Modelling Rust's Reference Ownership Analysis Declaratively in Datalog

Albin Stjerna

October 28, 2019

Abstract

Rust is a modern systems programming language that offers improved memory safety over traditional languages like C or C++ without introducing garbage collection. In particular, it guarantees that a program is free from data-races caused by memory-aliasing, use-after-frees, and accesses to deinitialised or uninitialised memory. At the heart of Rust's memory safety guarantees lies a system of memory ownership, verified statically in the compiler by a process called the borrow check. However, the current implementation of the borrow check is not expressive enough to prove that several desirable programs indeed does not violate the Rust memory ownership rules. This report describes an experimental reformulation of the borrow check called Polonius, which both increases the resolution of the analysis to reason at every point of the program flow, and enables a more expressive formulation of the borrow check itself through the use of a domain-specific language, Datalog.

Specifically, this thesis extends Polonius with initialisation and liveness computations for variables.Finally, the thesis describes an exploratory study of real-world input data for Polonius generated by analysing circa 12 000 popular Git repositories found on GitHub and the Crates.io Rust package index. Some central findings from the study are that deallocations are uncommon relative to other variable uses, and that the full flow-sensitive analysis is typically seldom needed. Indeed, surprisingly many functions (circa 64%) actually do not create any references at all, and therefore does not need (most of) the Polonius analysis, or even the borrow check.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Rust is a young systems programming language originally developed at Mozilla Research [1]. Its stated intention is to combine high-level features like automatic memory management and strong safety guarantees with predictable performance and pay-as-you-go abstractions in systems languages like C++. Particular attention is given to protection against data races in concurrent programs.

One of Rust's core features is the memory ownership model, which gives compile-time safety guarantees against access to uninitialised memory and data races in addition to enabling runtime-free automatic memory management. This model is enforced by a memory safety checker called the borrow check. The borrow check ensures that no memory access reaches uninitialised memory, and that any shared memory is never written for the duration of the sharing. Finally, it also protects against dangling references and references to stack-allocated memory that may be outside of the scope of an accessor, disallowing for example a function to return a reference to memory allocated on its stack. The rules themselves are discussed at a higher level in Section **??**, and related to the type system Oxide in Section 3.2. In essence, the borrow checker is a program (or module) that takes as its input a Rust program (possibly in reduced form) and produces a description of which parts of the code violates the rules of the memory ownership model.

This report describes a partial implementation of an experimental borrow checker called Polonius. Polonius specifically increases the reasoning power of the borrow check to reason at the level of individual program statements (a flow-sensitive analysis), allowing it to accept previously rejected but desirable patterns of Rust, such as the example shown in Listing **??**.

In practice, Polonius' analysis encompasses a variable liveness analysis (Section **??**), initialisation tracking (Section **??**), and may-reference analysis for validation of Rust's memory safety guarantees and alias control (Section 6.3), used to statically enforce safe use of shared memory. In practice, these rules can be understood as read and write-capable locks on a section of memory, placed by the compiler at compile-time.

1.1 Contributions

The contributions made in this thesis specifically include the implementation of liveness and initialisation calculations (Sections **??** and **??** respectively) in Polonius, previously computed in the Rust compiler before invoking Polonius. Additionally, this report analyses real-world Rust code found in ca 12 000 popular publicly available Git repositories found on Crates.io and GitHub (Section 7). It compares two optimised variants of Polonius to a baseline naive implementation, and produces statistics on which types of inputs to Polonius typically dominate, drawing some conclusions on common coding patterns.

For clarity, sections detailing components not developed as part of this thesis are marked with (†). They are nonetheless included, as there has not previously been any published complete description of Polonius.

Chapter 2

A Safe and Modern Systems Programming Language

A notable detail of the borrow check is what is meant by a "variable". In Rust, some data structures, such as `struct`s (complex data types, corresponding to objects without methods), and tuples, are analysed at the granularity of the individual components, which may have arbitrarily deep nesting (known statically). This means that the following code, for example, is sound, as the loans do not overlap:

```
struct Point(u32, u32);

let mut pt: Point = Point(6, 9);
let x = &mut pt.0;
let y = &mut pt.1;
// no error; our loans do not overlap!
```

In our instance, the root variable `pt` contains the *paths* `pt.1` and `pt.2`. Such paths constitute a tree with its root in the variable itself. Both the core borrow check and the initialisation tracking that we will discuss reasons about variables on the path level. It is worth pointing out that dynamic structures like vectors, as well as arrays, are not analysed at this granularity, but are considered one single object.

In Rust compiler parlance, we say that a path points to a *place*, corresponding literally to the place in memory holding its data.

Finally, we sometimes talk of a path having *prefixes*, where a prefix is anything above and including a "leaf" in the tree spanning all paths. For example, the path `x.y` would have the prefixes `x` and `x.y`. Prefixes will also come up in initialisation tracking, and in the borrow check itself.

Rust's memory safety guarantees are strictly speaking part of its type system (as further discussed in Section 3.2), but is verified in a separate step of the compilation process (see Chapter **??**) . This step is referred to as as the *borrow check*, and the component of the compiler that performs it the *borrow checker*. The specific rules enforced by the borrow checker are discussed in the next chapter (Chapter 3).

Due to limitations in its formulation, the current borrow checker rejects code such as the one in Listing 2.0.1, as it is unable to prove that there are no two overlapping write references to the same location in memory (namely `buffer`). This limitation stems from a more constrained reasoning around program flow, which introduces imprecision into the analysis. Polonius is designed to address this imprecision by extending the reasoning power of the borrow check to be flow-sensitive, that is reason at the power of each individual program statement.

Listing 2.0.1: A motivating example for Polonius, rejected by the current borrow checker. The code is sound, as the loaned `event` is either returned out of the loop, or overwritten at the next iteration. Therefore, there are no overlapping mutable loans of `buffer`. [2]

```
fn next<'buf>(buffer: &'buf mut String) -> &'buf str {
    loop {
        let event = parse(buffer);

        if true {
            return event;
        }
    }
}

fn parse<'buf>(_buffer: &'buf mut String) -> &'buf str {
    unimplemented!()
}
```

The other reason for Polonius is to more clearly capture the semantics of the borrow check in a domain-specific language.

Chapter 3

The Borrow Check: Enforcing Rust's Memory Model

Conceptually, the borrow check verifies that Rust's ownership rules of shared memory are respected. Memory is owned by the scope (typically a function, block, or data structure) that has allocated it, and will be deallocated upon leaving the scope of the owner. Memory can also dynamically change owners through a move. For example, the constructor of a data structure can capture its arguments and store them in the returned data structure, thus moving the memory without performing a reallocation. The borrow check verifies that each memory access is (definitely) owned (and initialised) at the point of the control-flow of each access. It also verifies that accesses to shared memory through loans respect the terms of that loan. A shared reference cannot be mutated, and must be guaranteed to be free from use-after-frees.

A sumary of the rules enforced by the borrow check can be found in Table 3.0.1. Many of these examples are taken directly or slightly modified from Weiss, Patterson, Matsakis, and Ahmed [3].

3.1 From Lifetimes to Provenance Variables

As the lifetime of its value is a part of a reference variable's type, it can be referred to by name using the syntax `&'lifetime`. In the literature, the terms "region" [4], "(named) lifetime" , and "reference provenance" [3] (provenance) are all employed. As the section heading suggests, we will use the last one of them as we believe it best captures the concept. Named provenances (such as `'lifetime` above) are referred to as "provenance variables". For historical reasons, the name "region" sometimes occurs in Polonius' code as well. Moreover, during the work on this thesis, a fourth term, "origin", was chosen to replace the term "provenance variables" used here. Additionally, a comprehensive re-naming of all the terms used is also underway at the time of writing.

From a type system perspective, the provenance is part of the type of any reference and corresponds to the borrow expressions (reference constructions) that might have generated it in the Polonius formulation of the borrow check. For example, if a reference `r` has the type `&'a Point`, `r` is only valid as long as the terms of the loans in `'a` are upheld. Take for example the annotated code of Listing 3.1.1, where `p` would have the type `&'a i32` where `a` is the set $\{L_0, L_1\}$.

Listing 3.1.1: An example of a multi-path loan where the value in `p` could point to either of the vector `x`'s values depending on the return value of the function `random()`. The code has been annotated with named provenance variables and would not compile as-is.

```
let x = vec![1, 2];

let p: &'a i32 = if random() {
  &x[0] // Loan L0
} else {
  &x[1] // Loan L1
};
```

If a reference is used in an assignment like `let p: &'b i32 = &'a x`, the reference, `p`, cannot outlive the referenced value, `x`. More formally the type of the right-hand side, `&'a i32`, must be a subtype of the left-hand side's type; `&'a i32 <: &'b i32`. In practice, this establishes that `'b` lives at most as long as `'a`, which means that the subtyping rules for variables establishes a set membership constraint between their provenance variables, as seen in Rule 3.2.6 of Section 3.2, which gives a brief introduction to the reference ownership analysis of Polonius from a type systems perspective.

Finally, when talking about the *liveness* of a provenance variable r at some point in the control-flow graph p, we will mean that r occurs in the type of at least one variable which is live at p. This has the semantic implication that any of the loans in r might be dereferenced at control-flow points reachable from p, and thus that the terms of the loans in r must be respected at that point. The possibility of a future access is not limited to direct access of a variable, but also concerns uses in the custom deallocator of a `struct` holding a reference. This scenario is further discussed in Section 6.1.1.

3.2 The Borrow Check as a Type System

In this section, we will relate Polonius to Weiss, Patterson, Matsakis, and Ahmed ongoing work of Weiss, Patterson, Matsakis, and Ahmed on formalising the reference ownership rules of Rust into the formally defined type system Oxide [3]. Oxide is notable in that it shares Polonius' view of variables as sets of possible points in the code that would give rise to the reference ("loans" of the variable). The rules presented here are based on the 2019 draft version of the paper, and will change substantially in the paper's final version.

The typing rules of this section are meant to be read top-to-bottom. They mean that as long as the conditions above the horizontal bar holds, the conclusion below it will hold; usually that an expression is sound with respect to the type system (it type-checks). Most of the conventions used in the Oxide formulation can be glossed over for our purposes here, but the most important ones are the type environment Γ, used to map places (roughly: variables) (π, π_1, and so on) to their types (τ, τ_1, etc). As reference types contain provenance variables (ρ), this type environment is stateful, in that for example typing a reference-constructing expression would modify the typing environment to add a new loan.

Judgments on the form $\Gamma \vdash_\omega \pi : \tau$ mean that "in the environment Γ, it is safe to use the variable π (of type τ) ω-ly" [3]. In other words, if ω is *unique*, it means that there are no live loans of any paths overlapping π, and of ω is *shared* that there are no overlapping loans in the provenance part of τ. The full type system handles degradation of these types of references, etc, but would be far beyond the scope of our comparison here.

