



# Place Capability Graphs: A General-Purpose Model of Rust's Ownership and Borrowing Guarantees

ZACHARY GRANNAN, University of British Columbia, Canada

AUREL BÍLY, ETH Zurich, Switzerland

JONÁŠ FIALA, ETH Zurich, Switzerland

JASPER GEER, University of British Columbia, Canada

MARKUS DE MEDEIROS, New York University, USA

PETER MÜLLER, ETH Zurich, Switzerland

ALEXANDER J. SUMMERS, University of British Columbia, Canada

Rust's novel type system has proved an attractive target for verification and program analysis tools, due to the rich guarantees it provides for controlling aliasing and mutability. However, fully understanding, extracting and exploiting these guarantees is subtle and challenging: existing models for Rust's type checking either support a smaller idealised language disconnected from real-world Rust code, or come with severe limitations in terms of precise modelling of Rust borrows, composite types storing them, function signatures and loops.

In this paper, we present *Place Capability Graphs*: a novel model of Rust's type-checking results, which lifts these limitations, and which can be directly calculated from the Rust compiler's own programmatic representations and analyses. We demonstrate that our model supports over 97% of Rust functions in the most popular public crates, and show its suitability as a general-purpose basis for verification and program analysis tools by developing promising new prototype versions of the existing Flowistry and Prusti tools.

CCS Concepts: • **Theory of computation** → **Program analysis**; **Type structures**.

Additional Key Words and Phrases: program analysis, type systems, Rust, borrow checking

## ACM Reference Format:

Zachary Grannan, Aurel Bíly, Jonáš Fiala, Jasper Geer, Markus de Medeiros, Peter Müller, and Alexander J. Summers. 2025. Place Capability Graphs: A General-Purpose Model of Rust's Ownership and Borrowing Guarantees. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 344 (October 2025), 28 pages. <https://doi.org/10.1145/3763122>

## 1 Introduction

Rust is a popular programming language, combining competitive performance with a rich and expressive type system that eliminates many common programming errors and unintended side-effects. Rust's type system applies a notion of *ownership* to all allocated memory, by default preventing access through any variable other than its owner, and defining its deallocation time (when its owner goes out of scope). Aliasing is restricted by a notion of *borrowed references* which may temporarily access memory that they alias but do not own. As the name suggests, the ability to

---

Authors' Contact Information: [Zachary Grannan](#), University of British Columbia, Vancouver, Canada, [zgrannan@cs.ubc.ca](mailto:zgrannan@cs.ubc.ca); [Aurel Bíly](#), ETH Zurich, Zurich, Switzerland, [aurel.bily@inf.ethz.ch](mailto:aurel.bily@inf.ethz.ch); [Jonáš Fiala](#), ETH Zurich, Zurich, Switzerland, [jonas.fiala@inf.ethz.ch](mailto:jonas.fiala@inf.ethz.ch); [Jasper Geer](#), University of British Columbia, Vancouver, Canada, [jgeer35@cs.ubc.ca](mailto:jgeer35@cs.ubc.ca); [Markus de Medeiros](#), New York University, New York City, USA, [mjd9606@nyu.edu](mailto:mjd9606@nyu.edu); [Peter Müller](#), ETH Zurich, Zurich, Switzerland, [peter.mueller@inf.ethz.ch](mailto:peter.mueller@inf.ethz.ch); [Alexander J. Summers](#), University of British Columbia, Vancouver, Canada, [alex.summers@ubc.ca](mailto:alex.summers@ubc.ca).



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2475-1421/2025/10-ART344

<https://doi.org/10.1145/3763122>

Table 1. Most-closely-related work, compared on the criteria motivating our model and paper.

| Selected Existing Work          | Our Key Criteria (cf. page 3) |                          |                                  |                               |                              | Other Borrowing Challenges (cf. page 1) |                       |                           |                         |
|---------------------------------|-------------------------------|--------------------------|----------------------------------|-------------------------------|------------------------------|-----------------------------------------|-----------------------|---------------------------|-------------------------|
|                                 | (1) Function-Modular Model    | (2) Path-Sensitive Info. | (3) Connect from + to Rust repr. | (4) Most Real-world Rust Code | (5) Use for Diverse Analyses | Struct Lifetime Params                  | Loop Reborrow Summary | Function Reborrow Summary | Defines Borrow Checking |
| Oxide [41]                      | ✓                             |                          | <sup>1</sup>                     |                               |                              |                                         | ✓                     | ✓                         | ✓                       |
| Aeneas [19]                     | ✓                             |                          | <sup>1</sup>                     |                               | ✓                            |                                         | <sup>2</sup>          | ✓                         | ✓                       |
| RustBelt [22]                   | ✓                             | ✓                        |                                  |                               |                              | ✓                                       |                       | ✓                         | ✓                       |
| Stacked / Tree Borrows [21, 38] |                               |                          | ✓                                | ✓                             |                              | ✓                                       |                       |                           |                         |
| PCGs                            | ✓                             | ✓                        | ✓                                | ✓                             | ✓                            | ✓                                       | ✓                     | ✓                         |                         |

<sup>1</sup> compilation from Rust only; results for custom program representations only. <sup>2</sup> partial heuristic support: see [19] for details.

access borrowed memory is taken temporarily, and then restored to its lender. Rust allows multiple aliases to the same memory to be usable at once only if none can mutate its content.

The restrictive aliasing discipline of Rust’s type system makes the language attractive as a static analysis and verification target, given the challenges of reasoning about aliasing in general (the infamous *frame problem*). A wealth of formal techniques have grown up for Rust, including formalisations [20, 22, 27, 41], (deductive) verification tools targeting specification-annotated Rust code [1, 11, 25, 26], and static analyses [2, 10, 37]. These projects leverage the rich guarantees provided by Rust’s type checking in the ways they internally model program memory, track aliasing and value information, and interpret user-provided specifications of desired program properties.

This paper does not define Rust’s type-checking *algorithms*, but rather defines a detailed model which substantially elaborates their *results* in a way which reifies Rust’s rich guarantees, and is exploitable for diverse program analysis and verification techniques and tools. Comprehensively modelling Rust’s type checking is technically challenging for a variety of reasons. The properties implied at different program points are *flow-sensitive*; for example, whether a variable currently owns memory is not visible from its declared type or its initialisation, as Rust allows *move assignments* to transfer ownership between *places* (Rust’s terminology for expressions denoting locations). Similarly, whether a place has been borrowed-from, and for how long, are flow-sensitive properties. Rust’s type checking includes *borrow checking*, which determines whether consistent choices can be made for the extents of each borrow; the points at which borrows end are invisible in source code, but important for code understanding and analysis: this is when side-effects previously performed via the borrowed references become observable via the borrowed-from places.

Capturing Rust’s notions of borrowing for general types and programs yields several sources of complexity: (1) instances of composite types (such as structs) may store borrows. These types are declared with *nested lifetimes*, which indicates to Rust’s type checker the possibly unbounded number of borrows stored within them; notions of borrowing are generalised to the sets of borrows stored in these types. (2) Rust allows for *reborrowing*: transitively creating chains of borrows to parts of memory previously borrowed. These borrows can be created and expired differently in different paths through the program, so precisely tracking their effects requires *path-sensitive* information. (3) Programs with loops can create (re)borrows unboundedly many times. To capture Rust’s type checking in such cases, loops must be summarised in a way which matches the compiler’s rules. (4) A model of Rust’s borrowing must respect the modularity of Rust function calls, including the subtle ways in which Rust allows programmers to abstract over constraints between (nested) lifetimes in function signatures. We explain the full details of these (and other) challenges in Sec. 2.

In this paper, we present a novel model of the results of Rust’s type checking, designed to address all of these technical challenges and to enable downstream tools to exploit the properties implied by Rust’s design. Our model does *not* define Rust type checking itself, but captures in detail the

*results* of successfully type checking a Rust program as defined by the standard Rust compiler. It seamlessly combines solutions to two technical objectives: (1) where computable, all information a downstream analysis might require is *precise*, and (2) where abstractions are necessary in general either for computability or modularity reasons (for loops, function calls and types containing sets of borrows), these abstractions are consistent with those employed by the compiler. While many existing works define a variety of formal models for different aspects of Rust's design (we discuss these in detail in Sec. 5), these do not handle the full set of challenges defined above. We provide a high-level overview of how some related work compares in terms of the objectives of our own work in Table 1. We note that many existing works have objectives we do not address, such as defining operational semantics for Rust, or defining how type checking itself can/should be done.

Overall, we have designed our novel model to satisfy the following criteria:

- (1) The model provides a detailed, function-modular account of *how and why* a program type-checks, tracking at each program point in a function's body which places own and borrow which memory, which are currently borrowed-from and how, and when *and how* these borrowed capabilities (to read or write) flow back to their origins. Our model uses a novel representation for sets of borrows, allowing us to connect to the compiler's borrow-checking concepts, while abstracting over the concrete details of any given borrow-checker algorithm or implementation, making it possible to connect to alternative Rust borrow checkers.
- (2) The model captures precise path-sensitive information for non-recursive control flow, and summarises function calls and loops with abstractions that precisely reflect the shape of constraints imposed by borrow checking (this turns out to be complex for loops that reborrow memory). The former grants consumers the freedom to demand as much information as desired, while the latter allows supporting general Rust code (e.g. that following a summarised loop) while providing consumers with an abstraction that could be enriched by further analysis or annotations. In particular, our loop and function summaries are suitable for connection with user specifications in a deductive verification setting.
- (3) All parts of the model have a clear and easily-checkable correspondence with notions in the Rust compiler, e.g. the program representation and type-checking features. Consequently, the model can be used to visualise and understand type checking itself. This close connection avoids potential gaps between the de-facto definition and semantics of Rust as defined by the compiler and the representations and information comprising instances of our model.
- (4) Our model handles a large fragment of real-world Rust code (over 97% of the functions in the top-500 crates), and is automatically generated from compiler runs, ensuring the practical applicability of tools built upon our model, and demonstrating its generality in practice.
- (5) The model is suitable for consumption by a wide variety of static analysis and verification tools, providing at least the information they currently require, and potentially extending their reach to a wider class of Rust programs; this makes its definition a significant advance for designing and implementing detailed Rust analysis and verification tools.

As we explain in Sec. 5 and illustrate in Table 1, existing models of Rust's type checking do not meet multiple criteria from this list. Conversely, many existing verification and analysis tools could extract the information they rely on from our model: our work makes building static tools for Rust a substantially simpler task. In summary, this paper's key contributions are as follows:

- (1) We define and present *Place Capability Graphs* (PCGs), as a detailed and general-purpose model of Rust's type checking satisfying the full criteria above; in particular, the model directly builds upon the Rust compiler's programmatic representations and analysis results.
- (2) We provide an implementation of our model, written as a standalone reusable artefact (crate) in Rust; this implementation is open-sourced [13] and the artefact is available online [16].

|   |                                                                 |   |                                                                         |
|---|-----------------------------------------------------------------|---|-------------------------------------------------------------------------|
| 1 | <code>struct Pos2D&lt;T&gt; { x: T, y: T }</code>               | 1 | <code>fn replace_x&lt;'a, T&gt;(pos: &amp;'a mut Pos2D&lt;T&gt;,</code> |
| 2 |                                                                 | 2 | <code>new_x: T) {</code>                                                |
| 3 | <code>fn replace_x_own&lt;T&gt;(mut pos: Pos2D&lt;T&gt;,</code> | 3 | <code>let x_ref = &amp;mut (*pos).x;</code>                             |
| 4 | <code>new_x: T) -&gt; Pos2D&lt;T&gt; {</code>                   | 4 | <code>*x_ref = new_x;</code>                                            |
| 5 | <code>let old_x = pos.x;</code>                                 | 5 | <code>}</code>                                                          |
| 6 | <code>pos.x = new_x;</code>                                     | 6 | <code>fn caller(mut original: Pos2D&lt;i32&gt;) {</code>                |
| 7 | <code>return pos;</code>                                        | 7 | <code>let pos = &amp;mut original;</code>                               |
| 8 | <code>}</code>                                                  | 8 | <code>replace_x(pos, 0);</code>                                         |
|   |                                                                 | 9 | <code>}</code>                                                          |

(a)

(b)

Fig. 1. Examples to showcase Rust move assignments and in-place mutations.

- (3) We evaluate our model's generality by applying it to the entire codebases of the top-500 Rust crates at time of writing; we demonstrate support for over 97% of the published real-world code, and successfully generate models for all program points in these functions.
- (4) We employ a form of mutation testing to generate Rust programs that the compiler should *not* accept according to our model, and verify that the restrictions captured by our model match the compiler's results; this provides confidence that our model accurately captures the *restrictions* imposed by Rust's type system in terms of memory accesses it disallows.
- (5) We develop new prototype versions of two existing Rust analysis tools as consumers of our model: the information-flow analysis Flowistry [10] and the deductive verifier Prusti [1]. Our PCGs replace Flowistry's alias analysis and Prusti's proof annotation algorithm. Integrating our PCGs dramatically simplified aspects of Prusti's implementation; all required annotations to drive its formal proofs are deterministically read off from our model.

Our paper<sup>1</sup> is organised as follows. Sec. 2 presents specific challenges that arise when reasoning about Rust ownership and borrowing, also presenting key background on Rust itself. Sec. 3 introduces our PCG model and defines the rules governing its construction based on information from the Rust compiler. Our evaluation in Sec. 4 demonstrates the accuracy and wide applicability of our model. In Sec. 5, we compare our model to related work, and hypothesise how our model could benefit existing analysis and verification tools. We conclude and discuss future work in Sec. 6.

