

Coupled Borrows

Modelling Rust's Aliasing Information with Capabilities

by

Markus de Medeiros

Co-supervisor: Alexander Summers (UBC)

Co-supervisor: Aurel Bîlý (ETH Zurich)

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE

in

The Faculty of Science

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

April 2023

© Markus de Medeiros 2023

Abstract

Prusti is a Rust verification tool which automatically infers a proof of memory safety from the Rust compiler internals. Prusti’s core inference was originally described in terms of a data structure called the *Place Capability Summary* (or *PCS*) which describes the *capabilities* the Rust program has to memory at each program point. The PCS is not implemented in the current version of Prusti; this thesis explores a method for inferring the PCS as a separate step from the rest of Prusti. In doing so, we define a new data structure we call the *Coupling Graph*, which establishes approximate aliasing relationships between places in a form that is straightforward to derive from the compiler. Our approach enables a PCS to be derived for programs involving reborrowing inside complex dataflow such as loops, which the release version of Prusti does not currently support.

Lay Summary

Rust is a programming language which checks strong safety properties about programs before they are run. Prusti is a tool which can verify much stronger properties of Rust programs by building off of the facts that Rust already has checked (called a *core proof*). Our work rethinks how Prusti infers core proofs, with the ultimate goal of robustly supporting more features from Rust in the tool.

Table of Contents

Abstract	ii
Lay Summary	iii
Table of Contents	iv
Acknowledgements	v
1 Overview	1
1.1 Background	3
1.1.1 Compiler Architecture	3
1.1.2 Owned data	3
1.1.3 The Polonius Borrow Checker	4
2 Owned Places	8
2.1 Places and Capabilities	8
2.2 The Free PCS	10
2.2.1 The Free PCS Operations	11
2.2.2 Defining The Free PCS	13
2.2.3 Optimization: removing mutability	14
3 Coupled Borrows	15
3.1 Borrow Coupling	15
3.2 The Coupling Graph	16
3.2.1 Calculating the Coupling Graph	17
3.2.2 Free PCS semantics for the Coupling Graph	20
4 Evaluation	30
4.1 Feature: Magic Wand Inference	30
5 Future Work	33
Bibliography	34

Acknowledgements

My deepest thanks to my co-advisors Alex Summers and Aurel Bîlý for their guidance at every stage in this project. I would also like to thank Jonáš Fiala, Federico Poli, and the rest of the Prusti team for their expertise and patience as I was learning about Prusti. Finally, I want to thank the rest of the wonderful people in the Software Practices Lab for the supportive environment and exciting conversations.

Chapter 1

Overview

Rust is a safety-oriented systems language whose strong type system ensures the absence of many common kinds of memory errors. Prusti^{[1][2]} is a deductive verifier for Rust which exploits Rust’s type system in order to automate proofs over Rust code with little annotation overhead. Prusti is written as a compiler plugin, and obtains its facts from directly from intermediate passes (notably the *Middle Intermediate Representation*, or *MIR*) in Rust’s compilation pipeline.

Rust is also a rapidly developing language whose compiler is largely underspecified. This poses a challenge for verification efforts; in the four years since Prusti was released it has accumulated several refactoring passes and patches to remain in sync with major changes to the compiler’s internal architecture. This poses several challenges for maintaining and growing Prusti. Patches which break abstraction boundaries end up leaking high-level information deep into Prusti’s pipeline, making the codebase brittle. Core algorithms (for example, the *pack/unpack algorithm*) become increasingly difficult for new contributors to understand and maintainers to debug. The implementation of new Rust features can be slowed by having to retool the major parts of the Prusti pipeline all at once.

Fortunately, Prusti was originally designed around a simple interface called the *Place Capability Summary* (*PCS*) which describes how a Rust program is permitted to interact with memory locations at each program point. The PCS expresses Rust’s *ownership* as a separating conjunction of so-called *accessibility predicates* in a fractional linear logic that can be encoded into the Viper verification language. Accessibility predicates are a logical Viper primitive that describe the *capability* we have to a place; mutable, writable, non-aliasing places are given the *exclusive* capability whereas immutable, read-only, possibly-aliasing places are given the *shared* capability. The original design for Prusti builds on a *core proof* of memory safety, which amounts to a translation of the Rust program into Viper such that the set of accessibility predicates entailed by the proof at each program point is

equal to the PCS at that point.

Currently, Prusti encodes the Rust program into Viper, leaving the PCS entirely implicit. The decision to deviate from the original design was made in a time when extracting the relevant information from the compiler was both a technical challenge and a low priority, however as both Prusti and the compiler have evolved the advantages of explicitly calculating a PCS are becoming clear. From a logical perspective, a mechanical derivation of the PCS gives a soundness criterion that the lower level passes can check themselves against. From a software engineering perspective, an explicit PCS calculation can serve as a reusable interface to the safety guarantees of the Rust compiler, separating concerns between reading compiler information and transforming those guarantees into a Viper proof.

In this thesis we present a model of the PCS which encapsulates the relevant parts of Rust’s program state for Prusti. To demonstrate its effectiveness, we will then apply this model to Rust programs that involve reborrowing inside loops, a feature which the release version of Prusti currently does not support. Our approach explicates the connection between the compiler’s information and our PCS by following this general strategy:

1. Understand the information Prusti needs to verify a core proof.
2. Link that precise information to an approximate version which is directly observable in the compiler.
3. Develop algorithms to fill the precise details back in.

We have attempted to design the PCS to be robust against some of the expected changes to the compiler in the near future. Several teams are actively working on formalizing the semantics of the MIR,¹ (the intermediate stage in the Rust compiler where Prusti’s translation begins) so we will not focus on that aspect of the project. We will also not attempt to accommodate unsafe code, though we are interested in exploring this as an application of our model in the future.

Our model is deeply linked to Polonius[4] which we treat as the canonical source of truth for the state of the borrow checker (in contrast to tools such as Aeneas[3] which reimplement borrow checking in their model). As

¹This includes our own *MicroMIR*, a fine-grained sequentialization of MIR statements we have been developing since the summer of 2022.

Polonius is under active development, we have made an effort to minimize our dependencies on the unstable aspects of that project. Furthermore, we have attempted to avoid leaking Prusti-specific encoding details into the model; we hope that our model can be generally useful to other projects that read and reuse capability-like information from the compiler.

1.1 Background

In this section we will provide a high-level summary of our observations about the parts of the compiler which pertain to memory safety, and introduce the terminology that we will use for the remainder of the thesis.

1.1.1 Compiler Architecture

The Rust compiler[6] uses multiple passes to lower Rust source code into its target language, LLVM IR. Of importance to Prusti are the *HIR* and *MIR* (*High- and Mid-level Intermediate Representation*) intermediate languages. Both Prusti’s translation into Viper and Polonius’s borrow-checking operate on the MIR before optimization passes are applied. Prusti also uses the *HIR* to encode its functional specifications lower down in its pipeline, so it is important that we maintain a connection the compiler by reusing its data structures and terminology where we can.

