Place Capability Graphs

A General-Purpose Model of Rust's Ownership and Borrowing System

Alex Summers

The University of British Columbia

Based on joint work with Zachary Grannan, Aurel Bílý, Jonáš Fiala, Jasper Geer, Markus de Medeiros, Peter Müller

What's so special about Rust's Type System?

- Ownership (plus borrowing) as a fundamental notion
  - Flexible ownership transfer ("move assignments")
- Clear mutability / immutability distinctions (immutable by default)
- Sophisticated *borrowed references* features
  - Temporarily allow access through aliased references
  - Borrows can cross function boundaries and be stored as first-class values
- "Aliasing XOR Mutability" Principle (mutable access is unique)
  - But this is subtle: uniqueness in terms of *currently-usable aliases*
  - Mutations via a borrowed reference will still become visible to the lender
  - This happens as and when borrows end (the lender becomes usable again)

Quick Rust intro on examples

Rust is attractive for analysis / verification

- Controlled aliasing (but aliasing is still significant)
- Many guarantees “for free” (memory safety, non-aliasing properties)
  - In fact, *the programmer still works hard* to demonstrate these
- But, type and borrow-checking are complex notions
  - *No accepted/complete formal definitions* (for any of the real language)
  - Sometimes unclear in practice why/how the compiler accepts a program
- Most complex aspect is the handling of “*borrow extents*”
  - By this, I mean the duration of certain (sets of) borrowed references
  - Intentionally distinct terminology from Rust (lifetimes, regions, origins, …)
  - Constraints generated on borrow extents; *borrow-checked* for satisfiability

Our new model: Place Capability Graphs

- Idea: we can explain (much of) Rust's design in terms of *capabilities*
  - In the sense of various memory-management type systems (e.g. Pony)
  - We call these *place* capabilities, as they connect to Rust *places* (locations)
- Place Capability Graphs are directed hypergraphs per program point
  - Various nodes represent capabilities as governed by the type-system
  - Several kinds of (hyper)edges: most are 1-many/many-1, especially in this talk
- Edges represent undoable reassignments of capabilities (e.g. borrows)
  - Nodes with children represent capabilities *not currently usable* (mostly)

Modelling Ownership

```
struct Pair { fst: String, snd: String }

fn replace_fst(mut p: Pair, s:String) -> Pair
{
1   let tmp = p.fst; // for illustration
2   p.fst  = s;
3   return p;
}
```

0;

Modelling Ownership

```
struct Pair { fst: String, snd: String }

fn replace_fst(mut p: Pair, s:String) -> Pair
{
1   let tmp = p.fst; // for illustration
2   p.fst  = s;
3   return p;
}
```

0;

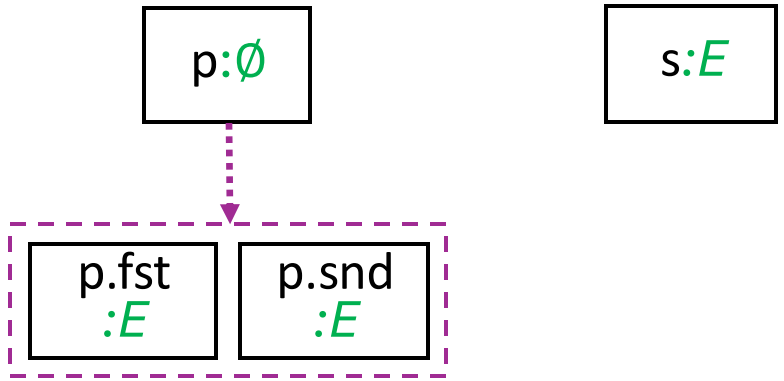
Modelling Ownership

```
struct Pair { fst: String, snd: String }

fn replace_fst(mut p: Pair, s:String) -> Pair
{
1   let tmp = p.fst; // for illustration
2   p.fst  = s;
3   return p;
}
```

0; unpack p;
1;

Modelling Ownership

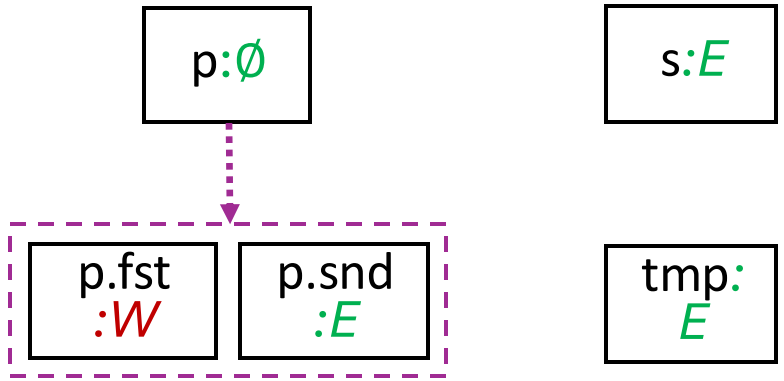
```
struct Pair { fst: String, snd: String }

fn replace_fst(mut p: Pair, s:String) -> Pair
{
1   let tmp = p.fst; // for illustration
2   p.fst  = s;
3   return p;
}
```



0; unpack p;
1;

Modelling Ownership

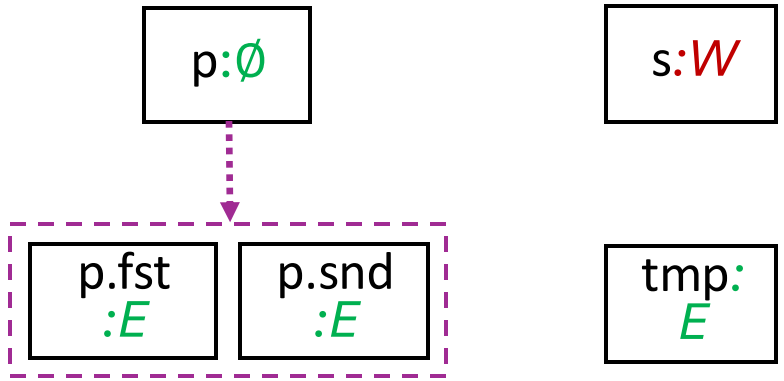
```
struct Pair { fst: String, snd: String }

fn replace_fst(mut p: Pair, s:String) -> Pair
{
1   let tmp = p.fst; // for illustration
2   p.fst  = s;
3   return p;
}
```



0; unpack p;
1;

Modelling Ownership

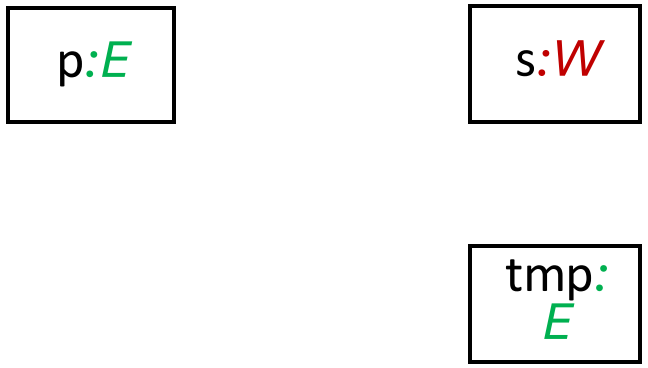
```
struct Pair { fst: String, snd: String }

fn replace_fst(mut p: Pair, s:String) -> Pair
{
1   let tmp = p.fst; // for illustration
2   p.fst  = s;
3   return p;
}
```



0; unpack p;
1;
2;

Modelling Ownership

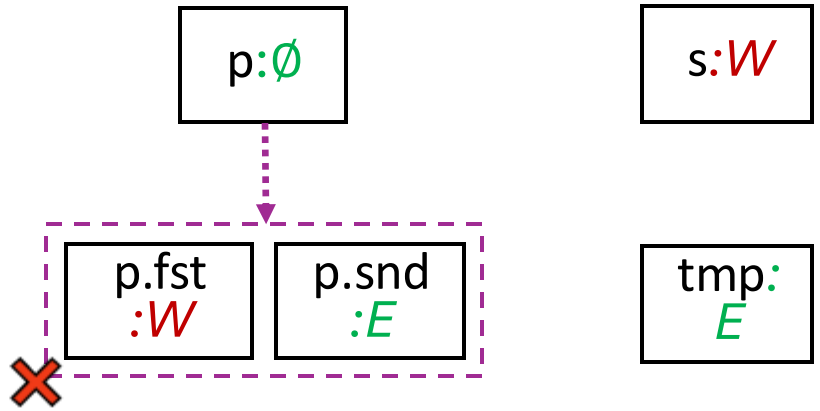
```
struct Pair { fst: String, snd: String }

fn replace_fst(mut p: Pair, s:String) -> Pair
{
1  let tmp = p.fst; // for illustration
2  p.fst  = s;
3  return p;
}
```




```
0; unpack p;
1;
2; pack p;
3;
```

Modelling Ownership

```
struct Pair { fst: String, snd: String }

fn replace_fst(mut p: Pair, s:String) -> Pair
{
1   let tmp = p.fst; // for illustration
2   // p.fst  = s;
3   return p;
}
```




```
- use of partially moved value: 'pair'
```

Modelling Ownership

```
struct Pair { fst: String, snd: String }

fn replace_fst(mut p: Pair, s:String) -> Pair
{
1   let tmp = p.fst;
2   p.fst  = s;
3   return p;
}
```

p:*E*

s:*W*

tmp:
E

- Graph state reflects ownership
- Can identify type incorrect steps
- Annotations for type manipulations
- Generalises easily to control flow (at least, for only ownership…)
- Simple “join”, keeping weakest capabilities in either branch
- Rust’s type system defines usable places ~~without path sensitivity~~

```
0; unpack p;
1;
2; pack p;
3;
```

