Extended Place Capabilities Summaries for Rust Programs

Dylan Wolff

***Abstract*— The advent of the Rust programming language and its borrowing and ownership model have created new opportunities in Program Analysis and Verification. In order to take advantage of Rust's advanced type system, however, it is necessary to extract detailed information from both the Rust compiler and borrow checker. Once this data has been collected, it then must be assembled 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 relying on various internal compiler or borrow checker functions and data structures.**

I. Introduction

The Rust programming language is primarily characterized by its unique approach to static compiler checks and memory management. By enforcing ownership and borrowing restrictions for memory locations, the type-system of Rust precludes whole classes of memory saftey 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's internals. The Rust compiler project, however, is under active development and its internal API's have not been stabilized (with no immediate plans to do so). As a result, whenever changes are made to the Rust compiler, which is on a nightly release schedule, they propagate as breakages in the dependent projects. Compounding the problem, these changes are considered[1] "non-breaking" because the internal compiler API's have not been stabilized. 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 concern, we developed a comprehensive summary of the information needed by Prusti and other tools as a separate library. This localizes breaking changes to a single, separated component and allows for many applications to utilize the 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 is a summary of 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 result in adding the corresponding place to the PCS, as that variable now has permissions to access the memory location it refers to. While initially, it may exclusively refer to that place (and thus be mutable[2]), if that place is later borrowed by another variable, 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 be the combination of accessibility information from the PCS with additional data regarding borrows at that program point. Thus an EPCS is a representation of part of the program state before or during a given statement in a Rust program. This state is evolved by a set of

[1]This is not to suggest that the maintenance issues are the fault of the Rust compiler team. The API's used by Prusti are internal for good reasons, and it is, and absolutely should be up to the individual project owners to adjust to frequent changes in the API's.

[2]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

inference rules, expanded upon in Section V. Most of these rules dictate how each kind of statement changes an EPCS. Some, however, such as the "unpack" and "pack" rules rules defined in Section V-A, prepare an EPCS such that it is in a proper state for the statement rules to take effect. These rules are dubbed "Ghost Rules", and when/how to apply them is determined 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.

The inference rules are applied from an initial empty EPCS state for each entrypoint into the Rust program, following the Control Flow Graph (CFG) statement by statement until a fixed point is reached. In practice, this is achieved with a single iteration given the design choice outlined in Section VI-C. With this fixed point, the EPCS API can be queried at any statement within the CFG and give a sound 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]. Places can be represented as place expressions, which are paths that contain local variables, static variables, dereferences (*expr), fields etc. Rust's type system also has a notion of ownership and a notion of borrowing. Because of borrowing, the "owning" place expression may not actually be able to access its underlying memory location at a given program point. To capture this feature, 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 line 3 were not commented, the 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, temporary reduction in capabilities due to a loan and subsequent restoration of those capabilities, are thus key components in forming a sound PCS.

Prusti handles loans separately from computing a limited PCS, but, to provide an accurate PCS as a separate component, this analysis must be done together with the classic PCS enumeration. We define this additional information concerning which loans block/restore which capabilities as the Reference Capabilites Summary (RCS). Ultimately, because it is already necessary to compute the RCS to obtain a complete PCS, and also because we predict that it will often also be used in combination with the vanilla PCS, we include both the RCS and PCS together in what we call the Extended Place Capabilities Summary (EPCS).

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

ECPS are split into a PCS, Γ, and an RCS, Σ:

$$EPCS_p ::= \Gamma\ \Sigma \tag{1}$$

PCS are composed of a set of place expressions:

$$\Gamma ::= \{\} \mid \{\Pi\} \tag{2}$$

RCS are composed of a set of references:

$$\Sigma ::= \{\} \mid \{\Lambda\} \tag{3}$$

Place expressions, π, have a type, τ, and capability, ω:

$$\Pi ::= \pi : \tau : \omega \mid \pi : \tau : \omega, \Pi \tag{4}$$

Place expressions are composed of variables and their subplaces (fields and dereference operations):

$$\pi ::= x \mid \pi.f \mid *\pi \tag{5}$$

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

$$\Lambda ::= \lambda \mid \lambda, \Lambda \tag{6}$$

$$\lambda ::= \pi \to \pi : \tau : \omega \mid \pi \to \varrho \tag{7}$$

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

$$\varrho ::= region[\,] \mid region[\Pi] \mid subplace[\pi : \tau : \omega] \tag{8}$$

Capabilites are either shared or exclusive:

$$\omega ::= e \mid s \tag{9}$$

Note that, for concision, τ and ω are often omitted 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 λ corresponds to a reference that is live at program point p. These references can be concrete, referring to only a single place ($\pi \to \pi$) or indefinite referring to either a region, or a subplace of some known place π.[3]. Because regions are

[3]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

not typically used by tools outside of the borrow checker, and not particularly informative on their own, the regions as 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 elements on the right are blocked by the referrer on the left until that reference is no longer live, at which point, when the left element is released (e.g. the corresponding borrow expires), the elements on the right are restored.