The MIR is organized into a control-flow graph of *basic blocks*, each of which consists of a sequence of *statements* that ends with exactly one *terminator* (some passes in the compiler relax this requirement). Each statement and terminator is uniquely identified by a *location*. Up to date information about the individual MIR statements can be found in the MIR documentation[7].

1.1.2 Owned data

The basic unit of memory at the MIR level is a *place*, which consists of a *local* and a sequence of *projections*. Locals which are allocated during the execution a MIR program have their live regions demarcated by `StorageLive` and `StorageDead` statements; any other locals are presumed to be live across the entire program. It is assumed that distinct locals do not alias; however, their projections (namely `deref`) might.

A subset of the places which appear syntactically in the program are assigned unique identifiers called *MovePaths*. As an optimization, the compiler only calculates liveness for places with *MovePaths*; this is an issue for us since the PCS must convey accessibility to places which do not occur in MIR source.² While we could obtain the compiler’s canonical liveness calculation by rewriting the Rust source program to include no-ops which reference the places that we might need to know the liveness for, we instead choose the zero-cost solution by generalizing and redoing Rust’s liveness calculations ourselves.

Like moves in Rust source programs, moves at the MIR level are assumed to deinitialize their moved-from places, as compared to copies which do not. The compiler performs standard fixed point analyses on the set of *MovePaths* to determine the ranges where they are inhabited by live values. Some *MovePaths* structures will require additional pieces of low-level code called *drop shims* to be inserted at drops where the compiler cannot statically deduce if a struct is fully initialized (specifically, if a *MovePath* is both *MaybeInitialized* and *MaybeUninitialized*). A consequence of Prusti and Polonius operating on the unoptimized MIR is they cannot access a version of the program which fully elaborates how variables are dropped,³ meaning we will have to make some simplifying assumptions about drops in Chapter 2. For more precise information about how the fixed point analyses are calculated, please refer to the `rustc_mir_dataflow` crate in the compiler or sections 40 through 50 in the compiler development guide[6].

1.1.3 The Polonius Borrow Checker

Polonius[4] is an experimental borrow checker which aims to replace the existing borrow checking code in the compiler. Polonius is implemented in Datalog; this is both fortunate and unfortunate for us. On one hand, we can reason about our model using the simple logical rules outlined as Horn clauses in Polonius’ implementation. On the other, it means much of the complexity of borrow checking is pushed into the compiler’s generation of

²A common example would be when a Prusti user asserts that an unused field in a struct is unchanged by the body of a loop. Because the struct field is unused the compiler will not generate a *MovePath* for it, even though the fact that the field is not changed (or even dropped) during the loop body may be crucial to a successful Viper proof.

³The Polonius team is considering lowering their analysis for their next major revision, so this may change in the future.

Polonius input facts which is both hard to understand and highly unstable. In this section, we will provide a brief overview of the parts of Polonius which are relevant to our model. We will not precisely classify how these facts are generated from the MIR.

The relevant atoms used by the Polonius analysis are

- **Loans:** unique identifiers for every borrow syntactically created by the program (that is, every MIR statement of the form `x = &y` or `x = &mut y` has an associated loan). At these program points, we say a loan is *issued*.
- **Origins:** Analogues of lifetimes from Rust’s surface language[5] which are fully elaborated by the compiler.
- **Locations:** A `Start` location and `Mid` location for each MIR statement and terminator, corresponding respectively to the point before and after a statement takes effect.


Every borrow-typed place in a MIR program is generated a unique origin. Every time a loan is issued, the compiler generates an *issuing origin* which lives only as long as the right-hand side of that assignment. Fresh origins are also provided for user type annotations and function arguments, but this version of our analysis does not support these features yet.

Loans are *invalidated* when their borrowed-from places are used in a way that is only legal if the loan is no longer live (written to, or read from in the case of mutable borrows). All invalidations of a loan are encoded in the `loan_invalidated_at` fact. Using a fixed-point analysis over the MIR control-flow graph Polonius seeks to prove that every borrow’s lifetime is definitely over before it is possibly invalidated, in which case the program is said to *borrow check*.⁴ As a first step, Polonius calculates a set of points in the control-flow graph for each borrow-typed values where that value is live, in the sense that it is both initialized and possibly used in the future.⁵ At runtime every initialized, borrow-typed place will alias some loan. To determine if a program borrow checks, Polonius must calculate the possible loans any given origin may alias at each program point, and then finally

⁴It also seeks to prove that declared relationships between lifetimes at function boundaries are respected, but our model does not use this fact yet.

⁵Currently, Polonius recomputes this itself, even though the data is accessible through the compiler.

check that none of them are invalidated.

Thus, determining how loans flow between origins is the main content of the Polonius analysis. The main dataflow facts in Polonius are the `subset_base` and `loan_killed_at` facts, which are generated as Polonius inputs by the compiler. An origin is a *subset* of another origin at a point when all loans in the subset must also be contained in the superset at that point. Dataflow between origins induces subsets; for example,  origin of an assigned-from place is a subset of the origin of the corresponding assigned-to place. Origins are explicitly *not* considered subsets at a point if the sets of loans they may alias are subsets by coincidence: at every program point, Polonius calculates an intermediate `subset` fact which represents the transitive closure of all declared subset relationships between live origins which serves to characterize the origins which *must* be subsets of each other at that point.⁶ For each origin and at each program point, the set of loans an origin may alias is updated and encoded in the `origin_contains_loan_at` fact. Polonius uses the `subset` fact to update the `origin_contains_loan_at` fact by propagating all the loans from subset origins into their superset origin at every point.

Finally, there is a subtlety regarding reassignments of reborrows which will be important for our analysis later: When a borrowed-from place is reassigned, invalidations of that place become permissible again. Polonius models this by removing that loan from all live origins as regulated by the `loan_killed_at` input fact.⁷ The archetypal example of this behaviour happens when a user reassigns a reborrowed-from place, which we present an example of in Program 1.1.

Polonius is necessarily an approximation of the real aliasing relationships in a program. Mirroring the approximations of owned data, Polonius takes a set union at each join point to determine the set of loans an origin possibly aliases. Note the similarities between the approximations performed for the Polonius analysis and those for owned data: borrow checking is no more fine-grained (or value dependent, for that matter) than the initialization

⁶Calculating this *subset* fact causes a known issue in Polonius: the borrow checker is supposed to differentiate between *immediate* subsets that flow one set of loans into another at a point, and *persistent* subsets which establish a subtype relationship between origins for their entire duration. Currently all subsets are interpreted as persistent in Polonius.

⁷More precisely Polonius does not remove loans from origins, it just flags them and does not propagate them to the next program point.

Program 1.1 An example of killing a borrow. At the conclusion of this program, `y` is still a borrow of the data `5`, but it no longer reborrows through `a` and thus invalidations of the borrow on line 2 are permitted.

```

1 let mut a = &mut (5 : u32);
2 let mut y = &mut (*a);
3 a = &mut (6 : u32);

```

checking described in Section 1.1.2. This correspondence will simplify how owned places (Chapter 2) and borrows (Chapter 3) interact in our model.

