

# Extended Place Capabilities Summaries for Rust Programs

Dylan Wolff

**Abstract**—The advent of the Rust programming language and its borrowing and ownership model has created new opportunities in Program Analysis and Verification. However, to take advantage of Rust’s advanced type system, tools need to extract detailed information from Rust’s compiler and borrow checker. Furthermore, these tools must assemble this data into a meaningful format. The Extended Place Capabilities Summary (EPCS) presents type and borrow checking data in just such a format for a meaningful subset of the Rust language while also performing analysis steps to provide an enriched summary of the data. Furthermore, the EPCS API implementation allows verification and analysis tool developers to depend on a centralized, stable API rather than directly on the various internal, unstable compiler or borrow checker functions and data structures.

## I. INTRODUCTION

The Rust programming language takes a unique approach to static compiler checks and memory management. By enforcing ownership and borrowing restrictions for memory locations, Rust’s type-system precludes whole classes of memory safety issues common in other languages that do not utilize a garbage collector. The recent popularity of Rust has created new opportunities to build analysis and verification tools that take advantage of the information and guarantees provided by its type system, such as the Prusti project [3].

However, while the Rust compiler provides descriptive negative-information for programs that do not compile, there is no canonical way to extract positive-information for a correct Rust program (i.e., why it type-checks in the first place). This kind of information is crucial for verification tools like Prusti [3] and may have other applications for visualization and analysis of Rust programs.

Prior work utilizing any such positive-information required direct interaction with the Rust compiler internals. The Rust compiler, however, is under active development, and its internal APIs are unstable. As a result, as the Rust compiler changes on a nightly release schedule, these changes frequently propagate as breakages in any dependent projects. The Rust compiler developers consider these changes considered “non-breaking” because the internal compiler APIs are not officially stable.<sup>1</sup> This constant churn requires extreme diligence from tool maintainers to fix issues scattered throughout the codebases of any projects depending on this information.

To alleviate this problem, we developed a comprehensive summary of information commonly needed by Prusti and

other tools as a separate library. This aggregation localizes breaking changes to a single, separated component. It also allows many applications to utilize a curated, stable API rather than needing to massage internal compiler data into a usable format.

The term “Place Capability Summary” (PCS) was initially coined in the Prusti paper [3], but was not precisely defined nor fully described. Generally, it summarizes the capabilities a Rust program has to memory locations (“places”) at various points throughout its execution.

For example, initializing and assigning to a variable would add the corresponding place to the PCS, as that variable now has permissions to access the memory location to which it refers. While initially, it may exclusively refer to that place (and thus be mutable<sup>2</sup>), if another variable later borrows that place, its capabilities may change based on the characteristics of the borrow.

So far, there have been several efforts towards formalizing the semantics of Rust (e.g., [11], [12], [7], [8]), but none have precisely outlined the notion of a PCS, nor implemented a way to extract this information from a Rust program. The primary goal of this project was to expand on these prior works in this area.

## II. CONTRIBUTIONS

- 1) Precisely defined the notion of a “Place Capability Summary” (PCS) and “Extended Place Capability Summary” (EPCS) for a given line of a Rust program at the Mid-level Intermediate Representation (MIR) level of abstraction
- 2) Developed a plan for extracting this information from MIR code and the Rust compiler/borrow checker
- 3) Implemented a tool that can derive the EPCS for a small subset of MIR as a proof of concept for a complete implementation of an EPCS API for Rust as a whole

## III. APPROACH

At a high level, we define an Extended Place Capability Summary (EPCS) to combine accessibility information from the PCS with additional data regarding borrows at that program point. Thus, an EPCS represents part of the program state before or during a given statement in a Rust program. This state evolves through a set of inference rules, expanded upon in Section V. Most of these rules dictate how each kind

<sup>1</sup>This is not to suggest that the maintenance issues are the fault of the Rust compiler team. The APIs used by Prusti, for example, are internal for good reasons. It is, and absolutely should be, up to the individual project owners to adjust to frequent API changes.

<sup>2</sup>As in [12], we treat mutability of a place as being determined by whether or not a single variable has exclusive access to it, not by the “mut” keyword. This keyword merely allows Rust programmers to be explicit about the exclusivity of access to places

of statement changes an EPCS. Some, however, such as the “unpack” and “pack” rules defined in Section V-A, prepare an EPCS such that it is in a proper state for the other rules to take effect. We call these rules “Ghost Rules” and determine when/how to apply them by the algorithms given in Section V. Ghost rules transform an EPCS to another EPCS with a different but equivalent representation of the state at that program point.