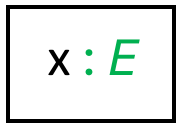
Modelling Borrowing

```
fn main() {
1   let mut x = 1;
2   let y = &mut x;
3   let z = &mut *y;
4   *z = 5;
5   *y = *y + 1;
6   println!("{}",x); // prints 6
}
```

Modelling Borrowing

```
fn main() {
1   let mut x = 1;    // x : i32
2   let y = &mut x;   // y : &'a mut i32
3   let z = &mut *y;  // z : &'b mut i32
4   *z = 5;
5   *y = *y + 1;
6   println!("{}",x); // prints 6
}
```

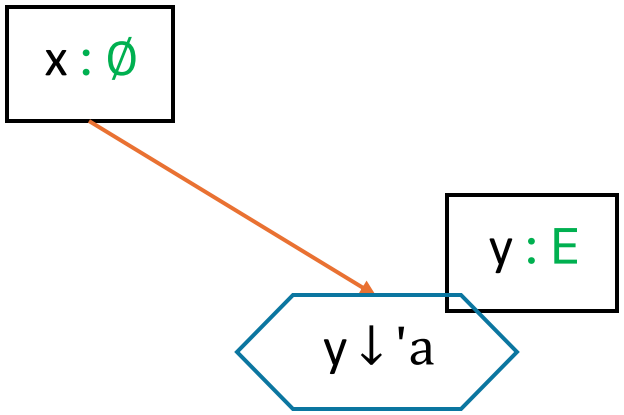
Modelling Borrowing



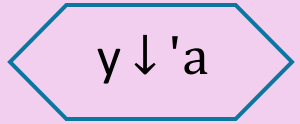

```
fn main() {
1  let mut x = 1;    // x : i32
2  let y = &mut x;   // y : &'a mut i32
3  let z = &mut *y;  // z : &'b mut i32
4  *z = 5;
5  *y = *y + 1;
6  println!("{}",x); // prints 6
}
```

Modelling Borrowing

```
fn main() {
1  let mut x = 1;    // x : i32
2  let y = &mut x;   // y : &'a mut i32
3  let z = &mut *y;  // z : &'b mut i32
4  *z = 5;
5  *y = *y + 1;
6  println!("{}",x); // prints 6
}
```



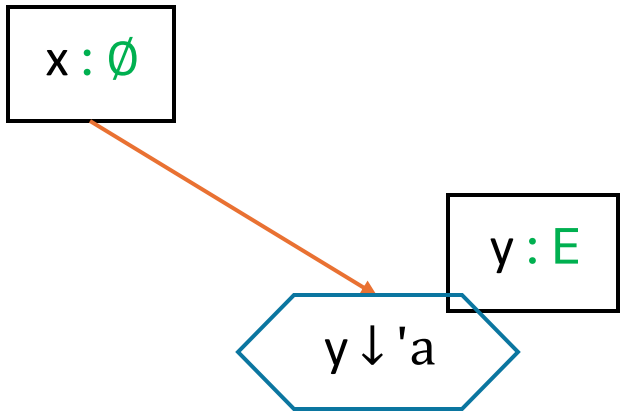
Lifetime Projection Nodes



- Represent a set of borrows:
  - Stored in the place (1st argument, here y)
  - In references with 2nd argument 'a as lifetime
    - i.e. in its declared type (& 'a mut i32)
  - At the current program point (implicit: this graph)
- Always occur simultaneously with their place's node
  - Think of them identifying parts of its capabilities
- Always occur when the *borrow checker* thinks the corresponding borrow extent is still live

Modelling Borrowing

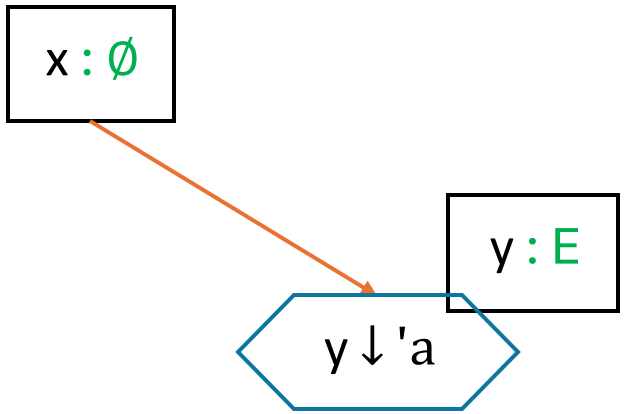
```
fn main() {
1  let mut x = 1;    // x : i32
2  let y = &mut x;   // y : &'a mut i32
3  let z = &mut *y;  // z : &'b mut i32
4  *z = 5;
5  *y = *y + 1;
6  println!("{}",x); // prints 6
}
```




```
0;
1;
2;
```

Modelling Borrowing

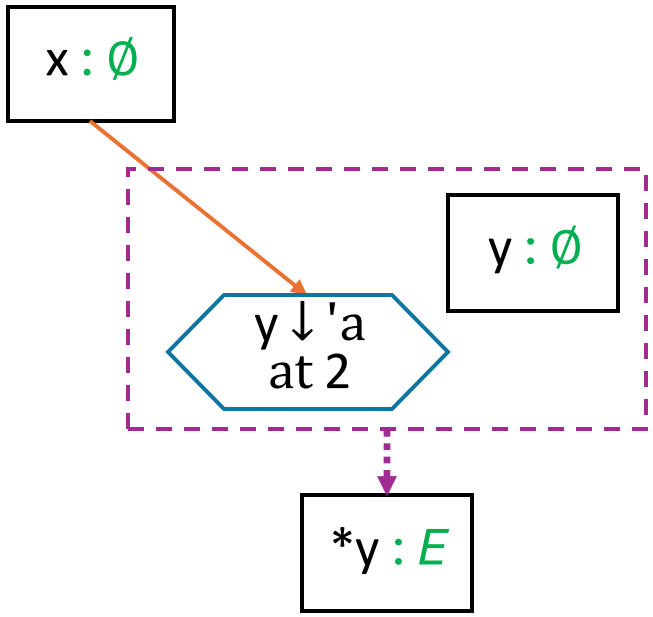
```
fn main() {
1  let mut x = 1;    // x : i32
2  let y = &mut x;   // y : &'a mut i32
3  let z = &mut *y;  // z : &'b mut i32
4  *z = 5;
5  *y = *y + 1;
6  println!("{}",x); // prints 6
}
```




```
0;
1;
2;
```

Modelling Borrowing

```
fn main() {
1  let mut x = 1;    // x : i32
2  let y = &mut x;   // y : &'a mut i32
3  let z = &mut *y;  // z : &'b mut i32
4  *z = 5;
5  *y = *y + 1;
6  println!("{}",x); // prints 6
}
```



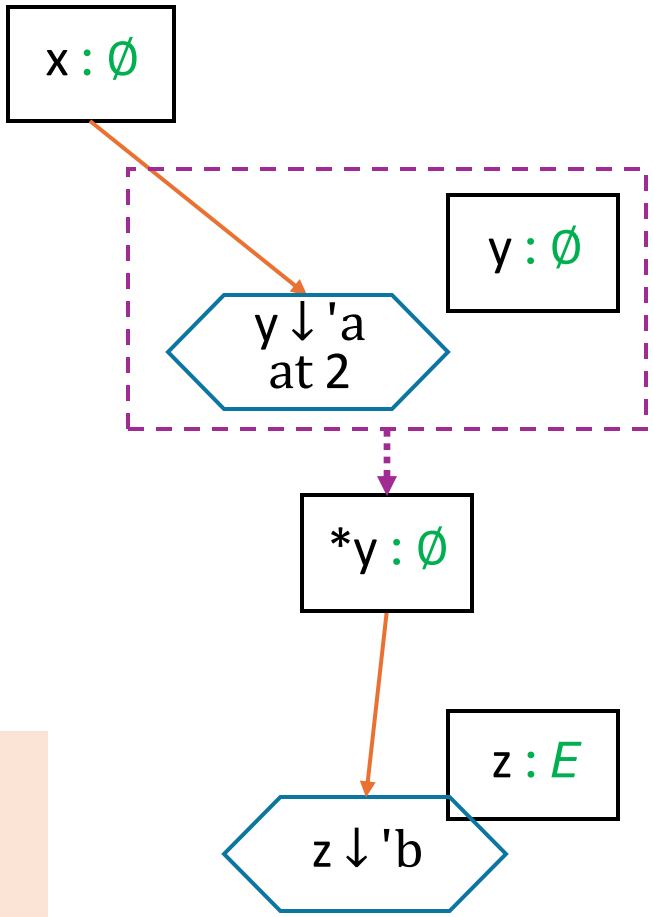
0;
1;
2; unpack y;

Labelled Nodes (at 2)

- Represent nodes as they *were* at an earlier program point (here: line 2)
- While y is unpacked, we don't have its lifetime projection (it's in pieces)
- Labelled node retains connections to where borrow came from, borrow extent

Modelling Borrowing

```
fn main() {
1  let mut x = 1;    // x : i32
2  let y = &mut x;   // y : &'a mut i32
3  let z = &mut *y;  // z : &'b mut i32
4  *z = 5;           // z's borrow ends
5  *y = *y + 1;      // y's borrow ends
6  println!("{}",x); // prints 6
}
```

