Modelling Rust's Reference Ownership Analysis Declaratively in Datalog

Albin Stjerna

December 29, 2019

Abstract

Rust is a modern systems programming language that offers improved memory safety over traditional languages like C or C++ as well as automatic memory management without introducing garbage collection. In particular, it guarantees that well-typed programs are 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 do not violate the Rust memory ownership rules. This report introduces an improved borrow check called Polonius, which increases the resolution of the analysis to reason at the program statement level, and enables a more expressive formulation of the borrow check itself through the use of a domain-specific language, Datalog. To the best of our knowledge, Polonius is the first use of Datalog for type verification in the compiler of a production language.

Specifically, this thesis extends Polonius with initialisation and liveness computations for variables, and constitutes the first complete description of Polonius in text. Finally, it describes an exploratory study of 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 a weaker (and therefore faster) analysis than Polonius is often sufficient to prove a program correct. Indeed, many functions (circa 64%) do not create any references at all, and therefore do not involve the reference-analysis part of the borrow check.

Contents

List of Figures

List of Tables

Acknowledgements

> We seek not the knowledges ruled by phallogocentrism (nostalgia for the presence of the one true Word) and disembodied vision. We seek those ruled by partial sight and limited voice—not partiality for its own sake but, rather, for the sake of the connections and unexpected openings situated knowledges make possible. Situated knowledges are about communities, not about isolated individuals. **The only way to find a larger vision is to be somewhere in particular**
>
> Donna Haraway, "Situated Knowledges: The Science Question in Feminism and the Privilege of Partial Perspective" (1988), emphasis mine

Being easily confused about reality in general, one of my epistemologic touchstones is Donna Haraway's notion of *situated knowledges*, as formulated in the quote above, incidentally almost exactly as old as I am [1]. I do not pretend to understand everything she wrote, but the core idea of situating knowledge within somewhere has always stuck with me.

It is not by accident I work outside industry and hope I never work there again. Their ontologies and epistemologies are toxic, and everything they claim to own they have stolen; from abstract *content* to the publicly funded fundamental research that made technologies like the smartphone possible. The IT industry as a whole is such an affront to humanity I often think it would have been better if we never invented computers at all. Computers help us produce effect from almost pure thought. Look at what the industry made of it! Spam, spam, canned spam, mind-bending spam, targeted spam.

At the time of writing Rust is very young. Despite this, it still has a surprisingly large and surprisingly nice fan base for an online community, and for tech in general. The Rust Book notes that "…the Rust programming language is fundamentally about empowerment: no matter what kind of code you are writing now, Rust empowers you to reach farther, to program with confidence in a wider variety of domains than you did before" [2]. It is around this notion, that systems programming can be *fun*, that we *can make new languages* for some interesting value of we, that Rust's community has grown.

In no other contemporary computer-associated community have I found so many non-binary trans cyber-witches, hackers, punks, philosophers, and anarchists. At times working on the Rust compiler has felt a lot more like a utopian project than it has any right to. Being in this community felt like my early memories of the Internet, before it all turned first boring (ca 2004–2007), then dystopian (2013–2015 and onwards). It is in this *somewhere* I want to situate my knowledge. It is within this community I want to place my work. Thank you, for reminding me that these stupid machines are supposed to be *fun*.

I would also like to thank specifically and in no particular order the Polonius Working Group, including my supervisor, Niko Matsakis, Lqd (pronounced "liquid"), and Matthew Jasper, who all answered my dumb questions at one point or other.

Chapter 1

Introduction

Rust is a young *systems programming language* originally developed at Mozilla Research [3] [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 through control of aliased memory.

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 special type verification step during Rust compilation called the borrow check. The borrow check ensures that no memory access reaches uninitialised memory, and that any shared memory is only shared immutably. Finally, it also protects against dangling references and references to stack-allocated memory that may be outside of the scope of an accessor. The rules of the memory ownership model are discussed at a higher level in Chapter 3, and related to the experimental formal type system Oxide in Section 3.2.

However, these rules represent a trade-off between static provability and expressive power. There exist several desirable Rust patterns, such as the example in Listing 2.1, that cannot be proven safe by the current borrow checker. This thesis describes a partial implementation of an experimental borrow checker called

---

[1] For a brief discussion of what it means for a language to be a systems language and how this applies to Rust, see Chapter 2.

Polonius, which increases the reasoning power of the borrow check to the level of individual program statements (a flow-sensitive analysis), allowing it to accept previously rejected coding patterns like the one in Listing 2.1. Additionally, Polonius has already proven useful for generating inputs to prove the correctness of Rust programs [4].

In practice, Polonius' analysis encompasses a variable liveness analysis (Section 6.2), initialisation tracking (Section 6.3), and may-reference analysis for validation of Rust's memory safety guarantees and alias control (Section 6.4), used to statically enforce safe use of shared memory.

1.1 Contributions

While this thesis is about developing Polonius rather than initially designing it, Polonius itself is novel in that is to our knowledge the first instance of Datalog use in the implementation of a compiler for a practical programming language. Systems like Doop [5], as well as other uses of the Soufflé system [6] have been deployed for practical analysis of large Java code bases, but their use has been in separate tools rather than in the language implementations themselves.

The contributions made in this work specifically include the implementation of liveness and initialisation calculations (Sections 6.2 and 6.3 respectively) in Polonius, previously computed by an earlier compiler pass and passed on to Polonius. Additionally, this thesis analyses real-world Rust code in ca 12 000 popular publicly available Git repositories found on Crates.io and GitHub (Chapter 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, and uses these to suggest future improvements of Polonius (Chapter 8).

For clarity, sections detailing components not developed as part of this thesis are marked with (†). They are nonetheless included (Section 6.4), as there exists no published complete description of Polonius. In other words, this thesis is itself the most complete account of Polonius design, implementation, and operation to date.

Chapter 2

A Safe and Modern Systems Programming Language

Be wary, then. Best safety lies in fear.

Laertes, on strong type systems. *Hamlet*, Act-I,
Scene-III.

Rust is a *systems programming language*, by which we mean that it is suitable for writing programs that require *little overhead*, or precise control over *hardware* or *memory allocation*. It is useful for tasks like developing embedded systems, video games, or components for operating systems. Its design was driven by the needs of the next-generation rendering engine for Mozilla's browser Firefox, Servo [7], where it replaces C++. Other notable uses of Rust for systems programming is in the research operating system Redox OS [8].

Rust is *modern* in that it has an advanced type system, as well as support for functional programming paradigms (higher-order functions, lambda expressions, closures, pattern matching). It also comes with the build tool and dependency manager cargo, which greatly improves the experience over using tools like automake, cmake, etc, in particular with respect to dependency management. Additionally, it also supports object-oriented like programming styles (without inheritance) through traits [9], a concept similar to Java's interfaces, combined with C-like `structs`.

Finally, we claim that Rust is *safe* in that its memory ownership model guarantees that a reference will always point to *initialised*, *valid* memory of the *correct type*, and that *references do not introduce data races*. In particular the data-race freedom is important for concurrent applications such as the Servo rendering engine. The memory ownership model also supports systems programming by making it possible for the Rust compiler to automatically manage memory without using a garbage collector. This is what enables applications like operating system kernels to be written in Rust.

Put simply, Rust tracks which scope *owns* allocated memory, and when that scope is no longer accessible, its memory is deallocated. This is similar to what a disciplined C programmer might do manually. Ownership over memory can

10

be transferred through an operation called a *move*, after which the memory is no longer accessible in the current scope, as it is now uniquely owned by the scope that took ownership over it. For example, the statement `vector.push(a)` would normally mean that the variable `a` is no longer accessible outside of the vector, as long as `a` does not implement the `Copy` trait, signalling that it should instead be copied into `vector`. Simple types that can be cheaply copied, such as integers, typically implement `Copy`.

Keeping with the terminology, creating a reference is known as a *loan* or a *borrow* in Rust parlance. Whenever a reference to a resource is created in Rust, its borrowing rules described in Chapter 3 must be respected for as long as the reference is alive, including across function calls [2]. These rules include memory lifetimes (i.e. the referenced memory must stay initialised as long as there is a reference to it), and unique access for a write-reference; if there is an active write-reference to some part of memory, no other read- or -write references must exist simultaneously. This is the restriction that guarantees freedom from data races.

As a whole, the rules governing memory ownership in Rust are verified by a process called the *borrow check*, and it is discussed in further detail in the following Chapter 3. As part of the verification system, additional information about the origin of a reference is embedded in any reference' type. The details of this concept are introduced in Sections 3.1 and (much) further discussed in Section 3.2.

Before moving further, we want to point out that Rust represents a different trade-off between static reasoning power of the compiler and expressive power for the programmer than other popular systems programming languages. By disallowing certain patterns of code, such as simultaneously overlapping references to writeable memory, Rust is able to make greater guarantees of safety than, for example, C. This reasoning power comes at the cost of restricting certain desirable (and safe) programming patterns that cannot be efficiently reasoned about by a compiler. In order to soften the blow from this, there is a continuous interest in increasing the reasoning power of the borrow check, and thereby the expressive power of Rust as a language. It is within this equation that Polonius fits.

2.1 Polonius: Addressing the Limitations of Rust's Borrow Checker

As explained in the previous section, the basis for Rust's automatic memory management is a static analysis tracking the ownership of each potentially shared object. Due to limitations in its formulation, the current borrow checker, *non-linear lifetimes* (NLL), rejects code such as the one in Listing 2.1, 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. In general terms, the current borrow checker reasons about references and their referents (the memory they refer

to) in terms of *lifetimes*, interpreted as the lines of a program where a reference may be used. 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 level of each individual program statement. For a slightly longer explanation of the differences between the current borrow checker and Polonius, see Section 3.1.

