = T{};
//           { E s.f, E s.g, R x, e y }
//           pack(E s)
//           { E s, R x, e y }
y = s;
//           { e s, R x, R y }
```

Example: Packing and Unpacking

```
struct T {}  
struct S { f: T, g: T }
```

```
StorageLive(s);  
StorageLive(x);  
s = S {f: T{}, g: T{}};  
//           { E s, e x }  
//           unpack(E s)  
//           { E s.f, E s.g, e x }  
x = move s.g;  
//           { E s.f, e s.g, R x }
```

```
StorageDead(s);
```

Example: Packing and Unpacking

```
struct T {}
struct S { f: T, g: T }

StorageLive(s);
StorageLive(x);
s = S {f: T{}, g: T{}};
//           { E s, e x }
//           unpack(E s)
//           { E s.f, E s.g, e x }
x = move s.g;
//           { E s.f, e s.g, R x }

//           drop(e s.f)
//           { e s.f, e s.g, R x }
//           pack(e s)
//           { e s, R x }
StorageDead(s);
//           { R x }
```

Join Points

```
fn test(b: bool, t: T) {  
    if b {  
        StorageLive(u);  
        u = move t;  
//        drop(u)  
        StorageDead(u);  
//  
//        { R b, r t }  
    } else {  
//  
//        { R b, R t }  
  
    }  
  
}
```

Join Points

```
fn test(b: bool, t: T) {  
    if b {  
        StorageLive(u);  
        u = move t;  
//        drop(u)  
        StorageDead(u);  
//  
//        { R b, r t }  
    } else {  
//  
//        { R b, R t }  
  
    }  
  
}
```

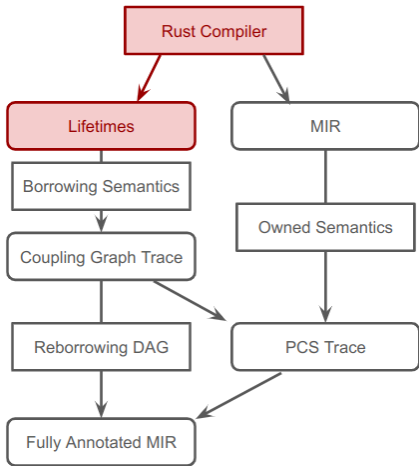
Which capabilities does the program have after the `if` statement?

Join Points

```
fn test(b: bool, t: T) {  
    if b {  
        StorageLive(u);  
        u = move t;  
        // drop(u)  
        StorageDead(u);  
        // { R b, r t }  
    } else {  
        // { R b, R t }  
        // drop(t)  
        // { R b, r t }  
    }  
    // { R b, r t }  
}
```

Which capabilities does the program have after the `if` statement?

Overview



Mutable Borrows

```
let x = &mut z;
```

x will have some kind of control over z's memory for some time.

Mutable Borrows

```
let x = &mut z;
```

x will have **some kind of control** over z's memory for some time.

- E *x, and no capability for z.

Mutable Borrows

```
let x = &mut z;
```

x will have some kind of control over z's memory for **some time**.

- E *x, and no capability for z.
- Eventually, we relinquish E *x and regain E z.

2nd notion of memory safety

```
let (x: & a mut T) = (&mut z: & bw0 mut T);
```

2nd notion of memory safety

```
let (x: & a mut T) = (&mut z: & bw0 mut T);
```

- x needs to be usable for a.

2nd notion of memory safety

```
let (x: & a mut T) = (&mut z: & bw0 mut T);
```

- x needs to be usable for a.
- The borrow can last for bw0 before it must be given back.
 - Called an *invalidation* of bw0.

2nd notion of memory safety

```
let (x: & a mut T) = (&mut z: & bw0 mut T);
```

- x needs to be usable for a.
- The borrow can last for bw0 before it must be given back.
 - Called an *invalidation* of bw0.
- Require $bw0 < a$ (read bw0 *outlives* a)

2nd notion of memory safety

```
let (x: & a mut T) = (&mut z: & bw0 mut T);
```

- x needs to be usable for a.
- The borrow can last for bw0 before it must be given back.
 - Called an *invalidation* of bw0.
- Require $bw0 < a$ (read bw0 *outlives* a)

Memory Safety in Rust

Fixed point analysis over MIR.

Memory is **definitely not borrowed from** when invalidated.

) No dangling pointers.

2nd notion of memory safety

```
let (x: & a mut T) = (&mut z: & bw0 mut T);
```

- x needs to be usable for a.
- The borrow can last for bw0 before it must be given back.
 - Called an *invalidation* of bw0.
- Require $bw0 < a$ (read bw0 *outlives* a)

Memory Safety in Rust

Fixed point analysis over MIR.

Memory is definitely not borrowed from when invalidated.

) No **dangling pointers**.