We apply the inference rules from an initial empty EPCS state for each entry point into the Rust program. To do so, we follow the Control Flow Graph (CFG) statement by statement until we reach a fixed point. In practice, we achieve this with a single iteration because of the design choice outlined in Section VI-C. With this fixed point reached, we can query the EPCS API at any statement within the CFG and obtain a summary of the type and borrow checking information at that program point.

#### IV. (E)PCS DEFINITIONS

In the Rust lexicon, a place corresponds to a memory location [1]. Place expressions, which contain local variables, static variables, dereferences ( $*\text{expr}$ ), and fields, represent a given place. Rust’s type system also has a notion of ownership and a notion of borrowing. Because of borrowing, the “owning” place expression may not be able to access its underlying memory location at all program points. To capture this behavior, the Prusti [3] authors distinguish the right of a place expression to access the value stored at its underlying place as a “Place Capability”. Thus, the “Place Capabilities Summary” or “Place Capability Set” (PCS) of a program at point  $p$  is the set of all usable place expressions at  $p$ , along with their associated capabilities.

However, it is impossible to accurately present this information without accounting for borrowed values. For example:

Listing 1: A Simple Rust Program

```

1  let mut x = 4;
2  let r = &mut x;
3  // print!("{}", x);
4  *r = 3;
5  print!("{}", x);

```

At line 1, the PCS includes an exclusive capability to use  $x$ . However, when the place described by  $x$  is mutably borrowed in line 2, this capability is temporarily lost. This loss occurs because the mutable borrow guarantees the borrower, in this case  $r$ , exclusive access to that place for the lifetime of the borrow. Thus, if we uncomment line 3, the Rust compiler would throw an error, as the borrow must maintain this exclusive capability until at least line 4, where it modifies the place. By line 5, the borrow has expired, which restores the initial capabilities of  $x$  and allows the final statement to compile with no errors. These two aspects, the temporary reduction in capabilities due to a loan and the subsequent restoration of those capabilities, are key components in forming a PCS that never shows capabilities that the program does not have.

Prusti handles loans separately from computing a limited PCS, but, to provide an accurate PCS as a separate component, it must do this analysis in addition to forming the limited PCS. We define this additional information concerning which loans block/restore which capabilities as the Reference Capabilities Summary (RCS). We include both the RCS and PCS in what we call the Extended Place Capabilities Summary (EPCS).

The EPCS at program point  $p$  is defined by the following grammar:

EPCS are split into a PCS,  $\Gamma$ , and an RCS,  $\Sigma$ :

$$EPCS_p ::= \Gamma \Sigma \quad (1)$$

PCS are composed of a set of place expressions:

$$\Gamma ::= \{ \} \mid \{ \Pi \} \quad (2)$$

RCS are composed of a set of references:

$$\Sigma ::= \{ \} \mid \{ \Lambda \} \quad (3)$$

Place expressions,  $\pi$ , have a type,  $\tau$ , and capability,  $\omega$ :

$$\Pi ::= \pi : \tau : \omega \mid \pi : \tau : \omega, \Pi \quad (4)$$

Place expressions are composed of variables and their sub-places (fields and dereference operations):

$$\pi ::= x \mid \pi.f \mid * \pi \quad (5)$$

References are place expressions leading to other place expressions ( $\pi \rightarrow \pi$ ), or indefinite regions  $\varrho$ :

$$\Lambda ::= \lambda \mid \lambda, \Lambda \quad (6)$$

$$\lambda ::= \pi \rightarrow \pi : \tau : \omega \mid \pi \rightarrow \varrho \quad (7)$$

Regions are a set of place expressions blocked by the corresponding reference:

$$\varrho ::= \text{region}[] \mid \text{region}[\Pi] \mid \text{subplace}[\pi : \tau : \omega] \quad (8)$$

Capabilites are either shared or exclusive:

$$\omega ::= e \mid s \quad (9)$$

Note that, for concision, we frequently omit  $\tau$  and  $\omega$  throughout this paper for contexts in which they are obvious or irrelevant. Additionally, for readability, the RCS set is written with parenthesis, rather than curly braces, to distinguish it visually from the PCS.