Listing 2.1: 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`*. [10]*

```
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. Rust's aliasing rules are not, strictly speaking, formalised. This means that future versions of Rust may change parts of the analysis. Therefore, having a clear, logic-like representation of the current rules is an advantage, besides being potentially easier to test, debug, develop, and profile as they are implemented.

Chapter 3

The Borrow Check: Enforcing Rust's Memory Model

> Neither a borrower nor a lender be
> For loan oft loses both itself and friend,
> And borrowing dulls the edge of husbandry.
>
> ───────────────────────────────
> Polonius in *Hamlet*, Act-I, Scene-III, Lines 75–77

In this chapter we will delve deeper into the borrow check. The idea is for the reader to develop an intuition for what it means in practice before moving on to actually implementing it in Chapters 4 and 6. We will also briefly explain how the formulation of the borrow check changes between NLL and Polonius (Section 3.1). This section introduces the finer points of Polonius, and is worth reading even if you are uninterested in the difference between Polonius and NLL.

Conceptually, the borrow check verifies that Rust's ownership rules of shared memory are respected. As mentioned in the previous section, memory is owned by the scope that has allocated it, and will be deallocated automatically when the owning scope is no longer needed. All of this is determined statically. Memory can also 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 mutable reference must not be (effectively) aliased.

A summary of the rules enforced by the borrow check can be found in Table 3.1, along with positive and negative examples. green and red boxes illustrate where the borrow check would occur with or without an error, respectively. Many of these examples are taken directly or slightly modified from Weiss, Patterson, Matsakis, and Ahmed [11].

Rule	Positive Example	Negative Example

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

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

Shared-Readonly	```struct Point(u32, u32);```
	 ```let mut pt = Point(13, 17);```
	 ```let x = |\goodexample{&pt}|;```
	 ```let y = |\goodexample{&pt}|;```
	 ```dummy_use(|\goodexample{x}|);```
		```struct Point(u32, u32);```
		 ```let mut pt = Point(13, 17);```
		 ```let x = |\goodexample{&pt}|;```
		 ```// ERROR: assigned to```
		 ```// borrowed value:```
		 ```|\badexample{pt}| = Point(1, 2);```
		 ```dummy_use(|\goodexample{x}|);```

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

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

Table 3.1: The Rules of the borrow check, with positive *(free from errors) and* ~~negative~~ *(with errors) examples. Highlighted code shows (parts of) 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.*

The initialisation and ownership tracking rules (Use-Init and Move-Deinit) guarantee that only initialised and owned memory is accessed. These are the rules that guarantee safe memory use at all, as well as garbage collection-free automatic memory management. In essence, they hold up the core ownership model, along with the corresponding reference lifetime rule (Ref-Live), which ensures that the points of access to memory through a reference does not happen after the scope owning the memory is left. This, for example, forbids us to return references to stack values from a function frame, as that frame would have been popped when the function survives. Semantically, this would mean that the returned reference *lives longer* (is accessible from at least the same scopes) than its referent, the value on the stack.

The rules of reference validity, Shared-Readonly and Unique-Write, together guarantee freedom from data-races, as well as access to valid data. They imply that only read-only memory can be shared between different, potentially concurrent, parts of the program, and read-only memory cannot have any data races. If a program mutates memory, it must have a unique name for that memory that cannot be accessible from another part of the program. The same rules also catch operations that would render a reference invalid, such as the owner of the borrowed memory moving or otherwise destroying it during a loan.

This approach and its associated type system comes from a long line of research into what is known as *linear*, or sometimes *affine* types [12]. A linear or affine type captures the concept that a given data structure is accessed precisely or at most once, respectively. In this sense, mutable memory is affine in Rust (used at most once); this is precisely the concept that guarantees freedom from data races and many other memory bugs. In comparison with other recent languages using capability types like Pony [13], [14] or Encore [15], Rust's memory ownership is relatively unsophisticated. Specifically, while capabilities attach to reference types of a language much like the borrow check's lending information (more closely discussed in the following section), capabilities have a higher resolution on operations they can allow or deny.

3.1   Polonius: From Lifetimes to Provenance Variables

The current borrow checker, NLL, treats the type of a reference, for example `&'db Database`, as a pair of values: a *(named) lifetime* (`'db`), and a type of the referent (`Database`). This type should be read as "a `Database` that lives for at least `'db`". Lifetimes are not always explicit and named like in the example; often they are inferred by the compiler. Rust allows functions to be generic with respect to a reference's lifetime. In essence, such a generic type means that the function takes a reference that must live for some duration, which must overlap the entire function body's life. Similar

15

notations allows for example a vector to hold references, as long as whatever they are referring to lives for at least as long as the vector they are stored in does. This is what the angle brackets in the motivating example of Listing 2.1 mean; the function `next()` takes a reference to a `String`, and, more importantly, guarantees that it will return something that lives for at least as long, which is `'buf`.

Lifetimes in NLL are *sets of lines of code a value lives for*. Given this notation, a value is said to *outlive* another if it lives for a larger number of lines. There is also a the named lifetime `'static`, which corresponds to the entire program, and so outlives every other lifetime. An error in NLL is reported if there is a violation of a loan overlapping its lifetime. However, the lifetimes formulation cannot express certain safe patterns, such as the one in Listing 2.1, where the lifetime of the loan ends up encompassing the entire loop unnecessarily.

This is why Polonius turns the relation between lifetimes and loans on its head. Its interpretation is that what was previously called a lifetime will now be called a *provenance* of the loan, and corresponds to the set of loans (calls to a reference constructor) that could have given rise to the reference. Named provenances, *neé* named lifetimes, (such as `'lifetime` above) are referred to as "provenance variables". [1] 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, where `p` would have the type `&'a i32` where `'a` is the set $\{\&x[0], \&x[1]\}$.

Listing 3.1: 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 3.7 of Section 3.2.

---

[1]In the literature, the terms "region" [16], "(named) lifetime" , and "reference provenance" [11] (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. However, during the work on this thesis, a fourth term, "origin", was chosen to replace the term "provenance variables" used here. Additionally, a comprehensive renaming of all the terms used is also underway at the time of writing, but was not sufficiently finished to be included in this report. For similar historical reasons, the name "region" sometimes occurs in Polonius' code as well.

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 and is further discussed in Section 6.2.1. This leads us back to our motivating example of Listing 2.1 again, where Polonius would conclude that `event` is being reassigned before use at the call to `parse()`, meaning that there would be no two overlapping live write-loans to the buffer, and therefore no error. The details of how this deduction is done is what will take the rest of this report to describe.

3.2 Polonius as a Type System

In this section, we will relate Polonius to Weiss, Patterson, Matsakis, and Ahmed on-going work of Weiss, Patterson, Matsakis, and Ahmed on formalising the reference ownership rules of Rust into the formally defined type system Oxide [11]. Oxide is notable in that it shares Polonius' use of provenance variables, as introduced in Section 3.1, in contrast to NLL. The rules presented here are based on the 2019 draft version of the paper and will change substantially for its 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).

$$\frac{\rho_1 \subseteq \rho_2 \qquad \tau_1 <: \tau_2}{\&\rho_1\tau_1 <: \&\rho_2\tau_2} \tag{3.1}$$

Before going into the complexities of Oxide, we will start with a simplified typing judgement in Rule (3.1), which says that a reference type is a subtype of another reference type iff their provenance variables are a subset. These typing judgements are implied on assignments with the intuition (based on Liskov's Substitution Principle) that you can only assign a right-hand side to a left-hand-side if the right-hand-side can function as a (is a subtype of) the left-hand-side. This means in terms of provenances that a reference type τ_1 is a subtype of another reference type τ_2 if it has a weaker dependency on loans, that is does not depend on additional loans. Such judgements will in practice be the main source of constraints on provenance variables in Polonius (the `outlives(R1, R2, P)` fact described in Chapter 4). In other words, if type systems are not your cup of tea, you may skip the rest of this section and proceed with Section 3.3. The only thing you need to remember is that assignments give rise to subset constraints that, as it were, "infects" provenance variables with loans from other provenance variables.

❅

Most of the conventions used in the Oxide formulation can be glossed over for the purposes of our understanding, but the most important ones are the type environment Γ, used to map places (π, π_1, and so on) to their types (τ, τ_1, etc). Oxide also needs to distinguish between types of statically known size (τ^S) and unknown size (τ^U) 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. Other notation used in the Oxide typing judgements discussed here is the one for $\boxed{\text{expressions}}$ such as $\boxed{x = 2}$, and the global (Σ) and type-variable environments (Δ). The two latter will not be of particular interest here, but are mentioned because they appear in the rules. Finally, variables that do not implement the `Copy` trait will be denoted with noncopyable. This is particularly important for moves, as they would not happen if the object being moved could instead be copied, and two copies be maintained.

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" [11]. 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 and 3.3, both taken from Weiss, Patterson, Matsakis, and Ahmed's paper (Figure 1). The first rule (3.2) 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 6.3, as well as part of the invalidation logic of Polonius.

$$\frac{\Gamma \vdash_{\text{mut}} \pi : \tau^s \quad \text{noncopyable } \tau^s}{\Sigma; \Delta; \Gamma \vdash \boxed{\pi} : \tau^s \Rightarrow \Gamma - \pi} \tag{3.2}$$

The second rule, Rule (3.3), 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(R, L, P)`, described in Section 4, and follows the intuition that if we create a reference, that reference is *known* to point to whatever we borrowed to create the reference.

$$\frac{\Gamma \vdash_\omega \pi : \tau}{\Sigma; \Delta; \Gamma \vdash \boxed{\&\omega\pi} : \& \{^\omega\pi\} \,\omega\tau \Rightarrow \Gamma} \tag{3.3}$$

Rules (3.2) and (3.3) 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 4.2 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 (3.4), 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.

$$\frac{\tau_1 \sim \tau_2 \Rightarrow \tau \quad \rho_1 \cup \rho_2 = \rho}{\&\rho_1\omega\tau_1 \sim \&\rho_2\omega\tau_2 \Rightarrow \&\rho\omega\tau} \tag{3.4}$$

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 (3.5) (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 expand π into all its references and perform the assignment. This would correspond to the `killed(L, P)` relation used in Polonius, where an old loan L 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.

$$\frac{\begin{array}{c} \Gamma \vdash_{\texttt{uniq}} \pi : \tau_o \quad \tau_o \sim \tau_u \Rightarrow \tau_n \\ \Sigma; \Delta; \Gamma \vdash \boxed{e} : \tau_u \Rightarrow \Gamma_1 \\ \texttt{places-typ}(\pi, \tau_u) = \overline{\pi : \tau} \end{array}}{\Sigma; \Delta; \Gamma \vdash \boxed{\pi = e} \; : \; \texttt{unit} \Rightarrow \Gamma_1 - \pi_1, \overline{\pi : \tau}} \tag{3.5}$$

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 (3.6)), 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 (3.7), is what actually introduces the relationship between provenance variables of references.

$$\frac{\begin{array}{c} \Sigma; \Delta; \Gamma \vdash \boxed{e_1} \; : \; \tau_1^s \Rightarrow \Gamma_1 \quad \Sigma \vdash \tau_1^s <: \tau_a^s \rightsquigarrow \delta \\ \texttt{places-typ}(\texttt{x}, \delta(\tau_a^s)) = \overline{\pi : \tau} \\ \Sigma; \Delta; \Gamma, \overline{\pi : \tau} \vdash \delta(e_2) : \tau_2^s \Rightarrow \Gamma_2 \end{array}}{\Sigma; \Delta; \Gamma \vdash \boxed{\texttt{let } \texttt{x} = \pi_2} \; : \; \tau_2^s \Rightarrow \Gamma_2 - \texttt{x}} \tag{3.6}$$