## 2 Capturing Rust

In this section, we provide an overview of Rust and present the challenges of modelling Rust's type checking<sup>2</sup> to be both useful to downstream consumers and accurately reflect the compiler.

### 2.1 Ownership

As explained in the introduction, ownership is a core concept in Rust's type system. In the simplest case, *move* assignments transfer ownership of the source's memory into the destination. Consider the program `replace_x_owned` in Fig. 1a, whose assignments are all *move*. Initially, `pos` owns an instance of the `Pos2D` struct, and `new_x` owns a `T`<sup>3</sup>. For many Rust types (including `Pos2D`) the ownership of a struct equates to owning each field, which can be moved into and out of individually. The move in Line 5 transfers ownership *only* from `pos.x` to `old_x`, therefore, attempting to read from `pos.x` or `pos` immediately after this line will cause a type-checking error.

<sup>1</sup>A version of our paper as a technical report is also available on arXiv [17].

<sup>2</sup>We will use "type checking" to refer to both classical type checking concerns and Rust-specific notions of borrow checking.

<sup>3</sup>Unlike in languages like Java/C++, function arguments in Rust always refer to owned, initialised, and distinct memory.

Line 6 also transfers ownership, this time from `new_x` to `pos.x`. In combination with the never-removed ownership to `pos.y` this move reconstitutes a complete `Pos2D` instance in `pos`, which is then free to be returned to the caller on Line 7. If Line 6 were omitted, this return would fail to type check for the same reason we couldn't read from `pos.x` immediately following Line 5.

## 2.2 Borrowing, Borrow Extents, and Borrow Checkers

Rust permits a controlled form of aliasing through *borrowing*, with which the exclusive ability to read or write to a piece of memory can be loaned to a reference aliasing it. The function `replace_x` in Fig. 1b implements an in-place variant of `replace_x_owned` using *mutable borrows*. Recall that in `replace_x_owned` we were able to mutate the field `x` because the function owned its `pos` argument. In Fig. 1b `pos` does not own its target (whose owner is on the call-stack), but still has the exclusive ability to read and write to it. This allows `replace_x` to mutate a `Pos2D` struct in-place.

The mutation in `replace_x` occurs under a *reborrow*. On Line 3 a new borrow `x_ref` temporarily obtains the exclusive ability to write to a field of the target of `pos`, which it does on Line 4.<sup>4</sup> Like the previous example, `x_ref` only borrows from the `x` field of the underlying struct, and neither `(*pos).x` nor `*pos` can be read from right after Line 3 (though `(*pos).y` can). Note that it is not ownership of `*pos` being lent, but instead the exclusive ability to read and write to it. Following prior literature on related type systems [3, 4, 7, 15, 18, 35] in this paper we refer to these rights as *capabilities*: the (possibly unique) ability to read or write to the underlying borrowed memory.

*Borrow Extents and Borrow Checking.* The Rust compiler assigns each borrow a contiguous portion of the program (in the sense of control flow) for which the borrow is allowed to use or loan out the capabilities to its target. We refer to these as *borrow extents*. Some borrow extents are annotated in Rust code by *named lifetimes* such as 'a in our example above, though in general a Rust program contains vastly more borrow extents than it does named lifetimes.<sup>5</sup>

Borrow extents can be constrained relative to each other. For example, a new reborrow (like `x_ref` in our example) cannot have a longer extent than the borrow it was created from. In addition to these *outlives constraints*, usages of a borrow's target will constrain borrow extents to include a set of program points. *Borrow checking* is the process of ensuring that there exists a solution to a system of borrow constraints. There are multiple borrow checking algorithms for Rust which provide varying degrees of detail about the system's solution, and in fact can describe different solutions altogether! Precisely modelling borrow extents (in particular, when borrows end) is critical for downstream analyses that reason precisely about how memory is accessible, aliased and updated via chains of borrows. For tools which build their analysis around how borrowed capabilities flow back to their lenders (e.g. Prusti [1] and Aeneas [20]), and any other analyses demanding precise aliasing information, it is essential to model how chains of borrows are related to each other, and how the capabilities from those borrows transfer when those borrows expire.

## 2.3 Path-Sensitivity

Although Rust's type checking is flow insensitive in terms of *outcomes* (e.g. whether a place currently has capability to read or write its underlying memory does not depend on the control flow taken),

<sup>4</sup>Rust allows many dereference operators to be elided, but in this section we make them explicit to aid understanding.

<sup>5</sup>Additional borrow extents can arise for a variety of reasons: (1) most lifetimes are elided in source programs, although elaborated types (with lifetimes) are explicit in Rust's Mid-level Intermediate Representation (MIR), (2) even after elaboration, some relevant lifetimes do not appear in the declared types of any variable, but are associated only with temporary sub-expressions, (3) Rust's newer borrow-checking algorithms, in particular Polonius [33, 34], work in terms of finer-grained notions of borrow extents than named lifetimes, distinguishing extents for separate sets of borrows that correspond to the same lifetime, e.g. when the same variable is assigned many times. The Polonius term for its notion of borrow extent is an *origin*; we use our terminology to remain generic over different borrow checkers' implementation choices.

```

1 fn dec_max(mut pos: Pos2D<i32>, mut z: i32) {
2     let max2 = if pos.x > pos.y { // c0/c1
3         &mut pos.x                // bb1 (c0)
4     } else {
5         &mut pos.y                // bb2 (c1)
6     };                            // bb3
7     let max3 = if z > *max2 {      // c2/c3
8         pos.x = 0;                // bb4 (c2) max2 borrow must end here!
9         &mut z
10    } else {
11        &mut *max2                 // bb5 (c3)
12    };
13    *max3 -= 1;                    // bb6
14 }

```

Fig. 2. An example designed to illustrate path-sensitivity with respect to (re)borrowing. The code is annotated with basic block and branch identifiers for use in Fig. 10.

the borrows at a given program point can depend on control flow. Rust analyses concerned with *side-effects* (e.g. verifiers such as Prusti [1] and Creusot [11]) require a model capturing flow-sensitive borrow information on top of the compiler’s more conservative, flow-insensitive checks.

The function `dec_max` given in Fig. 2 is designed to illustrate the differences between a flow-sensitive and flow-insensitive model of borrowing. At Line 2, `max2` borrows from a different field of `pos` depending on which branch of the `if` expression is taken. Following the join point at Line 6 `max2` is guaranteed to be initialised no matter which path was taken. Since its target `*max2` could borrow capabilities from either `pos.x` or `pos.y` the compiler will consider both `pos.x` and `pos.y` to be borrowed-from. In reality, `max2` borrows only from one of `pos.x` or `pos.y`, and an analysis tool such as a verifier may need to distinguish *which* and *when*. Borrows created *before* a control-flow branch may be forced to end during the branches themselves; it is even possible that this occurs in one branch and not the others. For example, at Line 8 the borrow stored in `max2` is forced to end in order to access `pos.x`, while the same borrow must be live in order to be reborrowed at Line 11.

Since code within branches affects constraints between borrows outside the branches, one might assume a precise path-sensitive analysis must analyse each path through a function separately. We will show that by careful design of our model, this is not the case. By exploiting the fact that the validity of all memory accesses depends only on the compiler’s path-insensitive analysis, we are able to efficiently combine our analysis state at each join point while retaining the ability to produce path-sensitive models of the aliasing relationships present in virtually all real-world Rust code.

## 2.4 Function Calls

To enable in-place mutation without sacrificing modularity, Rust allows functions to be parameterised by lifetimes which span their entire call. Based on the function signature, Rust computes constraints between the extents of borrows which are passed into the function and the extents of borrows which will be returned. From a caller’s perspective, this summary describes every relationship that can be known about the borrows returned by the function call. Conversely, the function body must ensure that it does not impose any additional constraints between its argument and return borrows. A model of Rust’s type checking should obey these same principles.

As an example, consider the program given in Fig. 3. The function `max` is parameterised by three named lifetimes (`'a`, `'b` and `'r`, for the extents of the two input parameters and output parameter,

```

1 fn max<'r, 'a: 'r, 'b: 'r>(rx: &'a mut i32, ry: &'b mut i32) -> &'r mut i32 {
2     if *rx > *ry { &mut *rx } else { &mut *ry }
3 }
4
5 fn dec_max_alt<'a>(pos: &'a mut Pos2D<i32>) {
6     let rx = &mut pos.x;
7     let ry = &mut pos.y;
8     let res = max(rx, ry);
9     *res -= 1;
10 }

```

Fig. 3. A function to illustrate modular type checking of calls concerning reborrowing.

respectively). The syntax `'a: 'r` declares an outlives constraint: the extent of the borrow stored in `rx` must last at least as long as the extent of the borrow returned by `max`. These constraints are sufficient to allow `rx` or `ry` to be reborrowed in the output of the function.

At `max`'s *call-site* on Line 8, the compiler sees from the outlives constraints in the function signature that the body was free to reborrow the output from either argument, so treats the returned reference as if it reborrowed from both. The returned borrow therefore gets an extent which ends after Line 9. At this point, both intermediate borrows `rx` and `ry` become usable again<sup>6</sup>.

Interactions between different Rust features make function calls involving lifetimes challenging to model. In general, (1) Rust types can use lifetimes as generic parameters: e.g. the type `&'a mut Pos2D<&'b mut i32>` denotes a mutable reference to a struct instance that itself stores references (whose extents may differ), (2) outlives constraints are often implicit (e.g. usage of a type `&'a mut Pos2D<&'b mut i32>` implicitly imposes `'b: 'a` to prevent live references from reaching expired borrows), (3) outlives constraints interact with *variance rules* for Rust types which themselves have lifetime parameters. We remark that modular summaries of calls must still connect with fine-grained reasoning about the borrows explicitly manipulated in the calling context.

## 2.5 Loops

Accounting for the borrowing that occurs across a loop raises further challenges. Consider the program given in Fig. 4. The function `penultimate_mut` returns a mutable reference to the second-to-last element of a list via the loop at Line 7, at each iteration borrowing `(*current).head` into `prev` and advancing `current` to point to the next node. Like the example in Fig. 2, the borrow stored in `prev` at Line 7 (if any) depends on which branch was taken. Since the number of loop iterations depends on the length of the list, the exact target of `prev` cannot be determined statically.

As with function calls, it is not possible to maintain precise borrowing information across zero or more iterations of a loop. Just as function signatures summarise the borrowing relationships across the call, modelling type checking across loops requires accurately summarizing the effect of arbitrarily many loop iterations: a loop invariant, for some suitable notion of the term.

## 2.6 Nested Lifetimes

Modelling borrows of types which in turn contain borrows raises many previously stated challenges. Consider the example given in Fig. 5. The first argument to `cl` is a borrow of the type `Pos2D<'a mut i32>`, a type that itself contains a *nested* borrow with lifetime `'a`. An analysis or verification

<sup>6</sup>Technically, both arguments to `max` are moved in and `rx/ry` should not be usable after the call. The compiler avoids this by moving in temporary reborrows of these `(&mut *rx/&mut *ry)` instead. We elide this here for simplicity, but our implementation handles this correctly.

```

1 struct List<T> { head: T, tail: Node<T> }
2 type Node<T> = Option<Box<List<T>>>;
3
4 fn penultimate_mut<'a>(list: &'a mut List<i32>) -> Option<&'a mut i32> {
5     let mut current = &mut *list;
6     let mut prev = None;
7     while let Some(next) = &mut (*current).tail {
8         prev = Some(&mut (*current).head);
9         current = &mut *next;
10    }
11    prev
12 }

```

Fig. 4. A function that takes a mutable reference to a list and returns a *reborrow* to its second-to-last element

```

1 fn replace_x_rf<'a, 'b>(rf: &'a mut Pos2D<&'b mut i32>, new_x: &'b mut i32) {
2     (*rf).x = new_x;
3 }
4
5 fn cl<'a>(r1: &'a mut i32, r2: &'a mut i32, r3: &'a mut i32, r4: &'a mut i32) {
6     let f = Pos2D { x: r1, y: r2 }; // f: Pos2D<&'a mut i32>
7     let rf = &mut f; // rf: &'a mut Pos2D<&'a mut i32>
8     (*rf).y = r3;
9     replace_x_rf(rf, r4);
10    *f.x = 0;
11 }

```

Fig. 5. Rust functions demonstrating nested borrows. The call to `replace_x_rf()` changes the target of `f.x`.

tool might need to know the set of memory locations which the write on Line 10 could have affected. Prior to the call, it is straightforward to identify that `(*rf).x` transitively borrows from the target of `r1` (the memory located at `*r1`). However, because `replace_x_ref` is allowed to change the targets of `*rf`, the target of `f.x` cannot be known precisely after the call without sacrificing modularity. A similar situation arises when types with nested lifetimes appear across loops.

This situation only arises for nested lifetimes: e.g. it is not possible for `replace_x_ref` to change the target of `rf` itself (which memory `rf` points to at the `cl` client side). Thus, our model must soundly over-approximate the set of capabilities borrowed from by types with nested lifetimes, while retaining more precise information for types without.