Finally, we make the following simplifying assumption about the fact generation in the compiler:


Definition 1.1.1 (Origin Characterization). Each Polonius origin can be characterized as either a *borrow temporary*, associated to the right-hand side of an issue of a borrow, or an origin for a *place*. We will use $lhs(o)$ to represent the place or borrow that an origin o represents.

This is not strictly true of all origins generated from Rust code. For one, a program can have *universal origins* and corresponding *universal subsets* that encode the relationships between borrow-typed function arguments. Furthermore, user-supplied type information (which occurs at type ascriptions, function invocation, and the aggregation of structs that contain borrows) can introduce additional origins that our model does not currently handle. We will restrict ourselves from using these features for now, and halt our analysis if we encounter an origin or `subset_base` fact which we cannot characterize according to Definition 1.1.1.

Chapter 2

Owned Places

As in the original design of Prusti[1] we seek to represent the ownership state at each program point as a set of *resources* called the *Place Capability Summary* (abbreviated *PCS*). To accomplish this we will define a new data structure called the *Free PCS*, which should be thought of as an approximation of the PCS that is straightforward to read off from the compiler.

A trace of the PCS across the execution of a MIR program is a constituent of the *core proof* for that program which the rest of the Prusti annotations will build on top of. The PCS can be viewed as a soundness test for programs which impose their own operational semantics on the MIR; for Prusti this means we should be able to **assert** the PCS at each program point in our translated Viper program. Importantly, this is a set of responsibilities that the compiler-provided information from Section 1.1.2 is not able to fulfill on its own: Rust does not track whether a user can access places that they do not reference in the source program (only those which have `MovePaths`), and the compiler's analyses do not distinguish between capability for a struct  capability for all of its fields. We must reconstruct this information to fully explain the program's execution in terms of permissions, and consequently, to enable Viper to automatically verify core proofs.

2.1 Places and Capabilities

Recall the definition of a *Place* from Prusti[1]:

$$p ::= x \mid p.f \mid (*p)$$

In the compiler MIR places are represented by a list of `ProjectionElems` from a base local, of which the Prusti model only supports `Deref` and `Field`. In our model, we will extend the notion of a place with support for the `Downcast` projection, though there are other projections which relate to

2.1. Places and Capabilities

features we do not support yet.

$$p ::= x \mid p.f \mid (*p) \mid p \text{ as } \tau \quad (2.1)$$

This extension serves two purposes. For one, downcasts are common in real Rust code, including some of the programs we will seek to calculate a PCS for later in this thesis. Another reason is to eliminate the slightly misleading idea that a place always has a unique unpacking at the MIR level: unlike product types such as structs which only have one legal PCS to unpack to, a sum type such as an enum can unpack to any one of its downcasts. The fact that an unpack of a place is not always uniquely specified by the place being unpacked is a meaningful change to how unpacking is thought of in the original presentation of Prusti[1].⁸ We believe our relaxation of unique unpacking is a key step towards supporting the remaining unimplemented projections (`Index`, `ConstantIndex`, `Subslice`, and `OpaqueCast`) in the future.

Our model also is designed to support reasoning about allocated but uninitialized places with resources. An uninitialized place can be thought of as having permission to write to a place, but not having permission to access that place's value. We say the former capability is *shallow*, whereas the latter is *deep*. This change has the advantage of bringing our model closer to parity with the compiler-provided information, and it will also become important for differentiating reborrows from iterated borrows or moves of borrows in Chapter 3.

Definition 2.1.1 (capabilities). A **capability** p is one of

$$p ::= E \mid S \mid e \mid s \quad (2.2)$$

We call E and S **deep** capabilities, and e and s are **shallow**. Furthermore, we introduce two functions to extract a capability associated to the declared mutability of a place:

$$\begin{aligned} \text{Deep}(x) &= \begin{cases} E\ x & x \text{ is mutable} \\ S\ x & x \text{ is immutable} \end{cases} \\ \text{Shallow}(x) &= \begin{cases} e\ x & x \text{ is mutable} \\ s\ x & x \text{ is immutable} \end{cases} \end{aligned} \quad (2.3)$$

⁸Furthermore, **due to undefined behavior** there are some situations where several distinct lists of projections can describe the same place in memory (for example, a field of ADT x with just one variant V may be described as either $(x \text{ as } V).0$ or $x.0$).

In the nomenclature of the original Prusti paper, the capabilities E and e entail *exclusive* (read and write capabilities that do not alias other PCS members) capabilities to a place whereas the capabilities S and s are *shared* (read-only capabilities that might alias other shared PCS members).

Following the Prusti paper, a **PCS** (place capability summary) is a set of capability-place pairs r interpreted as follows:

- The *value* of exclusive capabilities must not overlap in memory with the *value* of any other capability in the PCS.
- Shared capabilities may refer to the same *values* as another capability in the PCS, though we restrict the PCS from holding redundant capability to the same *place*.

The PCS is easiest to understand through examples. If x is a struct then $\{E\ x, E\ x.f\}$ is not a legal PCS because the value of x and the value of $x.f$ always overlap in memory. Conversely, we can infer from the PCS $\{E\ x, E\ y\}$ that x and y must not be aliases to the same piece of data. The definition precludes holding both exclusive and shared capabilities to the same place, as in $\{E\ x, S\ x.f\}$. It also disallows the PCS $\{S\ x, S\ x.f\}$, since the place $x.f$ is a subplace of x , though $\{S\ x\}$ is likely to be an acceptable alternative in this case. It is permissible for x and y to be aliases in the PCS $\{S\ x, S\ y\}$, and all of these examples apply in the same way to shallow permissions as well.

We write $x*y$ to denote that two places x and y satisfy these properties. A PCS then can be represented as a set of places, which is the notation we prefer in this thesis, or as a separating conjunction of capabilities as per Prusti[1].

2.2 The Free PCS

As mentioned in our introduction a trace which assigns a PCS to each program point is too granular to easily observe in the compiler; for initialization purposes the compiler does not distinguish between a struct and the set of all of its fields, nor can it query the state of places that are not MovePaths (places which Prusti users may nevertheless want to write assertions for). Our approach for PCS inference is as follows:

1. Define a new data structure, the *Free PCS*, which approximates together aspects of the PCS which the compiler does not distinguish.
2. Observe a trace of the free PCS from a simple fixed point analysis.
3. Reconstruct the fine-grained information by inserting *pack* and *unpack* annotations into the program.

Historically, an analogue to step 3 has been handled by the **notoriously hard to read *fold/unfold* algorithm in Prusti** (and more recently referred to as *refolding* in [2]). The Free PCS relieves this burden from the passes lower down in Prusti’s pipeline: any encoding which doesn’t violate the free PCS can have the rest of a Viper proof derived automatically, and the free PCS lays out Prusti’s assumptions about Rust’s semantics.

2.2.1 The Free PCS Operations

We define our *PCS Operations* similarly to Prusti’s original design[1].

Definition 2.2.1. (Free PCS Operations) A **free PCS operation** is one of

$$\text{unpack}(r) \quad \text{pack}(r) \quad \text{drop}(r) \quad (2.4)$$

where r is a capability-place pair.