At the heart of the type system lies the flow-sensitive typing judgments seen in Rules 3.2.1 and 3.2.2, both taken from Weiss, Patterson, Matsakis, and Ahmed's paper (Figure 1). The first rule (3.2.1) shows that for a given environment Γ, a move of a given variable π (occurring if π cannot be copied, which is what the right prerequisite says) is only valid if π is uniquely usable (that is, is not shared) (left prerequisite) in Γ. The typing itself removes π from the Γ, effectively barring it from future use as it has no type (conclusion). This corresponds to the initialisation tracking of Section **??**, as well as part of the invalidation logic of Polonius.

$$\frac{\Gamma \vdash_{\text{mut}} \pi : \tau^s \quad \text{noncopyable } \tau^s}{\Sigma; \Delta; \Gamma \vdash \boxed{\pi} : \tau^s \Rightarrow \Gamma - \pi} \tag{3.2.1}$$

The second rule, Rule (3.2.2), states that we may create an ω-reference to any variable π of type τ where ω-use is safe, and produce a reference of equal ω access to that variable of the type "reference to a value of type, τ, with its provenance variable being the set containing only that loan, denoted ${}^\omega\pi$". This corresponds to the input fact `borrow_region`, described in Section 4.0.1, and follows the intuition that if we create a reference, that reference is *known* to point to whatever we borrowed to create the reference.

$$\frac{\Gamma \vdash_\omega \pi : \tau}{\Sigma; \Delta; \Gamma \vdash \boxed{\&\omega\pi} : \& \{{}^\omega\pi\}\, \omega\tau \Rightarrow \Gamma} \tag{3.2.2}$$

Rules (3.2.1) and (3.2.2) constitute base cases for the ownership system, showing how variables get removed from the environment, and how provenance variables in reference types are created. In order to describe the full analysis, we need to also consider how these relations extend across program execution through sequencing or branching, of which the latter introduces the approximate aspect of provenances. Finally, we will also describe how provenance variables come into relation with each other through type unification and subtyping.

Since the borrow check is performed on the MIR, Polonius does not handle branchings in the normal sense. Therefore, the sequencing and branching rules of Oxide only translate analogously. As in Oxide, the type environment of the MIR is threaded through the typing of each expression, such that the sequence of expressions $\boxed{e_1; e_2}$ would first type-check e_1 and then e_2 in the resulting environment after type-checking e_1, each updating the typing environment as they go.

In Oxide, the typing rules for branch expressions uses a type unification of the value of the `if` expression such that its value unifies (that is, merges) the provenance variables of the environments in both branches. The MIR produced by such a branching would instead have a loop starting at the head of the `if` expression and ending with an assignment to the same variable in each branch before finally joining in a basic block where the assigned variable now could have come from either arm, as in Figure 4.0.2 but with references instead of regular values being assigned. Hence branching introduces the first source of imprecision into the provenance variables.

How, then, does this type unification work for references? The rule, T-Ref, Rule (3.2.3), tells us first that the two types τ_1, τ_2 that we want to unify must in turn unify into a single type τ, which of less interest to us; in principle it means that whatever the reference points to has compatible types. The conclusion of the rule is what is of interest here. It says that these two references' provenance variables must unify into the combined provenance ρ. Moreover, the access types of these references must be compatible; they must have the same use-type ω (meaning that we cannot use a non-unique reference as a unique one). In practice, this unification rule is what introduces the imprecision of this analysis on branchings, and would correspond to the propagation of relations across CFG edges in Polonius.

$$\frac{\tau_1 \sim \tau_2 \Rightarrow \tau \quad \rho_1 \cup \rho_2 = \rho}{\&\rho_1\omega\tau_1 \sim \&\rho_2\omega\tau_2 \Rightarrow \&\rho\omega\tau} \tag{3.2.3}$$

Finally, provenance variables comes into relation with each other during assignments and variable definitions. An assignment would have the form `x = y` and would give the already-defined variable `x` the value of `y`. If `y` is not `Copy`, it would be moved to `x` and be deinitialised. A definition would take the form `let x = y`, and would introduce a new variable `x` into the scope. The typing judgments for both kinds of statements in Oxide are complex, and we will therefore only gloss over them here.

Simply put, each assignment allows for different types on each side of the assignment, as long as the types unify, as seen in Rule (3.2.4) (Oxide's T-Assign rule), which says two things of interest to us. First, an assignment is only possible if the left-hand side of the expression can be unified with the right-hand side (the prerequisite), and second that assignment will remove the previous mapping of π_1 in Γ and replace it with the new expression. The call to the meta-function `places-typ` is used to ex-

pand π into all its references and perform the assignment. This would correspond to the `killed` relation used in Polonius, where an old loan is removed from the environment whenever one of its prefixes is assigned. Additionally, Polonius would also have assignment and initialisation inputs for the liveness and initialisation tracking respectively, but those are beside the point of this discussion.

$$\frac{\begin{array}{c}\Gamma \vdash_{\texttt{uniq}} \pi : \tau_o \quad \tau_o \sim \tau_u \Rightarrow \tau_n \\ \Sigma; \Delta; \Gamma \vdash \boxed{e} : \tau_u \Rightarrow \Gamma_1 \\ \texttt{places-typ}(\pi, \tau_u) = \overline{\pi : \tau}\end{array}}{\Sigma; \Delta; \Gamma \vdash \boxed{\pi = e} \ : \ \texttt{unit} \Rightarrow \Gamma_1 - \pi_1, \overline{\pi : \tau}} \tag{3.2.4}$$

Finally, variable binding is what introduces relations between provenance variables, which is another source of imprecision in the analysis. Glossing over the complexities of the typing rule for `let` expressions (Oxide's T-Let or (3.2.5)), we can see that a variable definition would update the variable's type in the environment and, the crucial part, imply a subtyping relationship between the left-hand side of the expression and the right-hand side, the $\Sigma \vdash \tau_1^s <: \tau_a^s \rightsquigarrow \delta$ prerequisite, which is then used in the new scope created by the binding. This subtyping rule, Rule (3.2.6), is what actually introduces the relationship between provenance variables of references.

$$\frac{\begin{array}{c}\Sigma; \Delta; \Gamma \vdash \boxed{e_1} \ : \ \tau_1^s \Rightarrow \Gamma_1 \quad \Sigma \vdash \tau_1^s <: \tau_a^s \rightsquigarrow \delta \\ \texttt{places-typ}(\texttt{x}, \delta(\tau_a^s)) = \overline{\pi : \tau} \\ \Sigma; \Delta; \Gamma, \overline{\pi : \tau} \vdash \delta(e_2) : \tau_2^s \Rightarrow \Gamma_2\end{array}}{\Sigma; \Delta; \Gamma \vdash \boxed{\texttt{let x} = \pi_2} \ : \ \tau_2^s \Rightarrow \Gamma_2 - \texttt{x}} \tag{3.2.5}$$

The subtyping rule for references, Rule (3.2.6), says that a reference type τ_1 is a subtype of a (reference) type τ_2 if the things they refer to are also subtypes (with the substitution δ), and, crucially here, if τ_1's provenance variable is a subset of τ_2's. The meaning here is that τ_1 can only act as a τ_2 if it points to something compatible (the rightmost prerequisite), if the uses are compatible (the middle prerequisite), and if the τ_1 does not require any loans except the ones in τ_1, the super-type. The intuition for this is that if we are to use τ_1 as a τ_2, the conditions of that loan must not include conditions (notably, liveness of the value at the other end of the reference) beyond what τ_2 promises. In Polonius, this is represented by the `outlives` fact, which is the major source of constraints on loans.

$$\frac{\rho_1 \subseteq \rho_2 \qquad \omega_1 \leq \omega_2 \qquad \Sigma \vdash \tau_1 <: \tau_2 \rightsquigarrow \delta}{\Sigma \vdash \&\rho_1\omega_1\tau_1 <: \&\rho_2\omega_2\tau_2 \rightsquigarrow \delta} \tag{3.2.6}$$

However, since we are working on just a part of the type validation of Rust, we can simplify this complex subtyping rule for our purposes into Rule (3.2.7), which says that a reference type is a subtype of another reference type iff their provenance variables are a subset.

$$\frac{\rho_1 \subseteq \rho_2 \qquad \tau_1 <: \tau_2}{\&\rho_1\tau_1 <: \&\rho_2\tau_2} \tag{3.2.7}$$

We now imagine this typing rule yielding these subtyping constraints for every assignment in the MIR we are verifying. Polonius only concerns itself with the attached provenance variables of the types, other parts of the compiler verifies the rest of the typing judgment. Equipped with this simpler rule, we now have enough understanding of the formal basis of Polonius to move forward with the actual implementation.

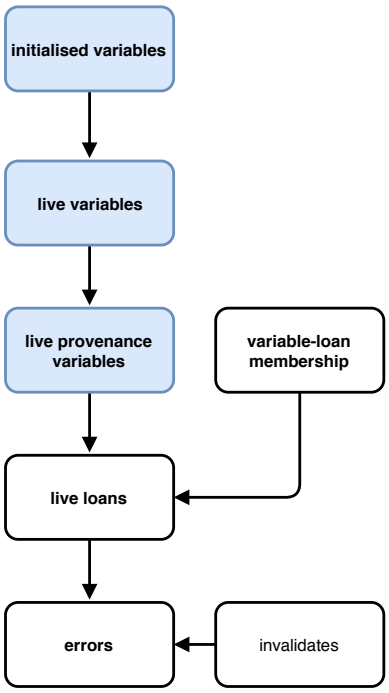


Figure 3.2.1: An overview of Polonius high-level structure; we compute liveness and members of provenance variables in order to find the potentially live loans at every given program point. These potential loans are used together with their potential violations to derive actual errors. A more precise representation can be found in Figure 6.0.1.

Rule	Positive Example	Negative Example

Use-Init	`let x: u32;`
	
	`if random() {`
	`  x = 17;`
	`} else {`
	`  x = 18;`
	`}`
	
	`let y = x + 1;`
		`let x: u32;`
		
		`if random() {`
		`  x = 17;`
		`}`
		
		`// ERROR: x not initialized:`
		`let y = x + 1;`

Move-Deinit	`let tuple = (vec![1], vec![2]);`
	
	`moves_argument(tuple.1);`
	
	`// Does not overlap tuple.1:`
	`let x = tuple.0[0];`
		`let tuple = (vec![1], vec![2]);`
		
		`moves_argument(tuple.0);`
		
		`// ERROR: use of moved value:`
		`let x = tuple.0[0];`

Shared-Readonly	`struct Point(u32, u32);`
	`let mut pt = Point(13, 17);`
	
	`let x = &pt;`
	`let y = &pt;`
	
	`dummy_use(x); dummy_use(y);`
		`struct Point(u32, u32);`
		`let mut pt = Point(13, 17);`
		
		`let x = &pt;`
		`// ERROR: assigned to`
		`//  borrowed value:`
		`pt.0 += 1;`
		`dummy_use(x);`

Unique-Write	`struct Point(u32, u32);`
	`let mut pt = Point(13, 17);`
	
	`let x = &mut pt;`
	`let y = &mut pt;`
	
	`//dummy_use(x);`
	`dummy_use(y);`
		`struct Point(u32, u32);`
		`let mut pt = Point(13, 17);`
		
		`let x = &mut pt;`
		`// ERROR: cannot borrow `pt``
		`// as mutable more than once:`
		`let y = &mut pt;`
		
		`dummy_use(x);`
		`dummy_use(y);`

Ref-Live	`struct Point(u32, u32);`
	`let pt = Point(6, 9);`
	`let x = {`
	`  &pt`
	`}; // pt still in scope`
	
	`let z = x.0;`
		`struct Point(u32, u32);`
		`let x = {`
		`  let pt = Point(6, 9);`
		`  &pt`
		`}; // pt goes out of scope`
		
		`// ERROR: pt does not live`
		`// long enough:`
		`let z = x.0;`

Table 3.0.1: The Rules of the borrow check, with positive (free from errors) and negative (with errors) examples. Highlighted code shows sub-expressions that would perform the borrow check, such as mutating, moving, or reading a variable. Green highlights show accepted uses and red ones show failed ones.