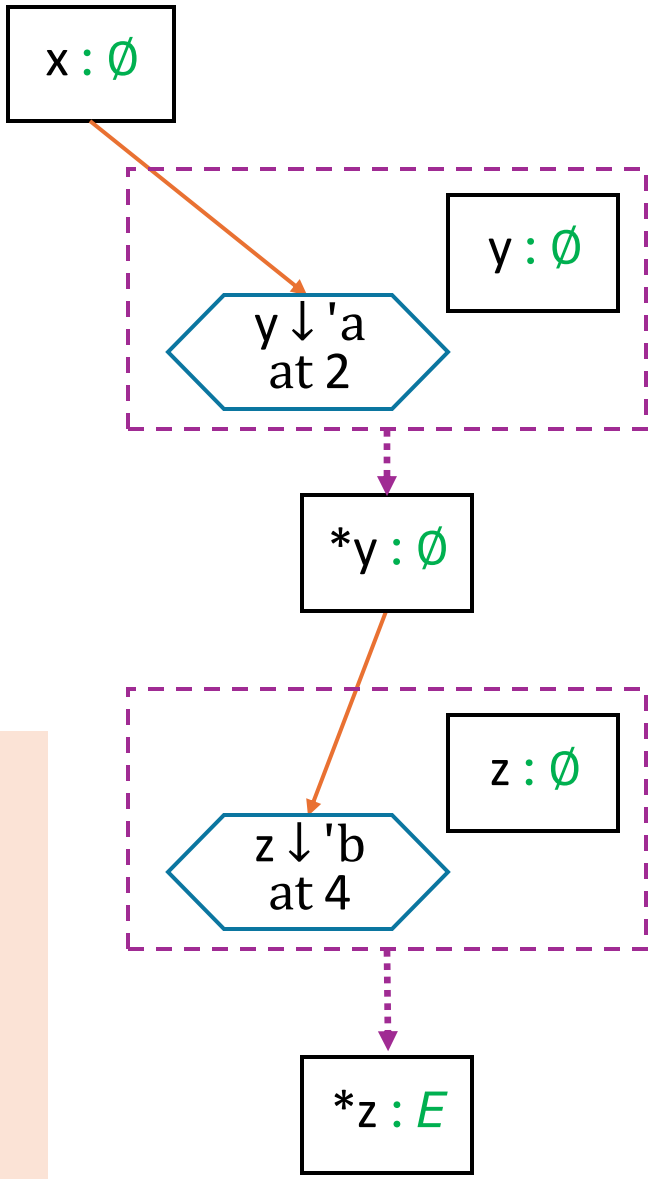



```
0;
1;
2; unpack y;
3;
```

Modelling Borrowing

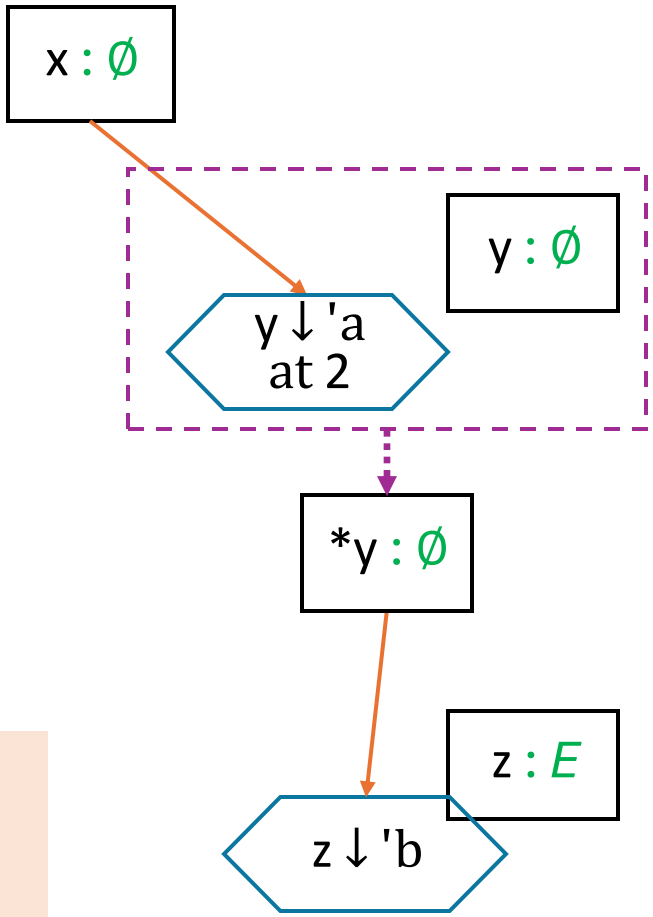
```
fn main() {
1  let mut x = 1;    // x : i32
2  let y = &mut x;   // y : &'a mut i32
3  let z = &mut *y;  // z : &'b mut i32
4  *z = 5;           // z's borrow ends
5  *y = *y + 1;      // y's borrow ends
6  println!("{}",x); // prints 6
}
```

```
0;
1;
2; unpack y;
3; unpack z;
4;
```



Modelling Borrowing

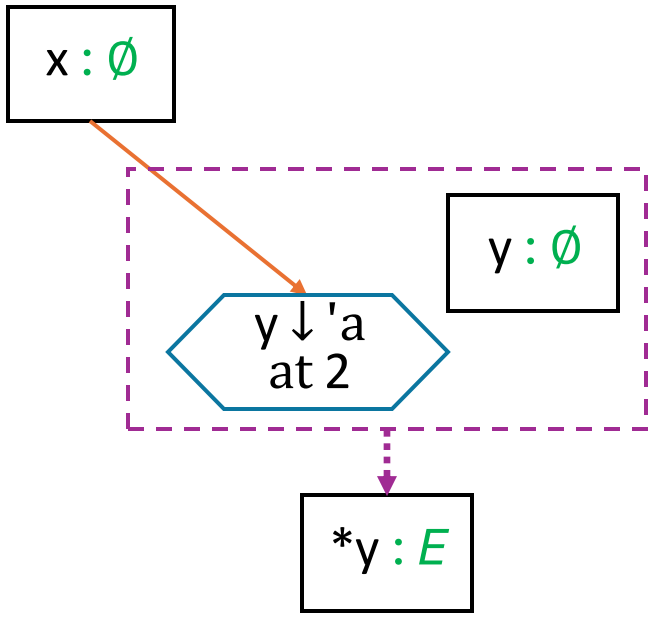
```
fn main() {
1  let mut x = 1;    // x : i32
2  let y = &mut x;   // y : &'a mut i32
3  let z = &mut *y;  // z : &'b mut i32
4  *z = 5;           // z's borrow ends
5  *y = *y + 1;      // y's borrow ends
6  println!("{}",x); // prints 6
}
```




```
0;
1;
2; unpack y;
3; unpack z;
4; pack z;
```

Modelling Borrowing

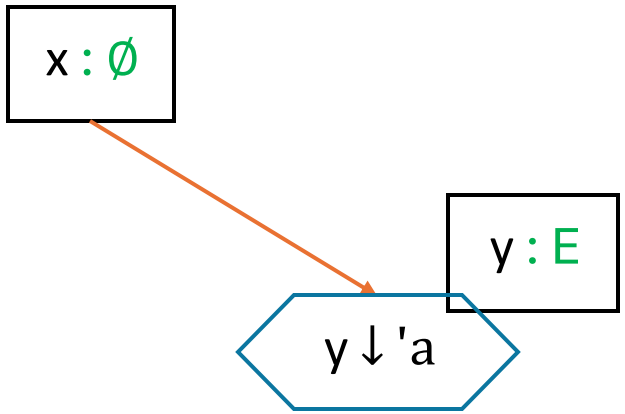
```
fn main() {
1  let mut x = 1;    // x : i32
2  let y = &mut x;   // y : &'a mut i32
3  let z = &mut *y;  // z : &'b mut i32
4  *z = 5;           // z's borrow ends
5  *y = *y + 1;      // y's borrow ends
6  println!("{}",x); // prints 6
}
```




```
0;
1;
2; unpack y;
3; unpack z;
4; pack z; expire z ↓ 'b;
```

Modelling Borrowing

```
fn main() {
1  let mut x = 1;    // x : i32
2  let y = &mut x;   // y : &'a mut i32
3  let z = &mut *y;  // z : &'b mut i32
4  *z = 5;           // z's borrow ends
5  *y = *y + 1;      // y's borrow ends
6  println!("{}",x); // prints 6
}
```



0;
1;
2; unpack y;
3; unpack z;
4; pack z; expire z ↓ 'b;
5; pack y;

Modelling Borrowing

x : *E*

```
fn main() {
1  let mut x = 1;    // x : i32
2  let y = &mut x;   // y : &'a mut i32
3  let z = &mut *y;  // z : &'b mut i32
4  *z = 5;           // z's borrow ends
5  *y = *y + 1;      // y's borrow ends
6  println!("{}",x); // prints 6
}
```

0;
1;
2; unpack y;
3; unpack z;
4; pack z; expire z ↓ 'b;
5; pack y; expire y ↓ 'a;