The analysis performed to calculate the EPCS is sound with respect to capabilities held at a given program point. That is, a program never holds less 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 these capabilities on the right side of the arrow can be restored after line 4, simply by placing them directly back into the curly brackets of the PCS. In this case, by looking at the code snippet, it is evident that r always points directly to x. This is indicated by the right side of the reference being a concrete place expression rather than a region such as $r[x : e]$.

The above definition and explanation is 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. The path dependent EPCS 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 a way that they can be accurately reassembled into their parent places as required.[4]

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

[4]A "parent place" of π is a place expression π' for which pi is a subplace of π'

Listing 3: Packing and Unpacking

```
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. Packing and unpacking can be applied recursively to nested structures to disassemble/reassemble their capabilities appropriately.

More formally, for a place π of type τ with fields $f_i, ..., f_n$ unpacking of a capability ω in from the initial PCS, Γ, to the required PCS, Γ', is defined by the following rule:

UNPACK:

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

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

PACK:

$$\frac{\begin{array}{c}\pi.f_1 : \tau_1 : \omega, \ldots, \pi.f_n : \tau_n : \omega \in \Gamma \\ \pi : \tau \quad fields(\tau) = f_1 : \tau_1, \ldots, f_n : \tau_n\end{array}}{\Gamma, \Sigma \rightarrow \Gamma[\pi.f_1, \ldots, \pi.f_n \ / \pi : \tau : \omega], \Sigma} \tag{11}$$

It should be noted that the "fields" of a type τ in this context are its actual fields if it is a struct type, but this is not always the case. Reference types are considered 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

```
fn unpack(prefix pcs place) -> new-pcs
    if place in pcs:
        return pcs

    pcs = try (pcs remove prefix)
    pcs = pcs add prefix.field
            for field in (fields prefix)

    prefix = place[0..len(prefix)+1]
    return (unpack prefix pcs place)
```

Whether or not to pack or unpack can be determined by checking if the current expression requires a place that is not in the EPCS. If so, and if that place is a prefix or extension[5] of a place in the PCS, then packing or unpacking can be attempted accordingly to acquire the required capabilities:

Listing 6: Unpacking When Required

```
...
for place required-by expression:
    if place not in epcs
        and one-of (prefixes-of place) in epcs:
            try (unpack prefix epcs place)
...
```

Listing 7: Packing When Required

```
...
for place required-by expression:
    if place not in epcs
        and one-of (extensions-of place) in epcs:
            try (pack epcs place)
...
```

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 the EPCS is assembled after most of the Rust compiler analysis steps, these errors will be caught by the compiler 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 or not they implement the Copy trait. In the case of a copy assignment the PCS is adjusted accordingly:

$$\text{COPY: } \pi' = Rvalue(\pi)$$

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

[5]A prefix of a place π is a place π' such that π is a sub-place, as defined in Equation 5, of π'. An extension of π is a sub-place of π

$$\text{COPY (REF): } \pi' = Rvalue(\pi)$$

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

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

$$\text{MOVE: } \pi' = Rvalue(\pi)$$

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

$$\text{MOVE (REF): } \pi' = Rvalue(\pi)$$

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

It should be noted that, for Rvalues that contain dereferences (*), those operations are carried out 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. This allows the EPCS to track exactly what the reference points to for the lifetime of that borrow.

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

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

$$\text{SHARED CONCRETE BORROW: } \pi' = \&\pi$$

$$\frac{\pi : \tau : \omega \in \Gamma}{\Gamma, \Sigma \to \Gamma[\pi : \tau : \omega \,/\, \pi : \tau : s] \cup \{\pi'\}, \Sigma \cup \{\pi' \to \pi\}} \tag{16}$$

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

$$\text{EXCLUSIVE CONCRETE BORROW: } \pi' = \&mut\, \pi$$

$$\frac{\pi : \tau : e \in \Gamma}{\Gamma, \Sigma \to \Gamma[\pi : \tau : e \,/\, \pi' : \tau : e], \Sigma \cup \{\pi' \to \pi\}} \tag{17}$$

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 λ or region ϱ 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 definte a region to be live if all of the borrows it encompasses are live. Both indefinite and definite references are assumed 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).[6] This liveness check is computed at each statement for each reference, such that if a borrow has expired, the removal of it and/or restoration of according capabilities are carried out before the next statement.

CONCRETE BORROW EXPIRY:

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

INDEFINITE BORROW EXPIRY:

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

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, it is approximated 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, and holds at entry in addition to 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 body of the loop, that reference becomes indefinite; at line 11, n could refer to $m.next$, or $m.next.next$ etc. The weakenings of a definite reference targeting y, in decreasing specificity are listed below.

$$y \subseteq subplace[y] \subseteq region[y, ...] \tag{20}$$

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 it is included 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 π directly, π' refers to a region including π given by the $ref\_invariant$ oracle function. The rule for shared indefinite borrows is given below as an example:

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

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

F. Reborrows and Dereferencing

Reborrows are not a special case of EPCS transform. They behave identically to their indefinite/concrete borrow counterparts, adding a new reference to the RCS and adjusting capabilities as necessary. However, it should be noted that, 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 is important for tracking the EPCS as it continues to evolve. If b is moved out after this line, the reference $a \to (*b).f$ 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, rather than continuing to track the referred-to place in terms of y (i.e. $z \to *y$). This is important to avoid "dangling" references as variables are changed throughout the program. In this example, y is assigned a different reference at line 3. If z's reference was kept as $z \to *y$, it would be dangling, because $*y$ no longer points to the place z refers to. In this context, z should still be a valid reference to what y **was** pointing to at the second line, x. These dereference operations are carried out 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, they need to be followed to prevent dangling references as well.

This behavior is relatively clear for concrete borrows. For indefinite borrows, following a dereference operation can result in an indefinite result. Thus a second dereference on such a result is perhaps non-obvious. Ultimately, the new

[6]It should be noted that these assumptions may be violated by the implementation of the borrow checker facts. Indeed, preliminary testing of the EPCS API has shown that the borrow checker does not hold to these assumptions. See the issues in the supplementary Gitlab Repository [13] for details.

reference will simply 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 subplace 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], ...)
```