### 3 Place Capability Graphs

In this section, we introduce Place Capability Graphs (PCGs), whose main components are shown in Fig. 6, and demonstrate how they address the problems presented in Sec. 2. A PCG is a directed hypergraph over two kinds of nodes. *Place nodes* associate a Rust *place* (expression denoting a memory location [8]) with *capabilities* (Fig. 6a), describing the actions allowed on these places. A place node with *exclusive* (E) capability can be freely read from, written to, borrowed and moved. Read (R) capability allows reading or immutable borrowing only; write (W) capability allows assignment only. The  $\emptyset$  capability allows no actions (typically due to an outgoing PCG edge).

The second type of PCG nodes are *lifetime projection* nodes. Lifetime projections  $p \downarrow 'a$  (in which  $'a$  is a lifetime from the type of  $p$ ) are a novel construct used to represent a set of borrows currently stored in the place  $p$  and whose borrow extents are defined by the borrow checker's current interpretation of lifetime  $'a$  for  $p$ . For example, if a variable  $x$  has type  $\&'a \text{ mut String}$ , then  $x \downarrow 'a$  denotes the (singleton) set of borrows stored in  $x$ . Lifetime projection nodes will allow us to mediate between precise reasoning about individual places and borrows, and abstractions over (potentially unbounded) sets of borrows as considered by Rust's borrow checkers. Rust's flow sensitive type checking means that the values and sets of borrows considered stored in places depend on the program point: in general we allow our graph nodes to refer e.g. to places as they were at a previous program point by labelling them with a program point label.

*Definition 3.1 (PCG Preliminaries).* We assume a labelling of each (MIR) program point with a distinct *program point label*, ranged over by  $l$ . We use  $p$  to range over *places*.

A *generalised place*  $P$  is defined by the cases:  $P ::= p \mid p \text{ at } l \mid \text{origin}(i)$ ; in the second case we call  $P$  a *labelled place*; in the third ( $i$  is a non-negative integer) the node is called a *remote place*<sup>7</sup>.

*Capability types*  $c$  are defined by:  $c ::= E \mid W \mid R \mid \emptyset$  (called *exclusive*, *write*, *read* and *none*).

A *lifetime projection*  $lp$  is defined by the cases:  $lp ::= P \downarrow \tilde{r} \mid P \downarrow \tilde{r} \text{ at } l \mid P \downarrow \tilde{r} \text{ at FUT}$

We assume an infinite set  $N$  of *node identifiers* (*node IDs*) ranged over by  $\alpha, \beta, \gamma$ , etc., and a (partitioning) predicate *placeNode* on node IDs, which is true (resp. false) for an infinite subset of  $N$ . When *placeNode*( $n$ ) is true,  $n$  is a *place node ID*; otherwise,  $n$  is a *lifetime projection node ID*.

Our PCG edges represent a variety of key notions arising in Rust's design. Edges from place nodes reflect a current flow of capabilities (that removing the edge would undo) from parent(s) to child(ren): the *leaf* place nodes of our PCGs are typically those with capabilities. From lifetime projection nodes, edges reflect instead where (sets of) borrows might flow to and be used.

*Definition 3.2 (PCG Edges).* A *PCG edge* is a tuple  $(t, S_0, S_1)$ , in which  $S_0, S_1$  are non-empty disjoint sets of node IDs, and  $t$  is an *edge type*: a label from one of the following cases (names in parentheses):  $t ::= b$  (*borrow*)  $\mid u$  (*unpack*)  $\mid f$  (*borrow-flow*)  $\mid a$  (*abstraction*)  $\mid l$  (*alias*)  $\mid d$  (*deref*).

The first set  $S_0$  are the *sources* of the edge; the second  $S_1$  are its *targets*; in general, PCG edges are *hyper-edges* (many-to-many). However, each node type comes with specific requirements on the arities and contents of the two node ID sets (place node IDs vs. lifetime project node IDs), as summarised by the following edge signatures (where  $P$  represents that a place node ID is expected;  $L$  that a lifetime projection node ID is expected;  $\rightarrow$  denotes zero or more of the element):  $(b, \{P\}, \{L\})$ ,  $(u, \{P\}, \{\vec{P}\})$ ,  $(f, \{L\}, \{L\})$ ,  $(a, \{\vec{L}\}, \{\vec{L}\})$ ,  $(l, \{P\}, \{P\})$ ,  $(d, \{P, L\}, \{P\})$ .

We define our graphs over a notion of node identifiers via several indirections, to simplify the presentation of operations which change node content but not graph structure.

*Definition 3.3 (Place Capability Graphs).* A *Place Capability Graph (PCG)* is defined by the six elements  $(nodes(G), edges(G), places(G), caps(G), lfProjs(G), branches(G))$ , where:

- $nodes(G)$  is a finite set of node identifiers;
- $edges(G)$  is a finite set of PCG edges, whose sources and targets are all in  $nodes(G)$ ;
- $places(G)$  is an injective map from the place node IDs in  $nodes(G)$  to generalised places;
- $caps(G)$  is a map from the image of  $places(G)$  to capabilities;
- $lfProjs(G)$  is an injective map from the lifetime projection node IDs in  $nodes(G)$  to lifetime projections; and
- $branches(G)$  is a map from  $edges(G)$  to a set of *branch-choice identifiers*; (cf. Sec. 3.5).

<sup>7</sup>Remote places are used to represent the borrows stored in original arguments of the function under analysis.

For node IDs  $m, n$ , we say  $m$  is *parent of  $n$*  in  $G$  iff, for some edge  $(t, S_0, S_1) \in \text{edges}(G)$ ,  $m \in S_0$  and  $n \in S_1$ . We derive the corresponding notions of *child* nodes, *reachability* (via children), and (hyper-)DAG accordingly. The *roots of  $G$* ,  $\text{roots}(G) \subseteq \text{nodes}(G)$  are its nodes without parents; dually the *leaves of  $G$* , written  $\text{leaves}(G)$  have no child nodes.

Our PCGs are acyclic (hyper-DAGs)<sup>8</sup>; their edges represent the flows of capabilities (to places/borrows) and borrows (into sets of borrows). The majority of our rules for constructing PCGs operate on a small handful of leaf nodes and their parents, leaving the rest of the graph unchanged. To simplify expressing these rules, we introduce a notion of *PCG context*, which can be thought of as a graph with some dangling downward (child) edges, to which a further graph can be connected:

*Definition 3.4 (PCG Contexts).* We define (PCG) Contexts  $C$  analogously to PCGs except for:

- An additional element  $\text{holes}(C)$  defines a subset of  $\text{leaves}(C)$ .
- The domains of  $\text{places}(C)$  and  $\text{lfProjs}(C)$  are restricted to only the nodes *not* in  $\text{holes}(C)$ .

For a context  $C$  and PCG  $G$ , the *composition of  $C$  with  $G$*  is a partial operation defining a PCG, written  $C[G]$ ; it is defined only when:

- $(\text{nodes}(C) \setminus \text{holes}(C)) \cap \text{nodes}(G) = \emptyset$  (no non-hole node in common).
- $\text{holes}(C) \subseteq \text{roots}(G)$
- The codomains of  $\text{places}(C)$  (resp.  $\text{lfProjs}(C)$ ) and  $\text{places}(G)$  (resp.  $\text{lfProjs}(G)$ ) are disjoint (and so these pairs of injective maps can be combined to an injective map).

When these conditions are met,  $C[G]$  is the PCG defined by taking the union of all sets and maps constituting  $C$  and  $G$ , ignoring the  $\text{holes}(C)$  set.

For a pair of contexts  $C$  and  $C'$ , the *context composition of  $C$  with  $C'$* , is a partial operation defining a PCG context, written  $C[C']$ , defined entirely analogously to the definition for  $C[G]$  above. When defined, the resulting context has the same holes as  $C'$ , i.e.  $\text{holes}(C[C']) = \text{holes}(C')$ . A *node content substitution*  $\sigma$  is a pair of maps: one from generalised places to generalised places, and one from lifetime projections to lifetime projections; we write e.g.  $\sigma(P)$  for the application of the appropriate map. We lift this definition pointwise to a graph or a context in the natural way, defining  $\sigma(G)$  to have the changed maps  $\text{places}(\sigma(G))(\alpha) = \sigma(\text{places}(G)(\alpha))$  and  $\text{lfProjs}(\sigma(G))(\alpha) = \sigma(\text{lfProjs}(G)(\alpha))$ . The operation  $\sigma(G)$  is defined only if  $\text{places}(\sigma(G))$  and  $\text{lfProjs}(\sigma(G))$  remain injective.

We use the following compact notations to describe the rules of our PCG analysis.

*Definition 3.5 (Graph and Rule Notations).* In a given graph  $G$ , we write  $\langle\langle P : c \rangle\rangle_\alpha$  to represent a place node: a place node ID  $\alpha \in \text{nodes}(G)$ , for which  $\text{places}(G)(\alpha) = P$  and  $\text{caps}(G)(P) = c$ . Similarly, we write  $\langle\langle lp \rangle\rangle_\alpha$  to represent a lifetime projection node: expressing  $\alpha \in \text{nodes}(G)$  and  $\text{lfProjs}(G) = lp$ . We will omit node IDs  $\alpha$  when it aids clarity.

We write  $S_0 \xrightarrow{t} S_1$  for an edge  $(t, S_0, S_1)$ , comma-separating this notation to describe multiple edges. We will present the nodes in  $S_0, S_1$  either with the notation above or only with their node IDs (and may also elide explicit set parentheses around node sets), as aids presentation. For example,  $\{\langle\langle P_1 : c_1 \rangle\rangle\} \xrightarrow{t} \{\langle\langle P_2 : c_2 \rangle\rangle_{\alpha_2}, \{\alpha_2\} \xrightarrow{t} \{\langle\langle P_3 : c_3 \rangle\rangle\}$  denotes a linked-list-like graph with three nodes and two edges; the explicit node ID  $\alpha_2$  captures the same node occurring in both edges.

Given a Rust (MIR) statement  $s$ , we denote a *PCG statement rule for  $s$*  with the syntax  $G_0 \xRightarrow{l:s} G_1$  in which  $l$  is the program point of the statement (often omitted for simplicity). We call  $G_0$  the *pre-state PCG* and  $G_1$  the *post-state PCG*. As is standard e.g., for Hoare Logic rules, we define PCG rules as rule *schemas*, in which meta-variables (e.g., for places, program points or portions of a

<sup>8</sup>Technically acyclicity can be violated due to our superimposed representations of path-sensitive graphs explained in Sec. 3.5; even so, acyclicity remains true when considering the edges corresponding to any single path through the program.

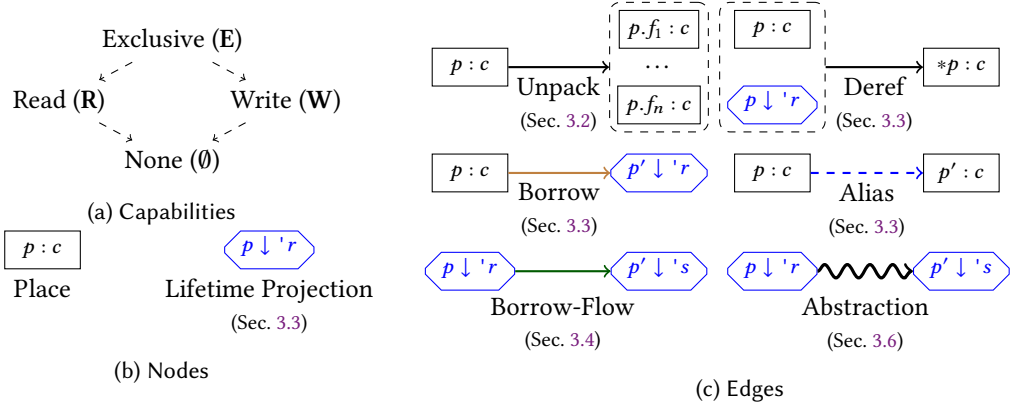


Fig. 6. The PCG capabilities, components of the graph and their associated permitted behaviours. Capabilities (a) describe the actions that the Rust compiler permits on a value via a particular place. PCG nodes (b) denote memory locations: either places, or *lifetime projections* which describe a region of memory that the borrow checker permits access to. Edges (c) in the PCG describe capability transfers, aliasing, and the flow of borrows.

PCG) can be instantiated to obtain specific rule instances. We denote a *PCG operation rule* with the analogous syntax  $G_0 \xRightarrow{l \text{ op}} G_1$ , in which  $l \text{ op}$  is a description of a PCG operation performed *between* program statements by our analysis; their definition and role will be explained shortly.

To reduce notational clutter further, we also adopt the following conventions for defining rules: when we write a statement rule (and analogously for operation rules) in the form  $C_0[G] \xRightarrow{s} C'_0[G']$  where  $C_0, C'_0$  are meta-variables, while  $G, G'$  constitute explicit notation for nodes and edges, we imply that the nodes and edges of  $G$  and  $G'$  are exactly those notated. Further, when it is clear which nodes in  $G$  and  $G'$  correspond to one another, we will omit their IDs; for nodes that are clearly fresh in  $G'$  (did not occur correspondingly in  $G$ ) we assume them to have fresh node IDs.

Some of our analysis rules do not apply only at the leaves of a PCG; to specify these rules locally we need to allow for extra graph portions may appear *both* above and below. When we write a rule in the form  $C_0[C[G_0]] \xRightarrow{s} C'_0[C'[G'_0]]$  where  $C_0, G_0, C'_0, G'_0$  are meta-variables, while  $C, C'$  use explicit notation for nodes and edges, we imply that the nodes and edges of  $C$  and  $C'$  are exactly those notated; those which are holes of  $C$  (resp.  $C'$ ) are notated with node content (places and capabilities, or lifetime projections) actually determined by the mappings defined in  $G_0$  (resp.  $G'_0$ ).