Each  $\lambda$  corresponds to a reference that is live at program point  $p$ . These references can be concrete, referring to only a single place ( $\pi \rightarrow \pi$ ) or indefinite, referring to either a region or a sub-place of some known place  $\pi$ .<sup>3</sup> Because regions are not typically used by tools outside of the borrow checker and not particularly informative on their own, the

<sup>3</sup>In the typical Rust terminology, the term lifetime is used to refer to this property. However, in the Polonius borrow checker implementation utilized by this paper ([4]), lifetimes have been re-dubbed “regions” and thus will be referred to as such

regions presented in the PCS include information about the variables that are blocked by them.

Thus the  $\rightarrow$  operator as defined above can also be thought of as the “magic wand” operator ( $-*$ ) as described in [5]. The referrer on the left blocks the element on the right until that reference is no longer live. At this point, the program releases the referrer’s capabilities (e.g. the corresponding borrow expires) and restores the capabilities of the elements on the right.

The analysis performed to calculate the EPCS is sound for capabilities held at a given program point. That is, a program never holds fewer capabilities for a place than presented by the EPCS at that point. The soundness of the EPCS is discussed further in Section V-G.2.

The notation can be seen in the previous example in Listing 1, now annotated with the EPCS after each line (with types elided for concision):

Listing 2: An Annotated Simple Rust Program

```

1  let mut x = 4;    // {x:e} ()
2  let r = &mut x;  // {r:e} (r -> x:e)
3  ...
4  *r = 3;          // {x:e} ()
5  print!("{}", x); // {x:e} ()

```

The reference created at the second line removes the capabilities of  $x$ , but tracks these capabilities with the reference  $r : e \rightarrow x : e$ . This reference indicates that  $r$  is blocking some capabilities for  $x$ . Thus, we can restore the capabilities on the right of the arrow after line 4 by merely placing them directly back into the PCS’s curly brackets. In this case, by looking at the code snippet, it is evident that  $r$  always points directly to  $x$ . This reference is direct because the right side of the arrow is a concrete place expression rather than a region, such as  $r[x : e]$ .

The above definition and explanation are sufficient for a path-independent EPCS. For a path-dependent EPCS, the summary at a given program point is composed of a set of sub-EPCS, one for each unique path through the program’s control flow graph. A path-dependent EPCS could also be extended to contain metadata about which conditions correspond to which version of the EPCS (i.e. path constraints).

## V. PCS RULES

### A. Packing and Unpacking Fields

Capabilities to a given place also imply the same capabilities to its sub-places (defined by the grammar in Equation 5). However, capabilities to sub-places can be acquired and given up throughout the program. Thus it is necessary to track these capabilities in such that we can accurately reassemble them into their parent places as required.<sup>4</sup>

To track this information, the EPCS can be transformed with “pack” and “unpack” operations into the state required by the current point. For example:

Listing 3: Packing and Unpacking

<sup>4</sup>A “parent place” of  $\pi$  is a place expression  $\pi'$  for which  $\pi$  is a sub-place of  $\pi'$

```

1 struct Simple {
2     f: u8,
3     g: u8,
4 }
5
6 fn subplaces() {
7     let mut c = Simple { 3, 4 }; // { c } ()
8     c.f = 2; // { c.f, c.g } ()
9     let d = c; // { c } ()
10 }

```

The above snippet contains one instance of unpacking, from  $c$  into  $c.f, c.g$ , at line 8. This unpacking is necessary because the assign statement at line 8 *requires* capabilities to  $c.f$  to be able to assign to it. At line 9, the assign statement needs a shared capability to the whole of the struct stored in  $c$ , so the sub-places are packed back into  $c$  in the PCS. We apply packing and unpacking recursively to nested structures to disassemble or reassemble their capabilities appropriately.

More formally, for a place  $\pi$  of type  $\tau$  with fields  $f_1, \dots, f_n$  unpacking of a capability  $\omega$  in from the initial PCS,  $\Gamma$ , to the required PCS,  $\Gamma'$ , is defined by the following rule:

UNPACK:

$$\frac{\pi : \tau : \omega \in \Gamma \quad \text{fields}(\tau) = f_1 : \tau_1, \dots, f_n : \tau_n}{\Gamma, \Sigma \rightarrow \Gamma[\pi / \pi.f_1 : \tau_1 : \omega, \dots, \pi.f_n : \tau_n : \omega], \Sigma} \quad (10)$$

Here  $\Sigma$  corresponds to the RCS as in Equation 1. Similarly, packing, the inverse operation, is defined by:

PACK:

$$\frac{\pi.f_1 : \tau_1 : \omega, \dots, \pi.f_n : \tau_n : \omega \in \Gamma \quad \pi : \tau \quad \text{fields}(\tau) = f_1 : \tau_1, \dots, f_n : \tau_n}{\Gamma, \Sigma \rightarrow \Gamma[\pi.f_1, \dots, \pi.f_n / \pi : \tau : \omega], \Sigma} \quad (11)$$

The “fields” of a type  $\tau$  in this context are actual fields if the place in question is a struct type, but not all data-structures are structs in Rust. We consider reference types to have a single “field”,  $*$ , corresponding to a dereference. Aggregate types, like tuples or enums, have “fields” corresponding to their indices. This uniformity in notation allows for definitions to be more concise.