The subtyping rule for references, Rule (3.7), 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.

$$\frac{\rho_1 \subseteq \rho_2 \qquad \omega_1 \leq \omega_2 \qquad \Sigma \vdash \tau_1 <: \tau_2 \rightsquigarrow \delta}{\Sigma \vdash \&\rho_1\omega_1\tau_1 <: \&\rho_2\omega_2\tau_2 \rightsquigarrow \delta} \tag{3.7}$$

3.3 Intuition: Polonius is Just a Bunch of Transitive Closures

Graphs/polonius-high-level-overview.pdf
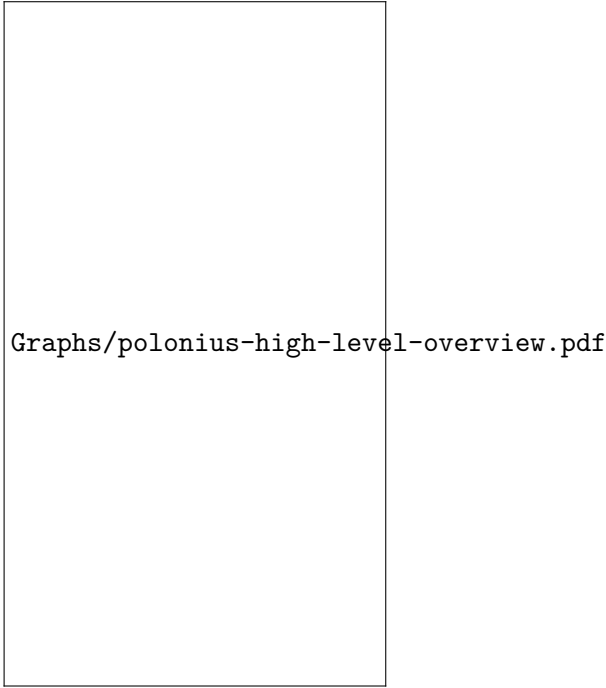
Figure 3.1: An overview of Polonius high-level structure; we compute liveness and members of provenance variables in order to find the potentially live loans at every given program point. These potential loans are used together with their potential violations to derive actual errors. A more precise representation can be found in Figure 6.1.

We now imagine this typing rule yielding these subtyping constraints for every assignment 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 type checking. Equipped with the simplified Rule (3.1), we can see that what would happen during type-checking is first a computation of sets of loans. Moving along with the program flow, we would add all loans from assignments until we couldn't find any more loans to add to any provenance variable; that is, we would have computed the *transitive closure* of the set membership constraints. This computation is described in Section 6.4.

There is one more rule to the computation of loan/provenance set memberships that we have not gotten into yet. If we see a loan be overwritten, that loan should not be propagated, and is said to be *killed*. This is the rule that says that removes `&x` from the provenance of `r`'s type upon assignment in cases like this:

```
let r = &x;
r = &y;
```

Now we know which loans belongs to which references. The rest of Polonius verification of loans is just this: an error is a violation of a *live loan*, that is a loan that might be used in the future. As it turns out, liveness is *another* transitive closure computation, described in Section 6.2, and the list of potential violations is generated by the Rust compiler and given as input to Polonius, a process which is described in Chapter 4.

Finally, we have the move errors, described in Use-Init and Move-Deinit. It turns out that their computation is *also* a transitive closure; at any point in the program, the set of initialised variables [2] is the set of variables that have been initialised so far, minus the ones that have been moved. This calculation is described in Section 6.3.

Overall, the process looks something like Figure 3.1. First, we compute a set of initialised variables (it turns out to be needed for computing live provenance variables), then we compute the set of initialised regular variables, and use both to find out which provenances are live. We then compute which loans belong to which provenance at which point in the program flow, which we use together with the liveness information to determine which loans may actually be used and so must be valid. We now have enough understanding of the formal basis of Polonius to move forward with the actual implementation, starting with the interaction between Polonius and the Rust compiler in the following chapter.

---
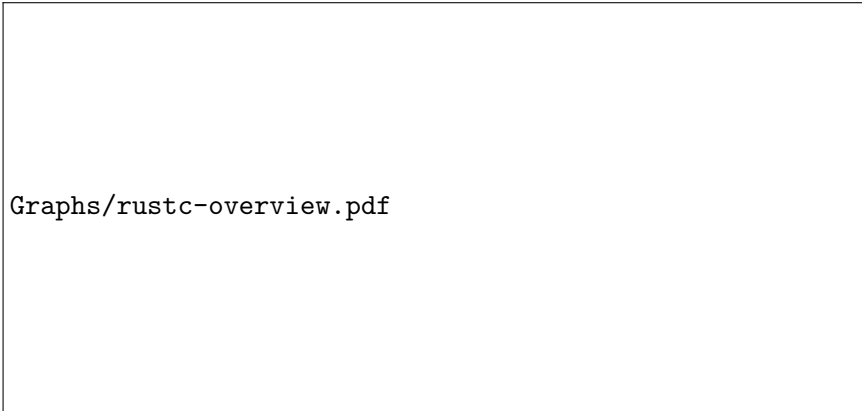
[2] We will use a more precise terminology than variables when it becomes time, but we are not there yet.

Chapter 4

The Borrow Check in the Rust Compiler

Get thee to a nunnery, go. Farewell.

Hamlet offering me career advice after my work
on this thesis. *Hamlet*, Act-III, Scene-I

Figure 4.1: An overview of the Borrow Check's place in the process of compiling Rust code, as described in the Rust Developer's Guide [18].

The logic of the borrow check as described in Chapter 3 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 4.1. 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.

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 one or two basic blocks (its *successors*) [19]. A side-to-side comparison between a small Rust program and its MIR can be seen in Figure 4.2. The Polonius analysis is performed at the level of these blocks, addressing each statement of the block in two phases: its start and mid-point. The start of a block is before the statement has taken effect, and the mid-point is just after. We use the notation `Start(bb0[0])` to refer to the starting point of block 0's first statement, and `Success(bb5)` to refer to the first statement of the block following the success branch of basic block 5.

4.1 Generating Inputs for Polonius

The Rust compiler analyses the MIR and emits *facts* that Polonius uses to derive its conclusions through the Datalog rules described in Chapter 6. An overview of the inputs with examples of when they are used can be seen in Table 4.2, but will be more closely introduced when they are used. Facts describe relationships between *atoms*, the objects in the world of Polonius. An overview of the atoms used in Polonius can be found in Table 4.1.

Atom	Example	Description
Loan[†]	`&x`	An individual borrow expression.
Provenance variable[†]	`'a`	The explicit or inferred part of a reference type that contains the set of loans it could have come from.
Node[†]	`Mid(bb1[5])`	A node in the control-flow graph.
Move path	`x.y.z`	A precise field that can be accessed into a variable; a field in a struct or a projection in a tuple.
Variable	`x`	A MIR (or Rust) variable.

Table 4.1: The atoms used in Polonius. Variables and move paths were introduced as part of this thesis.

While Polonius is a self-contained package with a single interface, the code translating compiler-internal data structures into Polonius facts has a much larger surface area. All the additions to the Rust compiler occurs in the `librustc_mir::borrow_check::nll` ule, alongside the current borrow checker ("NLL"). The module hierarchy and the
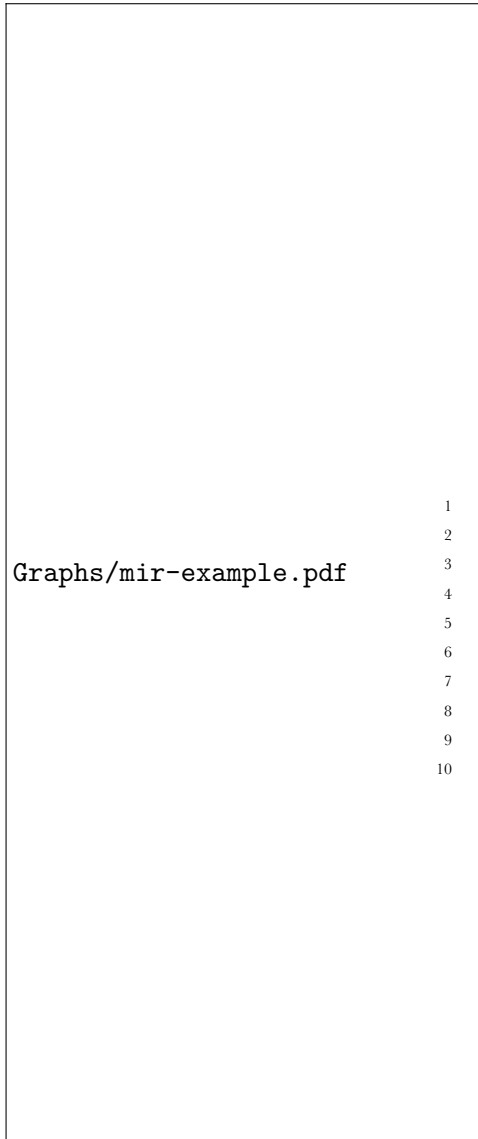
Graphs/mir-example.pdf
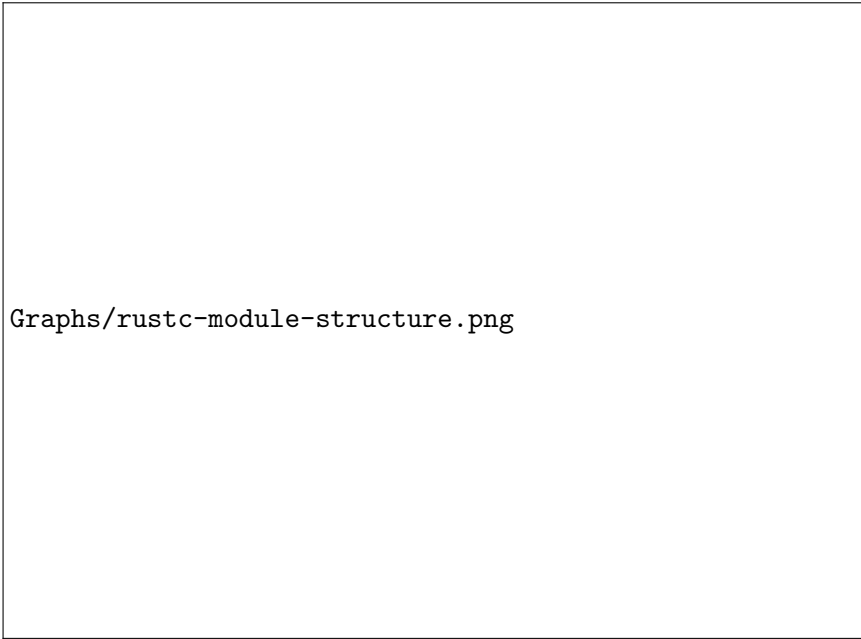
```rust
1  fn main() {
2      let x = 17;
3      let z = if x == 3 {
4          92
5      } else {
6          x
7      };
8
9      do_something(z);
10 }
```

Figure 4.2: 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).