### 3.1 PCG Analysis Workflow

Our PCG analysis requires as input a MIR-representation of a type-checked Rust program (for which we can access the results of the Rust compiler's analyses, especially borrow checking), in which all program points have been assigned distinct labels  $l$ , and all branches and corresponding joins, as well as loop-heads and corresponding back-edges have been identified.

Our analysis computes a PCG for every program point by forward analysis; the initial PCG used comprises, for each function argument  $p$ : (a) a place node ( $\langle p : E \rangle$ ), and (b) for each lifetime  $'a$  in the type of  $p$ , a lifetime projection node ( $\langle p \downarrow 'a \rangle$ ). Within a (MIR) CFG basic block, each statement is handled by its corresponding PCG analysis rule. Before doing so, PCG operations are performed (a) to remove edges that concern borrow extents that the borrow checker deems to have ended, and (b) to arrange the graph to satisfy the precondition of the analysis rule in question.

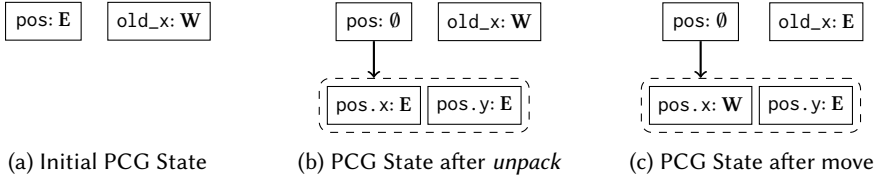


Fig. 7. PCGs at three points in Fig. 1a; Fig. 7a is the PCG at the outset of `replace_x_owned`, Fig. 7b is the PCG after `pos` is *unpacked*, and Fig. 7c is the PCG after the move assignment on Line 5.

Non-initial basic blocks are analysed starting from a PCG computed as the *join* (cf. Sec. 3.5) of the PCG graphs found for the final states of its direct ancestor blocks; our analysis follows a reverse postorder traversal (excluding loop back-edges) of the CFG blocks. For loops, the analysis is more-complex (but avoids an iterative fixpoint computation): we describe our algorithm in Sec. 3.7.

Overall, our PCG analysis results in a PCG for each program point in the body of the function. Additionally, it returns a list of *annotations* for each program point that reflect the operations it performed at that point, including those performed to remove edges when necessary.

### 3.2 Fine-Grained Place Accounting with Unpack Edges

As motivated in Sec. 2.1, Rust’s type checking decomposes capabilities for composite types into field-specific capabilities (and the reverse); consumer tools typically need to manage their representations for these (potentially recursive) types with similar operations. For these reasons, our model defines explicit *unpack* and *pack* PCG operations, which add (resp. remove) *unpack edges* that represent a decomposition of capabilities. In this subsection, we present simplified versions of our rules that are nonetheless equivalent when applied to composite types that do not contain borrows. In Sec. 3.4, we explain how our rules are extended to support general types.

Fig. 1a (page 4) shows an example in which unpacking is needed. The source-level assignment statement `let old = pos.x` will, in MIR, become a move assignment `old = move pos.x`. The PCG analysis rule for move assignments  $p = \text{move } q$ , in the case where  $q$  does not contain any borrows, is as follows:  $C[(p : W), (q : E)] \xrightarrow{p = \text{move } q} C[(p : E), (q : W)]$

This rule requires a pre-state PCG with leaf place nodes for the moved-in ( $p$ ) and moved-out ( $q$ ) places with capabilities  $W$  and  $E$ , respectively; in the post-state PCG the capabilities are exchanged. In our example, this rule cannot be applied to the initial graph in Fig. 7a, as it has no (leaf) place node for  $\text{pos.x} : E$ . A PCG operation must first be applied to rearrange the graph: in this case an *unpack*, defined by the rule schema:  $C[(p : E)] \xrightarrow{\text{unpack } p} C[(p : \emptyset) \xrightarrow{u} \overline{(p.f_i : E)}]$  in which  $p.f_i$  ranges over the immediate projections (e.g. fields of a struct) of  $p$ . Instantiated for our example (cf. Fig. 7a which is transformed into Fig. 7b), the corresponding rule instance is  $C[(\text{pos} : E)] \xrightarrow{\text{unpack } \text{pos}} C[(\text{pos} : \emptyset) \xrightarrow{u} \{(\text{pos.x} : E), (\text{pos.y} : E)\}]$  in which  $C$  is a context with no holes or edges and a single node (`old_x : W`). After this operation, the resulting PCG (shown in Fig. 7b) is suitable as the precondition of the move assignment rule. Following the move assignment, the PCG is as shown in Fig. 7c (our analysis proceeds with the next statement).

The inverse *pack* PCG operation is given by  $C[(p : \emptyset) \xrightarrow{u} \overline{(p.f_i : E)}] \xrightarrow{\text{pack } p} C[(p : E)]$  in which  $p.f_i$  again ranges over the immediate projections of  $p$ . Note that the PCG in Fig. 7c is not one in which this PCG operation could be applied; the exclusive capability for  $\text{pos.x}$  is missing. This reflects that Rust would not allow `list` to be moved, borrowed or dropped in this state. Our PCG analysis applies *pack* or *unpack* operations as needed between statements, favouring packing

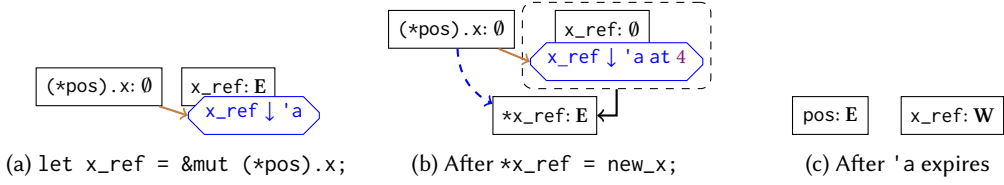


Fig. 8. Partial PCGs at three points in Fig. 1b. (a) the result of the mutable borrow on Line 3, (b) the result of the assignment on Line 4, and (c) the PCG after the mutable borrow expires and pos is packed up.

whenever possible (keeping the graphs smaller). We record which PCG operations are applied where, so that downstream consumers can drive their own analyses with analogous steps.

### 3.3 Tracking Borrows with Borrow Edges

As explained in Sec. 2.2, Rust's borrowed references are used to temporarily transfer capabilities (and later return them), enabling controlled aliasing and in-place mutation. The example in Fig. 1b shows the creation of such a borrow at Line 3. The PCGs for this example are shown in Fig. 8, where a *borrow edge* records the borrow of  $(*pos).x$  into the *lifetime projection*  $x\_ref \downarrow 'b$ <sup>9</sup>.

Lifetime projection nodes model a set (in this case a singleton) of borrows potentially stored in  $x\_ref$  whose extents have not yet been ended (according to the borrow checker). Our PCGs satisfy the invariant that, whenever a place node  $\langle p : c \rangle$  occurs such that  $c$  is either **R** or **E**, a corresponding lifetime projection node  $\langle p \downarrow 'a \rangle$  will occur for each lifetime 'a in the type of  $p$ , i.e. whenever we can read from a place, we also track the sets of borrows potentially stored in it.

The *borrow edge*  $\langle (*pos).x : E \rangle \xrightarrow{b} \langle x\_ref \downarrow 'a \rangle$  records the outstanding borrow of  $(*pos).x$  in the set of borrows represented by  $x\_ref \downarrow 'b$ : concretely, the borrowed memory is aliased by the place  $*x\_ref$ . This edge reflects *why*  $(*pos).x$  doesn't (currently) have its default capabilities.

As in the previous subsection, we present a simplified version of the rules that applies when borrows are not stored in composite data types or inside other borrows. We explain the full rules in Sec. 3.4. Our PCG analysis rule for creation of a mutable borrow via  $p = \&mut q$  is defined as:

$C[\langle p : W \rangle, \langle q : E \rangle] \xRightarrow{l: p = \&mut q} \sigma(C)[\langle p : E \rangle, \langle q : \emptyset \rangle \xrightarrow{b} \langle p \downarrow 'a \rangle]$  and the corresponding rule

for shared borrows is:  $C[\langle p : W \rangle, \langle q : R \rangle] \xRightarrow{l: p = \&q} \sigma(C)[\langle p : E \rangle, \langle q : R \rangle \xrightarrow{b} \langle p \downarrow 'a \rangle]$  where

'a is the lifetime from  $p$ 's type, and  $\sigma$  is a substitution defined to replace (unlabelled) places  $p'$  with corresponding labelled places  $p'$  at  $l$  exactly when  $*p$  is a prefix of  $p'$ , and analogously for the places in lifetime projection nodes. This substitution is needed in case  $*p$  had previously been *reborrowed*, and nodes referring to its prior value existed elsewhere in the graph.

In this simple example, our lifetime projection concept may seem overly complex given the (single) created borrow, the usage of these uniformly for any types storing borrows is what enables the generality of our model, and its handling in particular of general function calls and loops.

The rule for the assignment  $*x\_ref = new\_x$  on Line 4 requires  $*x\_ref$  to be present in the PCG with capability **W**; this capability is conceptually buried in the **E** capability for  $x\_ref$ . Our PCG operation *deref* for dereferencing a borrow-typed place, is similar to our unpack operation, but we design the rule to reflect that dereferences in Rust require *a justification from the borrow checking of the program* that the corresponding borrow extent has not ended. This dependency is captured with a lifetime projection node:  $C[\langle p : E \rangle, \langle p \downarrow 'a \rangle] \xRightarrow{l: deref p} C[\langle p : W \rangle, \langle p \downarrow 'a \text{ at } l \rangle] \xrightarrow{d} \langle *p : E \rangle]$

<sup>9</sup>To help match place nodes with corresponding lifetime projection nodes, we typically draw example diagrams with the latter overlaid under the former. Note however, that these are distinct graph nodes, with no edges between them.

In the above rule,  $\langle\langle p \downarrow 'a' \text{ at } l \rangle\rangle$  is a *labelled* lifetime projection, used to record a set of borrows that *used to* be associated with a place node at a prior program point; we label lifetime projection nodes whenever their place nodes no longer have at least read capability. This idea will be used and extended when we explain our handling of nested references in Sec. 3.4.

As a convenience (useful for some analysis applications), our PCG analysis also introduces *alias* edges whenever a pair of place nodes are available, one of which is borrowed, and one of which must be the dereference of the corresponding borrow. In general, this can only be known either for types that only store single borrows (references to types that don't contain lifetimes), or because the borrow itself was created explicitly in the function under analysis and can still (possibly via path-sensitivity) be tracked precisely. For example, in Fig. 8, our analysis adds the alias edge  $\langle\langle (*pos).x : \emptyset \rangle\rangle \xrightarrow{l} \langle\langle *x\_ref : E \rangle\rangle$  to reflect that  $*x\_ref$  is a definite alias of  $(*pos).x$ . Our alias edges are optional: they are never required or conditioned upon in our rule definitions. To simplify our presentation, we will omit them from the presentation from here onwards.

Returning to our example, our analysis next performs a PCG pack operation, specialised to reference types:  $C \left[ \{ \langle\langle p : W \rangle\rangle, \langle\langle p \downarrow 'a' \text{ at } l \rangle\rangle \} \xrightarrow{d} \langle\langle *p : E \rangle\rangle \right] \xrightarrow{\text{pack } p} C[\langle\langle p : E \rangle\rangle, \langle\langle p \downarrow 'a \rangle\rangle]$ .

At the last line of our example, the compiler prescribes that the borrow extent associated with  $x\_ref$  ends: conceptually, all PCG edges reflecting capabilities transferred by these borrows should be removed. Our analysis performs an iterative cleanup of the PCG, finding edges whose children are all leaves, and which are either unpack edges that can be packed, or edges whose children include either lifetime projection nodes for ended borrow extents or place nodes for places whose types correspond to ended borrow extents. The latter is partially handled by a PCG operation for expiring a borrow:  $C[\langle\langle p : E \rangle\rangle, \langle\langle q : \emptyset \rangle\rangle \xrightarrow{b} \langle\langle p \downarrow 'a \rangle\rangle] \xrightarrow{\text{expire } p \downarrow 'a} C[\langle\langle p : W \rangle\rangle, \langle\langle q : E \rangle\rangle]$  Applying this rule in our example,  $x\_ref$ 's capability will become  $W$ , reflecting that it can be reassigned but no longer read from (as the borrow it stored has expired).

### 3.4 Borrows Inside Composite Data Types

So far we have ignored the possibility of storing borrows inside composite data types, as well the possibility of *nested* borrows, i.e. creating references to places that themselves contain borrows (e.g. as in Fig. 5). We now describe how we systematically extend our rules (including those already presented) to handle these issues in two stages: first for code that stores borrows inside structs but does not nest borrows, and then finally the full rules that also account for nested borrows.

Because a composite datatype such as `List<mut i32>` can hold an arbitrary number of borrows, reflecting Rust's rules for tracking borrows requires *sets* of borrows that cannot (always) be enumerated. Our *lifetime projections* represent such sets: we explain next how to adapt our PCG rules to also account for the lifetime projections associated with a type containing lifetimes.