The principal difference to the operators defined in [1] is that we have replaced their *remove* operator with *drop*, to reflect a major change in its semantics.

Definition 2.2.2. (Semantics for free PCS operations) We define the semantics of a free PCS operation using a family of Hoare triples as outlined in Figure 2.1, and we lift a triple $\{X\} f \{Y\}$ to a relation of type $P \rightarrow P$ using framing in the usual way:

$$\begin{aligned} f(X) &= Y \\ f(X * w) &= Y * w. \end{aligned}$$

As in Prusti’s original presentation[1], the *pack* and *unpack* operations are inverses, and structs can unpack to the set of their fields. We deviate from the original design[1] with the inclusion rule for downcasts, and also in our rule for unpacking borrows. As mentioned, this enables our model to delineate between reborrows (`&mut (*x)`) and iterated borrows (`&mut x`). More specifically the latter should block write access to `x` while the former

$$\begin{array}{c}
 \frac{\{Z\} \text{ pack } (p \ x) \ \{p \ x\}}{\{p \ x\} \text{ unpack } (p \ x) \ \{Z\}} \qquad \frac{\{p \ x\} \text{ unpack } (p \ x) \ \{Z\}}{\{Z\} \text{ pack } (p \ x) \ \{p \ x\}} \\
 \\
 \frac{}{\{ \text{Deep } (x) \} \text{ unpack } (\text{Deep } (x)) \ \{ \text{Shallow } (x), \text{ E } *x \}} \ (x : \&\text{MUT } T) \\
 \\
 \frac{}{\{ \text{Deep } (x) \} \text{ unpack } (\text{Deep } (x)) \ \{ \text{Shallow } (x), \text{ S } *x \}} \ (x : \&T) \\
 \\
 \frac{}{\{p \ x\} \text{ unpack } (p \ x) \ \{p \ x.t1, \dots, p \ x.tn\}} \ (x : T, T \text{ HAS FIELDS } T1, \dots, Tn) \\
 \\
 \frac{}{\{p \ x\} \text{ unpack } (p \ x) \ \{p \ x.\text{downcast}(V, -)\}} \ (x : V) \\
 \\
 \{ \text{Deep } (x) \} \text{ drop } (\text{Deep } (x)) \ \{ \text{Shallow } (x) \} \qquad \{e \ x\} \text{ drop } (e \ x) \ \{s \ x\}
 \end{array}$$

Figure 2.1: Operational semantics for the free operations

should not; In our semantics for borrows the latter case shallow exclusive permission will remain in the free PCS whereas in the former it will not.

As per Section 1.1.2, our analysis operates on the MIR before drop-elaboration and thus we must make some simplifying assumptions about drops. For one, we will assume that all places which are not definitely initialized have their values dropped eagerly (this typically occurs at a join point in the control-flow graph). We also assume that drops have no side effects,⁹ so it is legal to drop a place anywhere between a point where it is not live, and where it is definitely dead. Under these assumptions, we can perform our own dataflow analysis on a MIR program to reconstruct liveness information for places. Our analysis computes live regions for all places, so that the PCS can query information about places not explicitly referenced by the MIR source. Our analysis also disambiguates some cases of known undefined behavior regarding duplicated `StorageDead` statements; this part of the model serves as the interface to the compiler and thus can adapt to the changing semantics of the MIR.

2.2.2 Defining The Free PCS

We formalize the approximations described in the prior sections as a pre-order on the set of legal PCS's.

Definition 2.2.3. (Free PCS) Let \lesssim be the preorder on P , where $P_0 \lesssim P_1$ when there exists a sequence of free PCS operations (whose semantics are described by Definition 2.2.2) which transform P_1 to P_2 when composed.

In the usual way, we define an equivalence relation \sim on P where $P_1 \sim P_2$ when $P_1 \lesssim P_2 \lesssim P_1$. Define the set of **Free PCSs** to be the quotient $F = P / \sim$. We can define a partial order $<$ on F by lifting \lesssim through the quotient in the usual way: $F_1 < F_2$ when $F_1 \lesssim F_2$ and $F_1 \not\sim F_2$.

One should think about \sim in this partial order as being equatable by some sequence of `pack` and `unpack` statements, and $<$ as some loss of capabilities (without deallocation). Our analysis is predicated on the following claim:

⁹This is true for code devices which are memory-constrained, and for Rust programs which do not write *drop handlers*

Claim 2.2.1. *For every MIR program **generated from safe Rust**, there exists a Free PCS trace $i \mapsto P_i$ where for every program point $mid(j)$ which has an edge in the control flow graph to $start(i)$, we have $P_{start(i)} \lesssim P_{mid(j)}$.*

This claim is partly an assurance that the compiler will not generate memory-unsafe MIR, and partly a promise that our operational semantics for each MIR statement will respect memory safety. For example, we assume that the operational semantics will not eliminate capabilities that we may still need in the future, or will not depend on a packed capability in such a way that an equivalent unpacked capability does not suffice. This claim allows us to assume that a trace of the free PCS exists for every legal Rust program.

We prototyped algorithms (based on unification) to construct an explicit sequence of free PCS operations in the summer of 2022.

2.2.3 Optimization: removing mutability

Note that it is sound to ignore mutability altogether, declaring

$$\text{Deep}(x) = E\ x \quad \text{Shallow}(x) = e\ x \quad (2.5)$$

instead of reading the mutability as in Equation 2.3; this optimization is sound as it amounts to rewriting the input program such that all locals are declared as `mut`, which is always permitted. We note this optimization because reading the declared mutability of a MIR local is currently challenging to access at the MIR level, and because it may be desirable for verification backends such as Prusti to not redundantly verify this aspect of the type system.

Chapter 3

Coupled Borrows

Our strategy for integrating borrows into this model is similar to that which led us to the free PCS. Historically, Prusti has tracked aliasing relationships between places in the *reborrowing DAG*, a directed acyclic hypergraph where each edge connects sets of capabilities that can be exchanged for each other during the course of the core proof. Like the free PCS, this information is too granular to read off of the compiler, so we build off of approximate information that we can observe instead.

3.1 Borrow Coupling

Following on from the claim made in Definition 1.1.1, we might hope to reconstruct a reborrowing DAG by reading the *outlives* (or `subset_base`) constraints as edges in the graph. Unfortunately, because Polonius is an approximate analysis, even with Definition 1.1.1 we cannot reconstruct the order of edges in a reborrowing graph just by inspection of the facts themselves. Consider for example Program 3.1. At the start of line 9, both variables `x` and `y` are initialized with live borrows, but statically we (and Polonius) do not know which. In this situation, the Polonius rules lose precise aliasing information and approximate each of `x` and `y` to alias *either* `t0` or `t1`. At the start of line 10 when the place `x` is no longer live, the compiler cannot soundly regain capability for either `t0` or `t1` even though it certainly must give up permission to the dead borrow `x`! Thus, if we intend for the PCS to represent the set of permissions which are accessible at a given program point then it must also contend with expiring borrows that we only have approximate aliasing information for.

We say that borrows are *coupled* when they are possibly aliased behind the same place. In Program 3.1 the borrows `bw0` and `bw1` are coupled at the join point of the conditional, because Polonius does not track data-dependent borrows.