Chapter 4

The Borrow Check in the Rust Compiler

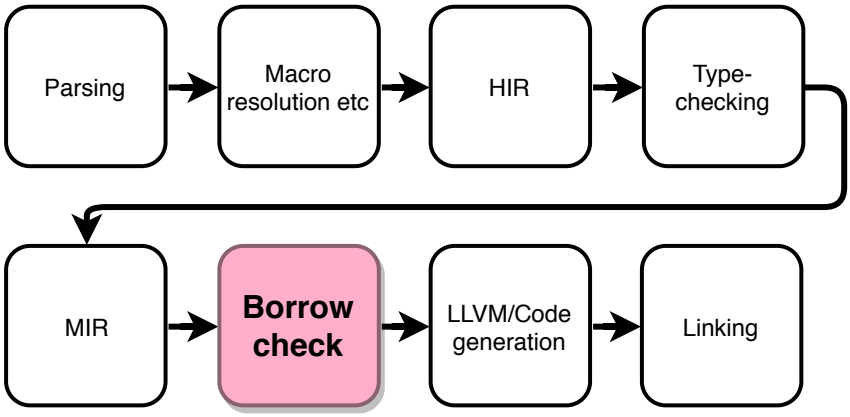
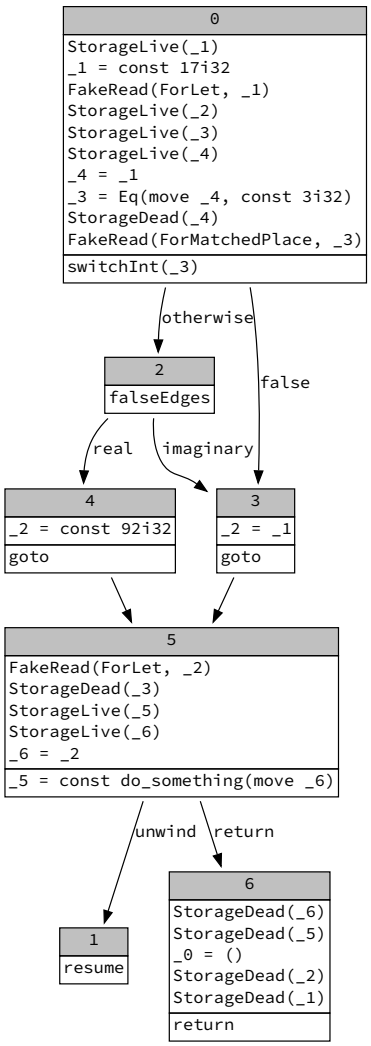


Figure 4.0.1: An overview of the Borrow Check's place in the process of compiling Rust code, as described in the Rust Developer's Guide [6].

The logic of the borrow check as described in Section 3.1 is calculated at the level of an intermediate representation of Rust called the Mid-Level Intermediate Representation (MIR), corresponding to the basic blocks of program control flow. Rust is lowered to MIR after regular type checking and after a series of earlier transformations, as seen in Figure 4.0.1. The Polonius analysis is executed at the function level, checking a function at a time.

The input data to Polonius is generated in the Rust compiler by analysing this intermediate representation. This means that we can safely assume to be working with simple variable-value assignment expressions, of the type `_1 = _2`, as opposed to complex expressions involving multiple variables on the right-hand side.



Listing 4.0.1: A minimal Rust program featuring branching and a function call.

```
1  fn main() {
2      let x = 17;
3      let z = if x == 3 {
4          92
5      } else {
6          x
7      };
8
9      do_something(z);
10 }
```

Figure 4.0.2: A graph rendering (left) of the `main()` function from a Rust program (right), illustrating branching (block 0, corresponding to lines 2–3), and a function call (5, corresponding to line 9). Note the `unwind` arm of block 5's terminator (last line), which will be followed if the function call panics, that is if something goes wrong during the call. Blocks 3 and 4 correspond to the assignment of the value of the `if` statement on line 3, assigning either `92` (block 4) or `x` (block 3) to `z`. The successful return block, 6, contains a number of stack deallocation hints for later compilation steps, and sets up the return value of `main()`, `_0` to be the empty tuple (corresponding to `void` in a C program).

The MIR consists of basic blocks in the traditional compilers sense, each containing a set of statements and usually ending with a *terminator*, an expression providing a branching to other basic blocks [7]. A side-to-side comparison between a small Rust program and its MIR can be seen in Figure 4.0.2.

4.0.1 Input Facts

The following short-hand names are used:

R is a provenance variable, a set of loans.

L is a loan, that is a `&v` expression creating a reference to `v`.

P, Q are points in the control-flow graph of the function under analysis.

V is a variable.

M is a move path, that is a part of a variable that can be accessed and, more importantly, moved. This can be the name of a variable (e.g. `a`), or an access to a field of a data structure or one of a tuple's projections (e.g. `a.b`, or `a.1`).

Fact	Code Example	Resulting Tuple(s)	Used
`borrow_region(R, L, P)`	`bb0[0]: _1 = &1;`	`('a, &1, p + 1)`	Bck
`universal_region(R)`	`fn parse<'b>(bfr: &'b str)`	`('b)`	Bck

`cfg_edge(P, Q)`	`bb0[0]: _1 = 1;`
	`bb0[1]: _2 = 3;`
		`(Start(bb0[0]), Mid(bb0[0])),`
		`(Mid(bb0[0]), Start(bb0[1])),`
		`(Start(bb0[1]), Mid(bb0[1]))`
			All

`killed(L, P)`			Bck
`outlives(R1, R2, P)`			Bck
`invalidates(P, L)`			Bck
`var_used(V, P)`			Lvs
`var_defined(V, P)`			Lvs
`var_drop_used(V, P)`			Lvs
`var_uses_region(V, R)`			Lvs
`var_drops_region(V, R)`			Lvs
`child(M1, M2)`			Init
`path_belongs_to_var(M, V)`			Init
`initialized_at(M, P)`			Init
`moved_out_at(M, P)`			Init
`path_accessed_at(M, P)`			Init

Table 4.0.1: Polonius Input Facts

`borrow_region(R, L, P)` the provenance variable R may refer to data from loan L starting at the point P. This is usually the point *after* the right-hand-side of a borrow expression, when the reference has been assigned and can be accessed. This corresponds to the $\{^{\omega}\pi\}$ part of the typing for borrowing (Rule (3.2.2)), and is used for relating an actual borrow to the first variable holding the created reference in Section 6.3, starting the initial propagation of the loan.

`universal_region(R)` for each named/parametrised provenance variable R supplied to the function. R is considered universally quantified, and therefore live in every point of the function.

`cfg_edge(P, Q)` whenever there is an edge $P \rightarrow Q$ in the control flow graph.

`killed(L, P)` when some prefix of the path borrowed in L is assigned at point P, meaning that reference is overwritten.

`outlives(R1, R2, P)` when $R_1 \subseteq R_2$ must hold at point P, a consequence of subtyping relationships as described in Rule (3.2.6). The term "outlives" has a historical origin in the previous terminology of lifetimes and the input will be renamed in future versions of Polonius.

`invalidates(P, L)` when the loan L is invalidated by some operation at point P.

`var_used(V, P)` when the variable V is used for anything but a drop at point P.

`var_defined(V, P)` when the variable V is assigned to (killed) at point P.

`var_drop_used(V, P)` when the variable V is used in a drop at point P.

`var_uses_region(V, R)` when the type of V includes the provenance R.

`var_drops_region(V, R)` when the type of V includes the provenance R, and V also implements a custom drop method which might need all of V's data, as discussed in Section 6.1.1. Notably, for the MIR in Listing **??**, `var_drops_region(_2, R)` would be emitted to indicate that the `struct` stored in `_2` contains a reference with the provenance variable R in its type, and that this reference could be accessed during the deallocation at this point, were it to happen.

`child(M1, M2)` when the move path M_1 is the child of M_2, That is, for example in the expression `x.y.z`, `x.y.z` is a child of `x.y` and `x`. In the implementation at the time of writing, `child` contained all descendants (children of children), but in future versions the intention is to just use direct relations and have Polonius infer the transitive closures.

`path_belongs_to_var(M, V)` if M is the root path into V.

`initialized_at(M, P)` when the move path M was initialized at point P, such as for example in the expression `x.y = 17`, which would initialise the path `x.y`. Note that the fact is emitted only for the specific path being initialised, and that the transitive initialisation of the prefix' children is implicit.

`moved_out_at(M, P)` when the move path M was moved out (deinitialised) at point P. The same logic about implicit moves as for `initialzed_at` applies here.

`path_accessed_at(M, P)` when the move path M was accessed at point P. This fact is not used in any of the current calculations, but is the final component needed to calculate erroneous accesses of (potentially) moved paths.

4.1 Generating Facts for Polonius in the Rust Compiler

As stated above, the Polonius analysis is performed on the MIR, and the results are then mapped back onto the source code when generating user-facing errors. While Polonius is a self-contained package with a well-defined interface, however, the interface to the code performing the translation of compiler-internal data structures into input facts for Polonius has a much larger surface area. All the additions to the Rust compiler occurs in the `librustc_mir::borrow_check::nll` module, that is alongside the current borrow checker ("NLL"). The module hierarchy and the location of emission of the various facts is shown in Figure 4.1.1. It is worth noting that the Polonius analysis piggy-backs off of previous analysises, notably the `outlives` constraints generated by the previous borrow checker during type-checking.

All facts except `invalidates`, `cfg_edge`, `killed`, `borrow_region`, `outlives`, and `universal_region` were added as part of the work on this thesis.

All inputs based on provenance variables (that is, the ones with "region" in their names from the previous terminology); `path_belongs_to_var`, `universal_region`, `borrow_region`, `var_uses_region`, and `outlives`, are all generated using information obtained during MIR type-checking. The rest of the inputs are generated either from walking the MIR directly (`invalidates`, `cfg_edge`, and all the facts concerning variable uses and drops), or from intermediary indices generated from the MIR in earlier parts of the compilation process (all facts related to move paths, which are identified by previous compilation steps). All of this suggests that the design shown in Figure 4.1.1 should be refactored to reflect these data dependencies, unifying the generation of most facts into a common Polonius module higher up in the hierarchy, and leaving only the ones needing the transient and internal output from the type checker (i.e provenance variables and their relations to each other and to variables) under the `type_check` submodule. This possible future design is discussed in more detail in Chapter 8.

Returning to one of the examples of the borrowing rules in Section **??**, we can describe some of the facts that would be output on each line. An annotated example can be seen in Listing 4.1.1.

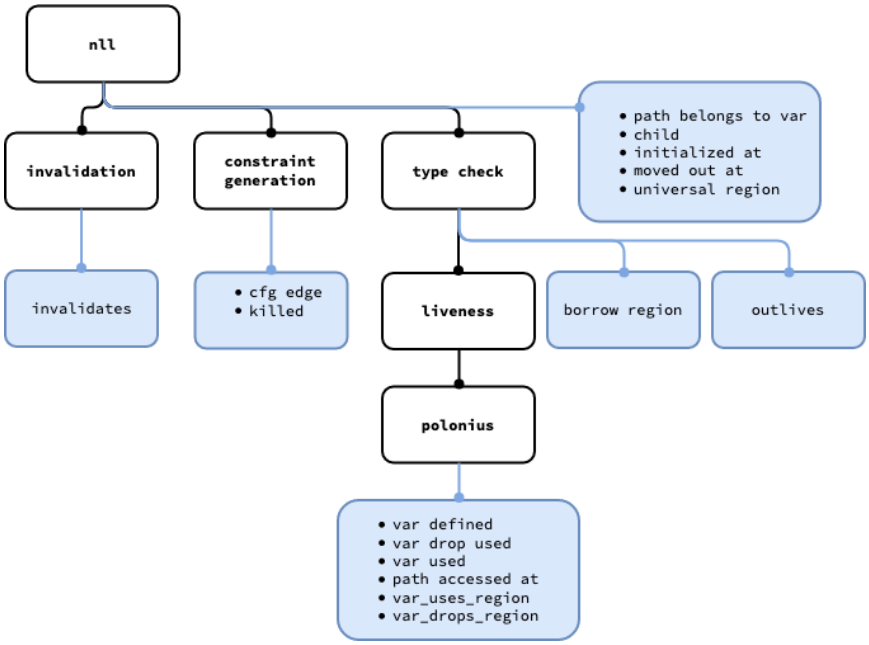


Figure 4.1.1: An illustration of where in the module hierarchy of the Rust compiler the various facts are emitted. Underscores are replaced with white space for readability. Blue boxes represent facts, and black boxes (sub-)modules.