The algorithms for packing and unpacking are given in pseudo-code below:

Listing 4: Unpacking Algorithm

```

1 fn pack(pcs place) -> option<new-pcs>;
2   if place in pcs:
3     return pcs
4
5   longest-extension = longest
6   (filter (extensions place) in pcs)
7   longest-prefix = longest-extension[0..-1]
8
9   for field in (fields longest-prefix):
10    pcs = try
11      (pcs remove longest-prefix.field)
12    pcs = pcs add longest-prefix
13
14   return (pack pcs place)

```

Listing 5: Packing Algorithm

```

1 fn unpack(prefix pcs place) -> new-pcs

```

```

2   if place in pcs:
3       return pcs
4
5   pcs = try (pcs remove prefix)
6   pcs = pcs add prefix.field
7       for field in (fields prefix)
8
9   prefix = place[0..len(prefix)+1]
10  return (unpack prefix pcs place)

```

We determine whether or not to pack or unpack by checking if the current expression requires a place not in the EPCS. If so, and if that place is a prefix or extension<sup>5</sup> of a place in the PCS, then we attempt packing or unpacking accordingly to acquire the required capabilities:

Listing 6: Unpacking When Required

```

1   ...
2   for place required-by expression:
3       if place not in epcs
4           and one-of (prefixes-of place) in epcs:
5               try (unpack prefix epcs place)
6   ...

```

Listing 7: Packing When Required

```

1   ...
2   for place required-by expression:
3       if place not in epcs
4           and one-of (extensions-of place) in epcs:
5               try (pack epcs place)
6   ...

```

Note that the *try* statement here indicates that the operation may not succeed. In essence, if the place cannot be packed or unpacked from the current EPCS, then there is an error in the program. In practice, however, because we assemble the EPCS after most of the Rust compiler analysis steps, the compiler will catch these errors before they have a chance to occur here.

### B. Copy and Move Assignments

In Rust, RValue operands can be either moved out or copied, depending on whether they implement the Copy trait. In the case of a copy assignment, we adjust the PCS accordingly:

COPY:  $\pi' = Rvalue(\pi)$

$$\frac{\pi : \tau : \omega \in \Gamma \quad copyable(\tau)}{\Gamma, \Sigma \rightarrow \Gamma \cup \{\pi'\}, \Sigma} \quad (12)$$

COPY (REF):  $\pi' = Rvalue(\pi)$

$$\frac{\pi : \tau : \omega \in \Gamma \quad copyable(\tau) \quad \pi \rightarrow \lambda \in \Sigma}{\Gamma, \Sigma \rightarrow \Gamma \cup \{\pi'\}, \Sigma \cup \{\pi' \rightarrow \lambda\}} \quad (13)$$

For values that are moved out, the transformation is as follows:

MOVE:  $\pi' = Rvalue(\pi)$

<sup>5</sup>A prefix of a place  $\pi$  is a place  $\pi'$  such that  $\pi$  is a sub-place, as defined in Equation 5, of  $\pi'$ . An extension of  $\pi$  is a sub-place of  $\pi$

$$\frac{\pi : \tau : e \in \Gamma \quad noncopyable(\tau)}{\Gamma \rightarrow \Gamma[\pi / \pi']} \quad (14)$$

MOVE (REF):  $\pi' = Rvalue(\pi)$

$$\frac{\pi : \tau : e \in \Gamma \quad noncopyable(\tau) \quad \pi \rightarrow \lambda \in \Sigma}{\Gamma, \Sigma \rightarrow \Gamma[\pi / \pi'], \Sigma[\pi \rightarrow \lambda / \pi' \rightarrow \lambda]} \quad (15)$$

For RValues that contain dereferences (\*), we carry out those operations in the context of the EPCS before applying these transforms. See V-F

### C. Concrete Borrows

Concrete borrows refer to variables assigned borrows at most once at any given program point. In essence, this means borrow assignments that are not dependent on control flow. In this case, the EPCS tracks exactly what the reference points to for that borrow's lifetime.