Modelling Borrowing

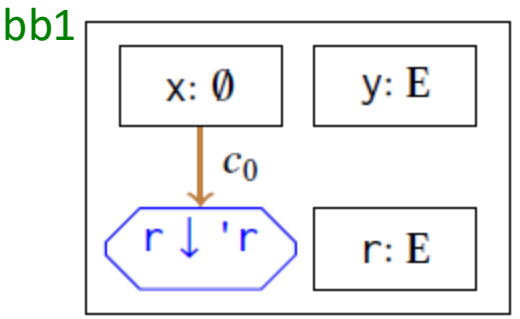
x : *E*

```
fn main() {
1  let mut x = 1;    // x : i32
2  let y = &mut x;   // y : &'a mut i32
3  let z = &mut *y;  // z : &'b mut i32
4  *z = 5;           // z's borrow ends
5  *y = *y + 1;      // y's borrow ends
6  println!("{}",x); // prints 6
}
```

0;
1;
2; unpack y;
3; unpack z;
4; pack z; expire z ↓ 'b;
5; pack y; expire y ↓ 'a;
6;

Path Sensitivity

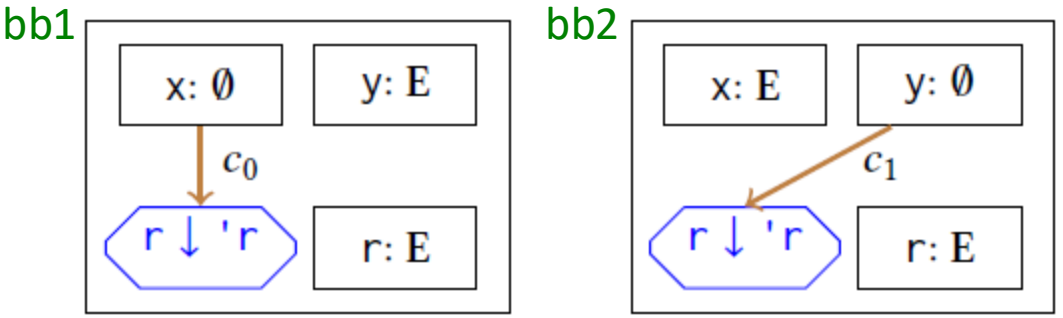
```
fn f(mut x: i32, mut y: i32, mut z: i32) {
  let r = if x > y {       // branch: c0/c1
    &mut x                 // bb1 (c0)
  } else {
    &mut y                 // bb2 (c1)
  };                       // bb3
  let s = if z > 5 {       // branch: c2/c3
    &mut *r                // bb4 (c2)
  } else {
    &mut z                 // bb5 (c3)
  };
  *s = 5; // bb6
}
```



- Borrows can start/end differently in each branch
- At each branch point, generate branch IDs
  - One for each branch choice (e.g. c_0/c_1)
- Edges created under branch are labelled with IDs

Path Sensitivity

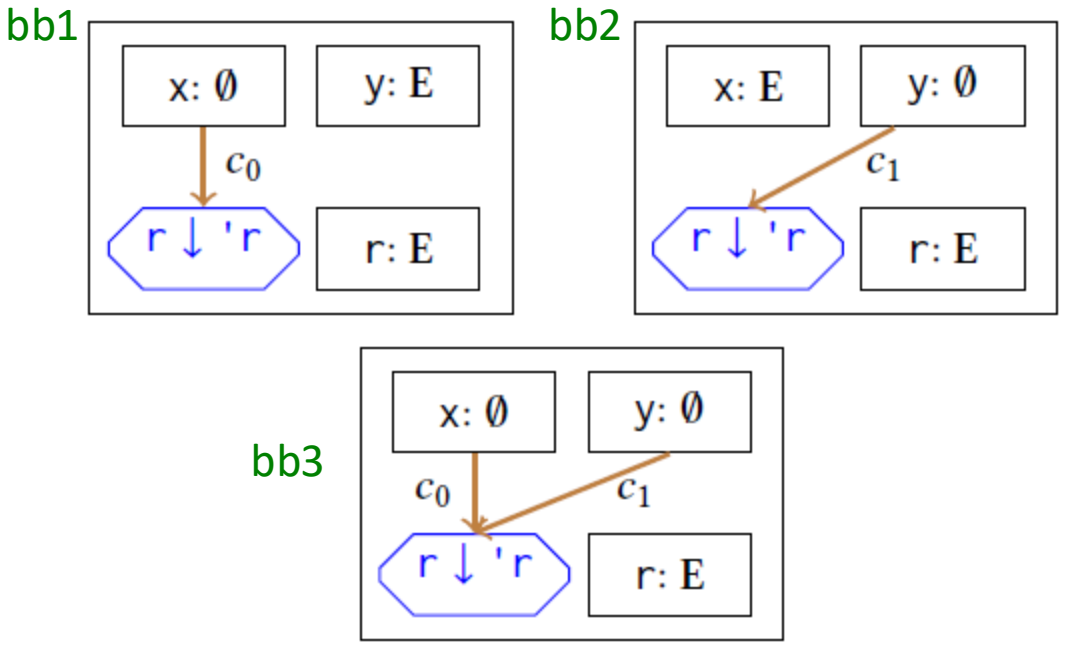
```
fn f(mut x: i32, mut y: i32, mut z: i32) {
  let r = if x > y {      // branch: c0/c1
    &mut x                // bb1 (c0)
  } else {
    &mut y                // bb2 (c1)
  };                      // bb3
  let s = if z > 5 {      // branch: c2/c3
    &mut *r               // bb4 (c2)
  } else {
    &mut z                // bb5 (c3)
  };
  *s = 5; // bb6
}
```



- Borrows can start/end differently in each branch
- At each branch point, generate branch IDs
  - One for each branch choice (e.g. c_0/c_1)
- Edges created under branch are labelled with IDs

Path Sensitivity

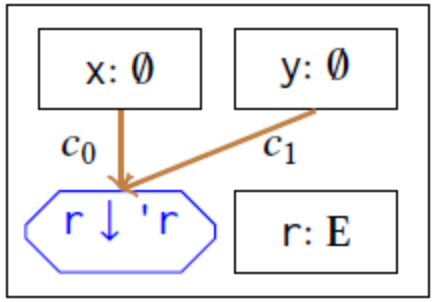
```
fn f(mut x: i32, mut y: i32, mut z: i32) {
  let r = if x > y {      // branch: c0/c1
    &mut x                // bb1 (c0)
  } else {
    &mut y                // bb2 (c1)
  };                      // bb3

  let s = if z > 5 {      // branch: c2/c3
    &mut *r               // bb4 (c2)
  } else {
    &mut z                // bb5 (c3)
  };
  *s = 5; // bb6
}
```



- Borrows can start/end differently in each branch
- At each branch point, generate branch IDs
  - One for each branch choice (e.g. c_0/c_1)
- Edges created under branch are labelled with IDs
- This way, we can compactly represent graphs with partially-shared structure at the join point
  - We keep edges from both graphs
  - Places no longer live at join are removed

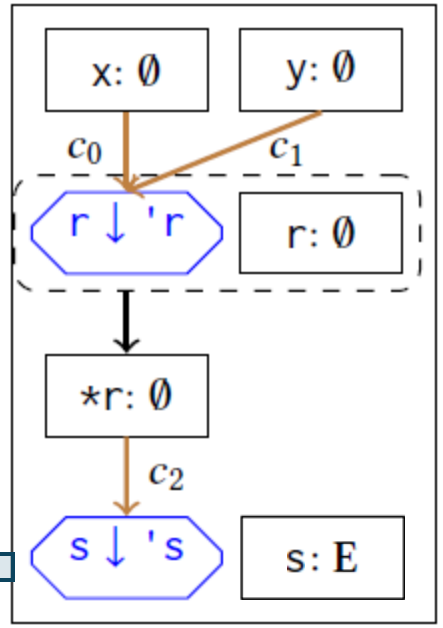
Path Sensitivity

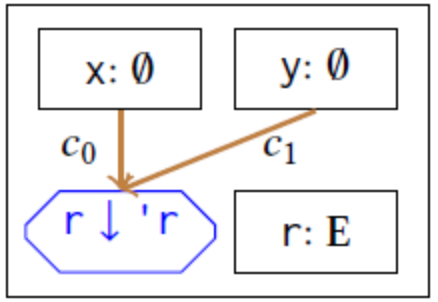
```
fn f(mut x: i32, mut y: i32, mut z: i32) {
  let r = if x > y {      // branch: c0/c1
    &mut x                // bb1 (c0)
  } else {
    &mut y                // bb2 (c1)
  };                      // bb3
  let s = if z > 5 {      // branch: c2/c3
    &mut *r               // bb4 (c2)
  } else {
    &mut z                // bb5 (c3)
  };
  *s = 5; // bb6
}
```



Path Sensitivity

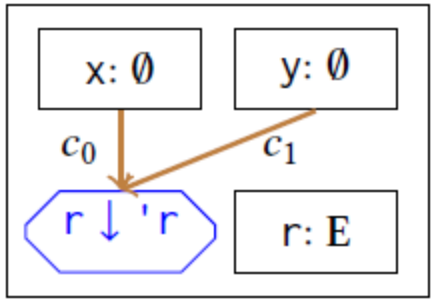
```
fn f(mut x: i32, mut y: i32, mut z: i32) {
  let r = if x > y {      // branch: c0/c1
    &mut x                // bb1 (c0)
  } else {
    &mut y                // bb2 (c1)
  };                      // bb3
  let s = if z > 5 {      // branch: c2/c3
    &mut *r               // bb4 (c2)
  } else {
    &mut z                // bb5 (c3)
  };
  *s = 5; // bb6
}
```