Fact	Code Example	Resulting Tuple(s)	Used
`borrow_region(R, L, P)`[†]	`bb0[0]: _1 = &1;`	`('a, &1, Mid(bb0[0]))`	Bck
`universal_region(R)`[†]	`fn f<'b>(x: &'b str)`	`('b)`	Bck
`cfg_edge(P, Q)`[†]	`bb0[0]:  _1 = 1;`	`(Start(bb0[0]), Mid(bb0[0])),`	All
	`bb0[1]:  _2 = 3;`	`(Mid(bb0[0]), Start(bb0[1])),`	
		`(Start(bb0[1]), Mid(bb0[1]))`	
`killed(L, P)`[†]	`bb0[0]:  _3 = &_1;`	`(&_1, Mid(bb0[1]))`	Bck
	`bb0[1]:  _3 = &_2;`		
`outlives(R1, R2, P)`[†]	`bb0[0]: p: &'p i32 = &'x x;`	`('x, 'p, Mid(bb0[0]))`	Bck
`invalidates(P, L)`[†]	`bb0[0]:  _2 = &_1;`	`(Mid(bb0[1]), &_1)`	Bck
	`bb0[1]:  _1 = 3;`		
`var_used(V, P)`	`bb0[0]: x + 1`	`(x, Mid(bb0[0]))`	Lvs
`var_defined(V, P)`	`bb0[0]: x = 7`	`(x, Mid(bb0[0]))`	Lvs
`var_drop_used(V, P)`	`bb0[0]: drop(x)`	`(x, Mid(bb0[0]))`	Lvs
`var_uses_region(V, R)`	`let x: &'x i32;`	`(x, 'x)`	Lvs
`var_drops_region(V, R)`	`struct Wrap<'p> { p: &'p i32 }`	`(x, 'p)`	Lvs
	`impl<'p> Drop for Wrap<'p> {...}`		
	`let x: Wrap = ...`		
	`drop(x)`		
`child(M1, M2)`	`let x = (17, (23, 29));`	`(x.0, x),`	Init
		`(x.1, x),`	
		`(x.1.0, x.1),`	
		`(x.1.1, x.1)`	
`path_belongs_to_var(M, V)`	`let x = (17, (23, 29));`	`(x, x)`	Init
`initialized_at(M, P)`	`bb0[0]: x.0 = 17;`	`(x.0, Mid(bb0[0]))`	Init
`moved_out_at(M, P)`	`bb0[0]: f(move x.0)`	`(x.0, Start(Success(bb0)))`	Init
`path_accessed_at(M, P)`	`bb0[0]: x.val + 7`	`(x.val, Mid(bb0[0]))`	Init

Table 4.2: Polonius input facts, with minimal code examples. All facts except `invalidates`, `cfg_edge`, `killed`, `borrow_region`, `outlives`, and `universal_region` were added as part of the work on this thesis.

Figure 4.3: 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.

location of emission of the various facts is shown in Figure 4.3. The reason for this complex intermingling of modules is that Polonius fact generation piggy-backs off of previous analysises, notably the `outlives` constraints generated by the previous borrow checker during type-checking. In other words, it is not a full replacement of the current implementation.

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 4.3 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.

27

Returning to one of the examples of the borrowing rules in Chapter 3, we can describe some of the facts that would be emitted on each line. An annotated example can be seen in Listing 4.2.

Listing 4.2: 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.
```

The example above is slightly simplified; the translation to MIR 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. Precisely how these errors are derived is the subject of Chapter 6.

Chapter 5

Selecting an Implementation Language for Polonius

> My liege, and madam, to expostulate
> What majesty should be, what duty is,
> What day is day, night night, and time is time,
> Were nothing but to waste night, day, and time;
> Therefore, since brevity is the soul of wit,
> And tediousness the limbs and outward
> flourishes,
> I will be brief.
> ───────────────────────────────
> Polonius, illustrating the succinctness and clarity
> of Datafrog compared to Datafrog, in *Hamlet*,
> Act-II, Scene-II

In this chapter, we will discuss Datalog, the language we use to express Polonius, and its implementation Datafrog. 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 [20]. It describes fixpoint calculations over logical relations as predicates, described as fixed input *facts* about objects, called *atoms* (upper-case names), computed *relations* (lower-case names), 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(Older, Younger)` 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 [21] is a minimalist Datalog implementation embedded in Rust, using a worst-case optimal join algorithm as described in [22]. The fact that Datafrog is embedded in Rust (an EDSL, or Embedded Domain-Specific Language) 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(Older, Younger)` relation in Datafrog can be seen in Listing 5.1.

Listing 5.1: `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, discussed in Section 6.3.

```
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 5.2: 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 Figure 6.3 becomes the code in Listing 5.2, and the corresponding join used for the first half of `region_live_at(R, P)` of Figure 6.6 can be seen in Listing 5.3.

Listing 5.3: 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 5.3 `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 5.3, 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 node. 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 5.2), 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 `Keys` in the `Relation` being (anti-)joined. In the case of `extend_anti`, any tuples matching `Key` are discarded.

In Listing 5.2, 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.

5.1 Datalog: a Natural Choice for Polonius

The most obvious reason why we chose Datalog for Polonius lies in the intuition developed in Section 3.3. If Polonius is a bunch of transitive closures computed on certain relations over the CFG, Datalog would be a natural fit.

Additionally, Datalog has a long history of use for static analysis, with a notable recent example being the Soufflé system, which synthesises performant C++ code from the Datalog specifications to show promising performance for analysis of large programs comparable to hand-crafted implementations in C++ [23]. This approach is similar to how Datafrog embeds a minimal solver as a Rust library. Soufflé in particular was an inspiration for Polonius, and with the current ongoing work on translating Soufflé-style Datalog into the more cumbersome Datafrog EDSL format Polonius can almost be said to converge into a specialised "Soufflé for Rust".[1]

Another inspiration was the Doop framework [24], developed by Smaragdakis and Bravenboer for a proprietary Datalog engine and later ported to Soufflé [25]. Doop specifically performs a may-point-to-analysis (much like Polonius), using explicit tuple storage (also like Polonius), as opposed to other common representations like Binary Decision Diagrams (BDDs). Much like Polonius, the authors of Doop sought a framework that would let them specify their analysis clearly in a high-level language. In fact, Datalog has been so successful for may-point-to-analysis that a 2016 paper developing a lattice-based declarative analysis for the same purpose described it as "the killer application" of Datalog [26], a career that took off around 2007 with the work of Benton and Fischer, who demonstrated the Dimple framework for static analysis of Java [27]. This was despite a case study showing already in 1996 that logic programming (in that case Prolog) would constitute a good trade-off between performance of the analysis and efficiency of development [28].

However, and as mentioned in Section 1.1, Polonius is to our knowledge the first attempt at using Datalog for pointer analysis (or any other kind of static analysis for that matter) in a production compiler. All the previous analyses and frameworks have been predominantly analysis of Java, either in the form of source code or byte code, by programs outside of the Java compiler.

---

[1]The experimental code to synthesise Datafrog rules can be found on GitHub in the separate repository `https://github.com/lqd/datapond`.

Chapter 6

Implementing Polonius

> Why, then, 'tis none to you, for there is nothing
> either good or bad, but thinking makes it so
>
> *Hamlet*, Act-II, Scene-II

In this chapter, we will describe the current implementation of Polonius in Datalog (forgetting the horrors of Datafrog in Chapter 5). We will start by discussing how Polonius computes liveness of provenance variables, one of the two main contributions of this thesis, in Section 6.2, then move on to move analysis in Section 6.3 before finishing with the loan violation computation in Section 6.4[†].

An overview of Polonius can be seen in Figure 6.1: 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 the provenance variable 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).

In all of these sections, we will introduce every non-trivial rule with a positive and negative example of use. By positive we mean an example that would generate more members to the closure being computed, and by negative, we mean mean an example that would not match. Each section will also start with a table of the facts and relations used in that section, both the target ones and intermediary relations introduced along the way.