3.2. The Coupling Graph

Program 3.1 Swapping borrows

```
1 fn swap(mut t0: T, mut t1: T, b: bool) {
2     let mut x = &mut t0; // borrow bw0
3     let mut y = &mut t1; // borrow bw1
4     if b {
5         let tmp = x;
6         x = y;
7         y = tmp;
8     }
9     let last_usage_x = x;
10    let last_usage_y = y;
11 }
12 struct T {} // non-copy type
```

3.2 The Coupling Graph

We will represent our interpretation of the aliasing information in a data structure we call the *coupling graph*, which can be thought of as a dataflow-insensitive approximation of the reborrowing DAG. First, we outline a *model-level* presentation of this data structure, which is straightforward to define and similar to a *Reborrowing DAG* structure used internally by tools such as Prusti and Aeneas. Later on, we will present this data structure in another way which is simpler to infer from Polonius.

Definition 3.2.1. (Coupling Graph, Model-level)

A **linear resource** is one of

- a place-capability pair
- a place-capability-location triple (called *tagged* resources), or
- a Polonius *loan* atom.

A **Coupling Graph** is a directed acyclic hypergraph on the set of tagged resources, where each hyperedge connects two legal PCS's. The directed edges in this graph are described by one of the rows in Figure 3.2.1: an edge annotated with an *Annotation* is directed from the set of *From* vertices to the set of *To* vertices.

3.2. The Coupling Graph

Annotation	From	To
Move(from, to)	{ from }	{ to }
Reborrow(from, to)	{ from }	{ to }
Borrow(loan, from)	{ from }	{ loan }
Pack(from, to)	from	{ to }
Unpack(from, to)	{ from }	to
Coupled(pre, id, post)	pre	post

Figure 3.1: Edge annotation rules for the coupling graph. Curly braces indicate that their contents are a single vertex, rather than a set of vertices.

Returning to step through our example, Figure 3.2 displays the coupling graph at several points in Program 3.1. At the start of line 4 we have complete information about which places alias `t0` and `t1`. At the start of line 9 we know that both of `x` and `y` are aliasing `t0` and `t1` but not which aliases which. Finally, at the start line 10, we know `x` is no longer aliasing `t0` or `t1` since it has been dropped, but we still cannot regain access to `t0` or `t1` while `y` is accessible. When `y` dies at the start of line 11, the coupling graph is cleared and the PCS can regain capabilities for both `t0` and `t1`.

In this thesis we will not present a formal connection between the coupling graph and the *reborrowing DAG*, a similar data structure in Prusti which encodes aliasing relationships between places. We note that a coupling graph can be seen as a fully approximate version of the reborrowing DAG which treats each edge as an opaque exchange between sets of capabilities, though this level of information loss may not be sufficient to verify core proofs in practice.¹⁰

3.2.1 Calculating the Coupling Graph

Unfortunately, while the presentation of the coupling graph in Section 3.2 is straightforward to interpret as a reborrowing DAG, it does not readily admit an inference algorithm. To rectify this, we use a different representation of the same mathematical object that is more tightly linked to the Polonius facts themselves.

Definition 3.2.2. (Coupling Graph, Representation) Let M be a mapping from live origins to directed acyclic hypergraphs on the set of capabilities

¹⁰We are currently exploring a flow-dependent extension to this system as one possible solution to this issue.

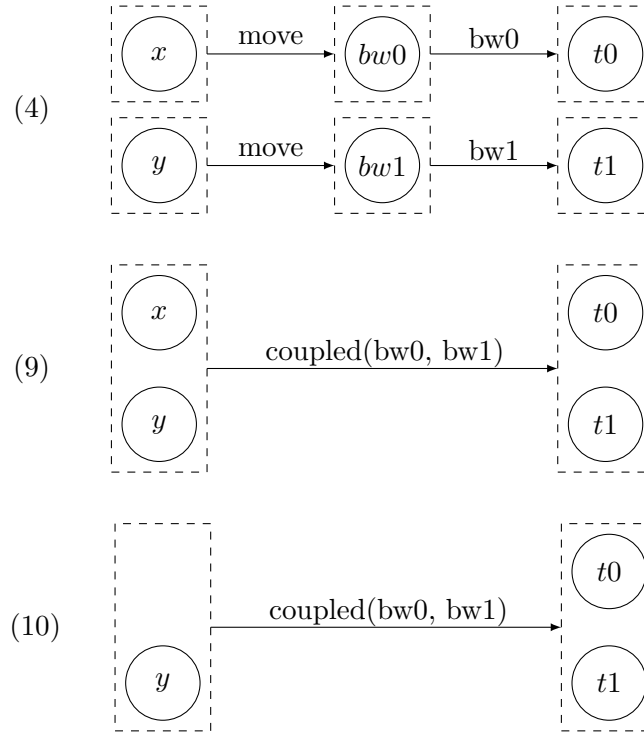


Figure 3.2: Model-level coupling graphs for Program 3.1 at the start of lines 4 (after the initialization of x and y), 9 (after the `if` block), and 10 (after borrow x expires).

3.2. The Coupling Graph

```
#7r: {E *_5} --* {E _1, E _2}
      {E *5} -coupled(cb1) -> {E _1, E _2}
#6r: {E *_4} --* {E _2}
      {E *4} -coupled(cb1) -> {E _1, E _2}
```

Figure 3.3: Implementation-level coupling graph for Program 3.1 at the start of line 9.

such that $leaves(M(o)) \sim lhs(o)$. For brevity, we extend the legal edge annotations to include $shared(o)$ for origins o whose *from* and *to* vertices are the leaves and roots of $M(o)$, respectively (this edge should be interpreted as the entire graph $M(o)$ in its place).

The coupling graph is the smallest directed acyclic hypergraph which contains the entire range of this map, where **Coupled** edges with the same **id** are joined by taking the union of their from- and to-nodes.

This representation of the graph makes a more explicit connection between lifetimes (or *origins*) and the edges they represent. Every lifetime in a program, including those which are left implicit from the source program, represents a particular subgraph of the coupling graph. If one lifetime outlives another, and they possibly alias the same place, the longer-lived lifetime will be a subgraph of the shorter-lived one.

The motivating difference between the coupling graph in its model-level and represented form is about the representation of edges. More precisely, a single edge in the model-level coupling graph can correspond to multiple edges in different origins at the implementation level. Thus a place can be blocked by several lifetimes, requiring all to end before the place is unblocked, as in our coupling example. Compare the coupled edge in Figure 3.3 to its representation in Figure 3.2: the single coupled edge at the model level becomes two edges in our representation. Each edge in the representation-level must be expired before we can interpret the edge as expired at the model-level.

The textual representation in Figure 3.3 comes from our implementation (Chapter 4) but we will clarify how to read it here as it is used extensively in the following section. Each origin has its associated subgraph described by a

list of edges¹¹. Its leaves and roots are described using the syntax `{leaves} --* {roots}`, which we call its *signature*, and is intentionally similar to the *magic wand* from separation logic. `shared` edges derive their to- and from-nodes from this signature.