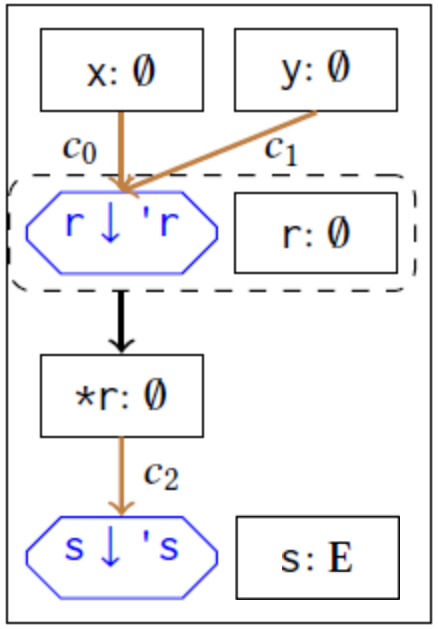


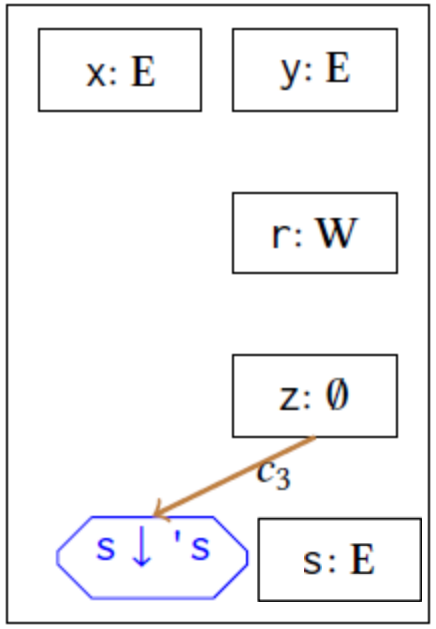


Path Sensitivity

```
fn f(mut x: i32, mut y: i32, mut z: i32) {
  let r = if x > y {      // branch: c0/c1
    &mut x                // bb1 (c0)
  } else {
    &mut y                // bb2 (c1)
  };                      // bb3
  let s = if z > 5 {      // branch: c2/c3
    &mut *r               // bb4 (c2)
  } else {
    &mut z                // bb5 (c3)
  };
  *s = 5; // bb6
}
```

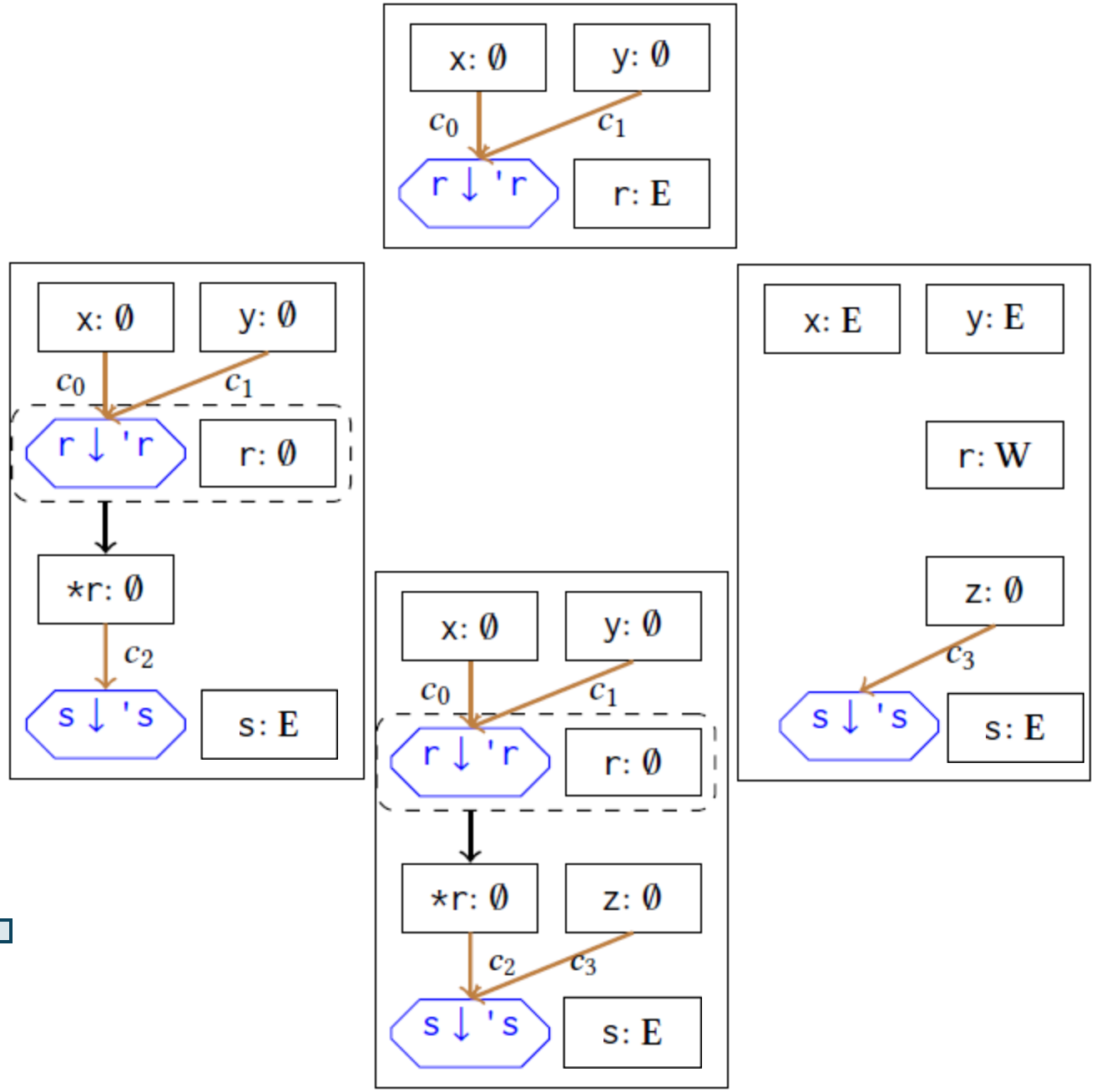






Path Sensitivity

```
fn f(mut x: i32, mut y: i32, mut z: i32) {
  let r = if x > y {      // branch: c0/c1
    &mut x                // bb1 (c0)
  } else {
    &mut y                // bb2 (c1)
  };                      // bb3
  let s = if z > 5 {      // branch: c2/c3
    &mut *r               // bb4 (c2)
  } else {
    &mut z                // bb5 (c3)
  };
  *s = 5; // bb6
}
```

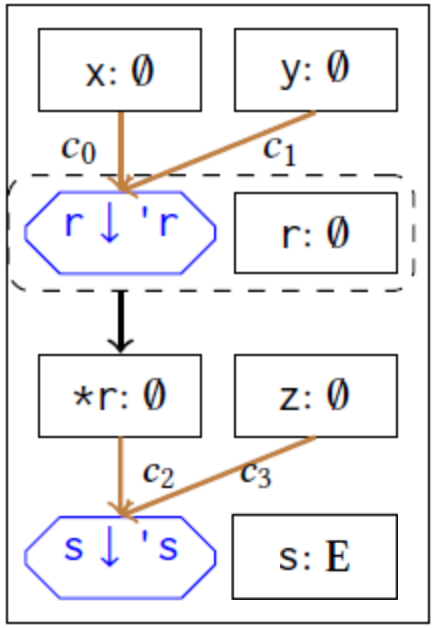


Path Sensitivity

```
fn f(mut x: i32, mut y: i32, mut z: i32) {
  let r = if x > y {      // branch: c0/c1
    &mut x                          // bb1 (c0)
  } else {
    &mut y                          // bb2 (c1)
  };                                // bb3
  let s = if z > 5 {      // branch: c2/c3
    &mut *r                         // bb4 (c2)
  } else {
    &mut z                          // bb5 (c3)
  };
  *s = 5; // bb6
}
```

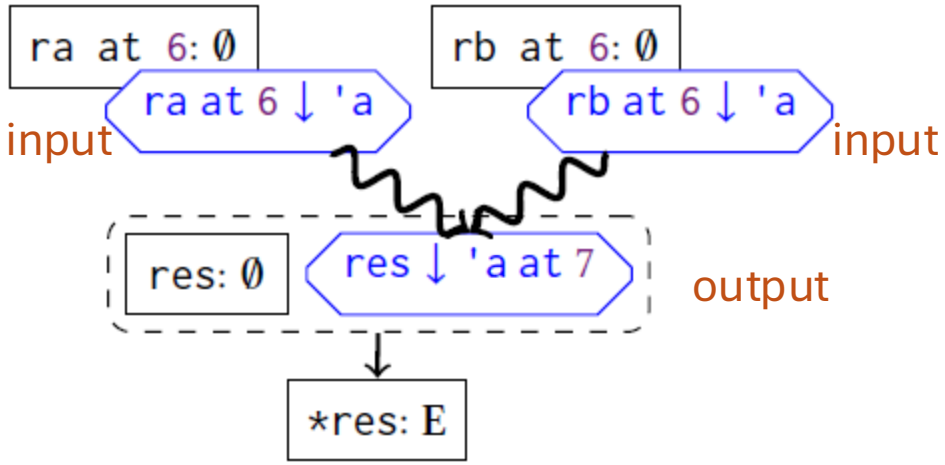
Maximal Sharing

- Resulting graph shares nodes and edges *unless* they differ in the two graphs
- Sharing the same graph from here *onwards* works because Rust is not path-sensitive
  - Graphs must agree on their leaves
  - So, we can extend them uniformly
- Sharing structure *above* a join is trickier
  - Edges may be removed during branch
  - But only the same edges! Feel free to ask..