But first, we need to clarify a detail we have previously glossed over, namely:
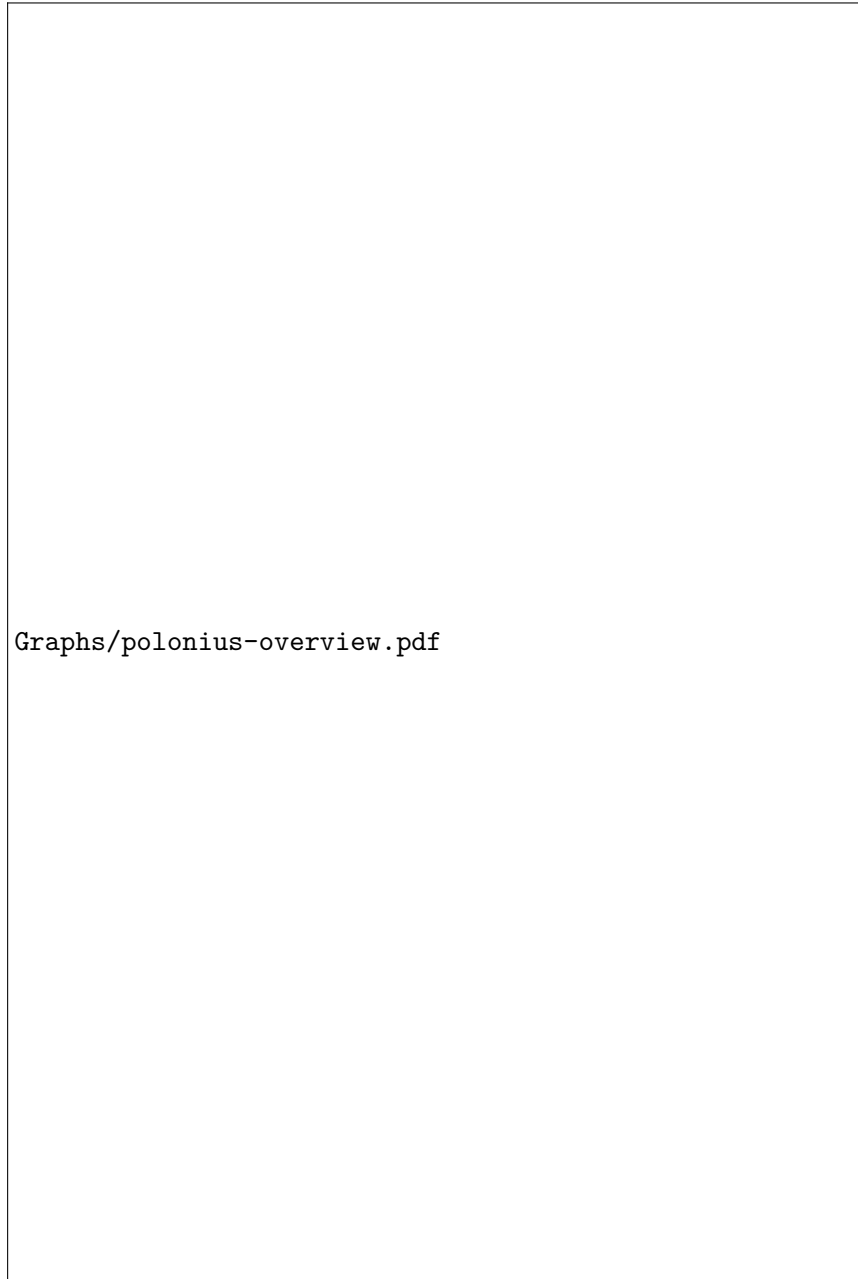
Figure 6.1: An overview of how the inputs and intermediate steps of Polonius combine into the final output. Blue boxes represent facts and relations implemented during the work on this thesis. Relations are shown using boldface, and facts in regular font. The historical term "region" is used here instead of provenance variables to match the convention used in the actual code.

6.1 Analysing Complex Data Structures

The borrow check specifically tracks memory at the resolution of *paths*, that is paths to concrete memory. Examples of paths is x (a variable on the stack), x.f (a field of a struct), *x.f (a field accessed through a reference), or (*x.f)[_] (an index of an array stored in a struct and accessed through a reference). However, an array counts as one path, ignoring its indices. Paths constitute trees, (or perhaps rather Russian nesting dolls) such as in the example of x.f, where x.f lies under x. We say that each path preceding (and including) a given path are its *prefixes*. The topmost path, the variable name itself, we call a *root path*. Finally, we say that paths *overlap* if one of them is a prefix of the other. Intuitively, this mean that they involve the same region in memory.

The fact that the borrow check is performed on paths means that the following code, for example, is sound, as the paths and therefore also 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!
```

6.2 Liveness, as Experienced by Polonius

Liveness of *variables* is computed in order to determine the liveness of *provenance variables*. Intuitively, a variable (provenance or regular alike) is live when it may be used at some later point in the program. We say that a provenance variable is *use-live* if it is live on account of being used (e.g. dereferenced). This is important, because if a provenance variable is dead, violating its loans is not an issue.

A summary of the facts, intermediate relations, and outputs of Polonius' liveness computations can be found in Table 6.1.

The use-liveness of a variable (Figure 6.3) is computed as a transitive closure backwards across the CFG. A variable is use-live where it is being used, and stays use-live (backwards) until it is defined. 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 every analysis in Polonius, it is imprecise with respect to branchings, as there is no way to know statically which branch is taken. That means that we over-approximate variable liveness: if only one branch uses a variable, it is live upstream of that branch. Perhaps the best illustration of this is a graph form, as seen in Figure 6.2.

However, recall that we are not interested in the liveness of *variables*; we care about the provenances in their types. And it turns out that there is another way in which a reference may be live, which is by being used during the deallocation of a data structure.

Atom/Fact	Type	Description
Provenance[†]	Atom	The explicit or inferred part of a reference type that contains the set of loans it could have come from.
Node[†]	Atom	A node in the control-flow graph.
Variable	Atom	A MIR (or Rust) variable.
`cfg_edge(From, To)`[†]	Fact	A transition in the CFG.
`var_used(Var, Node)`	Fact	A regular variable use happens here.
`var_defined(Var, Node)`	Fact	A variable is assigned here.
`var_uses_region(Var, Provenance)`	Fact	Connects provenance variables from types to their variables.
`var_drops_region(Var, Provenance)`	Fact	Deallocating this variable indirectly uses this provenance variable.
`var_initialized_on_entry(V, Node)`	Fact	This variable is initialised on entry to this CFG node.
`var_drop_live(Var, Node)`	Intermediate	This variable is used in a drop.
`var_use_live(Var, Node)`	Intermediate	This variable is used in an expression.
`region_live_at(Provenance, Node)`	Output	This provenance variable is live at this node, and the conditions of its loan must be accepted.

Table 6.1: Liveness Dramatis Personae.

```
Graphs/liveness.pdf
```
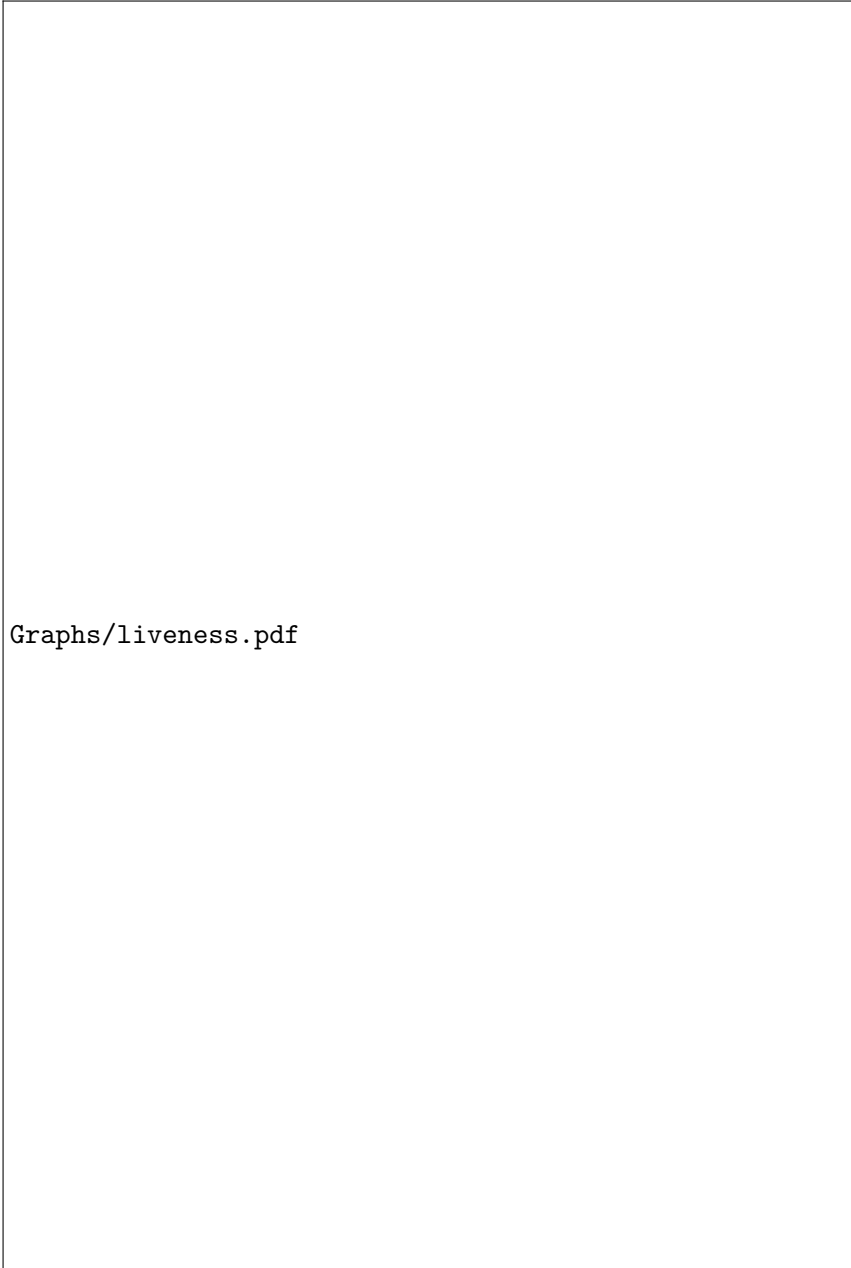
Figure 6.2: A graph representation of the the variable liveness calculation results, with relevant Polonius facts as they occur (a droplet symbolising `var_drop_used(V, _)`*, a wrench* `var_used(V, _)`*, and a skull and crossbones symbolising* `var_defined(V, _)`*). Variables are named by prefixing underscores, and edges annotated with the propagated live variable and its liveness type(s) (**D**rop or **U**se). The source code is one of the test cases for drop-liveness.*

```
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).
```

⊤	Input	Conclusion
	`var_used(x, Start(bb0[1]))`	`var_use_live(x, Mid(bb0[0]))`
	`cfg_edge(Mid(bb0[0]), Start(bb0[1]))`	`var_use_live(x, Start(bb0[1]))`
⊥	Input	Conclusion
	`var_use_live(x, Start(bb0[1]))`	(nothing)
	`var_defined(x, Mid(bb0[0]))`	
	`cfg_edge(Mid(bb0[0]), Start(bb0[1]))`	

Figure 6.3: The rules for calculating use-liveness: a variable is use-live if it was used at a node P, or if it was live in Q, there is a transition $P \rightarrow Q$, and it was not defined (killed) in P.

6.2.1 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.

Rust provides a default deallocator for data structures, which can be overridden. This has repercussions on liveness calculations. While the default deallocator for an object never accesses its fields, and therefore does not make them live, a custom deallocator might access any of them in arbitrary ways. This of includes references stored in the struct, whose provenance variables must be considered live. Intuitively, this follows from the fact that the deallocator may use the references in the `struct`, and that the conditions of their loans must therefore be respected, and we say that the variable holding the struct is *drop-live*. An example can be found in Figure 6.4.

Following the MIR translation of Listing 6.1 in Figure 6.4, we see across the re-borrows used to move the created references into the `struct`s that the function's single block terminates in 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, as its sole element, a reference, requires no deallocation.

38