3.2.2 Free PCS semantics for the Coupling Graph

The coupling graph is designed around the following principles, which will govern its interaction with the free PCS.

1. The leaves of the coupling graph are always contained in (an equivalent form under \sim of) the Free PCS.
2. To perform any operation in the coupling graph, the Free PCS must remove any leaves which are eliminated and regain any leaves which are exposed.
3. The coupling graph can only contain deep capabilities.

We will outline the semantics of the coupling graph alongside our current running example, though we will be brief about parts which only relate to the operational semantics for owned places. Program 3.2 is a slightly simplified version of MIR for our example; we will introduce the relevant Polonius facts from the compiler’s debug output as we step through the program.

Issuing Borrows

We begin at location `Start(bb0[0])` with an empty coupling graph, and a free PCS

$$\{E_1, E_2, E_3\}$$

that contains exclusive capability to `t0`, `t1`, and `b` respectively. Next, the local `_4`, representing `x`, comes into scope with a shallow permission

$$\{E_1, E_2, E_3, e_4\}$$

before a new borrow `bw0` of `_1` is assigned to it at `bb0[1]`. Here, Polonius reports one `subset_base` for the assignment; the issuing origin `#2r` (which is the issuing origin of `bw0`), is a subset of `#6r` which is the origin associated to `x`. Thus, our coupling graph will need

¹¹This can be done unambiguously for all examples presented in this thesis.

3.2. The Coupling Graph

Program 3.2 MIR program generated from Program 3.1

```
bb0:
  0:  StorageLive(_4),
  1:  _4 = &mut _1,
  2:  FakeRead(ForLet(None), _4),
  3:  StorageLive(_5),
  4:  _5 = &mut _2,
  5:  FakeRead(ForLet(None), _5),
  6:  StorageLive(_6),
  7:  StorageLive(_7),
  8:  _7 = _3,
  9:  switchInt(move _7) -> [0: bb2, otherwise: bb1]

bb1:
  0:  StorageLive(_8),
  1:  _8 = move _4,
  2:  FakeRead(ForLet(None), _8),
  3:  StorageLive(_9),
  4:  _9 = &mut (*_5),
  5:  _4 = move _9,
  6:  StorageDead(_9),
  7:  StorageLive(_10),
  8:  _10 = &mut (*_8),
  9:  _5 = move _10,
 10:  StorageDead(_10),
 11:  _6 = const (),
 12:  StorageDead(_8),
 13:  kind: goto -> bb3

bb2:
  1:  _6 = const (),
  2:  goto -> bb3

bb3:
  0:  StorageDead(_7),
  1:  StorageDead(_6),
  2:  StorageLive(_11),
  3:  _11 = move _4,
  4:  FakeRead(ForLet(None), _11),
  5:  StorageLive(_12),
  6:  _12 = move _5,
  7:  FakeRead(ForLet(None), _12),
  8:  _0 = const (),
  9:  StorageDead(_12),
 10:  StorageDead(_11),
 11:  StorageDead(_5),
 12:  StorageDead(_4),
 13:  return
```

3.2. The Coupling Graph

1. a new **borrow** edge for the issue of the borrow,
2. a new **move** edge from the moved-into place to the borrow,
3. a new **shared** edge that models the subset **#2r <: #6r**
4. as with any move, shallow exclusive capability for the moved-into place,
5. to remove the (deep) borrowed-from place from the PCS and initialize the moved-into place.

Together, we obtain the coupling graph

```
+ #2r: {E bw0} --* {E _1}
+      {E bw0} -borrow(bw0)-> {E _1}

+ #6r: {E *_4} --* {E _1}
+      {E *_4} -move-> {E bw0};
+      {E bw0} -shared(#2r)-> {E _1}
```

We see that the change in the coupling graph induces a change in its nodes: we have introduced a new root $\{E_1\}$ and a new leaf $\{E_*_4\}$. Following the steps outlined above, the free PCS should remove $\{E_1, e_4\}$ and regain $\{E_4\} \sim \{e_4, E_*_4\}$, which follows the principle that the changes in leaves and roots in the coupling graph match the changes in the Free PCS:

$$\{E_2, E_3, E_4\}.$$

Reading off the Free PCS and coupling graph, we now have complete capability to $_4$, and its presence in the Free PCS is blocking access to E_1 , as expected.

Expiring Origins

At program point `Mid[bb0[1]]` the origin **#2r** is dead, because the temporary resource **bw0** cannot be used outside of the assignment. Consequently, Polonius flags this origin, and does not propagate it forward to the next program point `Start[bb0[2]]`. It is simple for us to do the same in our graph: between `Mid[bb0[1]]` and `Start[bb0[2]]` we flag that origin **#2r** is not live by annotating it with the point where it died:

3.2. The Coupling Graph

```

+ #2r@bb0[1]: {E bw0} --* {E _1}
    {E bw0} -borrow(bw0)-> {E _1}

#6r: {E *_4} --* {E _1}
    {E *_4} -move-> {E bw0};
+    {E bw0} -shared(#2r@bb0[1])-> {E _1}

```

In general we keep dead origins around as long as they are referred to by a shared edge,¹² in the common case of programs with long reborrowing chains this reduces a quadratic number of duplicated edges into a linear one. For brevity in this example, we will rewrite the graph to an equivalent version that inlines killed origins:

```

- #2r@bb0[1]: {E bw0} --* {E _1}
-    {E bw0} -borrow(bw0)-> {E _1}

#6r: {E *_4} --* {E _1}
    {E *_4} -move-> {E bw0};
+    {E bw0} -borrow(bw0)-> {E _1}
-    {E bw0} -shared(#2r@bb0[1])-> {E _1}

```

Note that removing origin #2r did not change the set of leaves and roots in the coupling graph as a whole and so no changes to the PCS were needed. In general removing an origin may involve some sequence of repacks in order to assume the right permissions can be removed from the PCS, but we can trust the free PCS to insert these intermediary statements.

Repeating the procedure above, we can execute up to the end of basic block bb0, ending with PCS

{e _3, E _4, E _5, E _7}

and coupling graph

```

#6r: {E *_4} --* {E _1}
    {E *_4} -move-> {E bw0};
    {E bw0} -borrow(bw0)-> {E _1}

+ #7r: {E *_5} --* {E _2}
+    {E *_5} -move-> {E bw1};
+    {E bw1} -borrow(bw1)-> {E _2}

```

¹²Leaks of edges are not possible, because our data structure is acyclic.

Moving Borrows

Evaluation of the conditional jump does not change the PCS, and the `else` branch `bb2` only initializes `_6`, so we can move inside the `if` branch next. The local `_8` (representing `tmp`) comes into scope, and we see that it is being moved into from the borrow-typed place. To move a borrow in our system, several steps need to happen at once:

1. we need to add a `move` edge between the borrows' dereferences in the coupling graph,
2. we need to kill the moved-from edge in the coupling graph,
3. we need capability to initialize the moved-to place in the free PCS (that is, we need shallow exclusive capability for it),
4. we need to gain deep capability for the moved-to place in the coupling graph.