Two general principles underly our adaptations. Firstly, whenever a rule causes a place node to go from having a read/exclusive capability to no longer having one, its corresponding lifetime projection nodes are no longer meaningful as *current* representations of sets of borrows (they describe borrows stored in a place that cannot be accessed): we replace them with labelled versions that refer to the program point before this change. Secondly, when the place's capabilities have flowed elsewhere (e.g. via a move or unpack), we should reflect that the borrows may correspondingly also now be accessed by new/different places. This latter idea is reflected by *borrow flow* edges, which connect lifetime projection nodes to indicate where a set of borrows may have derived its borrows from. For example, when unpacking a struct we adapt the rule presented on page 12 to be:

$$C \left[ \langle\langle p : E \rangle\rangle, \overrightarrow{\langle\langle p \downarrow 'a_i \rangle\rangle} \right] \xrightarrow{l: \text{unpack } p} C \left[ \langle\langle p : \emptyset \rangle\rangle \xrightarrow{u} \langle\langle p.f_j : E \rangle\rangle, \overrightarrow{\langle\langle p \text{ at } l \downarrow 'a_i \text{ at } l \rangle\rangle \xrightarrow{f} \langle\langle p.f_j \downarrow 'a_i \rangle\rangle} \right]$$

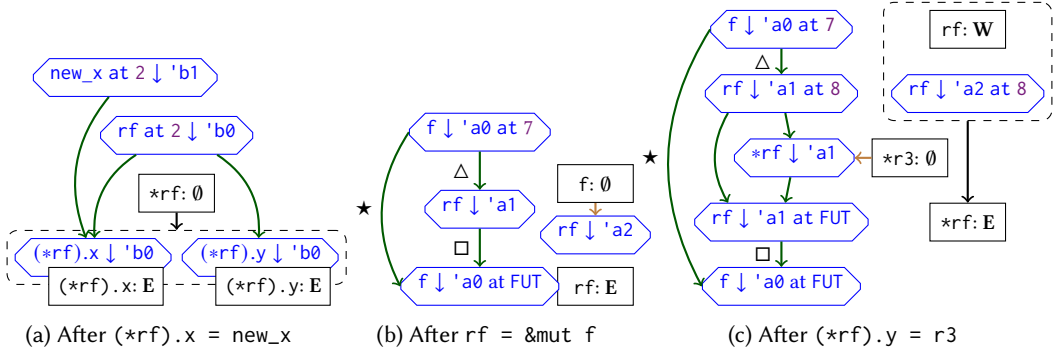


Fig. 9. Partial PCGs at three points in Fig. 5. Fig. 9a shows the result of unpacking a struct containing borrows, as is done at Line 2. Fig. 9b shows the result of the assignment on Line 7; the PCG analysis introduces a future lifetime projection  $f \downarrow 'a0$  at FUT and adds edges to it from  $f$ 's labelled lifetime projection  $f \downarrow 'a0$  at 7 ( $\square$ ) and the current lifetime projection  $rf \downarrow 'c1$  ( $\star$ ). Fig. 9c shows the state after the assignment  $(\ast rf).y = r3$ : the target of  $\Delta$  and the source of  $\square$  are updated to newly introduced labelled and future projections.

where  $'a_i$  ranges over the lifetimes in  $p$ ,  $p.f_j$  ranges over the immediate projections (fields) of  $p$ , and borrow flow edges are added for those  $p.f_j$  whose types contain a lifetime corresponding to  $'a_i$  (observe that in the case of a struct whose field types contain no lifetimes, we have our simplified rule). The result of applying this rule to unpack  $rf$  at Line 2 of Fig. 5 is given in Fig. 9a. Our earlier rule for moves is analogously extended to become:

$$C \left[ \frac{\langle p : W \rangle, \langle q : E \rangle, \langle q \downarrow 'a_i^q \rangle}{\langle q \downarrow 'a_i^p \rangle} \right] \xrightarrow{l: p = \text{move } q} \sigma(C) \left[ \frac{\langle p : E \rangle, \langle q : W \rangle, \langle q \text{ at } l \downarrow 'a_i^q \text{ at } l \rangle \xrightarrow{f} \langle p \downarrow 'a_i^p \rangle}{\langle q \downarrow 'a_i^p \rangle} \right]$$

in which  $'a_i^p$  denotes the  $i$ 'th lifetime in the type of  $p$ ,  $'a_i^q$  denotes the corresponding lifetime from  $q$ , and  $\sigma$  extends the substitution given for the borrow rules (page 13) such that  $\sigma(q.* \downarrow 'r) = q.* \text{ at } l \downarrow 'r \text{ at } l$  where  $q.*$  is any projection of  $q$ .

We now turn our attention to the final case we have postponed: nested borrows. Without the ability to nest borrows, the sets of borrows stored in a place  $p$  can only be modified by operations on  $p$  itself. However, in the presence of nested borrows, the set of borrows stored in a place  $p$  can be modified via *references* to (parts of)  $p$ . To support nested borrows, we must update our rules to account for changes in the sets of borrows denoted by our lifetime projection nodes in such cases.

We solve this issue using a special kind of placeholder node containing a *future lifetime projection*, denoted  $p \downarrow 'a$  at FUT. These nodes are used to track the borrow flow edges that *will* need to be incoming to a lifetime projection node  $p \downarrow 'a$  by the time it next becomes available (when  $p$  itself becomes a place node with read/exclusive capability). We employ such future lifetime projection nodes for each lifetime that occurs nested in the type of a borrow of  $p$ , for as long as  $p$  itself remains borrowed; once it becomes available, its lifetime projection nodes take the place of the placeholders.

To maintain appropriate borrow-flow edges into these future nodes, we systematically adapt each of our rules to: (1) introduce new future nodes and initialise their edges, (2) update edges into these future nodes as further borrows/moves occur that could contribute to their eventual sets, (3) replace the future nodes with regular lifetime projection nodes when the borrows end.

For point (1), whenever a place  $p$  is borrowed, we label its previous lifetime projection nodes (if any), and add fresh future nodes for each. To reflect the possible sources of borrows eventually stored in these new nodes, we add two edges into each: one from the labelled previous node, and one from the corresponding lifetime projection node in the newly-created borrow. For example, in

Fig. 9b, when the borrow  $\text{rf} = \&\text{mut } f$  is created, the PCG analysis introduces a future lifetime projection  $f \downarrow 'a0$  at FUT to represent borrows that will be stored in  $f$  under lifetime  $'a0$  once the borrow in  $\text{rf}$  expires, and connects this node to two newly introduced edges ( $\star, \square$ ).

For point (2), whenever a place  $q$  is moved or borrowed into place  $p$ , if  $q$ 's lifetime projections previously had outgoing edges to future nodes (say, for a place  $r$ ), these future nodes get updated edges from  $p$ 's corresponding lifetime projections, reflecting that  $p$  may now be used to change the stored borrows. When  $q$  is *borrowed* by  $p$ , these new dependencies are added *via* the new future nodes created for  $q$ , as  $q$  may become usable again before we unwind to  $r$ . This is shown in Fig. 9c: after  $(\star \text{rf}).y = r3$ , the target of  $\triangle$  and the source of  $\square$  are changed from  $\text{rf} \downarrow 'a1$  to the newly-introduced lifetime projections  $\text{rf} \downarrow 'a1$  at 8 and  $\text{rf} \downarrow 'a1$  at FUT respectively.

Finally, for point (3), when a place node newly becomes a leaf (with read/exclusive capability), any future nodes for its lifetime projections are replaced by current (unlabelled) lifetime projections: this is the moment that the new sets of borrows become available via the place once more.

Based on these principles, we extend our rule for creating borrows as follows:

$$C \left[ \begin{array}{c} \frac{\langle\langle p: W \rangle\rangle, \langle\langle q: E \rangle\rangle}{\langle\langle q \downarrow 'a_i \rangle\rangle_{\alpha_i}} \\ \alpha_i \xrightarrow{f} \delta_{ij} \end{array} \right] [G] \xRightarrow{p = \&\text{mut } q} \sigma(C) \left[ \begin{array}{c} \frac{\langle\langle p: E \rangle\rangle, \langle\langle q: \emptyset \rangle\rangle \xrightarrow{b} \langle\langle p \downarrow 'b \rangle\rangle,}{\langle\langle q \downarrow 'a_i \text{ at } l \rangle\rangle_{\alpha_i} \xrightarrow{f} \langle\langle p \downarrow 'b_i \rangle\rangle_{\beta_i}},} \\ \alpha_i \xrightarrow{f} \langle\langle q \downarrow 'a_i \text{ at FUT} \rangle\rangle_{\gamma_i}, \\ \beta_i \xrightarrow{f} \gamma_i, \gamma_i \xrightarrow{f} \delta_{ij} \end{array} \right] [\sigma(G)]$$

where  $a_i$  ranges over the lifetimes in  $q$  and  $b_i$  is the lifetime corresponding to  $a_i$  in  $p$ .  $\delta_{ij}$  is the target of  $j$ 'th edge originating from the  $i$ 'th lifetime projection of  $q$  (such a node is guaranteed to be a future lifetime projection). We extend the rules for moves, packs, and unpacks analogously.

### 3.5 Computing Joins and Tracking Path-Sensitive Borrows

When the control flow of a program branches, PCGs for each branch are computed separately. These results are *joined* to a single graph from which our analysis continues as usual. Our join is designed to retain relevant path-sensitive information (in particular concerning differences in borrow flow in the branches) while minimising duplication of both work and graph (sub-)structure.

Our join for place nodes is straightforward: only those which are leaves in both branches make up the place node leaves of the joined graph; this reflects the fact that Rust's rules for which places can be accessed are not path-sensitive. Although which places are considered borrowed is also not path-sensitive, *how and why* they are borrowed may depend on the path taken; as motivated in Sec. 2.3 this information is relevant for reasoning about the borrows and the aliases and side-effects they induce. We describe next how our PCGs track this path-sensitive information precisely and efficiently (by sharing maximal graph structure), using Fig. 10 as a demonstrative example.

Our analysis represents path-sensitive information by associating each choice at each branch in the CFG with a unique *branch-choice identifier*. Our analysis tracks the sequences of branch-choices taken to reach the current basic block; when a new branch is taken, each pre-existing edge is labelled with its identifier; when an edge is created, it is labelled with the tracked set.

At a join point, the edges from all joined graphs are included (retaining any labels) and edges shared between joined PCGs have their sets of labels *merged* (unioned); whenever this results in a *complete* set of branch-choice identifiers for any given branch point, these are removed (the edge exists for each choice at this branch point). As an example, the graph in Fig. 10c after the join point in bb3 retains edges from both branches (labelled  $c_0$  and  $c_1$ ). Edges which existed before the branch point and *survive* in either graph will be preserved without duplication: the graph for bb3 is not duplicated in the final result. As a result, a PCG can be seen as the superimposition of a set of

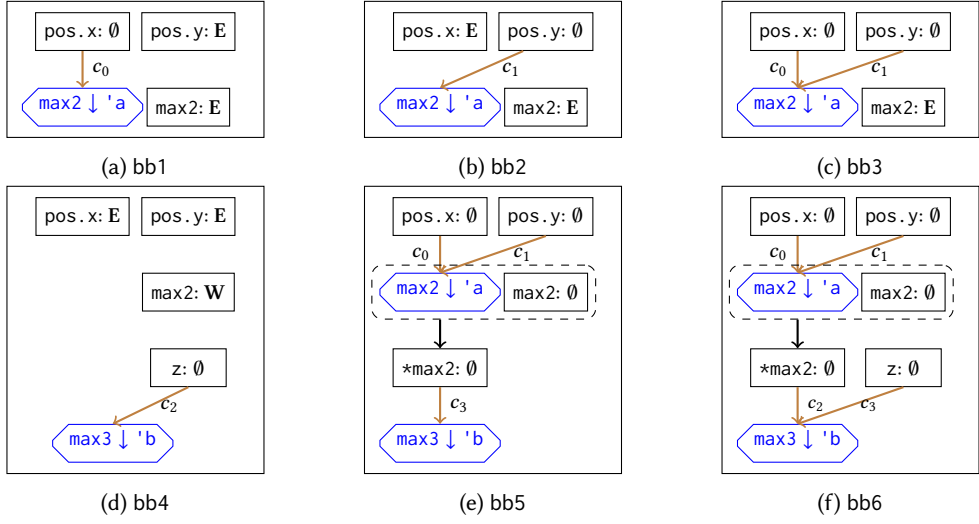


Fig. 10. Example demonstrating joins in the PCG for Fig. 2: the state at bb3 is computed by joining the states at bb1 and bb2 (where  $c_0$  and  $c_1$  are the respective branch-choice identifiers), and bb6 is the join of the states at bb4 and bb6 (where  $c_2$  and  $c_3$  are the respective branch-choice identifiers).

DAGs, one for each path through the CFG, with maximal sharing between each. Given a set of branch-choice identifiers describing a particular path through the program, one can recover the DAG for that path by considering only edges labelled with superset of the path's identifiers.

### 3.6 Tracking Reborrowing Across Function Boundaries with Abstraction Edges

Calling a Rust function can cause borrows to be created, ended and returned (via different places) to a caller (cf. Sec. 2.4). To support function-modular Rust analyses, our PCG analysis uses *abstraction edges* to capture, as precisely as is guaranteed by the function's *signature* (and Rust's type-checking rules), the borrow flows and corresponding aliasing potentially resulting from a function call. Abstraction edges are conceptually similar to borrow-flow edges, but are used to represent borrow flow that could not be (modularly) precisely-tracked: both for function calls and for loops (cf. Sec. 3.7).