The borrowing rules are similar for those of copy/move assignments. Indeed, moves could be modeled as exclusive borrows that are never returned, as in [3]. However, for this paper, moves, copies, shared borrows, and exclusive borrows are all treated as separate operations.

A shared borrow downgrades the place  $\pi$  to a shared capability, and adds the corresponding reference to the RCS:

SHARED CONCRETE BORROW:  $\pi' = \&\pi$

$$\frac{\pi : \tau : \omega \in \Gamma}{\Gamma, \Sigma \rightarrow \Gamma[\pi : \tau : \omega / \pi : \tau : s] \cup \{\pi'\}, \Sigma \cup \{\pi' \rightarrow \pi\}} \quad (16)$$

An exclusive borrow (e.g.  $x = \&mut y$ ) removes the place  $\pi$  being borrowed from the PCS, and also adds the corresponding reference to the RCS:

EXCLUSIVE CONCRETE BORROW:  $\pi' = \&mut \pi$

$$\frac{\pi : \tau : e \in \Gamma}{\Gamma, \Sigma \rightarrow \Gamma[\pi : \tau : e / \pi' : \tau : e], \Sigma \cup \{\pi' \rightarrow \pi\}} \quad (17)$$

### D. Borrow Expiry

To determine the liveness of borrows, we assume there exist oracle functions *borrow\_is\_live*( $p, \lambda$ ) and *region\_is\_live*( $p, \varrho$ ) that return *true* iff the borrow  $\lambda$  or region  $\varrho$  is live at program point  $p$ . Here we define a borrow to be “live” if it may be used to access the underlying place later in the program. Similarly, we define a region to be live if all of the borrows it encompasses are live. We assume both indefinite and definite references to be transferred to the LValue during assign statements, unless the borrow or region would naturally expire for other reasons (e.g. the borrow is no longer used afterwards).<sup>6</sup> We compute this liveness check at each statement for each reference. If a borrow has expired,

<sup>6</sup>Note that the specific implementation of the borrow checker facts may violate these assumptions. Indeed, preliminary testing of the EPCS API has shown that the Polonius borrow checker does not hold to these assumptions. See the issues in the supplementary Gitlab Repository [13] for details.

we remove it and restore of according capabilities before the next statement.

#### CONCRETE BORROW EXPIRY:

$$\frac{!borrow\_is\_live(p, \lambda) \quad \lambda = \pi' \rightarrow \pi \quad \lambda \in \Sigma}{\Gamma, \Sigma \rightarrow \Gamma \cup \{\pi\}, \Sigma \setminus \lambda} \quad (18)$$

#### INDEFINITE BORROW EXPIRY:

$$\frac{!region\_is\_live(p, \varrho) \quad \varrho = r[\Pi] \quad \pi' \rightarrow \varrho \in \Sigma}{\Gamma, \Sigma \rightarrow \Gamma \cup \{\Pi\}, \Sigma \setminus \lambda} \quad (19)$$

#### E. Indefinite References

When the target of a reference is not constant at a given program point because it occurs, for example, within the body of a loop, we approximate it in the RCS by a reference invariant. This reference invariant is a weakening of the value on the right-hand-side of an indefinite borrow assignment. It holds at entry and inductively throughout the body of the loop. For example:

Listing 8: Indefinite References

```

1 Node<'a> {
2   val: u8,
3   next: &'a Node<'a>,
4 }
5
6 fn simple_loop(m: Node) -> u8 {
7   let mut n = &m;
8   let mut v = &m.val;
9   while (n.val > 0) {
10    n = n.next;
11    v = &n.val;
12  }
13  return *v;
14 }
```

In the listing above, while  $n$  initially concretely references the input Node  $m$ , through the loop's body, that reference becomes indefinite; at line 11,  $n$  could refer to  $m.next$ , or  $m.next.next$ , etc. We list the weakenings of a definite reference targeting  $y$ , in decreasing specificity below:

$$y \subseteq subplace[y] \subseteq region[y, \dots] \quad (20)$$

Note that the differentiation between  $subplace[y]$  and  $region[y]$  is purely semantic, as they have no particular difference in meaning in the context of the EPCS itself. Indeed, the implementation discussed in Section VI-C does not distinguish between the two. Still, it is possible that differentiating between them may be important to downstream tools, and thus we include it here.