```
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
}
```
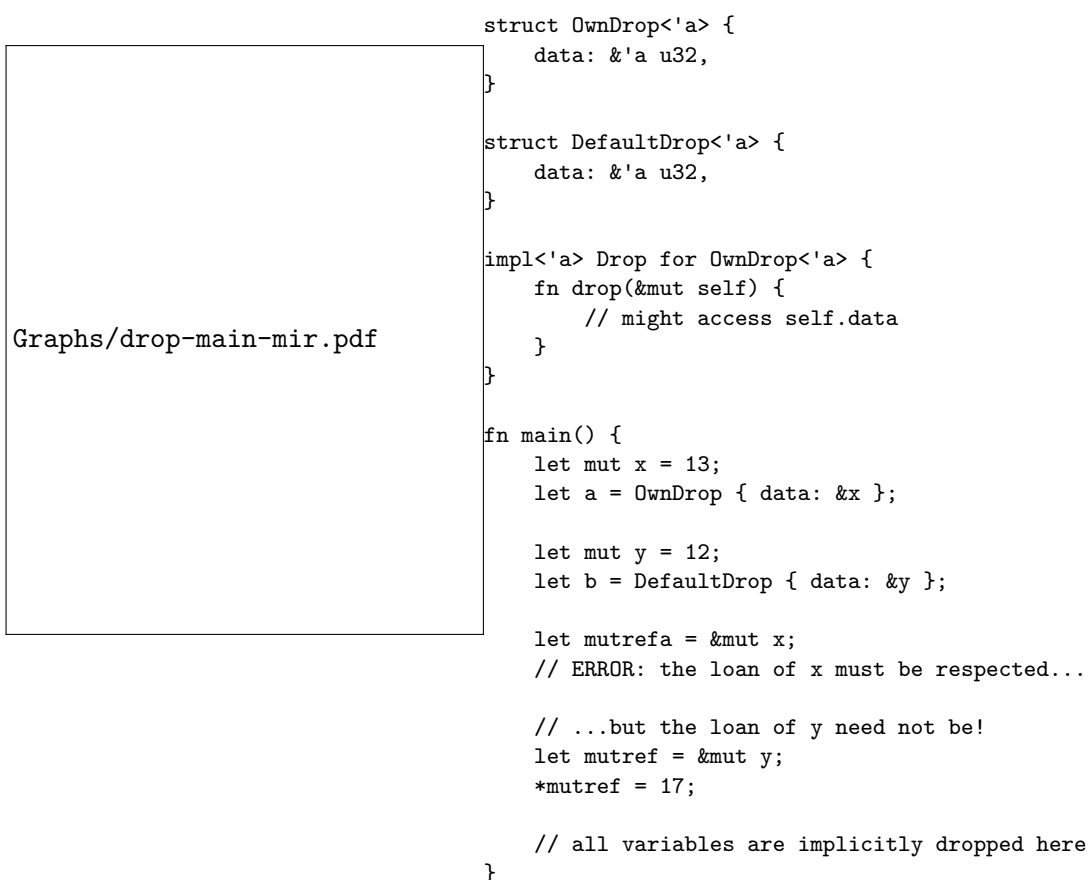
Figure 6.4: A graph rendering of the MIR produced from the `main()` function of the code to the right, illustrating a call to the custom deallocator of `_2` that would cause it to be drop-live during the block. Take special note of the lack of calls to `drop(_6)`; as `DefaultDrop`, the `struct` stored 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.

Drop-liveness is calculated in a similar fashion to use-liveness, with the exception that a moved variable is never dropped, as it is now owned (and therefore deallocated) by the context it was moved to. This is the reason for the computation of variables that might be initialised in Section 6.3. The rules can be found in Figure 6.5.

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)`.

6.2.2   Variable Liveness to Provenance Variable Liveness

The two kinds of liveness are then used to calculate the reference liveness relation (Figure 6.6), which serves as input for the rest of the borrow checker. A given provenance variable R is live at some node p if it is in the type of a use-live variable v, or if it is associated to a drop-live variable. While the connection between use-live variables and their provenances is direct, the connection between a use-live variable and its provenance variable(s) is *indirect*; any reference stored inside the drop-live `struct` in v is live at p if v is drop-live there. An example of this can be seen in Table 6.2.

Statement	Use-live	Drop-live	Provenance(s) live
`let mut x = 13;`			
`let own = OwnDrop { data: &'x1 x };`	x		
`let bad_ref = &'x2 mut x;`	x	own	'x1
`uses_var(bad_ref);`	bad_ref	own	'x2, 'x1
`// drop(own) implicit`		own	'x2

Table 6.2: An example of a drop-live `struct` causing an inner provenance variable to be live. This program would generate an error, as we have two overlapping loans of the same path (x), where one is mutable.

6.3   Move Analysis

The idea behind the move analysis is again a transitive closure computation across the CFG, much like the one for liveness in Section 6.2. Initialisation (`path_maybe_initialized_on_exit(P, _`) propagates forwards from an assignment (`path_assigned_at(P, _)`) across the CFG (`cfg_edge(P, Q)`) until the path is moved (`path_moved_at(P, _)`). We also transitively follow the move path tree downwards on each event, so that a use of x would also use `x.f`, for example. This transitive expansion of facts happens in a pre-computation described in Section 6.3.1, and is needed for erroneous path accesses to be guaranteed to overlap with a moved out path and generate an error. Finally, we trace root

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

⊤ Input	Conclusion
`var_drop_used(x, Start(bb0[1]))`	`var_drop_live(x, Start(bb0[1]))`
`var_maybe_initialized_on_exit(x, Mid(bb0[0]))`	
`cfg_edge(Mid(bb0[0]), Start(bb0[1]))`	
`var_drop_live(x, Start(bb0[1]))`	`var_drop_live(x, Mid(bb0[0]))`
`var_maybe_initialized_on_exit(x, Mid(bb0[0]))`	
`cfg_edge(Mid(bb0[0]), Start(bb0[1]))`	

⊥ Input	Conclusion
`var_drop_live(x, Start(bb0[1]))`	(nothing)
`var_defined(x, Mid(bb0[0]))`	
`cfg_edge(Start(bb0[0]), Mid(bb0[0]))`	
`cfg_edge(Mid(bb0[0]), Start(bb0[1]))`	

Figure 6.5: The rules for calculating drop-liveness: the rules are similar to those for to use-liveness (Figure 6.3), 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!

Atom/Fact	Type	Description
Provenance[†]	A	The explicit or inferred part of a reference type that contains the set of loans it could have come from.
Node[†]	A	A node in the control-flow graph.
Variable	A	A MIR (or Rust) variable.
Path	A	A move path.
`cfg_edge(From, To)`[†]	F	A transition in the CFG.
`child(Path2, Path1)`	F	`Path1` is a prefix of-, and not equal to `Path2`
`path_begins_with_var(Path, V)`	F	This path is the root path of variable V.
`path_accessed_at(Path, Node)`	F, I	This move path is used in an expression here.
`path_assigned_at(Path, Node)`	F, I	`Path` is given a value here. All non-locals (function arguments) are initialised at the start of a function.
`path_moved_at(Path, P)`	F, I	This path was moved from this scope in an expression here. All locals (non-arguments) are are "moved" at the start of the function.
`ancestor(Above, Below)`	I	`Above` is a prefix of `Below`.
`path_maybe_init___exit(Path, P)`	I	`Path` is initialized without subsequent deinitialisation on one or more branches reaching this CFG node (lowest upper bound on set membership).
`path_maybe_uninit___(Path, Node)`	I	`Path` may be uninitialised upon exiting this CFG node.
`var_maybe_init___(Var, Node)`	O	This variable is partially initialised along at least one branch reaching this node at the time of its exit.
`move_error(Path, Node)`	O	Error: `Path` is accessed here, but may not be initialised.

Table 6.3: Move Analysis Dramatis Personae

```
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).
```

⊤	Input	Conclusion
	`var_drop_live(x, Mid(bb0[1]))`	`region_live_at('nested, Mid(bb0[1]))`
	`var_drops_region(x, 'nested)`	
	`var_use_live(x, Mid(bb0[1]))`	`region_live_at('x, Mid(bb0[1]))`
	`var_uses_region(x, 'x)`	

Figure 6.6: 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.

paths (and therefore also their transitive children) back to their variables through `path_begins_with_var(P, V)` for use in drop-liveness computations. A summary of the inputs, outputs, and intermediary relationships involved in the computation can be found in Table 6.3. Additionally, an annotated example of Rust source code with relevant facts and conclusions can be found in Table 6.4.

Like liveness, initialisation tracking is necessarily imprecise upon branching; if one branch in the CFG has `Path` initialised and one does not, we must decide on whether to over-estimate (assume `Path` is initialised), or under-estimate (assume `Path` is deinitialised) after the branches join. As it turns out, the move analysis in contrast to variable liveness does both. When we want to determine if a variable might be involved in a deallocation for the purposes of later figuring out live provenances in Section 6.2.1, we over-estimate (Section 6.3.2). When we want to figure out if it is safe to access a variable or if we should generate a move error, we under-estimate (Section 6.3.3).

However, before the liveness computations can begin, we need to elaborate a few inputs.

6.3.1 Figuring Out The Move Tree

The analysis begins with a pre-computation step that expands the path-related facts so that an initialisation of a prefix also initialises its children, grandchildren, etc. The Datalog for this can be found in Listings 6.2 and 6.3. The transitive expansion

43

happens for all the path-related facts: `path_assigned_at(P, _)`, `accessed_at(P, _)`, `path_moved_at(P, _)`, and `child(P1, P2)` (which becomes `ancestor(P2, P1)`). The names are re-used in later steps to reflect the fact that we see this as a pre-computation█ step expanding compressed facts from Rust. These are, as you may have guessed, also transitive closure computations.

Listing 6.2: Calculating the move path tree; a path is the `ancestor(Older, Younger)` of all its children and their children transitively.

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

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

Listing 6.3: Calculating transitive moves: a move, initialisation, or an access to a prefix transitively moves, initialises, or accesses all of the prefix' children. Identical rules for for `path_assigned_at(P, _)` and `accessed_at(P, _)` have been omitted.