```
struct T {}  
fn test(b: bool, mut t1: T, mut t2: T) {  
    let mut x = &mut t1;  
    let mut y = &mut t2;  
    if b {  
        let tmp = x;  
        x = &mut (*y);  
        y = &mut (*tmp);  
    }  
    let usage_x = x;  
    let usage_y = y;  
}
```

```
struct T {}  
fn test(b: bool, mut t1: T, mut t2: T) {  
    let mut x = &mut t1;  
    let mut y = &mut t2;  
    if b {  
        let tmp = x;  
        x = &mut (*y);  
        y = &mut (*tmp);  
    }  
    let usage_x = x; ←  
    let usage_y = y;  
}
```

x? Yes
y? Yes
t1? No: borrowed
t2? No: borrowed

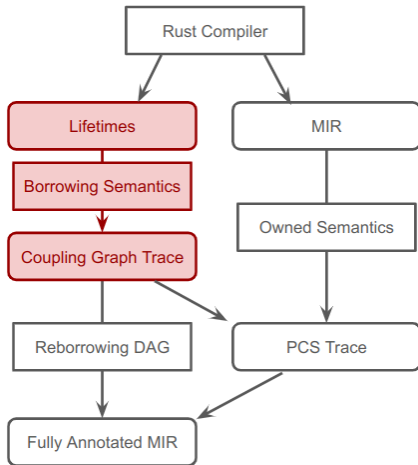
```
struct T {}  
fn test(b: bool, mut t1: T, mut t2: T) {  
    let mut x = &mut t1;  
    let mut y = &mut t2;  
    if b {  
        let tmp = x;  
        x = &mut (*y);  
        y = &mut (*tmp);  
    }  
    let usage_x = x;  
    let usage_y = y; ←  
}
```

x? No: moved out
y? Yes
t1? No: maybe borrowed
t2? No: maybe borrowed

```
struct T {}  
fn test(b: bool, mut t1: T, mut t2: T) {  
    let mut x = &mut t1;  
    let mut y = &mut t2;  
    if b {  
        let tmp = x;  
        x = &mut (*y);  
        y = &mut (*tmp);  
    }  
    let usage_x = x;  
    let usage_y = y; ←  
}
```

x? No: moved out
y? No: moved out
t1? Yes
t2? Yes

Overview



Setting: Mutable Borrows

Prusti's Approach to Explaining Borrowed Data

- 1 Model borrow checker state as *DAG* of capabilities.
- 2 Describe *rules* in terms of that state.
- 3 Exploit *approximations* from the borrow checker.

Capabilities for Borrows

Two complementary views:

- Temporary transfer of ownership
- Pointers with affine referent

Capabilities for Borrows

Two complementary views:

- Temporary transfer of ownership
- Pointers with affine referent

The second leads to a **repacking rule** for borrows:

- `let mut x = &mut t1;`
 $\{E\ x\} \text{unpack}(x) \{e\ x, E\ (*x)\}$

Capabilities for Borrows

Two complementary views:

- Temporary transfer of ownership
- Pointers with affine referent

The second leads to a **repacking rule** for borrows:

- `let mut x = &mut t1;`
 $\{E\ x\}$ `unpack(x)` $\{e\ x,\ E\ (*x)\}$
- `let tmp = x;`
 $\{R\ tmp\}$ `unpack(tmp)` $\{r\ tmp,\ E\ (*tmp)\}$

The Coupling Graph

- Directed Acyclic Hypergraph of Capabilities.

The Coupling Graph

- Directed Acyclic Hypergraph of Capabilities.
- Outlives relationships become **abstract** capability exchanges.
 - Explains how to give back capabilities when borrows expire.

The Coupling Graph

- Directed Acyclic Hypergraph of Capabilities.
- Outlives relationships become abstract capability exchanges.
 - Explains how to give back capabilities when borrows expire.
- Readily **approximated**, same way as the borrow checker.
 - Leaves correspond to **definitely not borrowed** places.
 - Rules for **coupling** edges (abstracting subgraphs at join points).

Coupling Graph Rewrite Rules