The EPCS transform assumes an oracle that calculates the reference invariant of a particular variable  $x$  at program point  $p$  in program  $P$ :  $ref\_invariant(P, p, EPCS, x)$ . Thus the transformation rules are the same as in Section V-C, but, instead of referring to  $\pi$  directly,  $\pi'$  refers to a region including  $\pi$  given by the  $ref\_invariant$  oracle function. The rule for shared indefinite borrows is given below as an example:

SHARED INDEFINITE BORROW:  $\pi' = \&x\pi$

$$\frac{\pi : \tau : \omega \in \Gamma \quad \varrho = ref\_invariant(P, p, (\Gamma, \Sigma), \pi')}{\Gamma, \Sigma \rightarrow \Gamma[\pi : \tau : \omega / \pi : \tau : s] \cup \{\pi'\}, \Sigma \cup \{\pi' \rightarrow \varrho\}} \quad (21)$$

#### F. Reborrows and Dereferencing

Reborrows are not a special case of EPCS transform. They behave identically to their indefinite or concrete borrow counterparts, adding a new reference to the RCS and adjusting capabilities as necessary. However, even if  $b$  is a reference, the following is NOT a reborrow because of Rust's auto-dereferencing:

Listing 9: Not a Reborrow

```
1 let a = & b.f // a.k.a. let a = &(*b).f
```

This observation is important for tracking the EPCS as it continues to evolve. If  $b$  is moved out after this line, the reference  $a \rightarrow (*b).f$  is actually still intact.

For example:

Listing 10: Dangling Reference RCS

```

1 let mut y = &x; // ( y -> x )
2 let z = &(*y); // ( y -> x, z -> x )
3 y = &w // ( y -> w, z -> x )
```

The transform at line 2 follows the dereference of  $y$  to  $x$  from the PCS after line 1. It does so instead of continuing to track the referred-to place in terms of  $y$  (i.e.,  $z \rightarrow *y$ ). This transformation is important to avoid “dangling” references as variables change throughout the program. In this example, we assign  $y$  a different reference at line 3. If we had kept  $z$ 's reference as  $z \rightarrow *y$ , the reference would be dangling because  $*y$  no longer points to the place to which  $z$  refers. In this context,  $z$  should still be a valid reference to what  $y$  was pointing to at the second line,  $x$ . We carry out these dereference operations insofar as is possible given the references tracked in the RCS, possibly multiple times if the RValue contains several dereferences. Similarly, for move and copy assignments, if the RValue contains dereferences, we must follow these dereferences to prevent dangling references.

This behavior is relatively clear for concrete borrows. For indefinite borrows, following a dereference operation can result in an indefinite result. Thus it is not obvious what the outcome of a second dereference on that result should be. We take this outcome to be the tightest indefinite reference that encompasses all possibilities (often a region that is a superset of places). In the example below,  $r[\alpha]$  and  $s[\alpha]$  are region and sub-place weakenings as discussed in Section V-E.

Listing 11: Following Indefinite References

```

1 // { ... } ( x -> r[w, y], y -> s[a], w -> s[b] )
2 z = &(*(*x)) // { ... } ( z -> r[a, b], ... )
```

We find the resulting region using the same reference invariant oracle as described in Section V-E.

### G. Control Flow Transfers and Merges

1) *Terminators*: Terminator instructions (e.g. *switchInt* etc.) typically result in copying the current EPCS to begin each basic block in the control flow graph that the instruction targets. We present a representative sample of terminator rules below:<sup>7</sup>

$$\text{GOTO } goto(p')$$

$$\frac{}{\Gamma, \Sigma \rightarrow_p \Gamma, \Sigma} \quad (22)$$

SWITCH INT (MOVE) *switch\_int*(*move*( $\pi$ ),  $p_1, \dots, p_n$ )

$$\frac{\pi \in \Gamma}{\Gamma, \Sigma \rightarrow_{p_1, \dots, p_n} \Gamma \setminus \pi, \Sigma} \quad (23)$$

CALL *call*( $f, move(\pi_{arg\_1}, \dots, \pi_{arg\_n}), \pi_{ret}, p_{ret}$ )

$$\frac{\pi_{arg\_1} \dots \pi_{arg\_n} \in \Gamma}{\Gamma, \Sigma \rightarrow_{p_{ret}} \Gamma \setminus \pi_{arg\_1} \dots \pi_{arg\_n} \cup \{\pi_{ret}\}, \Sigma \cup \{\pi_{ret} \rightarrow region(\pi_{ret})\}} \quad (24)$$

The  $\rightarrow_p$  and  $\rightarrow_{p_1 \dots p_n}$  notation indicates a transition to program point  $p$  or to program points  $p_1$  through  $p_n$ , respectively. Thus the left-hand side of the transition represents the EPCS state at the predecessor program point before the transition, and the right-hand side represents the EPCS at the point, or points, transitioned to.