To prepare for this sequence of steps, we unpack the free PCS to

$$\{e_3, e_4, E_*_4, E_5, E_7, e_8\}$$

so it contains the capability to be blocked (E_*_4) and the capability for the set (e_8). Noting that $\text{Deep}(e_8) = S_8$ and $\text{Deep}(e_*_8) = E_*_8$, the free PCS becomes

$$\{e_4, e_3, E_5, E_7, S_8\}$$

with a coupling graph

```

- #6r: {E *_4} --* {E _1}
+ #8r: {E *_8} --* {E _1}
+   {E *_8} -move-> {E@bb1[1] *_4};
+   {E@bb1[1] *_4} -move-> {E bw0};
+   {E bw0} -borrow(bw0)-> {E _1}

#7r: {E *_5} --* {E _2}
    {E *_5} -move-> {E bw1};
    {E bw1} -borrow(bw1)-> {E _2}

```

In this rewrite we implicitly removed the dead origin `#6r`. Note that the moved-out-from place `_4` is assignable again, as it is mutable and uninitialized. The operational semantics for moves of borrows are intentionally very similar to the semantics for moves of owned data; the central difference is that the *value* of borrows *b* are represented as a capability to a distinct place **b*.

Reborrows

After `_9` comes into scope, the next statement is a reborrow `_9 = &mut(*_5)`. A reborrow is similar to a move, but it does not kill the reborrowed-from place in the coupling graph. To perform a reborrow we must

1. repack so the reborrowed-from place is in the free PCS,
2. ensure we have shallow exclusive capability for the assigned-to place,
3. perform a borrow,
4. add a *reborrow* edge in the free PCS from the borrow to the reborrowed-from origin.

In this example step 4 is a trivial edge $\{E *_5\} \rightarrow \{E *_5\}$; in general this allows us to borrow one field out of a struct without borrowing the struct as a whole. The changes to the coupling graph and PCS leave us with

$\{e_3, e_4, e_5, E *_5, E *_7, S_8, E_9\}$

```
#8r: {E *_8} --* {E _1}
      {E *_8} -move-> {E@bb1[1] *_4};
      {Ebb1[1] *_4} -move-> {E bw0};
      {E bw0} -borrow(bw0)-> {E _1}

#7r: {E *_5} --* {E _2}
      {E *_5} -move-> {E bw1};
      {E bw1} -borrow(bw1)-> {E _2}

+ #9r: {E *_9} --* {E _2}
+      {E *_9} -move-> {E bw2};
+      {E bw2} -borrow(bw2)-> {E *_5};
+      {E *_5} -reborrow-> {E *_5};
+      {E *_5} -shared(#7r)-> {E _2}
```

While `E *_5` is no longer accessible in the free PCS, we could still regain access to `E *_5` by first expiring `#9`. This is in contrast to a move, where the moved-from resource is killed. In our case `#7r` dies anyways because `*_5` is not used in the future so we expire that origin from the graph (to no effect in the PCS).

It is typical for the MIR to translate reborrows from the source into a borrow into a fresh local followed by a move, which we handle as before, and then the fresh local `_9` goes out of scope. We are left with

3.2. The Coupling Graph

$\{e_3, e_4, E_*_4, e_5, E_*_5, E_7, S_8\}$

```
#8r: {E *_8} --* {E _1}
      {E *_8} -move-> {E@bb1[1] *_4};
      {Ebb1[1] *_4} -move-> {E bw0};
      {E bw0} -borrow(bw0)-> {E _1}

- #7r: {E *_5} --* {E _2}
- #9r: {E *_9} --* {E _2}
+ #6r: {E *_4} --* {E _2}
+      {E *_4} -move-> {E@bb1[5] *_9};
+      {E@bb1[5] *_9} -move-> {E bw2};
      {E bw2} -borrow(bw2)-> {E *_5};
      {E *_5} -reborrow-> {E *_5};
      {E *_5} -move-> {E bw1};
      {E bw1} -borrow(bw1)-> {E _2}
```

Killing Places

We have covered most of the operations the coupling graph can perform at this point. Now, the MIR reborrows $*_8$ into a new local $_10$, before moving $_10$ into $_5$. As $_5$ is newly initialized again, the resource in the coupling graph which used to be referred to as $_5$ and is still part of the graph's structure is out of date. We solve this by *killing* all subplaces of $_5$ by tagging them with the current location. Killed places behave the same as any linear resource in the coupling graph, but cannot be regained into the free PCS.

$\{e_3, e_4, E_*_4, e_5, E_*_5, E_7, s_8, e_10\}$

3.2. The Coupling Graph

```
- #8r: {E *_8} --* {E _1}
+ #7r: {E *_5} --* {E _1}
+   {E*_5} -move-> {E@bb1[8] *_10};
+   {E@bb1[8] *_10} -move-> {E bw2};
+   {E bw3} -borrow(bw3)-> {E *_8};
+   {E *_8} -reborrow-> {E *_8};
+   {E *_8} -move-> {E@bb1[1] *_4};
+   {Ebb1[1] *_4} -move-> {E bw0};
+   {E bw0} -borrow(bw0)-> {E _1}

#6r: {E *_4} --* {E _2}
+   {E *_4} -move-> {E@bb1[5] *_9};
+   {E@bb1[5] *_9} -move-> {E bw2};
+   {E bw2} -borrow(bw2)-> {E@bb1[9] *_5};
+   {E@bb1[9] *_5} -reborrow-> {E@bb1[9] *_5};
+   {E@bb1[9] *_5} -move-> {E bw1};
+   {E bw1} -borrow(bw1)-> {E _2}
```

Finally, the intermediate locals go out of scope (thus killing them in the graph) and after some repacking to simplify and initializing `_6` the state at the end of `bb1` is

$$\{e_3, E_4, E_5, E_6, E_7\}$$

```
#7r: {E *_5} --* {E _1}
+   {E*_5} -move-> {E@bb1[8] *_10};
+   {E@bb1[8] *_10} -move-> {E bw2};
+   {E bw3} -borrow(bw3)-> {E@bb1[12] *_8};
+   {E@bb1[12] *_8} -reborrow-> {E@bb1[12] *_8};
+   {E@bb1[12] *_8} -move-> {E@bb1[1] *_4};
+   {Ebb1[1] *_4} -move-> {E bw0};
+   {E bw0} -borrow(bw0)-> {E _1}

#6r: {E *_4} --* {E _2}
+   {E *_4} -move-> {E@bb1[5] *_9};
+   {E@bb1[5] *_9} -move-> {E bw2};
+   {E bw2} -borrow(bw2)-> {E@bb1[9] *_5};
+   {E@bb1[9] *_5} -reborrow-> {E@bb1[9] *_5};
+   {E@bb1[9] *_5} -move-> {E bw1};
+   {E bw1} -borrow(bw1)-> {E _2}
```


Coupling

We’ve shown that the true branch ends with a coupling graph with signatures

```
#6r: {E *_4} --* {E _2}
#7r: {E *_5} --* {E _1}
```

and the false branch has the signature it started with, namely

```
#6r: {E *_4} --* {E _1}
#7r: {E *_5} --* {E _2}
```