In the MIR, a function call is a statement of the form  $p = f(\text{operands})$ , where *operands* is a list of places that are either moved out or copied, and  $p$  is the place where the result of the function will be stored. From the caller's perspective, each place in the function arguments either is unchanged or becomes inaccessible; as a result, the caller cannot observe any changes that the function makes to the *owned* parts of these places. However, the function can make observable effects via the *borrows* stored within the operands (a) by returning a value that contains *reborrows* taken from borrows in the operands, and (b) by changing the sets of *nested* borrows passed in the operands.

We model the first case by introducing abstraction edges between lifetime projections in the operands and those in the result place: each lifetime projection in the operands is connected to the corresponding lifetime projections that it *outlives* in the result place. The (compiler-checked) outlives constraint captures whether borrows *could* be assigned in this way, according to the type system. Fig. 11 illustrates the abstraction edges which (modularly) model a call to the `max()` function: based on the types of `res`, `rx`, and `ry`, `res` may borrow from either variable.

The second case is closely related to Sec. 3.4: changes to a set of nested borrows may become relevant through a previously-borrowed place: in this case, future lifetime projections will be

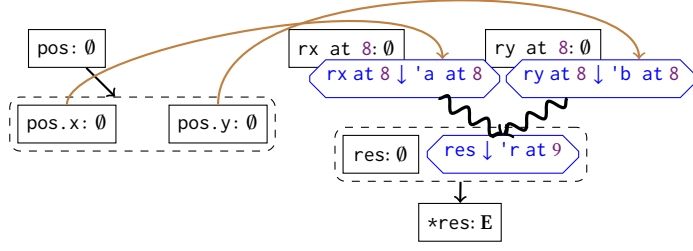


Fig. 11. PCG generated for the last line in Fig. 3, including the abstraction edges added for the call to max.

dependent on these sets before the call. As explained (Sec. 3.4), changes to these sets of borrows made by the call must be reflected onto these future nodes. In particular, changes made during a function call might necessitate additional dependencies on function arguments (cf. Fig. 5).

To directly model the potentially *changed* sets of borrows relevant to these concerns, our analysis of function calls introduces lifetime projections to represent the *post-state* of each lifetime projection in the operands. Each lifetime projection in the operands is connected with abstraction edges to its corresponding post-state projection as well as the post-state *nested* lifetime projections that it outlives (analogously to sets of borrows explicitly returned). In addition, any edges originating from lifetime projections in the arguments of the function to future nodes are redirected to originate from the corresponding post-state lifetime projection instead.

In summary, we define the PCG analysis rule for function calls as follows:

$$C \left[ \frac{\frac{\langle p : \mathbf{W} \rangle, \langle q_i : \mathbf{E} \rangle}{\langle q_i \downarrow 'r_j \rangle_{\alpha_{ij}}}}{\alpha_{ij} \xrightarrow{f} \delta_{ijm}} \right] [G] \xRightarrow{l:p=f(\vec{q_i})} C \left[ \frac{\frac{\langle p : \mathbf{E} \rangle, \langle q_i : \mathbf{W} \rangle, \langle q_i \text{ at } l \downarrow 'r_j \text{ at } l \rangle_{\alpha'_{ij}}}{\langle p \downarrow 's_k \rangle_{\beta_k}, \langle q_i \text{ at } l \downarrow 'r_j \text{ at } \text{POST}_l \rangle_{\gamma_{ij}}}}{\overrightarrow{RES}, \overrightarrow{POST}, \gamma_{ij} \xrightarrow{f} \delta_{ijm}} \right] [G]$$

Here,  $j$  indexes the lifetime projections in  $q_i$ ,  $k$  indexes the lifetimes in the type of  $p$ , and  $m$  indexes the target (future lifetime projection) nodes  $\delta_{ijm}$  (if any) of edges from  $\alpha_{ij}$ .  $RES$  is the set of edges to the lifetime projections of  $p$ ;  $RES$  contains the edge  $\alpha'_{ij} \xrightarrow{a} \beta_k$  for each  $i, j, k$  where the  $j$ 'th lifetime in the type of  $q_i$  outlives the  $k$ 'th lifetime in the type of  $p_k$ .  $POST$  is the set of edges to the post-state lifetime projections of the operands, containing the edge  $\alpha'_{ij} \xrightarrow{a} \gamma_{uv}$  for all  $i, j, u, v$  where either  $i, j = u, v$  or the  $j$ 'th lifetime of  $q_i$  is a nested lifetime that outlives the  $v$ 'th lifetime of  $q_u$ .

### 3.7 Reborrowing Across Loops

Loops can create, end and move borrows, and reborrow: the extent of borrows created *inside* the loop may extend beyond the loop. These behaviours cannot be tracked precisely; our analysis constructs a *loop invariant* graph structure to summarise them. The parts of a PCG changed by the loop are replaced by our summarised loop invariant graph. This must have leaf nodes for the places accessible at the loop head (and after the loop), and be connected above to the closest borrowed-from nodes which are *not* touched by the loop, which we call the *loop borrow roots*. Its structure must capture intermediate (re)borrowed places that can become leaves when other borrows end: these can be more-general DAGs than suffice to summarise function calls.

A key design choice for our lifetime projection nodes is to enable abstracting our detailed graphs to capture the constraints enforced by the borrow checker on its corresponding notions of tracked borrow extents: these constraints correspond to reachability between the corresponding lifetime

**Algorithm 1** ConstructLoopInvariant - Main Algorithm

---

```

1: procedure CONSTRUCTLOOPINVARIANT( $\mathcal{G}_{\text{pre}}, P_{\text{blocked}}, P_{\text{blockers}}, l_h$ )
2:    $P_{\text{loop}} \leftarrow P_{\text{blocked}} \cup P_{\text{blockers}}$  ▷ 1. Identify relevant loop places
3:    $P_{\text{blocked}} \leftarrow P_{\text{blocked}} \cup (\bigcup_{p \in P_{\text{loop}}} \text{FINDROOTS}(p, \mathcal{G}_{\text{pre}}))$  ▷ 2. Add loop borrow roots
4:    $\mathcal{A} \leftarrow \text{MAKEINVARIANTGRAPH}(P_{\text{blocked}}, P_{\text{blockers}}, l_h)$  ▷ 3. Make invariant for loop head  $l_h$ 
5:    $\mathcal{G}_{\text{cut}} \leftarrow \text{EXTRACTSUBGRAPH}(\mathcal{G}_{\text{pre}}, \mathcal{A})$  ▷ 4. Identify subgraph to abstract
6:   return  $(\mathcal{G}_{\text{pre}} \setminus \mathcal{G}_{\text{cut}}) \cup \mathcal{A}$  ▷ 5. Replace subgraph with abstraction

```

---

**Algorithm 2** MakeInvariantGraph

---

```

1: procedure MAKEINVARIANTGRAPH( $P_{\text{blocked}}, P_{\text{blockers}}, l_h$ )
2:    $\mathcal{A} \leftarrow$  new empty graph
3:   for all  $(p_{\text{blocker}}, p_{\text{blocked}}) \in P_{\text{blockers}} \times P_{\text{blocked}}$  do
4:     if  $p_{\text{blocker}}$  blocks  $p_{\text{blocked}}$  at  $l_h$  then:  $\text{CONNECT}(\mathcal{A}, p_{\text{blocked}}, p_{\text{blocker}})$ 
5:   Label all non-leaf lifetime projection nodes in  $\mathcal{A}$  with  $l_h$ ; return  $\mathcal{A}$ 

```

---

projections. This property is maintained by careful design of our analysis rules and overall workflow, including queries to the compiler itself for some rules, as explained throughout this section.

Exploiting this design, we define an algorithm to automatically find an appropriate loop invariant graph by combining our existing analysis rules with the compiler's borrow-checking results. Algorithm 1 provides a high-level overview of the algorithm, which we now describe in detail.

Our loop invariant graph concerns those borrows that are live at the loop head and may differ across iterations of the loop (all other borrows are either unchanged or only exist temporarily within the loop). It is defined in terms of places that could store these borrows (the *blocking places*), as well as the targets of the borrows themselves (the *blocked places*).

We start by defining the *live loop places*: those places that are live at the loop head and used inside the loop. We define the *blocking places* as the subset of live loop places whose types allow storing borrows. The *blocked places* are the union of (a) the live loop places that could become blocked during the loop (identified via a straightforward analysis of the borrow assignments in the loop), and (b) the closest ancestors of the blocking places which are live (but *not* used in the loop body) and borrowed at the loop head; this latter set are the *loop borrow roots*.

We then construct the loop invariant shape by adding abstraction edges from blocked places to blocking places, including edges between their corresponding lifetime projections (Algorithm 2). This shape is constructed by iterating over each blocked place  $p$ , and identifying the subset of blocking places  $\overline{p_b}$  that the borrow checker reports as blocking  $p$  at the loop head. For each blocking place  $p_b$ , a *borrow* edge is added between  $p$  and the lifetime projection corresponding to the borrows in  $p_b$  and *abstraction* edges are added between the lifetime projections of  $p$  and the lifetime projections of  $p_b$  (Algorithm 3). The rules for inserting such edges are analogous to those we described for direct borrows in Sec. 3.3 and Sec. 3.4 (including handling of future nodes and edges).

Having computed the loop invariant shape, our analysis identifies the corresponding subgraph in the PCG state  $\mathcal{G}_{\text{pre}}$  that this abstraction will replace. The subgraph is constructed by including each edge  $e$  in  $\mathcal{G}_{\text{pre}}$  where there exists nodes  $m$  and  $n$  such that  $e$  is along a path from  $m$  to  $n$  in  $\mathcal{G}_{\text{pre}}$  and  $m$  is an ancestor of  $n$  in the abstraction shape. Replacing the subgraph with the loop invariant yields the PCG for the program point at the loop head.

Fig. 12 shows the invariant generated for the loop in Fig. 4. The places *prev* and *current* are considered live at the loop head. Because both of their types can contain borrows, they are

**Algorithm 3** Connect - Add Edges to Abstraction Graph

---

```

1: procedure CONNECT( $\mathcal{A}, p_{\text{src}}, p_{\text{tgt}}$ )
2:   Insert  $\langle\langle p_{\text{src}} : \emptyset \rangle\rangle \xrightarrow{b} \langle\langle p_{\text{tgt}} \downarrow r \rangle\rangle$  into  $\mathcal{A}$  ▷ Where  $r$  is the lifetime of  $p_{\text{tgt}}$ 
3:   for all lifetime  $r$  in the type of  $p_{\text{src}}$  do
4:      $RP_{\text{mut}} \leftarrow \{ \langle\langle p_{\text{tgt}} \downarrow r' \rangle\rangle \mid r' \text{ is a lifetime of } p_{\text{tgt}}, r \text{ outlives } r' \text{ and } r' \text{ outlives } r \}$ 
5:     if  $RP_{\text{mut}} \neq \emptyset$  then;
6:       Add  $\langle\langle p_{\text{src}} \downarrow r \rangle\rangle_{\alpha} \xrightarrow{a} \langle\langle p_{\text{src}} \downarrow r \text{ at FUT} \rangle\rangle_{\beta}$ 
7:       for all  $lp$  in  $RP_{\text{mut}}$  do
8:         Add  $\alpha \xrightarrow{a} lp, lp \xrightarrow{a} \beta$ 
9:        $RP_{\text{flow}} \leftarrow \{ \langle\langle p_{\text{tgt}} \downarrow r' \rangle\rangle \mid r' \text{ is a lifetime of } p_{\text{tgt}}, r \text{ outlives } r' \} \setminus RP_{\text{mut}}$ 
10:      for all  $lp$  in  $RP_{\text{flow}}$  do
11:        Add  $\langle\langle p_{\text{src}} \downarrow r \rangle\rangle \xrightarrow{a} lp$ 

```

---

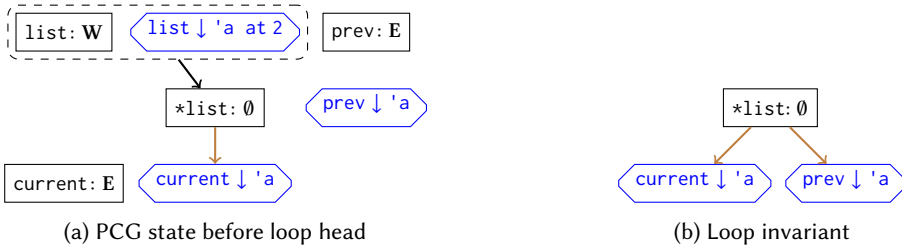


Fig. 12. The PCG loop invariant constructed for the `penultimate_mut` function in Fig. 4. The loop invariant is constructed at the loop head, without analysing the body of the loop.

considered as candidate blocker nodes in the graph. Neither is borrowed at the loop head, so they are not considered blocked nodes. We identify the place `*list` as a loop borrow root because it is the closest live ancestor of the lifetime projection  $\langle\langle \text{current} \downarrow 'a \rangle\rangle$  in the PCG just prior to the loop (Fig. 12a). Because the borrow checker indicates that `*list` is blocked by both `current` and `prev`, our algorithm adds borrow edges from `*list` to the lifetime projections of `current` and `prev`.