Function Calls

```
1 fn choose<'a, T>(c: bool, rx: &'a mut T, ry: &'a mut T) -> &'a mut T {
2   if c { rx } else { ry }
3 }
4
5 fn foo<'a>(ra: &'a mut u32, rb: &'a mut u32) {
6   let res = choose(true, ra, rb);
7   *res = 10;
8 }
```



Abstract Edges

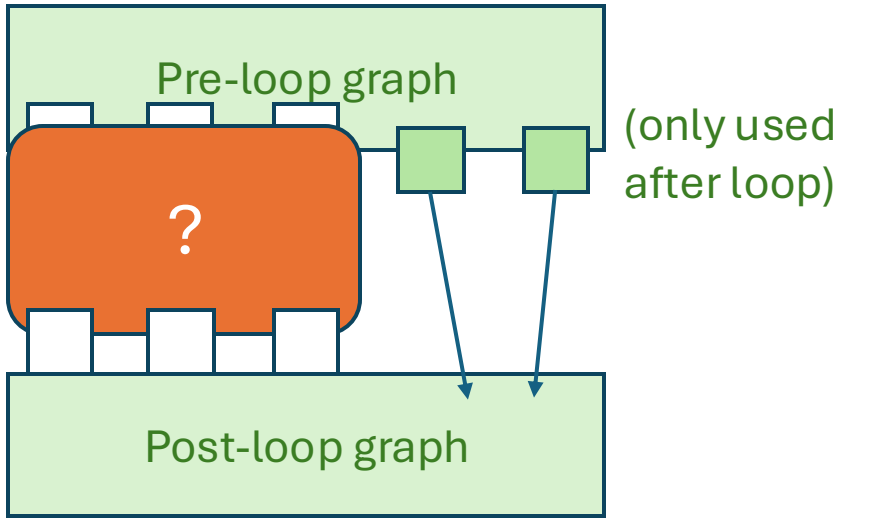
- Connect lifetime projection nodes (can be many-to-many)
- Represent (possible) flows of (re)borrows not tracked precisely
  - Some of the borrowed memory corresponding to source node(s) may currently be borrowed in the target node(s)
  - Corresponds to borrow-checking constraint on

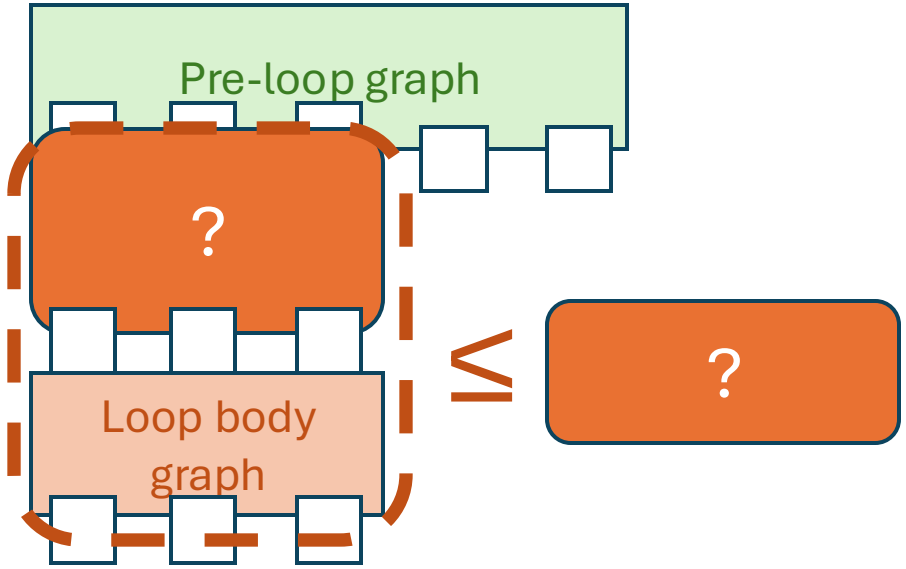
Handling Function Calls

- Calculate from function signature:
  - "input" lifetime projections
  - "output" lifetime projections
- For each input-output pair:
  - Connect with abstract edge iff borrow checker says source must outlive the target

Handling Loops – what is a loop invariant?

```
0 struct List { head: u32, tail: Option<Box<List>> }
1 fn penult<'a>(list: &'a mut List) -> Option<&'a mut u32>
2 {
3   let mut curr: &'a mut List = &mut *list;
4   let mut prev: Option<&'a mut u32> = None;
5   while let Some(ref mut tail) = curr.tail {
6     prev = Some(&mut curr.head); curr = &mut *tail;
7   }
8   prev
9 }
```



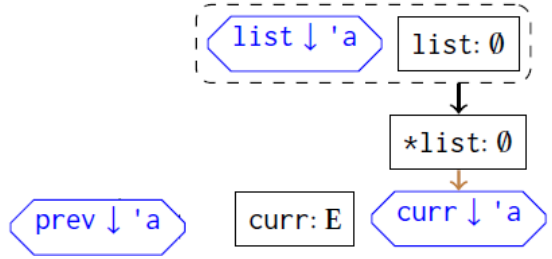


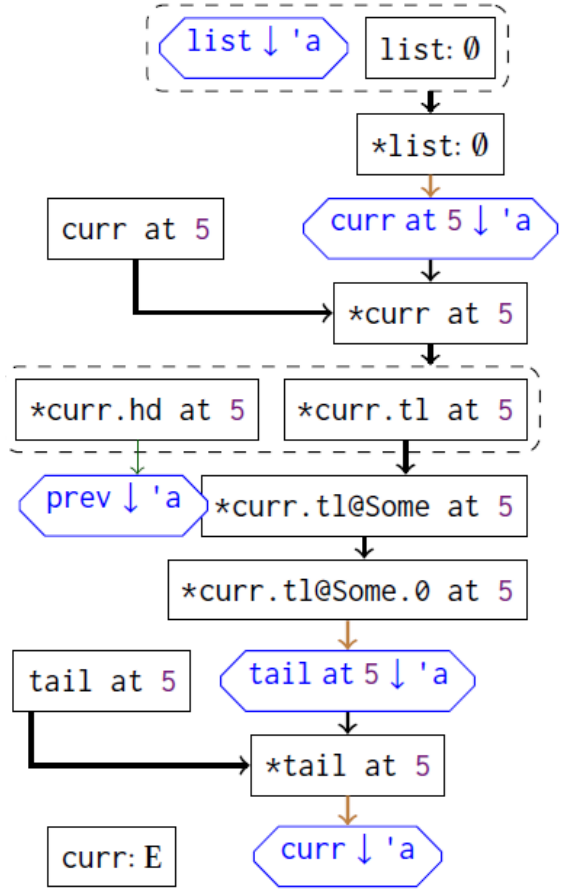
Handling Loops

```
0 struct List { head: u32, tail: Option<Box<List>> }
1 fn penult<'a>(list: &'a mut List) -> Option<&'a mut u32>
2 {
3   let mut curr: &'a mut List = &mut *list;
4   let mut prev: Option<&'a mut u32> = None;
5   while let Some(ref mut tail) = curr.tail {
6     prev = Some(&mut curr.head); curr = &mut *tail;
7   }
8   prev
9 }
```

Key Ideas

- Initial graph (before loop) is already known
- Identify places accessed in loop body
  - Reborrowing inside loops is the challenge
- Analyse loop body with “havoced” copies:



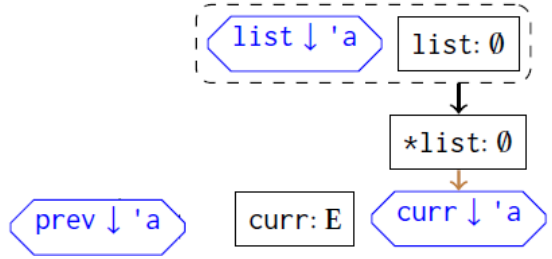


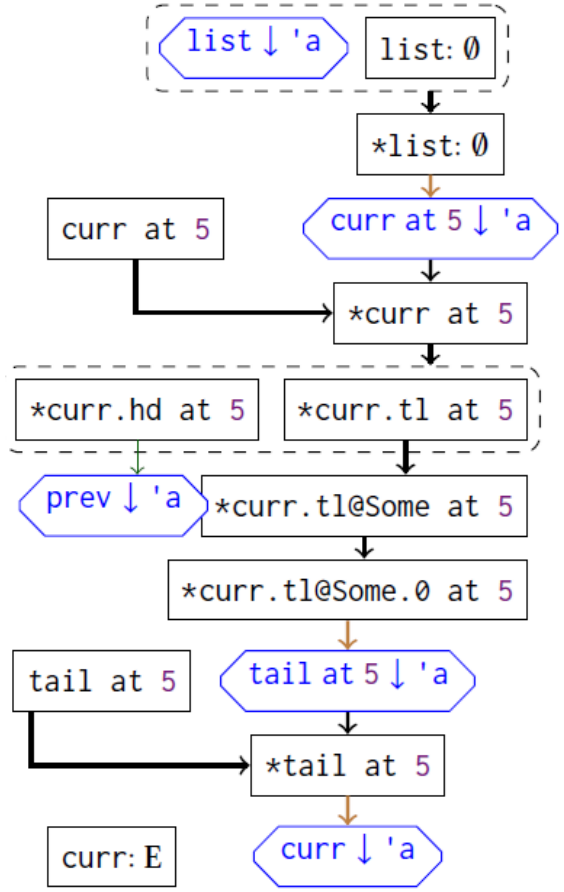
Handling Loops

```
0 struct List { head: u32, tail: Option<Box<List>> }
1 fn penult<'a>(list: &'a mut List) -> Option<&'a mut u32>
2 {
3   let mut curr: &'a mut List = &mut *list;
4   let mut prev: Option<&'a mut u32> = None;
5   while let Some(ref mut tail) = curr.tail {
6     prev = Some(&mut curr.head); curr = &mut *tail;
7   }
8   prev
9 }
```