Listing 4.1.1: A minimal example of a violated loan in Rust and the Polonius input facts it would produce during compilation.

```
    let mut pt = Point(6, 9); // var_defined(pt)
    let x = &mut pt;  // var_defined(x),
                      // var_used(pt),
                      // borrow_region('1, b0)
                      // outlives('1, 'x)
                      // var_uses_region(x, 'x)
    let y = &mut pt;  // invalidates(b0)
                      // ...

// we assume var_used(x), var_used(y) is emitted here.
```

In practice, this would happen at the MIR level, which would introduce intermediary variables. However, the core reasoning is the same: the right-hand side of the assignment is typed with a provenance variable `'1`, containing only that loan. The assignment to `x` then sets up a subtyping relationship with the corresponding `outlives('1, 'x)` fact that propagates it to `x`'s provenance variable `'x`, ensuring it is considered live when the loan on the next line generates a fact `invalidates(b0)`, resulting in the eventual derivation of an `error`.

Chapter 5

Datafrog, a Datalog Embedded in Rust

5.1 Why Datalog?

Datalog is a derivative of the logic programming language Prolog, with the desirable properties that any program terminates in polynomial time, and in some variants also with the power to express all polynomial-time computation [8]. It describes fixpoint calculations over logical relations as predicates, described as fixed input *facts*, computed *relations*, or *rules* describing how to populate the relations based on facts or other relations. For example, defining a fact describing that an individual is another individual's child might look like `child(mary, john).`, while computing the `ancestor` relation could then use the two rules, reflecting the fact that ancestry is respectively either direct parenthood or transitive parenthood:

```
ancestor(Mother, Daughter) :- child(Daughter, Mother).
ancestor(Grandmother, Daughter) :-
    child(Mother, Grandmother),
    ancestor(Mother, Daughter).
```

Datafrog [9] is a minimalist Datalog implementation embedded in Rust, providing an implementation of a worst-case optimal join algorithm as described in [10]. The fact that Datafrog is embedded in Rust means that standard Rust language abstractions are used to describe the computation. Static facts are described as `Relations`, while dynamic `Variables` are used to capture the results of computations, both of which are essentially sets of tuples, in our case tuples of integers. Rules are described using a join with either a `Variable` or a `Relation`, with an optimised join method used for joins with only one variable, but multiple relations. Only single-step joins on the first tuple element are possible, which means that more complex rules must be written with intermediary variables, and manual indices created whenever a relation must be joined on a variable which is not the first in the tuple. An example of the `ancestor` relation in Datafrog can be seen in Listing 5.1.1.

Listing 5.1.1: `ancestor(X, Y)` in Datafrog. Note the `map()` invocation, which reverses the tuples of the input `Vec` to fit the reversed target order. This example is used in work-in-progress Polonius code used to derive children of move paths, which is otherwise left outside of the scope of this thesis.

```
let ancestor = iteration.variable::<(T::Path, T::Path)>("ancestor");

// ...

// ancestor(Mother, Daughter) :- child(Daughter, Mother).
ancestor.insert(
    child
        .iter()
        .map(|&(child_path, parent_path)| (parent_path, child_path))
        .collect(),
);

// ancestor(Grandmother, Daughter) :-
ancestor.from_join(
    &ancestor, // ancestor(Mother, Daughter),
    &child,    // child(Mother, Grandmother).
    // select the appropriate part of the match:
    |&_mother, &daughter, &grandmother| (grandmother, daughter),
);
```

Listing 5.1.2: The implementation of `var_use_live(V, P)` in Datafrog.

```
var_use_live_var.from_leapjoin(
    &var_use_live_var,
    (
        var_defined_rel.extend_anti(|&(v, _q)| v),
        cfg_edge_reverse_rel.extend_with(|&(_v, q)| q),
    ),
    |&(v, _q), &p| (v, p),
);
```

Moving on with a more complex example, the Datafrog code for `var_use_live(V, P)` of Listing 6.1.1 becomes the code in Listing 5.1.2, and the corresponding join used for the first half of `region_live_at(R, P)` of Listing 6.1.4 can be seen in Listing 5.1.

Listing 5.1.3: The first half of the implementation of `region_live_at(R, P)` in Datafrog.

```
region_live_at_var.from_join(
    &var_drop_live_var,
    &var_drops_region_rel,
    |_v, &p, &r| {
        ((r, p), ())
    });
```

Joins in Datafrog are done using one of two methods on the variable that is to be populated (e.g. in Listing 5.1 `region_live_at_var`), a variable with tuples of the format `(Key, Val1)`. The first method, `from_join`, performs simple joins from variables

or relations into the (possibly different) target variable. Its arguments, in order, are a `Variable` of type `(Key, Val2)`, and either a second `Variable` or a `Relation` of type `(Key, Val3)`. The third and final argument is a combination function that takes each result of joining the two non-target arguments, a tuple of type `(Key, Val2, Val3)`, and returns a tuple of format `Key, Val1` to be inserted into the target variable.

In the example of Listing , the target variable `region_live_at_var` is populated by joining the `Variable` `var_drop_live_var` to the `Relation` `var_drops_region_rel`. Here, the combination function ignores the variable, returning only the resulting provenance variable and CFG point. The final result is stored in the first half of `region_live_at` with the empty tuple as the second half. This is a work-around to enable joins with two two-tuples.

For more complex joins where a single variable participates in the join and all other arguments are static `Relations` (such as is the case with the variable `var_use_live_var` of Listing 5.1.2), there is `from_leapjoin`. In this case, the input is the sole dynamic source variable, a tuple of "leapers", and a combining function like the one in `from_join`, but with the signature like the one above, mapping a matched tuple from the join to the target of the join.

A leaper is created from a `Relation` of type `(Key, Value)` by either applying the method `extend_with` or `extend_anti` for a join or an anti-join respectively. Both of these functions then take a function mapping tuples from the `Variable` to `Key`s in the `Relation` being (anti-)joined. In the case of `extend_anti`, any tuples matching `Key` are discarded.

In Listing 5.1.2, we can see a leapjoin populating `var_use_live_var` with tuples produced by joining the `Relations` representing `var_defined_rel` and the reversed CFG.

In this thesis, we will use the notation of Soufflé [11] for all examples for clarity and brevity, even though the actual code was written in Datafrog. In other words, starting from the next section, it would be safe to forget you ever read this one.

Chapter 6

Implementing Polonius

In this chapter, we will describe the current implementation of Polonius in Datalog (forgetting the horrors of Datafrog in Section **??**). We will start by discussing how information ("facts") extracted from the user-supplied program via the MIR (introduced in Section **??**) flows through the Polonius analysis to determine if and where the conditions of a loan are violated. We will then go into detail about the actual Datalog rules of Polonius in Section **??**, discuss how the input facts are generated in the Rust compiler (Section 4.1.1), and finish up by discussing what is missing in Polonius in order to perform a full analysis (Section 6.4).

An overview of Polonius can be seen in Figure 6.0.1: initialisation is calculated in order to calculate drop-liveness, which together with regular use-liveness is used to determine the actual liveness of variables. The liveness of variables is then used to determine the liveness of provenance variables in their types, and is used throughout the calculations. Subset relations between provenance variables are used to determine the set membership of loans, and those are then combined with the liveness information in order to determine which loans are live at which point of the program flow. Errors, finally, are generated whenever a potentially violating operation happens to a live loan (an observed tree falls in the woods, thus making a sound).

6.1 Liveness, as Experienced by Polonius

The basic liveness of a variable (Listing 6.1.1) is computed similarly to variable initialisation, except with variable uses instead of initialisations, assignments instead of uses, and backwards across the CFG. Specifically, the rule is as follows: if a variable v is live in some point q and q is reachable from p in the control-flow graph, then v is live in p too unless it was overwritten. Like initialisation, it is also imprecise with respect to branchings, as there is no way to know statically which branch is taken.

Listing 6.1.1: The rules for calculating use-liveness: a variable is use-live if it was used at a point P, or if it was live in Q, there is a transition $P \rightarrow Q$, and it was not defined (killed) in P.

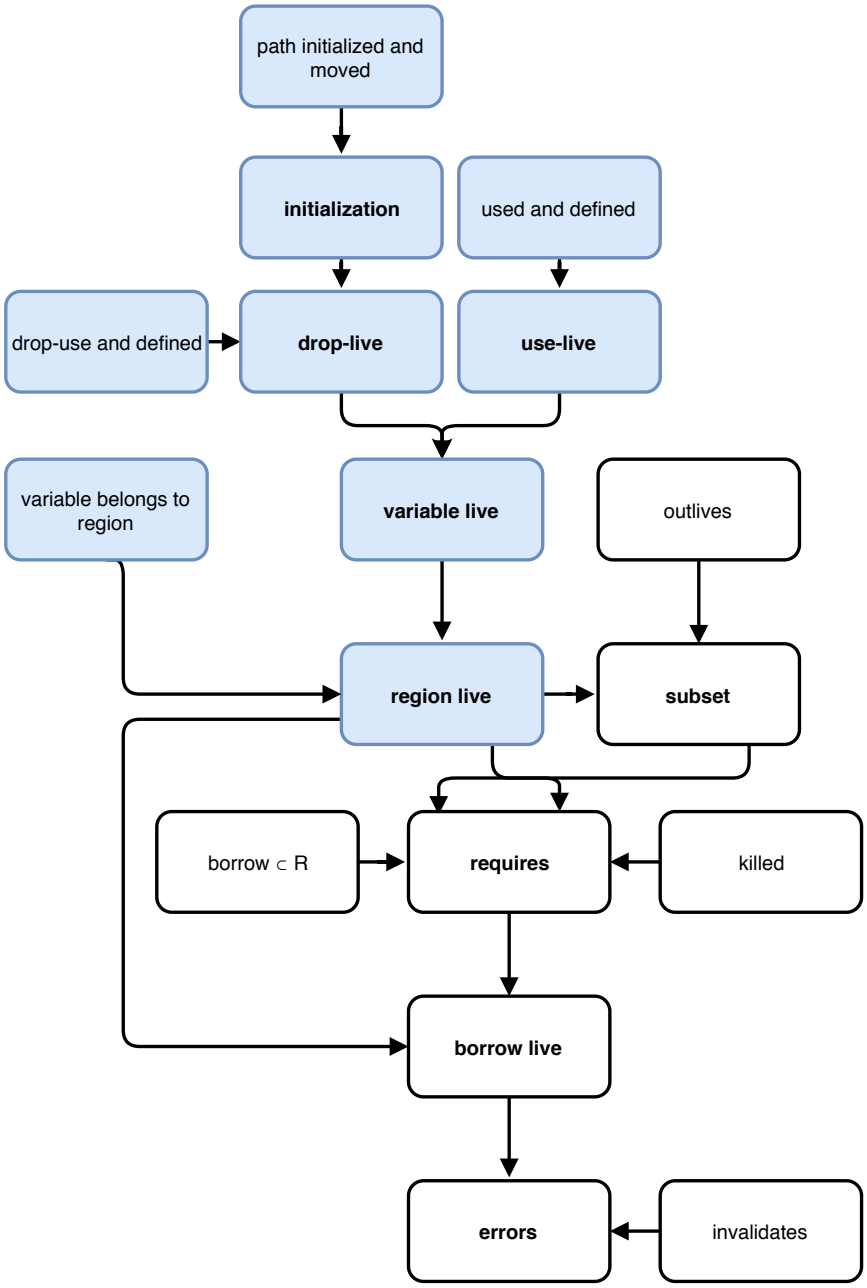


Figure 6.0.1: An overview of how the inputs and intermediate steps of Polonius combine into the final output. Blue boxes represent facts and relations implemented during the work on this thesis. Relations are shown using boldface, and facts in regular font. The historical term "region" is used here instead of provenance variables to match the convention used in the actual code.