```
path_moved_at(Path, P) :- path_moved_at(Path, P).

path_moved_at(Child, P) :-
    path_moved_at(Parent, P),
    ancestor(Parent, Child).
```

Equipped with these elaborations, we can proceed with both over-estimating liveness for drop-liveness calculations (Section 6.3.2) and under-estimating initialisation for error reporting (Section 6.3.3).

6.3.2 Over-Estimating Initialisation for Liveness

We begin initialisation tracking on assignments. A path is trivially initialised in a statement where it is initialised (`x.f = 17`), and stays initialised in the subsequent program points unless it is moved out by a move expression (`move x.f`, or `move x`). The imprecision is introduced by the join to `cfg_edge(From, To)`; it is enough for one connecting edge to `To` to have `x.f` initialised for it to be initialised at `To`.

Finally, we relate the initialisation of root paths back to their variables through `path_begins_with_var(P, V)`. This means that we track variables that are partially initialised; an over-approximation needed because a deallocation of a variable holding a partially initialised `struct` would only be guaranteed to be a no-op if the entire variable was moved at the time of the deallocation. The full listing for the code can be found in Figure 6.7.

6.3.3 Under-Estimating Initialisation for Move Errors

The principle behind extracting move errors is this: an *error* is an *access to a possibly moved or uninitialised path*. In order to compute this, we mark every function parameter as initialised and every local variable as uninitialised at the start of each function, and then propagate the status for each variable across the CFG.

```
path_maybe_initialized_on_exit(Path, Node) :-
    assigned_at(Path, Node).

path_maybe_initialized_on_exit(Path, TargetNode) :-
    path_maybe_initialized_on_exit(Path, SourceNode),
    cfg_edge(SourceNode, TargetNode),
    !moved_out_at(Path, TargetNode).

var_maybe_partly_initialized_on_exit(Var, Node) :-
    path_maybe_initialized_on_exit(Path, Node),
    path_begins_with_var(Path, Var).
```

⊤	Input	Conclusion
	`path_assigned_at(x, Mid(bb0[0]))`	`var_maybe_initialized_on_exit(x, Mid(bb0[0]))`
	`cfg_edge(Mid(bb0[0]), Start(bb0[1]))`	`var_maybe_initialized_on_exit(x, Start(bb0[1]))`
	`path_begins_with_var(x, x)`	

⊥	Input	Conclusion
	`path_maybe_initialized_on_exit(x, Start(bb0[0]))`	`var_m_i_e(x, Start(bb0[0]))`
	`path_moved_at(x, Mid(bb0[0]))`	
	`cfg_edge(Start(bb0[0]), Mid(bb0[0]))`	
	`path_begins_with_var(x, x)`	

Figure 6.7: The rules for over-approximating variable initialisation. A path is trivially initialised where it is actually initialised. It is transitively initialised in all nodes reachable from a node where it is initialised, and where it has not been deinitialised (moved out). Variables are initialised if their root path is initialised.

These relations are computed in Listings 6.4 (move errors) and 6.5 (possibly uninitialised paths).

Listing 6.4: A move error is a path access to any path that may be deinitialised. We use a join with the CFG to move the error one node ahead from the last provably initialised node. This is because if a path is initialised on exit from some statement, it is still initialised on entry to the next one, which would correspond to the node where the evaluation (but not the effect) of the statement happens. Without this "rolling up", we would have a move error on every statement that moves a path, as that statement also accesses the path it moves. With these rules, the path would be accessed at the starting-poing of the move statement, and moved at mid-point, thus avoiding generating an error where they overlap.

```
move_error(Path, TargetNode) :-
    path_maybe_uninitialized_on_exit(Path, SourceNode),
    cfg_edge(SourceNode, TargetNode),
    path_accessed_at(Path, TargetNode).
```

Listing 6.5: A `Path` may have been moved at a `Node` if it was moved on the way there without being subsequently reinitialised. This logic is identical to the one used for over-approximating initialisation in Figure 6.7.

```
path_maybe_uninitialized_on_exit(Path, Node) :-
    path_moved_at(Path, Node).

path_maybe_uninitialized_on_exit(Path, Node2) :-
    path_maybe_uninitialized_on_exit(Path, Node1),
    cfg_edge_(Node1, Node2)
    !path_assigned_at(Node1, Node2).
```

Statement	Maybe init	Maybe uninit	Accessed
`let x = (2, 3)`	`x.0, x.1, x`		
`move x.0`	`x.1, x`	`x.0`	`x.0`
`x.1 + 7`	`x.1, x`	`x.0`	`x.1`
`x.0 + 3`	`x.1, x`	`x.0`	`x.0`
`x.0 = 4`	`x.1, x.0, x`		
`if random() {move x.0}`	`x.1, x.0, x`	`x.0`	
`f(x)`	`x.1, x.0, x`	`x.0`	`x, x.0, x.1`

Table 6.4: An example of some facts and outputs of key relations during the type-verification of a small Rust program. Note that the example is slightly inauthentic; in reality, integers implement the `Copy` trait, and so would not be moved, and some statements have been shortened to just expressions. From top to bottom, we have a variable with paths being initialised, one of the paths being moved, an acceptable access to the remaining path, an erroneous access to the moved path, a reinitialisation of the moved path, an `if` statement triggering imprecision, and finally an erroneous use of a potentially partially moved variable (`x.0`).

Atom/Fact	Type	Description
Provenance[†]	Atom	The explicit or inferred part of a reference type that contains the set of loans it could have come from.
Node[†]	Atom	A node in the control-flow graph.
Loan[†]	Atom	A unique borrow expression (`&x`).
`cfg_edge(P, Q)`[†]	Fact	A transition in the CFG.
`region_live_at(Provenance, Node)`	Fact	This provenance variable is live at this node, and the conditions of its loan must be accepted.
`borrow_region(Provenance, Loan, Node)`[†]	Fact	A borrow expression at this node creates this loan and this provenance variable.
`invalidates(Node, Loan)`[†]	Fact	An operation at this node would violate this loan if it were live.
`subset(Provenance1, Provenance2)`[†]	Intermediate	There is a subset relationship between two provenances at this node.
`requires(Provenance, Loan, Node)`[†]	Intermediate	This provenance contains this loan at this node.
`loan_live_at(Loan, Node)`[†]	Intermediate	This loan is live at this node.
`error(Node)`[†]	Output	A borrow check error (a violated loan) occurred at this node.

Table 6.5: Loan Constraint Propagation Dramatis Personae

Statement	Killed	Invalidates	New Constraints	Live provenances
`let p: &'p i32 = &'1 x`			`'1 ⊆ 'p, &'1 x ∈ '1`	
`let q: &'q i32 = &'2 x`			`'2 ⊆ 'q, &'2 x ∈ '2`	`'p`
`let r: &'r i32`				`'p`
`q = &'3 y`	`&'2 x`		`'3 ⊆ 'q, &'3 x ∈ '3`	`'p, 'q, 'r`
`r=if f(){&'4 x}else{&'5 y}`			`'4, '5 ⊆ 'q, &'4 x ∈ ...`	`'p, 'q, 'r`
`x += 1`		`&'1 x, ..., &'4 x`		`'p, 'q, 'r`
`use(p, q, r)`				`'p, 'q, 'r`

Table 6.6: An example of the set membership constraints seen by Polonius during type-checking. Only two of the three invalidated loans would result in an error, as one of them has been $killed$ by an assignment. Therefore, when x is assigned at the fourth statement, the set memberships would be $'p = \{\&'1\ x\}$, $'q = \{\&'3\ y\}$, $'r = \{\&'4\ x, \&'5\ y\}$, resulting in only one two errors.< Additionally, note that the imprecision introduced by the if statement leads to dual subset constraints.

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

⊤	Input	Conclusion
	`subset('a, 'b, Mid(bb0[1]))`	`subset('a, 'c, Mid(bb0[1]))`
	`subset('b, 'c, Mid(bb0[1]))`	

⊥	Input	Conclusion
	`subset('a, 'b, Mid(bb0[1]))`	(nothing)
	`subset('a, 'c, Mid(bb0[1]))`	

Figure 6.8: Subset relations are transitive (as you would expect).

6.4 Loan Constraint Propagation[†]

The heart of Polonius is the loan constraint propagation where we figure out which loans belong to which provenances through the subset constraints introduced by subtyping relationships (as discussed in Section 3.2). A summary of the facts and relations involved can be found in Table 6.5. Additionally, a Rust snippet annotated with relevant facts and conclusions can be found in Table 6.6.

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 node 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 nodes where the constraints are generated by the Rust compiler, as seen by the input parameter `outlives(R1, R2)`. The brief one-liner in Listing 6.6 captures this fact, providing a "base case" for the computation. Additionally the mathematical fact that the subset relation is transitive is captured in Figure 6.8.

Listing 6.6: Subset relations hold at the node where they are introduced.

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

Finally, Polonius needs logic to trace 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 Figure 6.9, using the output of the liveness calculations described in Section 6.2.

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

⊤	Input	Conclusion
	`subset('a, 'b, Mid(bb0[1]))`	`subset('a, 'b, Start(bb0[2]))`
	`cfg_edge(Mid(bb0[1]), Start(bb0[2]))`	
	`region_live_at('a, Start(bb0[2]))`	
	`region_live_at('b, Start(bb0[2]))`	

⊥	Input	Conclusion
	`subset('a, 'b, Mid(bb0[1]))`	(nothing)
	`cfg_edge(Mid(bb0[1]), Start(bb0[2]))`	
	`region_live_at('a, Start(bb0[2]))`	

Figure 6.9: Subset relations propagate across CFG edges iff both of their provenance variables are live.

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 6.7, which just says that each provenance variable R contains the loan L that created it at node the node P where the borrow occurred.

Listing 6.7: 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 6.8.

Listing 6.8: A subset relation between two provenance variables R_1, R_2 propagates the loans of $R1$ to $R2$. This illustrates the mathematical rule $(l \in R_1 \land R_1 \subseteq R_2) \implies l \in R_2$.

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

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

$\top$	Input	Conclusion
	`requires('a, &x, Start(bb0[0]))`	`requires('a, &x, Mid(bb0[0]))`
	`region_live_at('a, Mid(bb0[0]))`	
	`killed(&x, Start(bb0[0])) = \bot`	
	`cfg_edge(Start(bb0[0]), Mid(bb0[0]))`	

$\bot$	Input	Conclusion
	`requires('a, &x, Start(bb0[0]))`	(nothing; the loan was killed)
	`killed(&x, Start(bb0[0]))`	
	`region_live_at('a, Mid(bb0[0]))`	
	`cfg_edge(Start(bb0[0]), Mid(bb0[0]))`	
	`requires('a, &x, Start(bb0[0]))`	(nothing; prov.var not live)
	`region_live_at('a, Mid(bb0[0])) = \bot`	
	`cfg_edge(Start(bb0[0]), Mid(bb0[0]))`	

Figure 6.10: Propagate loans across CFG edges for live provenance variables and loans whose references are not overwritten.

Finally, Polonius performs the flow-sensitive propagation of these membership constraints across edges in the CFG. This is done using the rule in Figure 6.10, where the requirements propagate across CFG edges for every loan L as long as the reference corresponding to L is not overwritten (`killed(L, _)`), and only for provenance variables that are still live. This corresponds to the T-Assignment rule of Oxide, seen in Rule (3.5), as discussed in Section 3.2.

With these relations figured out, we now know for each program point which provenance variable can contain which loan (and therefore which path each reference may have been borrowed from). All that is left is translating this information into errors.

6.4.1 Detecting Loan Violations

The compiler produces a set of nodes 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(P, L)`. All that remains for Polonius is to figure out which loans are live where (Listing 6.9), and determine if any of those nodes intersect with an invalidation of that loan (Listing 6.10).

Listing 6.9: Loans are live when their provenance variables are.

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

Listing 6.10: It is an error to invalidate a live loan.

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

6.5   Missing Features in 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.

6.5.1 Detecting Access to Deinitialised Paths

Most of the work required to support the full move initialisation is described in Section 6.3.3. However, this code is untested, not integrated into Polonius' error reporting, and lacks support for some edge cases. This feature also depends on the design of an interface for Polonius to report these errors back to Rust, and such an interface has not yet been agreed upon.

6.5.2 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 [29]. This feature also depends on the design of an extended interface for reporting errors from Polonius to the Rust compiler.

6.5.3   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 [30]. 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.

6.5.4   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 6.11 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 6.4. 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 6.11: 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).
```