Key Ideas

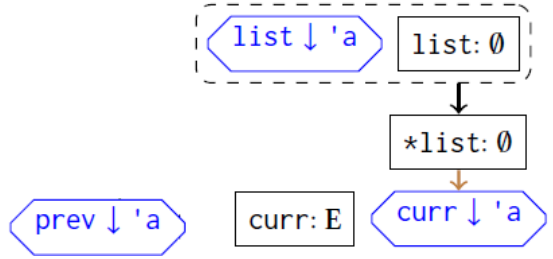
- Initial graph (before loop) is already known
- Identify places accessed in loop body
  - Reborrowing inside loops is the challenge
- Analyse loop body with “havoced” copies:
- Abstract to only lifetime projection nodes:

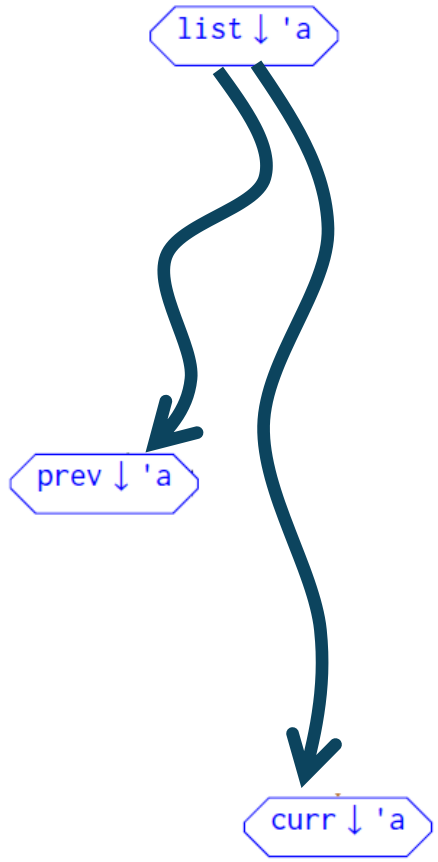




Handling Loops

```
0 struct List { head: u32, tail: Option<Box<List>> }
1 fn penult<'a>(list: &'a mut List) -> Option<&'a mut u32>
2 {
3   let mut curr: &'a mut List = &mut *list;
4   let mut prev: Option<&'a mut u32> = None;
5   while let Some(ref mut tail) = curr.tail {
6     prev = Some(&mut curr.head); curr = &mut *tail;
7   }
8   prev
9 }
```



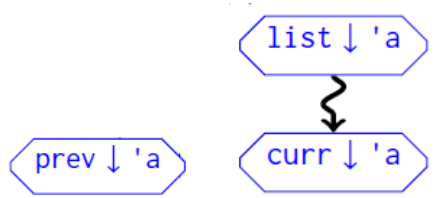


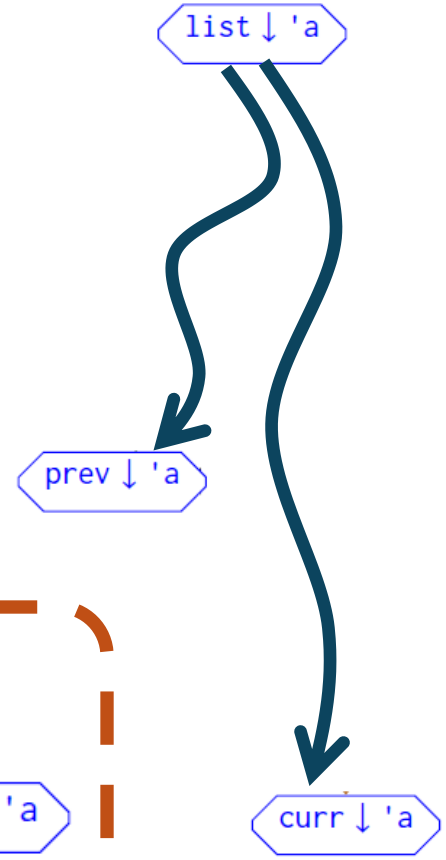
Key Ideas

- Initial graph (before loop) is already known
- Identify places accessed in loop body
  - Reborrowing inside loops is the challenge
- Analyse loop body with “havoced” copies:
- Abstract to only lifetime projection nodes:
- Abstract further - those live in current state:

Handling Loops

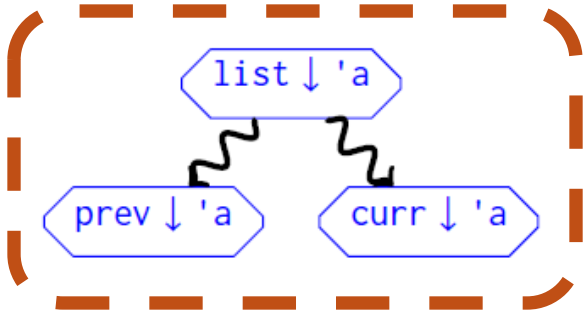
```
0 struct List { head: u32, tail: Option<Box<List>> }
1 fn penult<'a>(list: &'a mut List) -> Option<&'a mut u32>
2 {
3   let mut curr: &'a mut List = &mut *list;
4   let mut prev: Option<&'a mut u32> = None;
5   while let Some(ref mut tail) = curr.tail {
6     prev = Some(&mut curr.head); curr = &mut *tail;
7   }
8   prev
9 }
```





Key Ideas

- Initial graph (before loop) is already known
- Identify places accessed in loop body
  - Reborrowing inside loops is the challenge
- Analyse loop body with “havoced” copies:
- Abstract to only lifetime projection nodes:
- Abstract further - those live in current state:
- Apply same abstraction to initial graph
- Merge the graphs, keeping all edges:



Other features: no time, but ask if you like!

- PCG model supports general (safe Rust) types
  - Unpack notions generalise to e.g. enum -> cases projections
- Trickier: Rust composite types (structs, enums) can store borrows
  - The relevant source-level lifetimes are part of type declarations
  - Our lifetime projection nodes represent sets of borrows (for this reason)
  - e.g. ListRefs<‘a> places will get a lifetime projection node for ‘a
    - Represents entire set of borrows stored in the list, abstracted to one node
- Some generalisations are harder (and still being improved):
  - The notion of “inputs” and “outputs” to functions
  - Need for distinct versions of lifetime projection nodes in some cases
  - e.g. for a type &‘b mut ListRefs<‘a> passed into a function, the set of borrows represented by lifetime projection node ‘a for can itself be mutated!
- We have rules for this, but they are complex (currently on version

Evaluation: using the PCG model

Experimental evaluation(s)

- Automatically analysed functions in top 500 Rust crates (crates.io)
  - Successfully generated PCGs for (all of the code of) *98.4% of all functions*
  - Most unsupported functions involve *raw pointers* (unsafe code)

- How to gain confidence that these PCGs reflect Rust's *restrictions*?
  - e.g. that generated PCGs precisely reflect borrow-checker constraints
- We developed a form of automated mutation testing based on PCGs
  - Analyse the structure of our generated graphs for existing code
  - Use these to generate modified programs the graphs say *shouldn't* type-check
  - Re-run the compiler on these, and compare errors with these

PCG generation: functions in top 500 crates

Status	Details	Count
Success		119,338
Unsupported	DerefUnsafePtr	1,629
	~~ClosureCall~~	~~215~~
	InlineAssembly	80
	ExpansionOfAliasType	6
	IndexingNonIndexableType	5
	AssignBorrowToNonReferenceType	1
	Timeout	1
Total		121,274