#### 4 Evaluation and Implementation

In this section, we describe our Rust implementation and its evaluation. We designed our evaluation to answer the following three questions:

- (Q1) Can our implementation handle most real-world Rust code? (Sec. 4.2)
- (Q2) Can our PCGs serve as the basis for other analysis tools? (Sec. 4.3 and Sec. 4.4)
- (Q3) Does our implementation accurately reflect Rust's borrow-checker constraints? (Sec. 4.5)

##### 4.1 Implementation

Our implementation is written in Rust and available as a crate that can be included as a library. It supports both the current Rust NLL borrow checker as well as the Polonius borrow checker.

Our library provides multiple interfaces for clients to consume the results of the analysis. Clients can analyse the PCGs themselves to e.g., identify aliasing relationships (as we do in our Flowistry implementation). Clients can also consume *annotations* describing the operations that transition between consecutive PCG states; our prototype Prusti implementation consumes these annotations

Table 2. Per-Function Results of PCG Analysis. Failures `ExpansionOfAliasType`, `IndexingNonIndexableType`, and `AssignBorrowToNonReferenceType` occurred due to limitations in the Rust compiler: normalising a type alias erases lifetime information that is required for the PCG analysis.

| Status      | Details                             | Count   |
|-------------|-------------------------------------|---------|
| Success     |                                     | 118,483 |
| Unsupported | DerefUnsafePtr                      | 1,584   |
|             | CallWithUnsafePtrWithNestedLifetime | 1,039   |
|             | InlineAssembly                      | 80      |
|             | MoveUnsafePtrWithNestedLifetime     | 73      |
|             | ExpansionOfAliasType                | 6       |
|             | IndexingNonIndexableType            | 5       |
|             | AssignBorrowToNonReferenceType      | 1       |
| Total       |                                     | 121,271 |

to generate Viper assertions. Note that both interfaces expose path-sensitive information: clients can filter graphs and annotations to include only those relevant for a particular path.

## 4.2 Top 500 Crates

To evaluate the PCG implementation on a wide range of code, we ran our implementation on functions from the top 500 most downloaded crates on `crates.io`, excluding nine crates that did not compile on our test system<sup>10</sup>. Table 2 presents statistics of the PCG analysis per function. Our analysis generated PCGs at every program point for the vast majority of functions (97.7%) in the evaluated crates. The majority of the remaining functions failed because they used features currently unsupported by our model: unsafe pointers and inline assembly. For 12 functions, PCG construction failed due to a limitation in resolving type aliases in the Rust compiler: the compiler's interface for normalising a type also erases lifetime information, which is necessary for our analysis.

Our evaluation completed in 94 minutes on a six-core Intel Xeon E-2236 with 64 gigabytes of RAM, using the default borrow checker and Rust nightly 2024-12-15. Although our implementation supports the usage of the next-generation Polonius borrow checker, we used the standard borrow checker for our evaluation as Polonius takes more than an hour on some crates in our suite.

## 4.3 Flowistry

Flowistry [10] performs an information-flow analysis to identify statements in a Rust program that might affect the value of a given expression. Their implementation includes a Rust alias analysis.

To demonstrate that PCGs can form the basis of other analyses, and to provide evidence for the correctness of our analysis, we replaced Flowistry's alias analysis with a new one that uses our PCG implementation. The resulting version of Flowistry passed all tests in Flowistry's test suite.

## 4.4 Prusti

Prusti [1] is a verifier that translates Rust programs into Viper [30], a verification language based on separation logic. In order to check functional properties, Prusti first verifies that the borrow checker correctly enforced memory safety by constructing a *core proof* of memory safety. To automate this, Prusti uses an ad-hoc analysis to infer a variety of annotations required for the Viper proof.

<sup>10</sup>The crates `winreg`, `criterion-plot`, `tiny-keccak`, `redox_users`, `plotters`, `darling`, `derive_more`, `tokio-native-tls`, and `clang-sys` did not compile either because the operating system was not supported, necessary external libraries were missing, or the compilation required additional configuration via cargo.

Table 3. Results of Mutation Testing on top 250 crates. *tot* lists the number of mutants produced by the mutation. *kill* lists the number of mutants which failed borrow checking, as expected. *live* lists the number of mutants which passed borrow checking. *%kill* lists the percentage of mutants which failed borrow checking.

| Mutation            | <i>tot</i> | <i>kill</i> | <i>live</i> | <i>%kill</i> |
|---------------------|------------|-------------|-------------|--------------|
| MutablyLendShared   | 1172766    | 1172421     | 345         | 99.97%       |
| ReadFromWriteOnly   | 5078080    | 5078080     | 0           | 100.00%      |
| WriteToShared       | 1172336    | 1171973     | 345         | 99.96%       |
| MoveFromBorrowed    | 283649     | 283496      | 153         | 99.94%       |
| BorrowExpiryOrder   | 17119      | 17119       | 0           | 100.00%      |
| AbstractExpiryOrder | 4817       | 4817        | 0           | 100.00%      |
| <i>Total</i>        | 7728767    | 7727906     | 843         | 99.99%       |

We developed a prototype version of Prusti that instead uses our PCG implementation to generate proof annotations. This significantly simplified the Prusti codebase while also enabling support for some Rust features (such as references in structs or reborrowing in loops) that the previous analysis did not support. Our PCGs enable the inference of all annotations required for Prusti’s core proof:

- Predicate *fold* and *unfold* statements. These are a key feature that enables support for recursive data types. Due to the similarity of separation logic to Rust’s ownership system, these have a one-to-one correspondence with our pack and unpack operations.
- Loop invariants that specify the loop’s memory footprint. These are required for all loops and correspond to the PCG state at the head of the loop.
- Magic wand *package* and *apply* statements. These describe how permissions flow from an expired reference to the original owning data structure and are required for reborrowing methods and reborrowing in loops. They correspond to our abstraction edges.

Evaluating our prototype against a subset of the Prusti test suite (the should-verify category with overflow checks disabled), encoding *only* the core proof and not the functional correctness specifications, yielded a result of 155 tests passed, 40 failed, and 67 ignored due to unsupported features. Based on manual inspection, the remaining failures are unrelated to the PCG analysis. We also successfully verified simplified versions of all the examples presented in this paper.

#### 4.5 Mutation Testing

To demonstrate that PCGs accurately reflect the restrictions enforced by the Rust compiler, we built a mutation testing tool which uses the PCG to mutate the MIR of well-typed Rust programs into mutants which violate particular ownership or borrowing rules:

- MutablyLendShared creates a mutable borrow to a place behind a shared borrow.
- WriteToShared writes to a place behind a shared borrow.
- ReadFromWriteOnly reads from a place with **W** capability.
- MoveFromBorrowed moves out of a place behind a mutable borrow.
- BorrowExpiryOrder identifies a place  $p_1$  which blocks another place  $p_2$  via a mutable borrow and attempts to use  $p_2$  before  $p_1$ .
- AbstractExpiryOrder identifies a place  $p_1$  which blocks another place  $p_2$  via a mutable borrow and through an abstraction edge and attempts to use  $p_2$  before  $p_1$ .

Table 3 reports the results of running our tool on the top 250 most downloaded crates on crates.io. Note that kill and live do not add up to tot; WriteToShared caused 18 compiler panics not counted towards kill since their cause may be unrelated to type checking. We also have no data for 9 crates on which our mutation tool crashed. 99.96% of mutants fail to type check,

demonstrating that PCGs reflect Rust's type rules precisely. Of the live mutants we examined, all appeared to violate some rule. However, we were unable to replicate any live mutants in surface Rust. We suspect this is because the compiler performs some initial borrow checking before generating MIR<sup>11</sup>. As we mutate downstream of these checks, the compiler may not reject certain mutants.

## 5 Related Work and Potential Consumers

We discuss two aspects of related work: we compare existing models for Rust's type checking to PCGs and discuss how users of these models could benefit from the information provided by PCGs.

*Prusti*. Prusti's encoding [1] was originally presented in terms of *place capability summaries*<sup>12</sup>, presented semi-formally as a representation analogous to the place node leaves in our PCGs. Rules for modelling borrows etc. are formalised only indirectly via the target separation logic. The resulting proofs require detailed annotations, especially for reborrowing (handled by proving *magic wands*), demanding a step-by-step justification for *how* borrowed capabilities are returned. Prusti previously inferred such annotations through complex algorithms that proved hard to generalise beyond simple reborrowing patterns [12]. The prototype developed in our evaluation generates all necessary annotations directly from our PCG analysis results and can analyse programs that are not supported in the current version of Prusti, such as those that reborrow inside loops.

*Oxide*. Oxide [41] is (to our knowledge) the most mature attempt at a formal semantics for a large portion of Rust's type system, including (unlike our work) formalising how type checking can be *performed*: it defines and implements type checking rules, an operational semantics, and proves a type soundness result for a small language including challenging features such as closures.

For Rust *analysis and verification*, however, several key features are absent. Structs and lifetime parameters are not supported: desugaring of structs to tuples is not compatible with general Rust struct definitions which may feature recursive or opaque types. Rust uses lifetime parameters to abstract over *where* in a type borrows may be stored; our lifetime projections capture these abstractions directly and modularly. Oxide's rules also do not capture path-sensitive information, nor show how loop invariants can be computed to summarise loops that interact with (re-)borrowing. The formalism is also for a custom language representation; it's unclear how these results can be connected with practical analysis tools that typically work on Rust's representations (usually MIR).

*Aeneas*. Aeneas [19] defines formal models of Rust's type checking, and a verification approach mapping (safe) Rust into a functional language for manual proof in proof assistants. Rust moves and borrows are handled by copying values; where reborrows span function calls and loops, additional *backward functions* are generated, defining how to copy corresponding values back again.

Our comparison with Aeneas is similar to that with Oxide: they formalise more in terms of how Rust type checking works, but do not support as large a fragment of Rust (e.g. no struct lifetime parameters). Reborrowing inside loops is supported by heuristics that need not always succeed [19], and joins do not retain precise path-sensitive information. Connecting with other analyses is (as for Oxide) limited by the use of a custom program representation separate from Rust.

On the other hand, our PCGs could in principle generate the information needed by Aeneas for its verification approach: managing recursive representations of types is similar, and the generation of backward functions requires steps similar to the proofs of magic wands in Prusti, for which we have demonstrated proof-of-concept support. Consequently, it may be possible to lift some of the existing limitations of Aeneas by using the information provided by PCGs (as with our Prusti prototype), especially with respect to general loops and nested lifetimes.

<sup>11</sup>For example, in the `mir_built` query.

<sup>12</sup>The name for our model here is inspired by this notion.

*Aquascope.* Aquascope [9] is an analysis that visualises the type checking of Rust programs. The tool extracts ownership and borrowing information from a customised version of the Rust compiler, even for programs that do *not* type check. While a notion of capabilities (called permissions in their work) per program point is visible in Aquascope, information about *why* and *how* a place is borrowed is not available, nor is any information path-sensitive. Consistent with these choices, loops and function calls are not summarised. We believe that the information needed by Aquascope could be easily extracted from PCGs, except that our analysis requires programs that type check.

*RustBelt, RustHornBelt and Creusot.* RustBelt [22] is an early formal model for Rust, and defines a logical interpretation of Rust’s type checking as an instantiation of the Iris framework [23]. This interpretation models Rust ownership using separation logic predicates, and borrows via a custom logic of ghost state called the *lifetime logic*; in particular, this setup allows proofs about *unsafe* Rust, where the standard meanings of borrows might be customised for specific library implementations.

RustHornBelt [27] extends RustBelt to enable functional specifications on verified libraries, and uses *prophecy values* to reflect the value of borrows as they *will* be eventually modified. Although prophecies are an alternative means of reasoning about borrows to showing how capabilities flow back, the *soundness proof* for RustHornBelt requires showing the existence of an acyclic relation between the borrows; this is similar to the borrowing information our model produces.

These models are not directly comparable with ours: they do not connect directly to Rust programs or the compiler’s representations; their languages are loop-free. However, the models are fully formalised and can reason about unsafe code, while ours focuses on safe Rust. Our rules for creating and dereferencing borrows take some inspiration from RustBelt: the dual usage of place nodes and lifetime projection nodes is reminiscent of RustBelt’s rules, which require not only access to the borrowed memory but a justification that the corresponding lifetime is currently live.

Creusot [11] is a Rust verifier that translates Rust programs into functional programs but in a different manner to Aeneas; it exposes the notion of prophecy values in specifications, so that users can write specifications concerning (re)borrows that span function calls and loops by relating their current and *future* values. While this technique does not require summaries of borrow information to the same extent, we believe that visualising and understanding these summaries (for example for loops) might still help tool users to understand which specifications could be meaningful to express. We are confident that Creusot’s translation could also be driven by the information in our model.

*Stacked Borrows and Tree Borrows.* Stacked Borrows [21] and Tree Borrows [38] define a runtime semantics using graphs (stacks or trees) to represent dynamic borrow information in each execution state, with a focus on defining when memory accesses via *unsafe Rust code* are allowed. In contrast, PCGs define a modular static analysis for safe Rust that precisely captures static borrow-checking information. This different goal requires us to address challenges that do not arise for operational semantics such as Stacked or Tree Borrows, where only bounded memory and behaviours are explored, and modularity is not a concern. In particular, our PCGs must represent (possibly unbounded) sets of borrows stored in e.g., structs, provide (re)borrowing rules that integrate with these representations, and use them to summarise the effect of function calls and loops. Our PCGs are also designed to efficiently represent path-sensitive borrowing information.