```
// { E t1, e x, e z, e w }
```

```
x = &mut t1; // { E x, e z, e w }
```

```
{ E *x } → { E t1 }
```

Coupling Graph Rewrite Rules

```
// { E t1, e x, e z, e w }
```

```
x = &mut t1; // { E x, e z, e w }
```

$\{ E *x \} \rightarrow \{ E t1 \}$

```
z = move x; // { e x, E z, e w }
```

$\{ E *z \} \rightarrow \{ E@1 *x \} \rightarrow \{ E t1 \}$

Coupling Graph Rewrite Rules

```
// { E t1, e x, e z, e w }
```

```
x = &mut t1; // { E x, e z, e w }
```

$\{ E *x \} \rightarrow \{ E t1 \}$

```
z = move x; // { e x, E z, e w }
```

$\{ E *z \} \rightarrow \{ E@1 *x \} \rightarrow \{ E t1 \}$

```
unpack(z); // { e x, e z, E (*z), e w }
```

Coupling Graph Rewrite Rules

```
// { E t1, e x, e z, e w }
```

```
x = &mut t1; // { E x, e z, e w }
```

$$\{ E *x \} \rightarrow \{ E t1 \}$$

```
z = move x; // { e x, E z, e w }
```

$$\{ E *z \} \rightarrow \{ E@1 *x \} \rightarrow \{ E t1 \}$$

```
unpack(z); // { e x, e z, E (*z), e w }
```

```
unpack(*z); // { e x, e z, E (*z).f, E (*z).g, e w }
```

$$\begin{array}{l} \{ E (*z).g \} \\ \{ E (*z).f \} \end{array} \triangleright \{ E *z \} \rightarrow \{ E@1 *x \} \rightarrow \{ E t1 \}$$


```
w = &mut (*z).f; // { e x, e z, E (*z).g, E w }
```

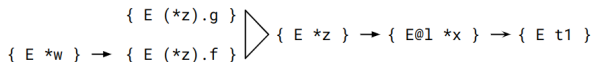
$\{ E (*z).g \}$
 $\{ E *w \} \rightarrow \{ E (*z).f \} \triangleright \{ E *z \} \rightarrow \{ E@1 *x \} \rightarrow \{ E t1 \}$

```
w = &mut (*z).f; // { e x, e z, E (*z).g, E w }
```

$$\{ E *w \} \rightarrow \{ E (*z).f \} \left\{ \begin{array}{l} \{ E (*z).g \} \\ \{ E *z \} \end{array} \right. \rightarrow \{ E@1 *x \} \rightarrow \{ E t1 \}$$

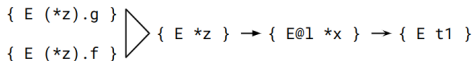
```
unpack(w); // { e x, e z, E (*z).g, e w, E (* w) }
```

```
w = &mut (*z).f; // { e x, e z, E (*z).g, E w }
```



```
unpack(w); // { e x, e z, E (*z).g, e w, E (* w) }
```

```
/* w expires */ // { e x, e z, E (*z).g, E (*z).f, e w }
```



```
w = &mut (*z).f; // { e x, e z, E (*z).g, E w }
```

$$\begin{array}{l} \{ E (*z).g \} \\ \{ E *w \} \rightarrow \{ E (*z).f \} \end{array} \triangleright \{ E *z \} \rightarrow \{ E@1 *x \} \rightarrow \{ E t1 \}$$

```
unpack(w); // { e x, e z, E (*z).g, e w, E (* w) }
```

```
/* w expires */ // { e x, e z, E (*z).g, E (*z).f, e w }
```

$$\begin{array}{l} \{ E (*z).g \} \\ \{ E (*z).f \} \end{array} \triangleright \{ E *z \} \rightarrow \{ E@1 *x \} \rightarrow \{ E t1 \}$$