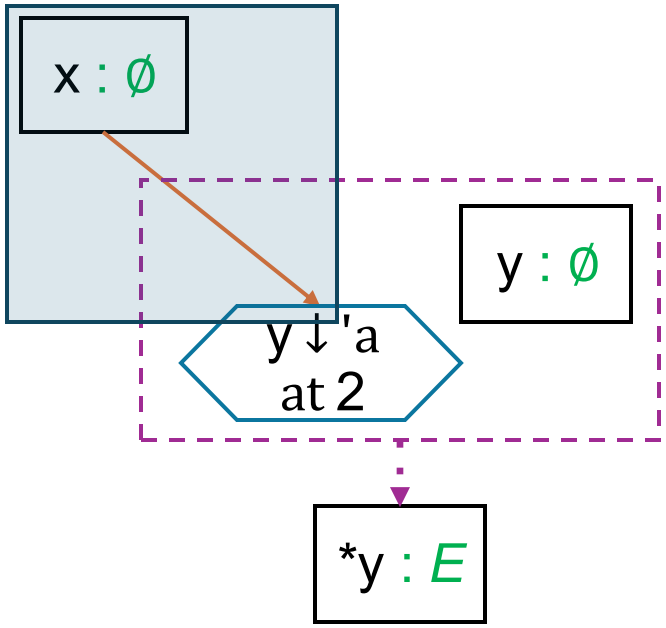
Hot off the press: now supported

Engineering Issue:
Normalizing type aliases erases regions

Macro-Generated Code
Fixable with performance optimizations

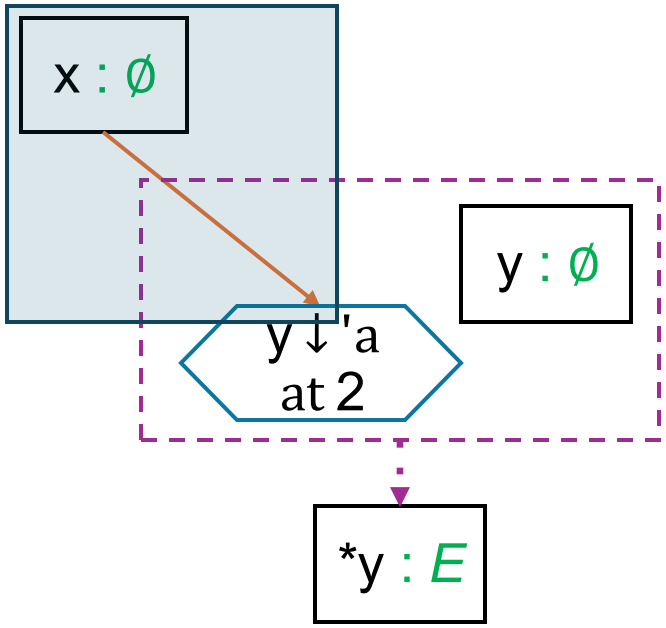
Mutation testing

```
fn main() {
1   let mut x = 1;
2   let y = &x;   // x is immutably borrowed
3   let z = &*y;  // x remains immutably borrowed
...
}
```



Mutation testing

```
fn main() {
1   let mut x = 1;
2   let y = &x;   // x is immutably borrowed
3   let z = &*y;  // x remains immutably borrowed
...
}
```



Mutation testing

```
fn main() {
1   let mut x = 1;
2   let y = &x;   // x is immutably borrowed
3   let m = &mut x; // mutation
4   let z = &*y;  // x remains immutably borrowed
…
}
```

```
cannot borrow `x` as mutable because it is also
borrowed as immutable
```

Mutation testing results

Mutation	*tot*	*fail*	*pass*	*%fail*
MutablyLendShared	1009867	1009497	370	99.96%
ReadFromWriteOnly	395299	395299	0	100.00%
WriteToShared	1009396	1008980	370	99.96%
MoveFromBorrowed	419555	419264	291	99.93%
BorrowExpiryOrder	196470	196374	96	99.96%
AbstractExpiryOrder	38220	38203	17	99.96%
Total	3068807	3067617	1144	99.96%

Note: failure is good!

Found bugs (fixed) in our implementation
But also very strange rustc behaviours - reporting

Integrating PCG into Rust Analysis Tools

- Used PCG to develop new prototype forks of Flowistry and Prusti tools
  - PCG is a standalone Rust crate supporting various queries; easy to hook up

- Flowistry – Information/Data Flow Analysis Tool
  - Replaced Flowistry's Alias Analysis with PCG queries
    - Thanks to Will Crichton for suggesting this possibility!
  - All existing test cases pass

- Prusti – Deductive Verifier for Rust
  - PCGs suitable for automating all core proof steps
  - Support for new features:
    - structs containing borrows, reborrowing across loops, etc.

Consuming the PCG in practice

- PCG implemented as a Rust crate – no external dependencies

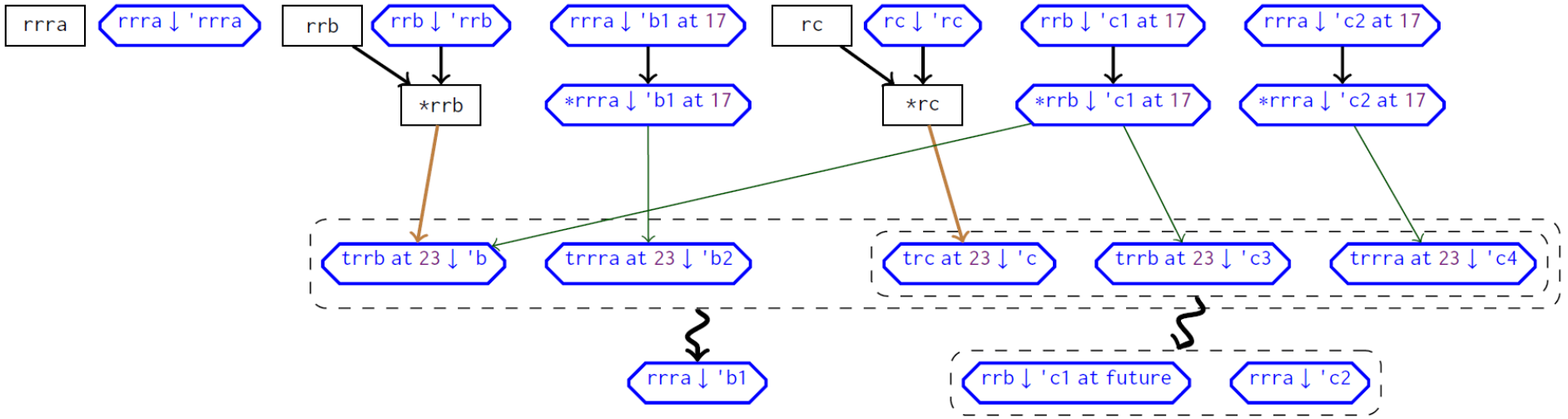
- Clients can query PCG at a program point, to ask questions like:
  - *What is the capability of place x?*
  - *What places do *y alias (and under what conditions)?*

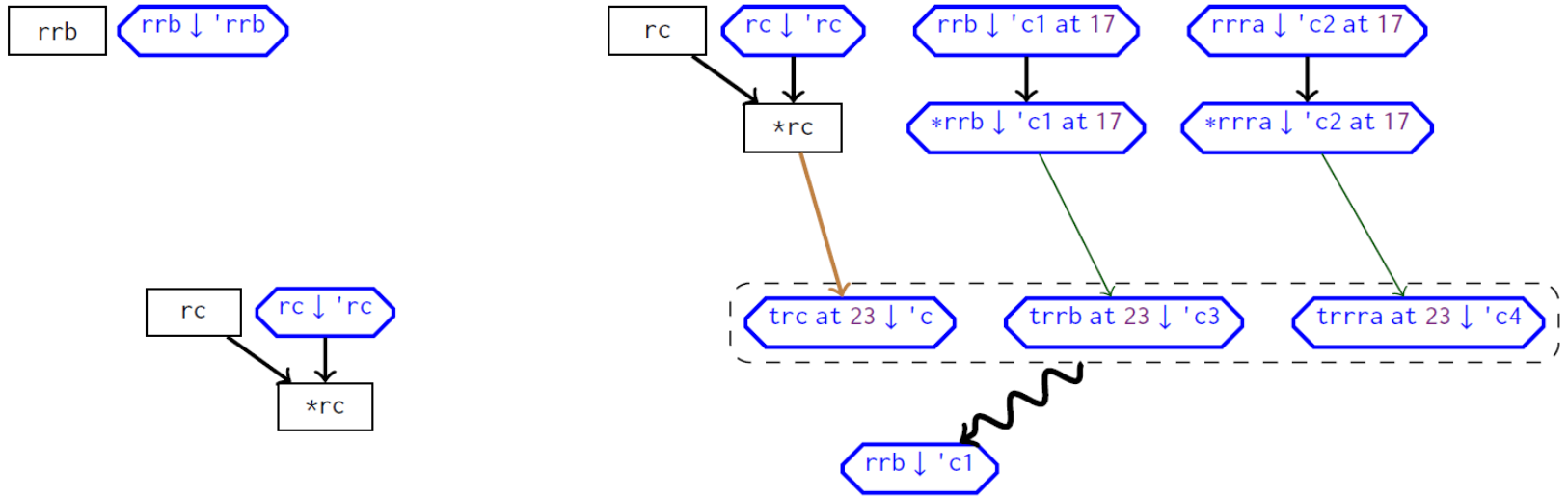
- Clients can query for PCG *annotations* between program points
  - *What capabilities were transferred at this program point?*
  - *What is the shape of the function call / loop invariant at this point?*
  - e.g. Prusti prototype translates PCG annotations into Viper commands

Future Work (ideas)

- Formalizing our PCG model
  - The graphs themselves (including nice properties around abstract edges)
  - The underlying capabilities (treated linearly, etc.)
  - Framing/immutability guarantees for actual program state
  - Extensions to symbolic execution (value information) over graphs
- Using PCG to visualize/debug/evaluate future borrow-checkers
- Possibly using the model to define borrow-checking
- Trying to simplify/justify rules for handling nested lifetimes
- Extension of the model to reason about some kinds of unsafe code
- Supporting other consumers of the model (analysis tools)
- .. your ideas? ☺

```rust
fn f<'a, 'b, 'c>(
    x: &'a mut &'b mut &'c mut i32,
    y: &'b mut &'c mut i32,
    z: &'c mut i32,
) { *y = z; *x = y; }

fn caller() {
    let mut a = 1;
    let mut b = 2;
    let mut c = 3;
    let mut ra = &mut a;
    let mut rb = &mut b;
    let mut rc = &mut c;
    let mut rra = &mut ra;
    let mut rrb = &mut rb;
    let mut rrra = &mut rra;

    f(rrra, rrb, rc);

    ***rrra = 1;
    **rrb = 1;
    *rc = 4;
}
```





alex.summers@ubc.ca / https://prusti.zulipchat.com/