We join these graphs together by creating a coupled edge. More precisely, we make one coupled edge for every largest family of leaves which may alias the same place. In this case, the leaves `E *_4` and `E *_5` may alias either `E _1` or `E _2` and so we create one coupled edge `cb1` to encompass them both:

```
- /* both graphs are combined */
+ #7r: {E *_5} --* {E _1, E _2}
+   {E *_5} -coupled(cb1) -> {E _1, E _2}
+
+ #6r: {E *_4} --* {E _1, E _2}
+   {E *_4} -coupled(cb1) -> {E _1, E _2}
```

Applying Coupled Edges

Finally, the end of the program drops the coupled borrows in sequence. The coupled borrows are expired in sequence, using the same rules as before. We will not go into the step-by-step details here since they are similar to what we’ve already covered, but at `bb3[4]` the origin `#6r` expires, changing the coupling graph to be

```
#7r: {E *_5} --* {E _1, E _2}
    {E *_5} -coupled(cb1) -> {E _1, E _2}

- #6r: {E *_4} --* {E _1, E _2}
-   {E *_4} -coupled(cb1) -> {E _1, E _2}
```

and thus consuming `E *_4` from the free PCS without giving back either `E _1` or `E _2`. Compare the nodes and edges to Figure 3.2: the change in coupled edge that we expect at the model-level is exactly captured by the rules we have already established for origin expiry.

Interpreted as Viper, we could either represent the coupled edge as a single magic wand, or as a pair of graphs which are control-flow dependent¹³.

¹³In the summer of 2022 we explored a variant of the latter option which efficiently

3.2. *The Coupling Graph*

In either case no edge in the reborrowing graph associated with this coupled edge should be expired until the edge is eliminated from the model-level coupling graph, that is, all instances of that coupled edge are eliminated from our implementation-level coupling graph.

Chapter 4

Evaluation

One of our principal motivations for this work has been supporting new features in Prusti. In this section we will present how our new PCS inference can support cases that Prusti does not handle.

As a proof of concept, we have implemented the inference algorithm in Chapter 3 as a prototype extension to Prusti’s existing fixed point analyses. Our extension reads the MIR and Polonius facts from the compiler, and outputs a Graphviz trace of the coupling graph at each program point in the MIR.

4.1 Feature: Magic Wand Inference

Loop invariant inference is a challenging problem for automatic verification tools. In Prusti, this problem manifests in trying to infer a *magic wand* (abstract objects from separation logic which can be *applied* to exchange some set of capabilities for another) which describes an invariant on which capabilities the live borrows are blocking at the loop head. Fortunately, our analysis so far can provide such a description. Because coupled borrows can encapsulate several concrete edges in a reborrowing graph, they translate naturally into magic wands that a verifier such as Viper can use directly.

As an example, consider Program 4.1, a modification of Program 3.1 which changes the branching conditional into a loop. Let us perform a fixed point analysis with the procedure as outlined in Chapter 3. After the first iteration of the loop body, we perform a join at the loop head which is precisely the same as the join exiting the *if* statement as at the end of Section 3.2.2: the *else* branch in Program 3.1 is the same as performing zero iterations through the loop in Program 4.1. A similar analysis to Section 3.2.1 obtains the same state by iterating the loop body starting with the system of coupled borrows, and so we find ourselves at a fixed point in only two

iterations¹⁴.

Program 4.1 A looping permutation of borrows.

```

1 fn swap(mut t0: T, mut t1: T, b: Bool) {
2     let mut x = &mut t0; // borrow bw0
3     let mut y = &mut t1; // borrow bw1
4     while b {
5         let tmp = x;
6         x = y;
7         y = tmp;
8     }
9     let last_usage_x = x;
10    let last_usage_y = y;
11 }
```

At the loop head in this program the coupling graph describes the compiler’s knowledge of which places are blocking which other places, so is a good candidate for a loop invariant; In Viper terms, every coupled edge can be represented by a magic wand which is applied at the point where it is expired. While not all aspects of the loop body are represented in the coupling graph (for example, the fact that each iteration always swaps the borrows with each other), the coupling graph encodes the coarsest possible relationship between borrows which still sound with respect to the assumptions made by Polonius. As such it serves as a good foundation for additional refinement either by user annotations, or by heuristic inference passes lower down in the Prusti pipeline.

Another example of a program which is challenging for Prusti is reborrowing inside loops, as presented by a linked list traversal in Program 4.2. Our analysis can infer the magic wand $\{E_7\} * \{E_1\}$ for the loop in this program, where `_7` is the local associated to `list` and `_1` is the local associated to `l`. Surprisingly, no modifications from the procedure in Chapter 3 are needed to infer a magic wand for this example, which Prusti could not previously handle.

¹⁴We have observed this same constant bound on the number of iterations in all of our test programs.

Program 4.2 Linked List Traversal

```
1 struct Link {next: Box<List>}
2 type List = Option<Link>;
3 fn test(mut l: List)
4 {
5     let mut list = &mut l;
6     while let Some(next_link) = list {
7         list = &mut next_link.next;
8     }
9     let list_live = &mut (*list);
10 }
```

Chapter 5

Future Work

In the near term, we hope to expand our analysis to handle `subset_base` facts that do not originate from dataflow, in order to support type ascriptions, borrows inside structs, and function calls. We also hope to extend the prototype to compute the full PCS rather than just the coupling graph, in order to evaluate the model on examples from crates outside of our hand-crafted examples.

In this model we have made significant progress towards modelling shared borrows, another feature with only partial support in Prusti. Shared borrows exhibit the property that multiple read-only borrows which simultaneously block write access to a place, the same behavior we observed with coupled borrows. Furthermore, the Polonius facts for mutable and shared borrows are nearly identical: their only difference is in the `loan_invalidated_at` fact (since shared borrows are not invalidated on reads) which has no bearing on our inference of the coupling graph whatsoever. Extending the Free PCS semantics to handle shared borrows is slightly more delicate, and we hope to develop and test our ideas for doing so through our prototype over the next few months.

In the longer term, we hope that this model can be a part of Prusti's current major refactoring project. We believe that the PCS is a suitable abstraction to separate compiler details from the rest of Prusti, and that our model proves that a mechanical derivation of the PCS is feasible under the current architecture of the Rust compiler.

Bibliography

- [1] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers. Leveraging Rust types for modular specification and verification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA) 2019*, pages 147:1–147:30. ACM, 2019.
- [2] Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. The prusti project: Formal verification for rust. In Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez, editors, *NASA Formal Methods*, pages 88–108, Cham, 2022. Springer International Publishing.
- [3] Son Ho and Jonathan Protzenko. Aeneas: Rust verification by functional translation. *Proc. ACM Program. Lang.*, 6(ICFP), aug 2022.
- [4] The Rust Compiler Team. The polonius book, 2023.
- [5] The Rust Compiler Team. The rust book, 2023.
- [6] The Rust Compiler Team. Rust compiler development guide, 2023.
- [7] The Rust Compiler Team. rustc_middle::mir, 2023.