```
/* z expires */ // { E t1, e x, e z, e w }
```

```
struct T {}  
fn test(b: bool, mut t1: T, mut t2: T) {  
    let mut x = &mut t1;  
    let mut y = &mut t2;  
    if b {  
        let tmp = x;  
        x = &mut (*y);  
        y = &mut (*tmp);  
    }  
    let usage_x = x;  
    let usage_y = y;  
}
```

Coupling Graph “if” branch:

$$\{E (*x: T)\} \longrightarrow \{'\theta\} \longrightarrow \dots \longrightarrow \{bw1\} \longrightarrow \{E (t2: T)\}$$
$$\{E (*y: T)\} \longrightarrow \{'1\} \longrightarrow \dots \longrightarrow \{bw\theta\} \longrightarrow \{E (t1: T)\}$$

Coupling Graph “else” branch:

$$\{E (*x: T)\} \longrightarrow \{'\theta\} \longrightarrow \{bw\theta\} \longrightarrow \{E (t1: T)\}$$
$$\{E (*y: T)\} \longrightarrow \{'1\} \longrightarrow \{bw1\} \longrightarrow \{E (t2: T)\}$$

Coupling Graph “if” branch:

$$\{E (*x: T)\} \longrightarrow \{'\theta\} \longrightarrow \dots \longrightarrow \{bw1\} \longrightarrow \{E (t2: T)\}$$

$$\{E (*y: T)\} \longrightarrow \{'1\} \longrightarrow \dots \longrightarrow \{bw\theta\} \longrightarrow \{E (t1: T)\}$$

Coupling Graph “else” branch:

$$\{E (*x: T)\} \longrightarrow \{'\theta\} \longrightarrow \{bw\theta\} \longrightarrow \{E (t1: T)\}$$

$$\{E (*y: T)\} \longrightarrow \{'1\} \longrightarrow \{bw1\} \longrightarrow \{E (t2: T)\}$$

Coupled:

$$\begin{array}{l} \{E (*x: T)\} \longrightarrow \{'\theta\} \quad \begin{array}{c} \text{blue triangle} \\ \text{red triangle} \end{array} \quad \begin{array}{c} \text{white arrow} \\ \text{white arrow} \end{array} \quad \begin{array}{c} \text{dark blue triangle} \\ \text{dark red triangle} \end{array} \quad \begin{array}{c} \{bw1\} \longrightarrow \{E (t2: T)\} \\ \{bw\theta\} \longrightarrow \{E (t1: T)\} \end{array} \end{array}$$

Coupling Graph “if” branch:

$$\{E (*x: T)\} \longrightarrow \{'\theta\} \longrightarrow \dots \longrightarrow \{bw1\} \longrightarrow \{E (t2: T)\}$$
$$\{E (*y: T)\} \longrightarrow \{'1\} \longrightarrow \dots \longrightarrow \{bw\theta\} \longrightarrow \{E (t1: T)\}$$

Coupling Graph “else” branch:

$$\{E (*x: T)\} \longrightarrow \{'\theta\} \longrightarrow \{bw\theta\} \longrightarrow \{E (t1: T)\}$$
$$\{E (*y: T)\} \longrightarrow \{'1\} \longrightarrow \{bw1\} \longrightarrow \{E (t2: T)\}$$

Coupled:

$$\begin{array}{l} \{E (*x: T)\} \longrightarrow \{'\theta\} \quad \begin{array}{c} \text{blue triangle} \\ \text{white arrow} \end{array} \quad \{bw1\} \longrightarrow \{E (t2: T)\} \\ \{E (*y: T)\} \longrightarrow \{'1\} \quad \begin{array}{c} \text{red triangle} \\ \text{white arrow} \end{array} \quad \{bw\theta\} \longrightarrow \{E (t1: T)\} \end{array}$$





Lifetime '0 expires:

- Consumes $E (*x: T)$





Lifetime $\backslash 0$ expires:

- Consumes $E (*x: T)$



Lifetime $\backslash 1$ expires

- Remainder of graph can expire
- Consumes $E (*y: T)$; $\forall E (t1: T), E (t2: T)g$ regained.

Coupled Borrows as Loop Invariants

```
struct T {}  
fn test(f: fn() -> bool, mut t1: T, mut t2: T) {  
    let mut x = &mut t1;  
    let mut y = &mut t2;  
    while f() {  
        let tmp = x;  
        x = &mut (*y);  
        y = &mut (*tmp);  
    }  
    let usage_x = x;  
    let usage_y = y;  
}
```

Coupled Borrows as Loop Invariants

```

struct T {}
fn test(f: fn() -> bool, mut t1: T, mut t2: T) {
  let mut x = &mut t1;
  let mut y = &mut t2;
  while f() {
    let tmp = x;
    x = &mut (*y);
    y = &mut (*tmp);
  }
  let usage_x = x;
  let usage_y = y;
}

```



Coupled Borrows as Loop Invariants