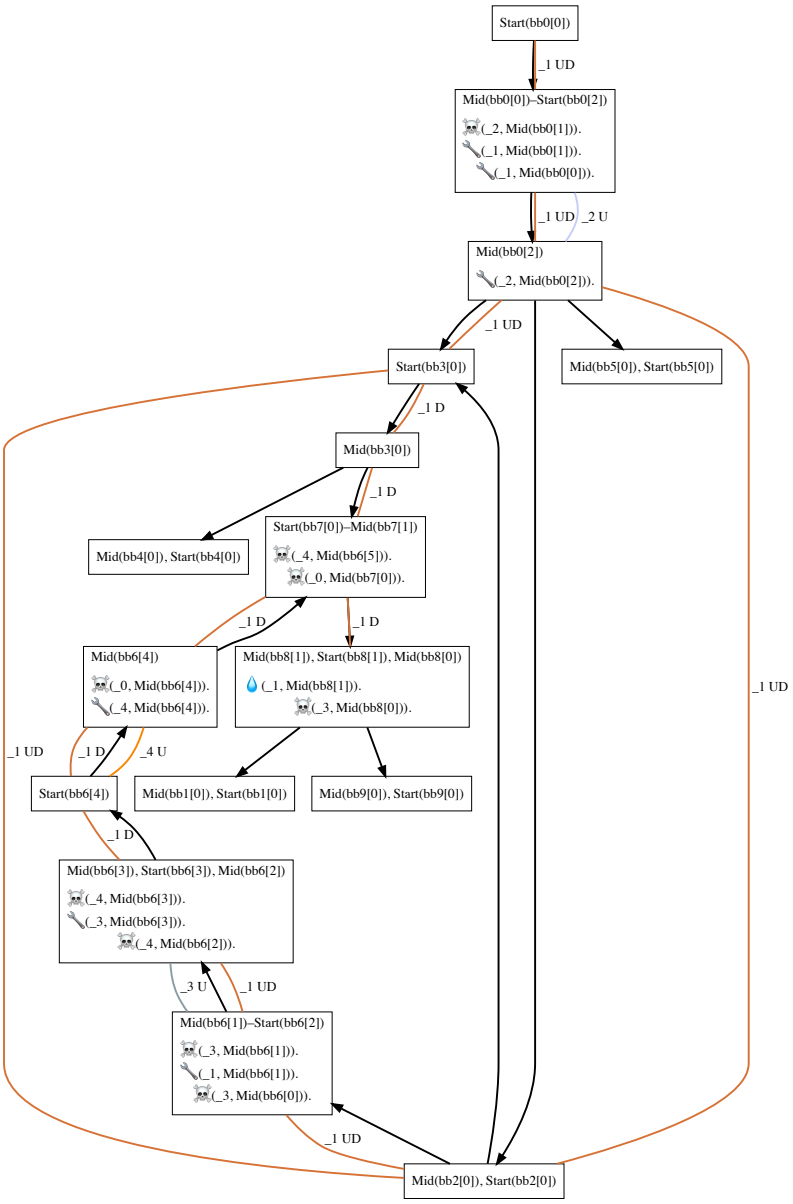


Figure 6.1.1: A graph representation of the the variable liveness calculation results, with relevant Polonius facts as they occur (a droplet symbolising `var_drop_used`*, a wrench* `var_used`*, and a skull and crossbones symbolising* `var_defined`*). Variables are named by prefixing underscores, and edges annotated with the propagated live variable and its liveness type(s) (**D**rop or **U**se).*

```
var_use_live(V, P) :- var_used(V, P).

var_use_live(V, P) :-
    var_use_live(V, Q),
    cfg_edge(P, Q),
    !var_defined(V, P).
```

6.1.1 Deallocation As a Special Case of Variable Use

When Rust's variables go out of scope, they are implicitly deallocated, or dropped in Rust parlance. Explicit deallocation is also possible by calling the function `drop()`, which takes ownership of a variable (that is, deinitialises it) and performs deallocation, or, for complex objects, calls the `drop()` method. For some types such as integers, deallocation is not necessary and the compiler generates no actual `drop()`:s in the MIR. However, the process of inferring this, called drop elision, happens after Polonius is invoked, and therefore Polonius needs to calculate which `drop()` statements would be no-ops.

Rust provides a default deallocator for data structures, but it can be overridden. This has repercussions on liveness calculations, because while the default deallocator for an object never needs to access its fields except to deallocate them, a custom deallocator might access any of them in arbitrary ways. This means that any conditions of a loan that resulted in a reference r stored in a `struct` s instance a must only be respected as far as `a.drop()` is concerned if s implements a custom deallocator. Otherwise the loan of r may be safely violated, as the default deallocator never dereferences r and thus does not require r to be valid. An illustration of this can be seen in Figure 6.1.2.

Following the MIR translation of Listing **??** in Figure **??**, we see across the slightly confusing re-borrows used to move the created references into the `struct`s that the only block of the function ends with a call to `drop()` that would invoke the custom deallocator. Here, the deallocator for `b`, our instance of `DefaultDrop`, is never even called at all.

Drop-liveness is calculated in a similar fashion to use-liveness, with the exception that a deinitialised variable is never dropped, and therefore is not considered drop-live. This is the reason for the computation of variables that might be initialised in the previous section. The rules can be found in Listing 6.1.3.

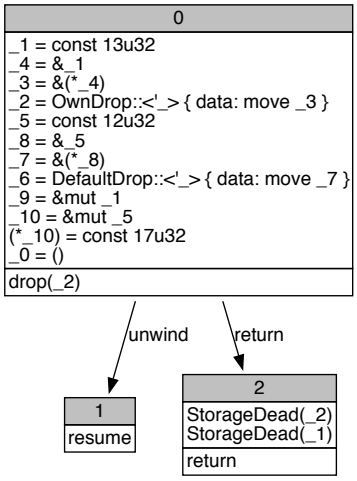
Note the use of the first rule, which is not transitive, to shift the point of the initialisation from the input's mid-point index (which is where a (de)initialisation would take effect) to the statement's starting-point. This is because a drop-use would only happen if the `x` was initialised on *entry* to the instruction `drop(x)`.

An example of the output from this calculation can be seen in Figure 6.1.1.

Listing 6.1.3: The rules for calculating drop-liveness: the rules are similar to those for to use-liveness (Listing 6.1.1), but propagation of liveness only happens if the variable being dropped may be initialised. Note that the rule for calculating initialisation on entry is not transitive!

```
var_maybe_initialized_on_entry(V, Q) :-
    var_maybe_initialized_on_exit(V, P),
```

Listing 6.1.2: The custom deallocator for `OwnDrop` enforces the loan giving the reference `data` until the struct is deallocated, but the loan in `DefaultDrop` is effectively dead as soon as it has no direct uses in the code and thus can be violated.




```rust
struct OwnDrop<'a> {
    data: &'a u32,
}

struct DefaultDrop<'a> {
    data: &'a u32,
}

impl<'a> Drop for OwnDrop<'a> {
    fn drop(&mut self) {
        // might access self.data
    }
}

fn main() {
    let mut x = 13;
    let a = OwnDrop { data: &x };

    let mut y = 12;
    let b = DefaultDrop { data: &y };

    let mutrefa = &mut x;
    // ERROR: the loan of x must be respected...

    // ...but the loan of y need not be!
    let mutref = &mut y;
    *mutref = 17;

    // all variables are implicitly dropped here
}
```

Figure 6.1.2: A graph rendering of the MIR produced from the `main()` function of the code to the right, illustrating a call to the custom deallocator of `_2` that would cause it to be drop-live during the block. Take special note of the lack of calls to `drop(_6)`; as `DefaultDrop`, the `struct` stored in `_6`, uses the default deallocator and contains only a reference, deallocating it is a no-op. Some irrelevant details, such as hints about stack allocations and deallocations of intermediate variables, have been pruned.

```
    cfg_edge(P, Q).

var_drop_live(V, P) :-
    var_drop_used(V, P),
    var_maybe_initialzed_on_entry(V, P).

var_drop_live(V, P) :-
    var_drop_live(V, Q),
    cfg_edge(P, Q),
    !var_defined(V, P)
    var_maybe_initialized_on_exit(V, P).
```

The two kinds of liveness are then used to calculate the reference liveness relation (Listing 6.1.4), which serves as input for the rest of the borrow checker. A given provenance variable R is live at some point p if it is in the type of a variable v which is either drop-live or use-live at p, with some notable caveats for drop-liveness (discussed in Section 6.1.1) embedded in the `var_drops_region` relation. In essence, even if v is a `struct` containing a reference with a provenance variable R, this point would have `var_drops_region(V, R)`, showing that the drop-use of v would require the liveness of the variable holding a reference with R in its type.

Listing 6.1.4: A provenance variable is live if it either belongs to a use-live variable, or if it might be dereferenced during the deallocation of a drop-live variable.

```
region_live_at(R, P) :-
    var_drop_live(V, P),
    var_drops_region(V, R).

region_live_at(R, P) :-
        var_use_live(V, P),
        var_uses_region(V, R).
```

6.2 Move Analysis

The idea behind variable initialisation calculations is a fairly straightforward transitive closure computation. Initialisation for a path propagates forwards from an initialisation event across the CFG until the path is deinitialised. As a consequence of this, initialisation tracking is imprecise (over-estimating) upon branching; if one path to a node in the CFG has v initialised and one does not, v is considered initialised for the purposes of this analysis.

For the purposes of this analysis, we mean by initialisation also partial initialisation of a complex variable. Therefore, initialisation also propagates upwards through the path tree, such that `x` is (partially) initialised whenever `x.y` is. An expression like `move x` would, in this example, only emit `moved_out_at(x, P)` as a starting

fact. This means that currently, initialisation tracking is imprecise with respect to parts of the variable as well as across branchings, a strictly speaking unnecessary imprecision. Future versions of Polonius will instead use a precise calculation here, but for the purposes of determining drop-liveness in the next section these calculations will suffice.

Finally, `path_belongs_to_var(M, V)` connects paths to their root variables. It is worth noting here that this fact only needs to contain a mapping of the root path to a variable, as initialisation always bubbles up through the tree due to the imprecision mentioned above. The full Datalog code is shown in Listing 6.2.1.

Listing 6.2.1: The rules for computing possible partial variable initialisation. A path is trivially initialised where it is actually initialised. It is transitively initialised in all points reachable from a point where it is initialised, and where it has not been deinitialised (moved out). Initialisation propagates upwards in the move path tree, until it reaches the variable at the root of the path.

```
path_maybe_initialized_on_exit(Path, Point) :-
    initialized_at(Path, Point).

path_maybe_initialized_on_exit(M, Q) :-
    path_maybe_initialized_on_exit(M, P),
    cfg_edge(P, Q),
    !moved_out_at(M, Q).

path_maybe_initialized_on_exit(Mother, P) :-
    path_maybe_initialized_on_exit(Daughter, P),
    child(Daughter, Mother).

var_maybe_initialized_on_exit(V, P) :-
    path_belongs_to_var(M, V),
    path_maybe_initialized_at(M, P).
```

6.3 Loan Constraint Propagation†

The first relation used in Polonius is the `subset(R1, R2, P)` relation, which states that $R_1 \subseteq R_2$ for two provenance variables R_1, R_2 at point p in the CFG, and correspond to the constraints generated during validation of expressions involving subtyping, as discussed in Section 3.2. Initially, these have to hold at the points where the constraints are generated by the Rust compiler, as seen by the input parameter `outlives`. The brief one-liner in Listing 6.3.1 captures this fact, providing a "base case" for the computation. Additionally the mathematical fact that the subset relation is transitive is captured in Listing 6.3.2.