2) *Merging*: Merging is necessary to unify the EPCS at a given point. For example, at the beginning of a basic block with multiple incoming edges, it is necessary to merge each incoming EPCS to obtain a single, path independent EPCS.<sup>8</sup> Merging the EPCS of two basic blocks involves taking the intersection of each PCS and the union of each RCS:

$$\text{MERGE: } merge(p, p')$$

$$\frac{}{\Gamma_p, \Sigma_p \rightarrow_{p'} \Gamma_p \cap \Gamma_{p'}, \Sigma_p \cup \Sigma_{p'}} \quad (25)$$

Keeping only the intersection of place capabilities is appropriate, as it only presents the capabilities definitively held at a given program point. Verification tools, for example, are typically interested in proving things based on certain capabilities. Thus if the analysis of capabilities is not sound—if *at least* those capabilities listed are not held—then verification based on those unsound capabilities may not be sound either.

<sup>7</sup>There are many kinds of very similar terminators in MIR, and enumerating each of them would be of little additional value to the selected sample presented here

<sup>8</sup>Even for a path-dependent EPCS, depending on the structure of the CFG, merging operations may be required

Conversely, we take references as a union. This choice is primarily to mirror the borrow checker’s soundness guarantees; a resource is blocked so long as the responsible borrow *might* be live. Again, this is conservative to ensure the soundness of the PCS; if a reference might be blocking a certain capability, for soundness, we must assume that it does. We calculate the union of indefinite borrows according to the set-union of the weakenings listed in Equation 20.

### H. Closures

We did not consider how closures impact the EPCS for this project.

## VI. IMPLEMENTATION

The implementation is primarily documented with `rustdoc` and can be found in the supplemental Gitlab Repo [13]. At a high level, the implementation consists of three components: a CLI, a wrapper around the Polonius borrow checker and Non-Lexical Lifetime (NLL) compiler outputs, and the EPCS API itself.

A few key design choices are discussed in this section, however.

### A. MIR

We constructed the EPCS API from Rust’s MIR rather than source code for several reasons. Foremost, the borrow checker operates at the MIR level, so performing the EPCS analysis on MIR allowed us to directly use the borrow checker output without mapping it back to source code first. Secondly, the reduced syntax and semantics of MIR simplified the analysis itself. Finally, the Rust compiler does provide mappings back to source Rust code in its MIR data structures. Thus, while not in the initial implementation of the API, it is possible to obtain a source-level EPCS from the EPCS of MIR code.

However, when analyzing MIR code, there are many *StorageLive* and *StorageDead* instructions generated for the compiler’s own static analysis. Because these instructions are known to have semantics that can be difficult to reason about [6], we have opted to ignore them in our implementation. We determine liveness information instead from the Polonius borrow checker, which contains facts for variable liveness and borrow liveness, as discussed in the following section.

### B. Borrow Checker Oracle

The oracle for determining the liveness of borrows is a thin wrapper around the Polonius [4] experimental borrow checker implementation. The Polonius engine takes in a series of Datalog facts emitted by the `nll-facts` compiler flag and computes several predicates relating to loan and region liveness. The EPCS API uses the Polonius engine as a dependency to compute and access this information.

### C. Reference Invariant Oracle

For the reference invariant oracle, we considered two proposals for the implementation.

a) *Option 1: Pulling the Invariant from the Borrow Checker:* Assuming that the borrow checker is sound, then any facts output from it within the loop body should be invariant across iterations. From lifetime/region liveness, loan-liveness, and the mappings from variables to regions and loans, it is possible to piece together an appropriate invariant for references in the EPCS.

Specifically, we determine concrete borrows by checking the region the LValue is associated with from the borrow checker’s output. We then recursively search that region and its sub-regions for a borrow instantiated at that program point, again using the facts output by the borrow checker. By associating the LValue with a “borrow id” we can query the borrow checker oracle as described in Section V-C for liveness information as necessary.

For indefinite borrows, there is no sub-region containing a borrow at the program point in question. Thus, instead of collecting information about a single loan, we return a collection of variables as being blocked by the LValue’s region. To do so, we again recursively search the sub-regions of the associated region.