```

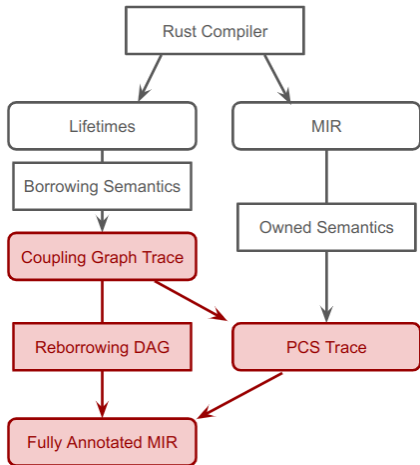
struct T {}
fn test(f: fn() -> bool, mut t1: T, mut t2: T) {
    let mut x = &mut t1;
    let mut y = &mut t2;
    while f() {
        let tmp = x;
        x = &mut (*y);
        y = &mut (*tmp);
    }
    let usage_x = x;
    let usage_y = y;
}

```



stable under the loop body!

Overview



The rest of the story

- Coupling graph edges govern subsets of a *reborrowing DAG*.

The rest of the story

- Coupling graph edges govern subsets of a *reborrowing DAG*.
- Viper: Coupled edges can be packaged as magic wands
 - Apply wands at expiry or repackage of last coupled edge.
- Coupled edges at simple join points do not need magic wands.

Shared Borrows, S subtyping

```
fn test(n: u32) {  
    let b0 = &n;  
    let b1 = &n;  
    /* clone n */  
    let _ = n;  
    let _ = b0;  
    let _ = b1;  
}
```

Shared Borrows, S subtyping

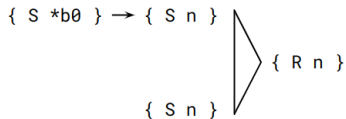
```
fn test(n: u32) {  
    let b0 = &n;  
    let b1 = &n;  
    /* clone n */  
    let _ = n;  
    let _ = b0;  
    let _ = b1;  
}
```

$f R n, e b_0, e b_1 g$

Shared Borrows, S subtyping

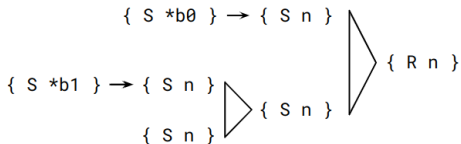
```
fn test(n: u32) {
  let b0 = &n;
  let b1 = &n;
  /* clone n */
  let _ = n;
  let _ = b0;
  let _ = b1;
}
```

- $f R b0 g$ `unpack(b0)` $f r b0, S (*b0) g$
- $f R n, e b0 g$ `b0 = &n;` $f R b0, S n g$

 $f S n, R b0, e b1 g$


Shared Borrows, S subtyping

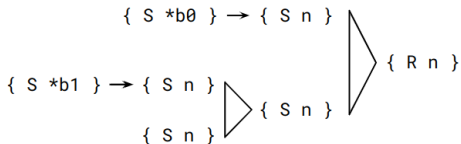
```
fn test(n: u32) {
  let b0 = &n;
  let b1 = &n;
  /* clone n */
  let _ = n;
  let _ = b0;
  let _ = b1;
}
```

 $f \text{ S } n, \text{ R } b0, \text{ R } b1 \text{ } g$


- $f \text{ R } b0 \text{ } g \text{ unpack}(b0) \text{ } f \text{ r } b0, \text{ S } (*b0) \text{ } g$
- $f \text{ R } n, \text{ e } b0 \text{ } g \text{ } b0 = \&n; \text{ } f \text{ R } b0, \text{ S } n \text{ } g$
- $f \text{ S } n, \text{ e } b1 \text{ } g \text{ } b1 = \&n; \text{ } f \text{ S } b0, \text{ S } n \text{ } g$

Shared Borrows, S subtyping

```
fn test(n: u32) {
  let b0 = &n;
  let b1 = &n;
  /* clone n */
  let _ = n;
  let _ = b0;
  let _ = b1;
}
```

 $f \text{ S } n, \text{ R } b0, \text{ R } b1 \text{ } g$


- $f \text{ R } b0 \text{ } g \text{ unpack}(b0) \text{ } f \text{ r } b0, \text{ S } (*b0) \text{ } g$
- $f \text{ R } n, \text{ e } b0 \text{ } g \text{ } b0 = \&n; \text{ } f \text{ R } b0, \text{ S } n \text{ } g$
- $f \text{ S } n, \text{ e } b1 \text{ } g \text{ } b1 = \&n; \text{ } f \text{ S } b0, \text{ S } n \text{ } g$
- $f \text{ R } n, \text{ - } g \text{ - } = \text{clone } n; \text{ } f \text{ R } n, \text{ - } g$
- $f \text{ S } n, \text{ - } g \text{ - } = \text{clone } n; \text{ } f \text{ S } n, \text{ - } g$

Function Calls, Expiry Tools