Listing 6.3.1: Subset relations hold at the point where they are introduced.

```
subset(R1, R2, P) :- outlives(R1, R2, P).
```

Listing 6.3.2: Subset relations are transitive.

```
subset(R1, R3, P) :-
    subset(R1, R2, P),
    subset(R2, R3, P).
```

Finally, Polonius needs logic to carry these subset relations across program flow. However, as mentioned before, we are only interested in detecting violations of loans that are actually live. Therefore, subset relation should be propagated across an edge of the control-flow graph if and only if its provenance variables are live, otherwise we are in a "if a tree falls in the woods" situation where the conditions of the loans can be safely violated as there is no live reference to be affected. Therefore, the rule for propagating the subset constraint across a CFG edge $P \rightarrow Q$ becomes the formulation seen in Listing 6.3.3, using the output of the liveness calculations described in Section **??**.

Listing 6.3.3: Subset relations propagate across CFG edges iff their provenance variables are live.

```
subset(R1, R2, Q) :-
    subset(R1, R2, P),
    cfg_edge(P, Q),
    region_live_at(R1, Q),
    region_live_at(R2, Q).
```

These rules describe how provenance variables relate to each other. The other part of the logic describes which loans belong to which provenance variable. The trivial base case is shown in Listing 6.3.4, which just says that each provenance variable R contains the loan L that created it at point the point P where the borrow occurred.

Listing 6.3.4: A provenance variable trivially contains (`requires`) the loan which introduced it.

```
requires(R, L, P) :- borrow_region(R, L, P).
```

Additionally, the `requires` relation needs to be propagated together with subset constraints; after all $R_1 \subseteq R_2$ implies that R_2 must contain (`require`) all of R_1's loans. This is captured by the rule in Listing 6.3.5.

Listing 6.3.5: A subset relation between two provenance variables R_1, R_2 propagates the loans of $R1$ to $R2$.

```
requires(R2, L, P) :-
  requires(R1, L, P),
  subset(R1, R2, P).
```

Finally, Polonius performs the flow-sensitive propagation of these membership constraints across edges in the CFG. This is done using the rule in Listing 6.3.6, where the requirements propagate across CFG edges for every loan L as long as the reference corresponding to L is not overwritten (`killed`), and only for provenance variables that are still live. This corresponds to the T-Assignment rule of Oxide, seen in Rule (3.2.4).

Listing 6.3.6: Propagate loans across CFG edges for live provenance variables and loans whose references are not overwritten.

```
requires(R, L, Q) :-
  requires(R, L, P),
  !killed(L, P),
  cfg_edge(P, Q),
  region_live_at(R, Q).
```

Detecting Loan Violations

The compiler produces a set of points in the CFG where a loan could possibly be violated (e.g. by producing a reference to a value that already has a unique reference) in `invalidates`. All that remains for Polonius is to figure out which loans are live where (Listing 6.3.7), and determine if any of those points intersect with an invalidation of that loan (Listing 6.3.8).

Listing 6.3.7: Loans are live when their provenance variables are.

```
loan_live_at(L, P) :-
  region_live_at(R, P),
  requires(R, L, P).
```

Listing 6.3.8: It is an error to invalidate a live loan.

```
error(P) :-
  invalidates(P, L),
  loan_live_at(L, P).
```

6.4 What is Missing from Polonius?

In addition to polish, comprehensive benchmarking, and performance optimisations, all discussed later, there are three important features missing in Polonius before it reaches parity with NLL, the current borrow checker.

6.4.1 Detecting Access to Deinitialised Paths

The current Polonius implementation only uses move data to derive conditional initialisation of variables in order to determine if they would be deallocated. However, the full borrow check would also calculate paths that *may have been moved out* and emit errors on access, such as in this code:

```
let tuple: (Vec<u32>, Vec<u32>) = (vec![], vec![]);
drop(tuple.0); // moved out of `tuple`
println!("{:?}", tuple.0); // ERROR
```

All the necessary input facts are already collected but the actual implementation and testing of the logic depends on a re-designed interface between the Rust compiler and Polonius, which would have required extensive interaction with the rest of the compiler team. However, the lion's part of the work is in place.

6.4.2 Illegal Subset Relations

Polonius currently does not verify that a subset relationship it finds between provenance variables is actually valid in itself. For example, this unsound code would not generate an error in today's Polonius:

```
fn pick_one<'x, 'y>(x: &'x [u32], y: &'y [u32]) -> &'x u32 {
    &y[0]
}
```

In this case, `pick_one()` takes two slices with some unknown provenance variables at least known to live for the duration of the function body. The subtyping rules would give that `'y` $\subseteq$ `'x` at the end of the function, because the reference into `y` must be a subtype of `&'x u32`, the return type. However, this cannot be guaranteed to hold in general, as Polonius (currently) knows nothing about the relationship between these two provenance variables, and in fact, as `pick_one()` is polymorphic over these provenance variables, this must hold for *any* pair of provenance variables `'x`, `'y`, which it certainly does not [12].

6.4.3 Analysis of Higher Kinds

The final missing functionality in Polonius is interaction with higher-ranked (generic, etc) subtyping arising from generic functions or trait-matching. The problem was described in a blog entry by Matsakis and will require extensions in the Rust compiler, which would produce simpler constraints than the universally and existentially quantified constraints generated by the type checker for Polonius to solve [13]. The current plan is to use the already existing infrastructure in Rust for this, but at the time of writing work on this has not even reached the planning stage.

6.4.4 Addressing a Provenance Variable Imprecision Bug

During the work for this thesis, a shortcoming in both Polonius and (probably) Weiss, Patterson, Matsakis, and Ahmed's Oxide, discussed in Section 3.2 was discovered, which would generate spurious errors in examples like Listing 6.4.1 where an imprecision in the tracking of subset relations would cause a loan to be propagated to a provenance variable erroneously, leading to effectively dead loans being considered live. Correcting this problem would require modifications to how the propagation of subset relations across the CFG works, which would not concern the liveness or initialisation tracking implemented as part of this thesis, but would affect the solution described in Section 6.3. At the conclusion of the work for this thesis, the Polonius working group had not yet produced a final reformulation of Polonius that would address this issue.

Listing 6.4.1: An example where the current Polonius loses precision and emits a spurious error, as it conflates the provenance variables `'x` and `'y`.

```
let mut z: u32;
let mut x: &'x u32;
```

```
let mut y: &'y u32;

if something {
  y = x; // creates `'x subset-of 'y`.
}

if something {
  x = &z; // creates {L0} in 'x constraint.
          //
          // at this point, we have
          //   `'x subset-of 'y` and `{L0} in `'x`,
          //   so we also have `{L0} in 'y` (wrong).
  drop(x);
}

z += 1; // Polonius: false positive error

drop(y);
```

6.5 Conclusion

Chapter 7

A Field Study of Polonius Inputs

We selected for analysis roughly 20 000 publicly available Rust packages ("crates") from the most popular projects as defined by number of downloads from Crates.io and number of stars on GitHub. [1] Of the initially selected repositories only about 1 000 were from other sources than GitHub. Only crates that compiled under recent versions of Rust nightly builds with non-linear lifetimes enabled were kept. This was due to the difficulty of isolating compilation errors due to missing dependencies on external C libraries or syntactically invalid code, both of which would happen long before Polonius in the compilation process, from errors that would involve Polonius. The source code of the packages was then translated to Polonius input files for a total of 340 GBs of tuples for 3 939 171 Rust functions (user-written as well as compiler-generated), which we used to measure Polonius runtime performance as well as for finding common patterns in the input data. Only complete data sets were considered; a repository with more than one target where at least one target did not compile was discarded, as was any repository where the analysis of input facts took more than 30 minutes, required more memory than what was available, or where the initial fact generation phase took longer than 30 minutes. After this selection process, 12 036 repositories remained for the final study, each of which contained at least one, but possibly multiple crates. The analysis assumed that all functions in all crates and all targets of a repository were unique, as the outputs were stored per-repository. The median number of functions in the dataset was 48, including functions generated by desugaring as well as user-written functions.

All experiments were run on a dedicated desktop computer running a 64-bit version of Ubuntu 19.04 with Linux 5.0.0-20-generic. The machine had 16 GBs of 2666 MHz CL16 DDR4 RAM, and a AMD Ryzen 5 2600 CPU running at a base clock of 3.4 GHz (max boost clock 3.9 GHz) with cache sizes of 576 KB (L1), 3 MB (L2), and 16 MB (L3). Executing the full set of jobs took around two weeks.

[1] Source code for the analysis as well as listings of the repositories are available at `https://github.com/albins/msc-polonius-fact-study`.

Additionally, we also excluded all functions that had no loans at all from the analysis, a surprisingly large portion; slightly above 64%. This is most likely due to code generation producing short "functions" that does not actually involve any borrowing at all. After discarding these, 11 687 repositories remained.

The main metric of "performance" in this study is the time it would take Polonius to solve a given set of inputs from a cold start. This also includes the time it takes to parse the files of tab-separated input tuples, initialisation, liveness, and the borrow check. In practical scenarios the peak memory usage of the analysis would also be an interesting metric. Additionally, a future benchmarking scenario should use Polonius to benchmark itself rather than an external wall-clock, allowing for more precise measurements excluding parsing and deserialisation and reporting separate runtimes for the three phases of the calculation.

When studying inputs to Polonius, we are mainly interested in two properties; how large and how complex the function under analysis is. Neither of these can be measured directly, but potentially useful proxy variables would be sizes of input tuples, the number of variables, loans, and provenance variables, as well as common and cheaply computed graph complexity metrics such as the node count, density, transitivity, and number of connected components of the control-flow graph.

Three variants of Polonius were included in the study; a Naive implementation, which is the one described in Section **??**, an optimised variant (DatafrogOpt †), and a variant that first executes a simpler analysis assuming lexical lifetimes and falls back to the full Polonius analysis only when that one produces an error (Hybrid). The intention is to have such a hybrid algorithm re-use the information gained by the simpler analysis to accelerate the more advanced analysis, but such functionality was not yet implemented at the time of the experiments. This mode also performs the full liveness and initialisation analysis twice, penalising it in the comparison.

The box plots in Figures 7.0.1, 7.0.2, and 7.0.4 are all Tukey plots; the green line shows the median, the box the 1 and 3rd quartile, and the whiskers are placed at 1.5 times the interquartile range. Outliers are not plotted, as the size of the input resulted in too many outliers for the plots to be readable.

7.0.1 Performance

In general, all three algorithms finished quickly for almost all functions, with both of the optimised algorithms already showing improvements in runtimes, as seen in Figure 7.0.1. Apparently, Naive has a wider spread of runtimes than the others. Additionally, geometric means of the observed runtimes show improvements from hybridisation (Figure 7.0.3), though it should be noted that the algorithm's worst-case of an input that fails both the simple and the full analysis was left out of the sample as that would have failed compilation, possibly inflating the results artificially. We can also see clearly that Hybrid outperforms its fallback flow-sensitive DatafrogOpt implementation even when excluding smaller inputs 7.0.2.