We chose to use this option in the EPCS API because of its simplicity. It allowed us to forgo writing code to do an iterative weakening fixed point analysis, even if we have slightly less control over the kind of information we can determine.<sup>9</sup>

While not achieved in the initial version of the EPCS API, the validity of the invariant can be checked inductively by asserting that it is a fixed point with respect to the loop execution (and that it holds on entry).

b) *Option 2: Iterative Weakening Fixed Point Approximation:* Another proposed method of finding an appropriate reference invariant is to perform an iterative weakening of references created inside the loop body until reaching a fixed point.

For example the first iteration of the fixed point computation at a single line loop body might be:

```
// EPCS: { curr } ( curr->list )
curr = curr.next;
// EPCS: { curr } ( curr->list.next )
```

Because this is not a fixed point, the reference for the place assigned to is weakened according to Equation 20 on the next iteration:

```
// EPCS: { curr } ( curr->sub-place(list) )
curr = curr.next;
// EPCS: { curr } ( curr->sub-place(list) )
```

Now that we have reached a fixed point, we take this reference as the reference-invariant.

<sup>9</sup>It is non-trivial to determine whether or not to take e.g. *subplace*[ $\alpha$ ] as a reference invariant. As mentioned in section V-E, there is no difference from the EPCS computation point of view between this and a region-based weakening. Because of this, using the borrow checker information instead of rolling a custom fixed point analysis seemed to be a clear win

## D. Alternative Designs Considered: Datalog

This kind of program analysis tends to lend itself very well to Datalog. Indeed, the Polonius borrow checker [4] is implemented in Datalog via a Rust library. While we believe a Datalog implementation would be a very clean, natural solution, several practical considerations lead to the final decision to implement the EPCS computation in Rust, rather than in Datalog (or a Rust library utilizing Datalog constructs).

The primary reason for not using a Datalog implementation was that it was an additional layer of complexity in the implementation that was, ultimately, unnecessary for the task. One of the main attractions for using Datalog is that fixed-point computations are essentially “free” and relatively performant. The EPCS computation itself is relatively lightweight, as it mostly involves gathering data from several sources. And by opting against using a fixed-point computation to find reference invariants, there was little motivation to incorporate an additional layer of translation from a Rust representation to Datalog to perform the analysis.

## VII. FUTURE WORK

The EPCS API implementation and specification, as it stands, is still incomplete. Expanding the EPCS to produce summaries for Rust code containing closures, for example, would be an important next step.

Additionally, because EPCS is constructed from MIR code, adding a mapping of an EPCS back to Rust source code level would also be helpful, as it would allow for source-level linting or visualization tools to utilize the API. Indeed, building such analysis or visualization tools that leverages the EPCS API to provide information to developers or researchers would be an excellent way to exercise the possibilities the unified interface presents. This project could involve something like the lifetime visualization ideas discussed in [10], [2] or [9].

## REFERENCES

- [1] The rust reference.
- [2] Borrow visualizer for the Rust language service, Oct 2016.
- [3] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. In *to appear in Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2019.
- [4] Niko Matsakis et. al. Polonius. <https://github.com/rust-lang/polonius>, 2020.
- [5] Samin S Ishtiaq and Peter W O’hearn. Bi as an assertion language for mutable data structures. *ACM SIGPLAN Notices*, 36(3):14–26, 2001.
- [6] Ralf Jung. Exploring mir semantics through miri, Jun 2017.
- [7] Shuanglong Kan, David Sanan, Shang-Wei Lin, and Yang Liu. K-rust: An executable formal semantics for rust, 2018.
- [8] Eric Reed. Patina: A formalization of the rust programming language. *University of Washington, Department of Computer Science and Engineering, Tech. Rep. UW-CSE-15-03-02*, 2015.
- [9] Jeff Ruffwind. Graphical depiction of ownership and borrowing in rust, Feb 2017.
- [10] Jeff Walker. Rust lifetime visualization ideas, Feb 2019.
- [11] Feng Wang, Fu Song, Min Zhang, Xiaoran Zhu, and Jun Zhang. Krust: A formal executable semantics of rust. In *2018 International Symposium on Theoretical Aspects of Software Engineering, TASE 2018, Guangzhou, China, August 29-31, 2018*, pages 44–51, 2018.
- [12] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. Oxide: The essence of rust. *CoRR*, abs/1903.00982, 2019.

- [13] Dylan Wolff. Rust-eps. <https://gitlab.inf.ethz.ch/OU-PMUELLER/student-projects/rust-pcs/>, 2020.