```
fn test< a, b, c, d>(x: & a mut T, y: & b mut T)
    -> (& c mut T, & d mut T)
where c <: a, c <: b, d <: b { /* ... */ }

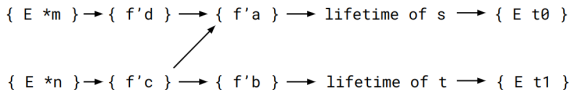
let s = &mut t0;
let t = &mut t1;
let (m, n) = test(s, t);
```

¹*Extended Support for Borrowing and Lifetimes in Prusti*

Function Calls, Expiry Tools

```
fn test< a, b, c, d>(x: & a mut T, y: & b mut T)
  -> (& c mut T, & d mut T)
where c <: a, c <: b, d <: b { /* ... */ }
```

```
let s = &mut t0;
let t = &mut t1;
let (m, n) = test(s, t)
```



- Caller: Lorenz Gorse showed how to encode to Viper¹.

¹*Extended Support for Borrowing and Lifetimes in Prusti*

Function Calls, Expiry Tools

```
fn test< a, b, c, d>(x: & a mut T, y: & b mut T)
    -> (& c mut T, & d mut T)
where c <: a, c <: b, d <: b { /* ... */ }

let s = &mut t0;
let t = &mut t1;
let (m, n) = test(s, t);
```

- Caller: Lorenz Gorse showed how to encode to Viper¹.
- Callee: Polonius ensures we couple the above edges.

¹*Extended Support for Borrowing and Lifetimes in Prusti*

Function Calls, Expiry Tools

```
fn test<a, b, c, d>(x: &a mut T, y: &b mut T)
    -> (&c mut T, &d mut T)
where c <: a, c <: b, d <: b {      /* ... */    }

let s = &mut t0;
let t = &mut t1;
let (m, n) = test(s, t);
```

- Caller: Lorenz Gorse showed how to encode to Viper
- Callee: Polonius ensures we couple the above edges.
- Expiry tools reify coupling graph expiry rules.

¹Extended Support for Borrowing and Lifetimes in Prusti

Borrows in Structs

```
struct BorrowsLL< a, S> {  
    data: & a mut T,  
    next: Option<Box<BorrowsLL< a, S>>>  
}  
let x: BorrowsLL< a, T> = /* ... */ ;
```

Borrows in Structs

```
struct BorrowsLL< a, S> {  
    data: & a mut T,  
    next: Option<Box<BorrowsLL< a, S>>>  
}  
let x: BorrowsLL< a, T> = /* ... */ ;
```

- Lazily add unpacking edges only on dereferences.

Borrows in Structs

```
struct BorrowsLL< a, S> {  
    data: & a mut T,  
    next: Option<Box<BorrowsLL< a, S>>>  
}  
let x: BorrowsLL< a, T> = /* ... */ ;
```

- Lazily add unpacking edges only on dereferences.

Borrows in Structs

```
struct BorrowsLL< a, S> {  
    data: & a mut T,  
    next: Option<Box<BorrowsLL< a, S>>>  
}  
let x: BorrowsLL< a, T> = /* ... */ ;
```

- Lazily add unpacking edges only on dereferences.
- $f \in E \times g \text{ unpack}(x) \ f \in E \times x.data, E \times x.next \ g$

Borrows in Structs

```
struct BorrowsLL< a, S> {  
    data: & a mut T,  
    next: Option<Box<BorrowsLL< a, S>>>  
}  
let x: BorrowsLL< a, T> = /* ... */ ;
```

- Lazily add unpacking edges only on dereferences.
- $f \in E \ x \ g \text{ unpack}(x) \ f \in E \ x.\text{data}, E \ x.\text{next} \ g$
- $f \in E \ x.\text{data} \ g \text{ unpack}(x.\text{data}) \ f \in E \ x.\text{data}, E \ *(x.\text{data}) \ g$
- Coupling and PCS Joins ensure graph and PCS are always finite.

Thank you for your attention!

Status/next steps:

- Implementation in Prusti
- Completeness proof
- Explore connections to other tools

Documentation and examples: Will be ready soon!