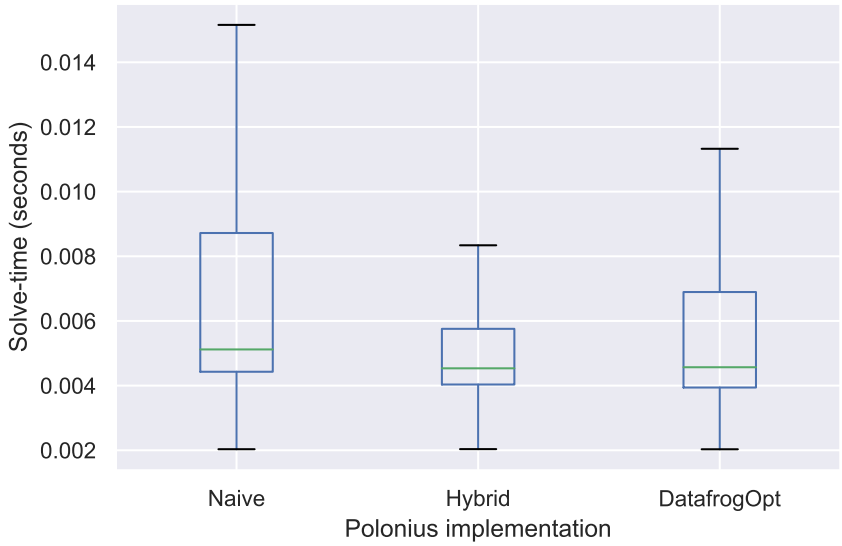


Figure 7.0.1: A box plot showing the distribution of runtimes per function for three implementations of Polonius. As can be seen here, the vast majority execute very quickly.

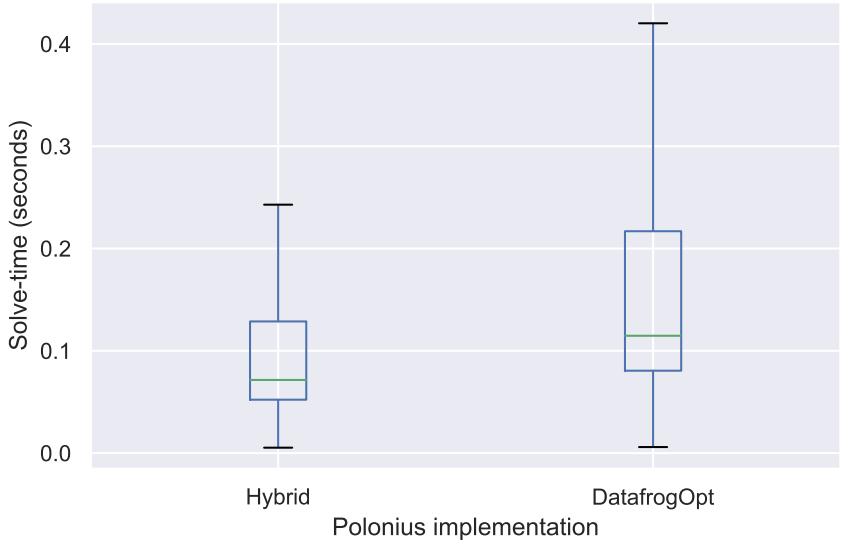


Figure 7.0.2: A box plot showing the distribution of runtimes per function for the two optimised Polonius implementations on just functions that executed in between 1–50s on Naive.

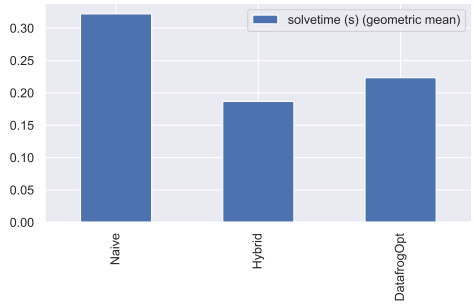


Figure 7.0.3: Geometric means of the runtimes per repositoriy and implementation.

7.0.2 Characteristics of Real-World Polonius Input Data

A typical Polonius input consists of a small number of tuples for most relations, as seen in Figure 7.0.4. In particular, most control-flow graphs are small in terms of number of nodes, and most functions only contain a small number of variables, with an even smaller number of loans. Drops are particularly rare, with circa 70% of all studied functions having no (potential) drop-uses at all (0 median, 7.6 mean), and only very few loans (2 median, 5 mean). This can also be seen in Figure 7.0.5 showing the distribution of number of (potential) drop-uses per function. In practice, this means that users generally do not override the built-in deallocators, do not explicitly deallocate their variables. The low number of loans also means that functions in general do not use complicated reference-sharing, typically only manipulating a few references.

This points towards a need to have a low starting overhead for Polonius, as much of its analysis would have to be performed on very small inputs, where the runtime would be dominated by a high constant setup time.

However, repositories can be assumed to be typically compiled all at once. Therefore, it is also interesting to say something about the maximum input size per repository, under the assumption that few large functions would dominate the runtime for that repository. After collecting the maximum values per repository, the median number of loans was 24, and the median number of potential drop-uses was 45 (regular uses was, for comparison, 177).

We attempted to perform a principal-component analysis (PCA) of the input data in order to visually identify possible clusterings of types of inputs, but the results were unusable as the inputs had no visually discernible patterns in neither 2 nor 3 dimensions, suggesting that most inputs are in some sense typical, or that PCA is ineffective here.

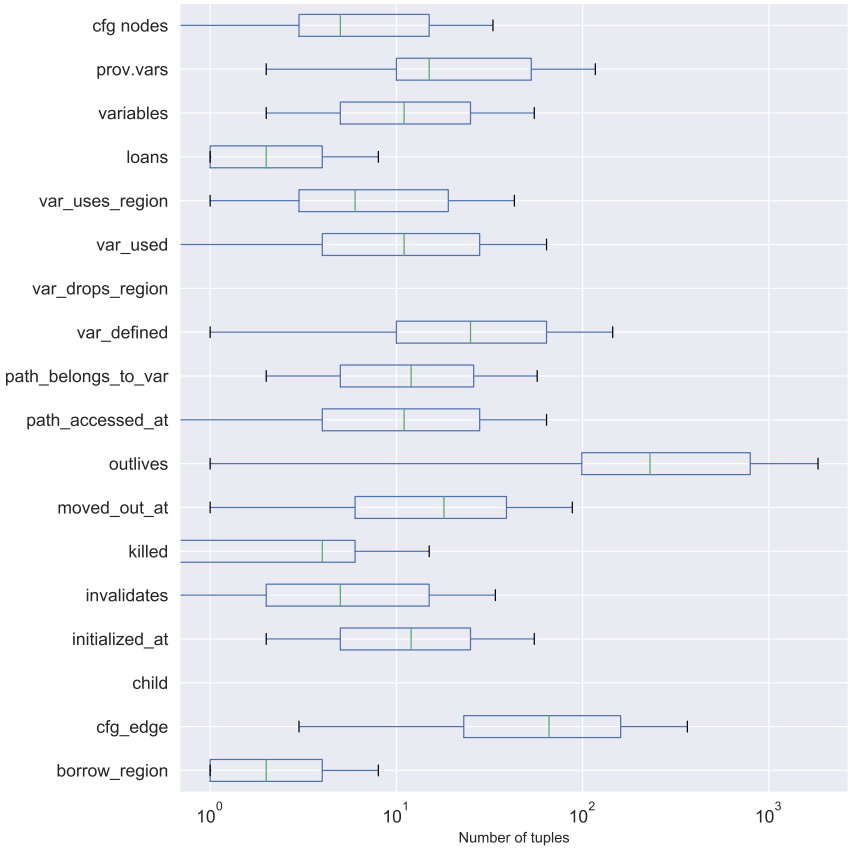


Figure 7.0.4: A box plot showing the distribution of the various input sizes.

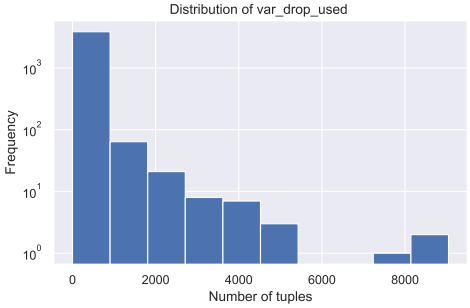


Figure 7.0.5: A plot showing the distribution of `var_drop_used`.

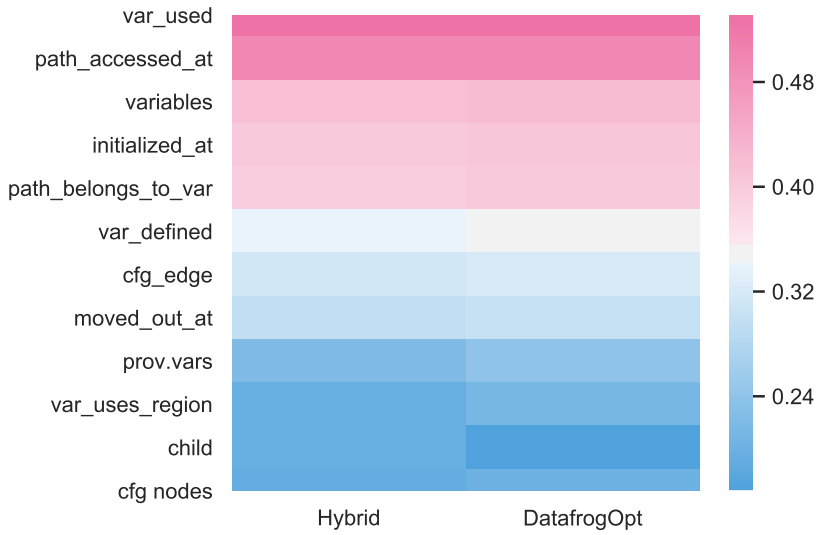


Figure 7.0.6: Heatmap of Pearson correlations between various input size metrics and runtimes for all three Polonius implementations, suggesting in particular that variable uses, number of variables, and the number of provenance variables heavily affect runtime.

7.0.3 How Inputs Affect Runtime

A heatmap of the (Pearson) correlation between input size and runtime for the various variants on long-running jobs (as previously defined to be jobs taking at least 1s and no more than 50s to run under Naive) can be seen in Figure 7.0.6 and Table 7.0.1, while a scatter plot of the results with a linear regression for some interesting pairs of inputs can be seen in Figure 7.0.7.

It is clear here that inputs affecting all parts of the computation have a larger influence, notably variable uses, number of variables, and the number of provenance variables. In particular, input sizes affecting the liveness computation time affects Hybrid, which should be no surprise as it does that computation twice. The same goes for the number of provenance variables, which figure in the second two parts of the analysis. Another conclusion from Table 7.0.1 is that the number of nodes of the CFG has a lower impact on runtime than its number of edges, reflecting that complex CFGs with many branchings take more time to compute than linear ones.

Both results suggest only a weak linear relation between input sizes and and the runtime with Naive, while a clearer relation can be found between DatafrogOpt and input sizes respectively. Naive, on the other hand, does not show similarly clear correlations between runtime and input sizes of any kind (Table 7.0.1).