The resulting region can be found by 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 is a target of the instruction. A representative sample of terminator rules are presented below.[7]

GOTO $goto(p')$

$$\frac{}{\Gamma, \Sigma \rightarrow_p \Gamma, \Sigma} \tag{22}$$

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

$$\frac{\pi \in \Gamma}{\Gamma, \Sigma \rightarrow_{p_1, \ldots, p_n} \Gamma \setminus \pi, \Sigma} \tag{23}$$

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

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

Note that the $\rightarrow_p$ and $\rightarrow_{p_1 \ldots p_n}$ 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.[8] Merging the EPCS of two basic blocks involves taking the intersection of each PCS and the union of each RCS:

MERGE: $merge(p, p')$

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

Keeping only the intersection of place capabilities is appropriate, as it only presents the capabilities that are definitively held at a given program point. Verification tools, for example, are typically interested in proving things based off of 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.

Conversely, references are taken as a union. This 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. The union of indefinite borrows are calculated according to the set-union of the weakenings listed in Equation 20.

H. Closures

Closures are somewhat of a special case, and thus were not considered for the purposes of 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

The EPCS API was built using Rust's MIR rather than source code for a number of 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 first. Secondly, the reduced syntax and semantics of MIR simplified the analysis itself. Finally, while not in the initial implementation of the API, the Rust compiler does provide mappings back to source Rust code in its MIR data structures and it is possible to obtain a source level EPCS from this information.

When analyzing MIR code, however, 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. Liveness information can instead be determined from the borrow checker, which contains facts for variable liveness, in addition to borrow liveness as discussed in the following section.

[7] 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

[8] Even for a path dependent EPCS, depending on the structure of the CFG, merging operations may be required

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, two proposals were considered 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, concrete borrows are determined by checking the region the Lvalue is associated with from the nll-facts output. The region is searched for a sub-region containing a borrow instantiated at that program point, again using information from the nll-facts. This allows the association of the Lvalue with a "borrow id", which can be used to 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, a collection of all the variables that use each of the sub-regions in the search space are returned as being blocked by the Lvalue's region. This route was ultimately chosen primarily for 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.[9]

While not achieved in the initial version of the EPCS API, the validity of the invariant can be checked by asserting that it is a fixed point w.r.t. 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 an iterative weakening of references created inside the loop body until a fixed point is reached.

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 a fixed point has been reached, the reference is taken as the reference-invariant

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, there were several practical considerations that 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 that was ultimately unecessary for the task. More specifically, one of the main attractions for using Datalog is that fixed-point computations are essentially "free" and relatively performant. The EPCS computation itself is actually relatively lightweight, as it mostly involves gathering data from a number of sources. And by opting against using a fixed-point computation to find reference invariants, there was little need to incorporate an additional layer of translation from a Rust representation to Datalog to perform the analysis.

VII. Future Work

Completing the implementation of a tool that constructs EPCS for a wider swath of Rust code would be the logical continuation of this project. Also, expanding the inference rules to include behavior for closures would be an important extension of this work.

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 could involve something like the lifetime visualization ideas discussed in [10], [2] or [9].

[9] 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

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-epcs. https://gitlab.inf.ethz.ch/OU-PMUELLER/student-projects/rust-pcs/, 2020.