*Other Rust Semantics.* We are aware of several older attempts to formalise aspects of Rust’s type system (e.g., [24, 32, 39, 40]); to our knowledge, these do not cover the detailed borrowing concerns addressed by our model, and are typically for idealised definitions of Rust-like languages. Some are operational semantics, which do not require the same complex rules to summarise Rust’s implied properties for general path-sensitive information, loops or function calls.

*Other Rust Analyses.* Many Rust verification and analysis tools could potentially make use of our model. Verus [25] is a verifier whose translation to SMT requires similar information to Prusti; it does not currently support reborrows that span function calls or loops, but there is long-standing interest in supporting them in the future [14], which would require information similar to that provided by PCGs. Flux [26] is a Rust verifier based on Liquid Types which also requires compiler-generated information for its translation that our model can provide. Rudra [2] is a mature program analysis for Rust, whose analyses require type-system-related information; to our knowledge these analyses are all flow-insensitive, and our model could easily provide (more than) the required basis.

Pearce [31] presents an early formalisation of Rust's type system, but does not support more-intricate features of Rust types (such as structs storing references) nor control flow. His work aims to define borrow checking, whereas ours focuses on the results of the borrow checker. Rust Analyzer [36] is a library for semantic analysis of Rust code. Its analyses are primarily intended to support IDE features, e.g. finding usages of a variable, and in general are lightweight versions of analyses that the Rust compiler itself performs. In contrast, our analysis builds on top of the compiler's existing analyses to provide an elaborated model of its type and borrow checking.

*Ownership Types.* Ownership type systems were first designed in the context of object-oriented languages. Clarke et al. [6] present a type system that organises objects in a tree-shaped ownership structure. They enforce that each reference chain from a root of the ownership graph to an object goes through all of the object's (transitive) owners; all other references are ruled out by the type system. Follow-up work proposed various variations of this idea. For instance, Universe Types [28] distinguishes between write and read-only references and imposes ownership restrictions only on write references. Other systems [5, 29] allow ownership to change dynamically, akin to move assignments in Rust. Some of these early ownership type systems permit restricted forms of borrowing (e.g. External Uniqueness [5] permits lexically-scoped borrows), but none can express the control-flow dependent borrow extents tracked by the Rust compiler.

Boyland's alias burying [3] supports unique variables, similar to Rust's owning places. In contrast to prior work, which maintained uniqueness through destructive reads (related to Rust's move assignments), references can be borrowed temporarily. A modular static analysis ensures that the borrowed-from unique variable is not used before the borrow goes out of scope.

## 6 Conclusions and Future Work

We have presented Place Capability Graphs, a novel model of Rust's type and borrow checking. We demonstrated the construction of PCGs from Rust's internal programmatic representation. In our evaluation, we showed that PCGs support a large subset of real-world Rust code, accurately reflect the constraints of the borrow checker, and can be used as the basis for program analysis tools.

There are several promising avenues of future work. We plan to continue the formal development of our work, and consider its suitability as a specification of type checking itself. We hope the open-source implementation of Place Capability Graphs can serve as the basis for a wider range of Rust analysis tools, beyond our prototype integrations into Prusti and Flowistry.

## Acknowledgements

We thank Rui Ge, Yanze Li, and Youssef Saleh for helpful feedback on this work and its presentation, and Rui Ge, Yanze Li and Randy Zhu for helping to test our artefact.

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). This work was funded in part by the Stellar Development Foundation.

## Data Availability Statement

We have made available all aspects of our evaluation (including the stand-alone implementation of our model) as an artefact [16], including sufficient instructions to reproduce our various evaluations. Our implementation also produces visualisations similar to the figures used in our paper, which can be used to manually inspect instances of our model as generated on chosen Rust code.

## References

- [1] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 2019. Leveraging Rust types for modular specification and verification. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 147:1–147:30. doi:10.1145/3360573
- [2] Yechan Bae, Youngsuk Kim, Ammar Askar, Jungwon Lim, and Taesoo Kim. 2021. Rudra: Finding Memory Safety Bugs in Rust at the Ecosystem Scale. In *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, Robbert van Renesse and Nikolai Zeldovich (Eds.). ACM, 84–99. doi:10.1145/3477132.3483570
- [3] John Boyland. 2001. Alias burying: Unique variables without destructive reads. *Softw. Pract. Exp.* 31, 6 (2001), 533–553. doi:10.1002/SPE.370
- [4] Elias Castegren and Tobias Wrigstad. 2016. Reference Capabilities for Concurrency Control. In *30th European Conference on Object-Oriented Programming (ECOOP 2016) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 56)*, Shriram Krishnamurthi and Benjamin S. Lerner (Eds.), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 5:1–5:26. doi:10.4230/LIPIcs.ECOOP.2016.5
- [5] Dave Clarke and Tobias Wrigstad. 2003. External Uniqueness Is Unique Enough. In *ECOOP 2003 - Object-Oriented Programming, 17th European Conference, Darmstadt, Germany, July 21-25, 2003, Proceedings (LNCS, Vol. 2743)*, Luca Cardelli (Ed.). Springer, 176–200. doi:10.1007/978-3-540-45070-2\_9
- [6] David G. Clarke, John Potter, and James Noble. 1998. Ownership Types for Flexible Alias Protection. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications, OOPSLA 1998, Vancouver, British Columbia, Canada, October 18-22, 1998*, Bjørn N. Freeman-Benson and Craig Chambers (Eds.). ACM, 48–64. doi:10.1145/286936.286947
- [7] Sylvan Clebsch, Sophia Drossopoulou, Sebastian Blessing, and Andy McNeil. 2015. Deny Capabilities for Safe, Fast Actors. In *International Workshop on Programming Based on Actors, Agents, and Decentralized Control*. ACM, 1–12.
- [8] Rust Community and Language Team. 2025. Place Expressions and Value Expressions (The Rust Reference). <https://doc.rust-lang.org/reference/expressions.html#place-expressions-and-value-expressions> Accessed July 2025.
- [9] Will Crichton, Gavin Gray, and Shriram Krishnamurthi. 2023. A Grounded Conceptual Model for Ownership Types in Rust. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 265 (Oct. 2023), 29 pages. doi:10.1145/3622841
- [10] Will Crichton, Marco Patrignani, Maneesh Agrawala, and Pat Hanrahan. 2022. Modular information flow through ownership. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (San Diego, CA, USA) (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 1–14. doi:10.1145/3519939.3523445
- [11] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 2022. Creusot: A Foundry for the Deductive Verification of Rust Programs. In *Formal Methods and Software Engineering - 23rd International Conference on Formal Engineering Methods, ICFEM 2022, Madrid, Spain, October 24-27, 2022, Proceedings (LNCS, Vol. 13478)*, Adrián Riesco and Min Zhang (Eds.). Springer, 90–105. doi:10.1007/978-3-031-17244-1\_6
- [12] Prusti Developers. 2024-2025. Personal Communication.
- [13] Prusti Developers. 2025. PCG Repository. <https://github.com/prusti/pgc> Accessed August 21, 2025.
- [14] Verus Developers. 2022. Returning mutable references (discussion). <https://github.com/verus-lang/verus/discussions/35> Accessed March, 2025.
- [15] Colin S. Gordon, Matthew J. Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. 2012. Uniqueness and reference immutability for safe parallelism. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (Tucson, Arizona, USA) (OOPSLA '12)*. Association for Computing Machinery, New York, NY, USA, 21–40. doi:10.1145/2384616.2384619
- [16] Zachary Grannan, Aurel Bílý, Jonáš Fiala, Jasper Geer, Markus de Medeiros, Peter Müller, and Alexander J. Summers. 2025. Artefact for "Place Capability Graphs: A General-Purpose Model of Rust's Ownership and Borrowing Guarantees". doi:10.5281/zenodo.16597989
- [17] Zachary Grannan, Aurel Bílý, Jonáš Fiala, Jasper Geer, Markus de Medeiros, Peter Müller, and Alexander J. Summers. 2025. Place Capability Graphs: A General-Purpose Model of Rust's Ownership and Borrowing Guarantees. arXiv:2503.21691 [cs.PL] <https://arxiv.org/abs/2503.21691>

- [18] Philipp Haller and Martin Odersky. 2010. Capabilities for Uniqueness and Borrowing. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer, Berlin, Heidelberg, 354–378. doi:10.1007/978-3-642-14107-2\_17
- [19] Son Ho, Aymeric Fromherz, and Jonathan Protzenko. 2024. Sound Borrow-Checking for Rust via Symbolic Semantics. *Proc. ACM Program. Lang.* 8, ICFP, Article 251 (Aug. 2024), 29 pages. doi:10.1145/3674640
- [20] Son Ho and Jonathan Protzenko. 2022. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.* 6, ICFP (2022), 711–741. doi:10.1145/3547647
- [21] Ralf Jung, Hoang-Hai Dang, Jeehoon Kang, and Derek Dreyer. 2020. Stacked borrows: an aliasing model for Rust. *Proc. ACM Program. Lang.* 4, POPL (2020), 41:1–41:32. doi:10.1145/3371109
- [22] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. 2018. RustBelt: securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.* 2, POPL (2018), 66:1–66:34. doi:10.1145/3158154
- [23] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 2018. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* 28 (2018), e20. doi:10.1017/S0956796818000151
- [24] Shuanglong Kan, David Sanán, Shang-Wei Lin, and Yang Liu. 2018. K-Rust: An Executable Formal Semantics for Rust. *CoRR abs/1804.07608* (2018). arXiv:1804.07608 doi:10.48550/arXiv.1804.07608
- [25] Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. 2023. Verus: Verifying Rust Programs using Linear Ghost Types (extended version). *CoRR abs/2303.05491* (2023). arXiv:2303.05491 doi:10.48550/arXiv.2303.05491
- [26] Nico Lehmann, Adam Geller, Gilles Barthe, Niki Vazou, and Ranjit Jhala. 2022. Flux: Liquid Types for Rust. *CoRR abs/2207.04034* (2022). arXiv:2207.04034 doi:10.48550/arXiv.2207.04034
- [27] Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. 2022. RustHornBelt: a semantic foundation for functional verification of Rust programs with unsafe code. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 841–856. doi:10.1145/3519939.3523704
- [28] P. Müller. 2002. *Modular Specification and Verification of Object-Oriented Programs*. LNCS, Vol. 2262. Springer.
- [29] Peter Müller and Arsenii Rudich. 2007. Ownership transfer in universe types. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, Richard P. Gabriel, David F. Bacon, Cristina Videira Lopes, and Guy L. Steele Jr. (Eds.). ACM, 461–478. doi:10.1145/1297027.1297061
- [30] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 2016. Viper: A Verification Infrastructure for Permission-Based Reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings (LNCS, Vol. 9583)*, Barbara Jobstmann and K. Rustan M. Leino (Eds.). Springer, 41–62. doi:10.1007/978-3-662-49122-5\_2
- [31] David J. Pearce. 2021. A Lightweight Formalism for Reference Lifetimes and Borrowing in Rust. *ACM Trans. Program. Lang. Syst.* 43, 1 (2021), 3:1–3:73. doi:10.1145/3443420
- [32] Eric W. Reed. 2015. *Patina: A Formalization of the Rust Programming Language*. Technical Report UW-CSE-15-03-02. University of Washington.
- [33] Rust contributors. 2019. The Polonius Reference Implementation for the Rust Borrow-Checker. <https://github.com/rust-lang/polonius> Accessed April 4, 2019.
- [34] Rémy Rakic and Niko Matsakis on behalf of The Polonius Working Group. 2023. Polonius Update. <https://blog.rust-lang.org/inside-rust/2023/10/06/polonius-update.html> Accessed March 24, 2025.
- [35] Sven Stork, Karl Naden, Joshua Sunshine, Manuel Mohr, Alcides Fonseca, Paulo Marques, and Jonathan Aldrich. 2014. Æminium: A Permission-Based Concurrent-by-Default Programming Language Approach. *ACM Trans. Program. Lang. Syst.* 36, 1, Article 2 (March 2014), 42 pages. doi:10.1145/2543920
- [36] Ferrous Systems and contributors. 2025. Rust-analyzer. <https://rust-analyzer.github.io/> Accessed July 30, 2025.
- [37] Alexa VanHattum, Daniel Schwartz-Narbonne, Nathan Chong, and Adrian Sampson. 2022. Verifying Dynamic Trait Objects in Rust. In *44th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2022, Pittsburgh, PA, USA, May 22-24, 2022*. IEEE, 321–330. doi:10.1109/ICSE-SEIP55303.2022.9794041
- [38] Neven Villani, Johannes Hostert, Derek Dreyer, and Ralf Jung. 2025. Tree Borrows. *Proc. ACM Program. Lang.* 9, PLDI, Article 188 (June 2025). doi:10.1145/3735592
- [39] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. 2018. KRust: A Formal Executable Semantics of Rust. *CoRR abs/1804.10806* (2018). arXiv:1804.10806 doi:10.48550/arXiv.1804.10806
- [40] Aaron Weiss, Daniel Patterson, and Amal Ahmed. 2018. Rust Distilled: An Expressive Tower of Languages. *arXiv preprint arXiv:1806.02693* (2018). doi:10.48550/arXiv.1806.02693
- [41] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. 2019. Oxide: The Essence of Rust. *CoRR abs/1903.00982* (2019). doi:10.48550/arXiv.1903.00982

Received 2025-03-25; accepted 2025-08-12