	Naive	Hybrid	DatafrogOpt
var_used	0.363947	0.531098	0.531099
path_accessed_at	0.349286	0.497498	0.498295
variables	0.304380	0.418331	0.423401
initialized_at	0.309505	0.403256	0.407881
path_belongs_to_var	0.298802	0.398564	0.403784
var_defined	0.279185	0.340064	0.348787
cfg_edge	0.296521	0.313906	0.322021
moved_out_at	0.286129	0.295944	0.302707
prov.vars	0.193118	0.222522	0.239422
var_uses_region	0.175650	0.195924	0.212868
cfg nodes	0.279858	0.193068	0.200334
child	0.270583	0.195513	0.168010
loans	0.137711	0.133432	0.151154
borrow_region	0.137711	0.133432	0.151154
killed	0.080521	0.101943	0.102579
invalidates	0.044914	0.082414	0.084853
outlives	0.206378	0.062142	0.082695
var_drop_used	0.195021	0.031959	0.043753
var_drops_region	0.135435	0.013908	0.023246

Table 7.0.1: Pearson correlations between size of inputs and the runtime of Naive, Hybrid, *and* DatafrogOpt *respectively, from high correlation to* DatafrogOpt *runtime to low.*

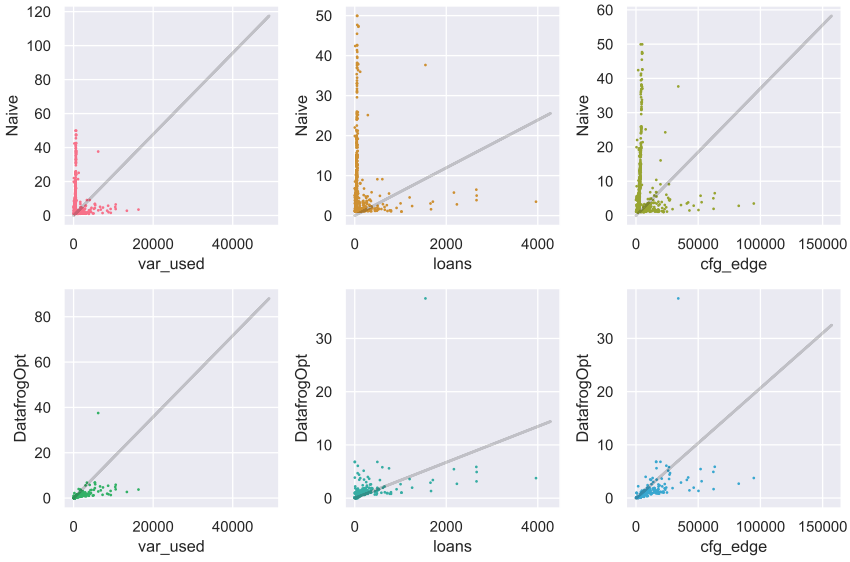


Figure 7.0.7: Scatter plot of runtimes under the naive and optimised algorithms compared to variables and CFG edge count after having pruned extreme values (runtimes below 1 s or above 13 minutes). Y axis is runtime in seconds.

Chapter 8

Conclusions and Future Work

Before Polonius can replace NLL as the Rust borrow checker, it would need considerable performance improvements in both its fact generation process as well as the solving itself. In its current condition, the fact generation code, in particular, performs multiple walks across the CFG, needlessly increasing runtime. Additionally, many of the inputs are computed unnecessarily, and, for example, the CFG could be compressed for some cases.

Returning to the analysis of Section 7, we can see from the performance of even the current naive Hybrid implementation, which first performs a non-flow sensitive analysis and then falls back to the full Polonius analysis, outperforms both the optimised analysis alone and Naive. We can also see that inputs without any loans at all are common, and in those cases the analysis can typically terminate before performing any analysis at all. Finally, Naive could be improved in two ways. First, in the current implementation initialisation and liveness analysis is performed twice for purely architectural reasons. A better implementation would calculate them once and re-use the results. Second, the current analysis does not use the errors from the flow-insensitive analysis when it falls back to the full flow-sensitive Polonius. Recycling the errors from the first analysis could in many cases reduce the search space for Polonius significantly, as any other error has already been ruled out in the simpler analysis.

Finally, Datafrog itself could be optimised, including using faster vector instructions or parallelisation techniques. Additionally, several of the input relations used in Polonius are only used to exclude values, and never used to propagate them. This suggests it would be possible to use more compact data structures for representing them, such as Bloom filters.

In this report, we have described a first implementation of the Rust borrow check in Datalog. We have shown how partial initialisation tracking was used along with variable-use and definition data to determine live references, which were then used to detect which potential loan violations happening in the code would actually be of a live reference, therefore causing an error.

Building on top of this, we then analysed Rust code from ca 12 000 popular Git repositories to determine what a characteristic Polonius input would look like. The study found that relatively few functions use any references at all, suggesting that the borrow check should be able to terminate early in a significant number of cases. On the same note, we also found that foregoing the full flow-sensitive analysis and falling back on a simpler analysis, even naively, in many cases improves performance significantly. Finally, the study concluded that the number of transitions in the control-flow graph and the number of variables both would be good proxies for the difficulty of solving an input in Polonius, in terms of run-time.

Left to do in Polonius before it is feature-complete is integrating it with the Rust type checker for higher-order kinds, finishing the full initialisation tracking, and extending the analysis to also include illegal subset constraints on reference type provenance variables. Finally, we also briefly discussed a recently discovered shortcoming believed to exist in both Polonius and the Oxide formulation [3], related to provenance variable imprecision in the analysis causing spurious errors. This issue is currently under investigation, and addressing it would likely impact the performance of Polonius, though possibly in a positive direction as a less precise formulation would potentially (in some cases) produce fewer tuples to propagate during analysis.

Finally, there is a need to refactor both Polonius itself (whose interface is outside the scope of this thesis), and the fact generation code of Figure 4.1.1. Such a refactoring could even reduce the number of iterations over the MIR during input generation, decreasing the runtime of that part of the code. A proposal for how the fact-generation code could be reorganised is shown in Figure 8.0.1. The key idea is to divide the fact generation code according to where in the compilation process it takes is inputs, such that only the parts needing access to the internal parts of the type-checker are executed during type-checking. This grouping of code according to the data it operates on also means that costly operations, notably CFG iteration, can be performed all at once.

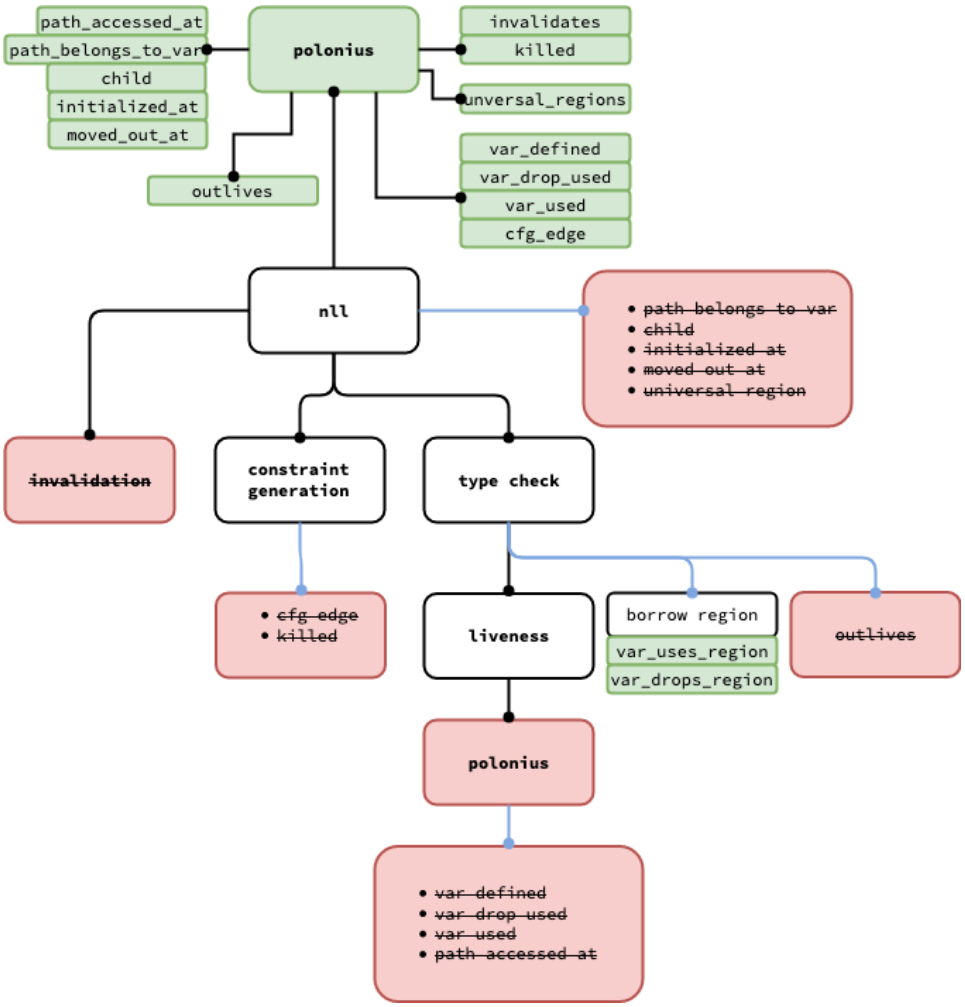


Figure 8.0.1: A suggestion for how the Polonius fact generation in Rust can be reorganised. Green boxes show inputs, black boxes Rust modules, and red modules (re)moved components. Note that boxes are grouped together according to the inputs necessary for producing them.

Bibliography

[1] N. D. Matsakis and F. S. Klock II, "The Rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '14, event-place: Portland, Oregon, USA, New York, NY, USA: ACM, 2014, pp. 103–104, isbn: 978-1-4503-3217-0. doi: 10.1145/2663171.2663188. (visited on 03/29/2019) (cit. on p. 7).

[2] *Issue 51132: Borrowing an immutable reference of a mutable reference through a function call in a loop is not accepted*, Aug. 14, 2015. [Online]. Available: https://github.com/rust-lang/rust/issues/51132 (visited on 04/01/2019) (cit. on p. 10).

[3] A. Weiss, D. Patterson, N. D. Matsakis, and A. Ahmed, "Oxide: The essence of Rust," Mar. 3, 2019. [Online]. Available: https://arxiv.org/abs/1903.00982v1 (visited on 04/21/2019) (cit. on pp. 11–13, 36, 46).

[4] N. D. Matsakis. (Apr. 27, 2018). An alias-based formulation of the borrow checker, Baby Steps, [Online]. Available: http://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/ (visited on 04/01/2019) (cit. on p. 11).

[5] *RFC 2094: Non-lexical lifetimes*, original-date: 2014-03-07T21:29:00Z, Apr. 1, 2019. [Online]. Available: https://github.com/rust-lang/rfcs (visited on 04/01/2019).

[6] Rustc developers. (Jul. 17, 2019). Guide to Rustc development, [Online]. Available: https://rust-lang.github.io/rustc-guide/ (visited on 07/17/2019) (cit. on p. 18).

[7] *RFC 1211: Mid-level IR (MIR)*, Aug. 14, 2015. [Online]. Available: http://rust-lang.github.io/rfcs/1211-mir.html (visited on 04/01/2019) (cit. on p. 20).

[8] F. Afrati, S. S. Cosmadakis, and M. Yannakakis, "On Datalog vs polynomial time," *Journal of Computer and System Sciences*, vol. 51, no. 2, pp. 177–196, Oct. 1, 1995, issn: 0022-0000. doi: 10.1006/jcss.1995.1060. (visited on 03/29/2019) (cit. on p. 24).

[9] *Datafrog: A lightweight Datalog engine in Rust*, Mar. 29, 2019. [Online]. Available: https://github.com/rust-lang/datafrog (visited on 04/01/2019) (cit. on p. 24).

[10] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra, "Worst-case optimal join algorithms," in *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI symposium on Principles of Database Systems*, ACM, 2012, pp. 37–48 (cit. on p. 24).

[11] B. Scholz, H. Jordan, P. Subotić, and T. Westmann, "On fast large-scale program analysis in datalog," in *Proceedings of the 25th International Conference on Compiler Construction*, ACM, 2016, pp. 196–206 (cit. on p. 26).

[12] N. D. Matsakis. (Jan. 17, 2019). Polonius and region errors, Baby Steps, [Online]. Available: http://smallcultfollowing.com/babysteps/blog/2019/01/17/polonius-and-region-errors/ (visited on 04/01/2019) (cit. on p. 36).

[13] ——, (Jan. 21, 2019). Polonius and the case of the hereditary harrop predicate, Baby Steps, [Online]. Available: http://smallcultfollowing.com/babysteps/blog/2019/01/21/hereditary-harrop-region-constraints/ (visited on 04/01/2019) (cit. on p. 36).