52

```
  drop(x);
}

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

drop(y);
```

6.6 Conclusion

In this chapter we have seen how Polonius starts with an analysis of move paths
and their initialisation status. The analysis proceeds across program flow and is
simultaneously both over- and underapproximating. Initialisation information is
then used together with variable liveness to determine if a provenance variable is live
at any given node of the control flow. This information is then used together with
the subset relationships created from subtyping relations discovered during type-
checking to determine which loans may be live at which nodes of the program flow.
Joining these two insights together then helps us figure out which loans are live at
which program point, which allows us to determine if violating them is an error
or not. An annotated version of the motivating example of Listing 2.1 showing
Polonius deductions can be found in Listing 6.12 below.

*Listing 6.12: Polonius deductions on the motivating example. As almost every interesting transition happens
inside of the assignment, events happening at mid-point are written in comments after the line, and events
happening at the start of the node are written at before the line.*

```
fn next<'buf>(buffer: &'buf mut String) -> &'buf str {
    loop {

        // Start-point: invalidates(bw0)
        // Start-point: borrow_live_at = {}: no error!
        let event = parse(buffer);
        // Mid-point: bw0 dies through assignment to event
        // borrow_live_at(bw0)
        if true { // borrow_live_at(bw0)
            return event; // borrow_live_at(bw0)
        }
    }
}

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

Chapter 7

A Field Study of Polonius Inputs

> There are more things in heaven and earth, Horatio,
> Than are dreamt of in your philosophy.
>
> *Hamlet*, Act-I, Scene-V

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.

---

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

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.

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.
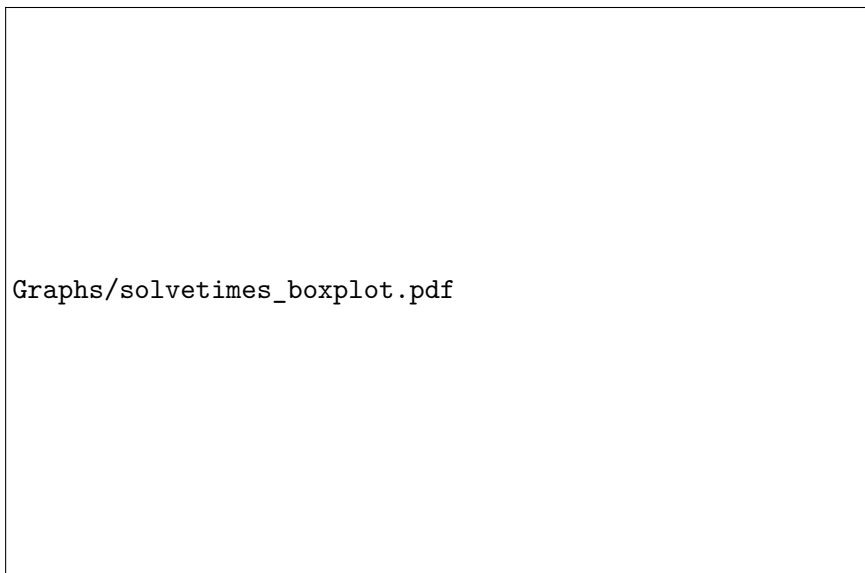
When studying inputs to Polonius, we are mainly interested in two properties; how large and how complex the function under analysis is. Neither of these can be measured directly, but potentially useful proxy variables would be sizes of input tuples, the number of variables, loans, and provenance variables, 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 Chapter 6 without the move error code of Section 6.3.3, 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 7.1, 7.2, and 7.4 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.
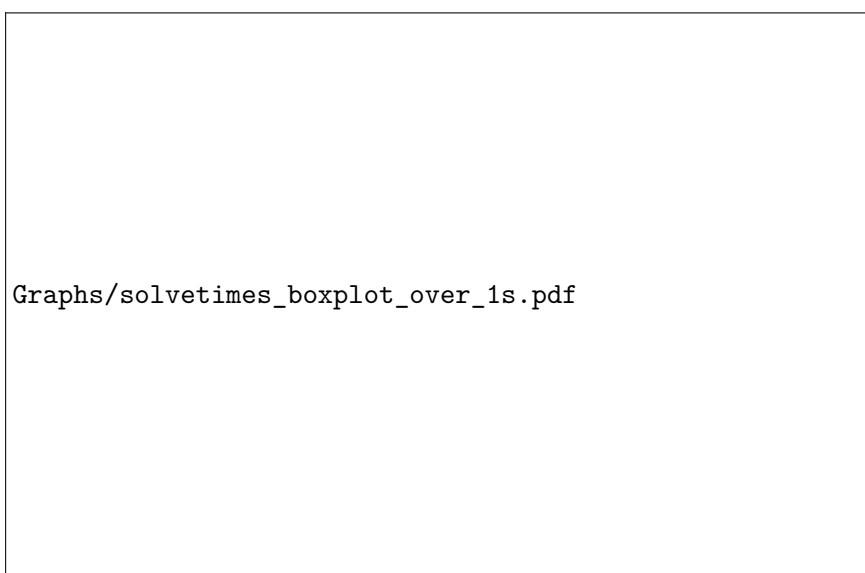
7.1 Performance

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 7.1. Apparently, Naive has a wider spread of runtimes than the others. Additionally, geometric means of the observed runtimes show improvements from

Figure 7.1: 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 7.2: 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.
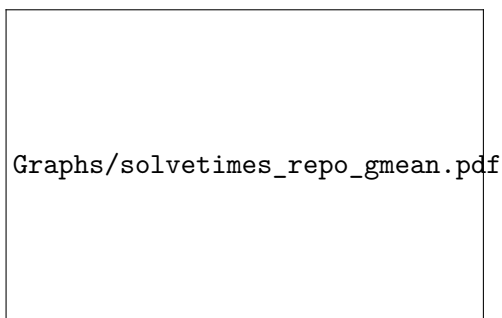
Figure 7.3: Geometric means of the runtimes per repositoriy and implementation.

hybridisation (Figure 7.3), 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 7.2.

7.2 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 7.4. 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 7.5 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).
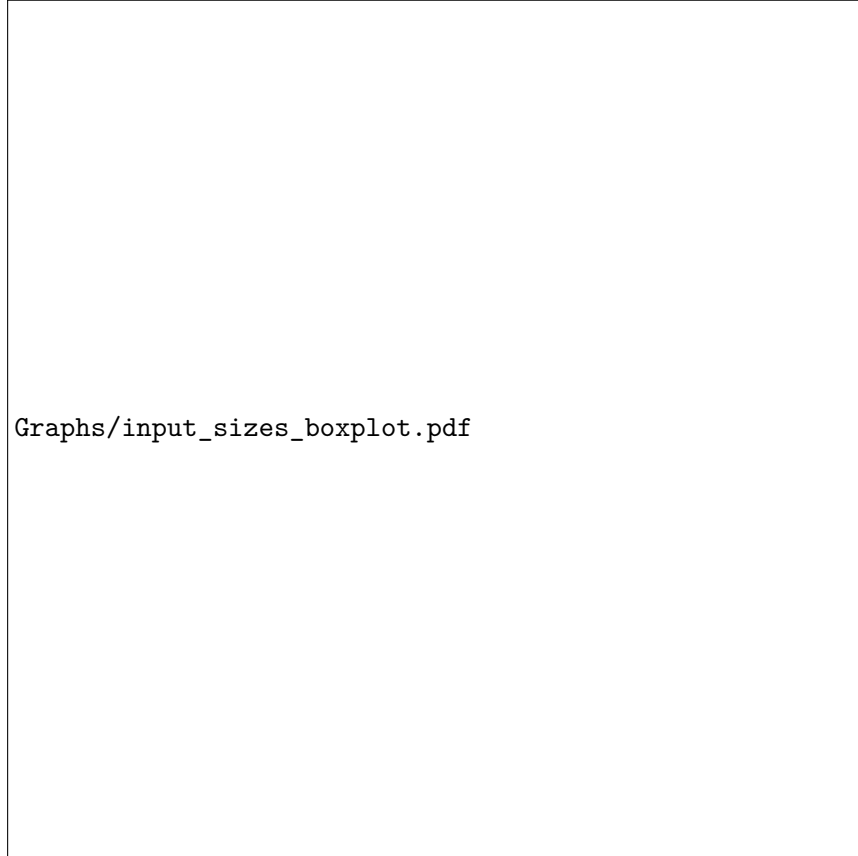
Graphs/input_sizes_boxplot.pdf

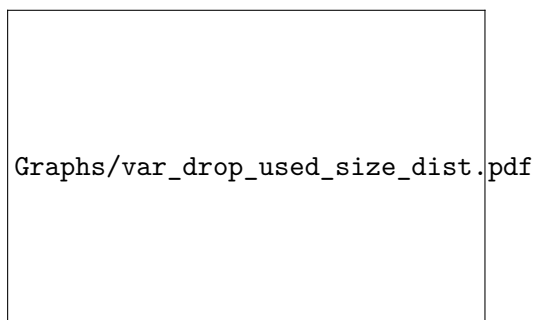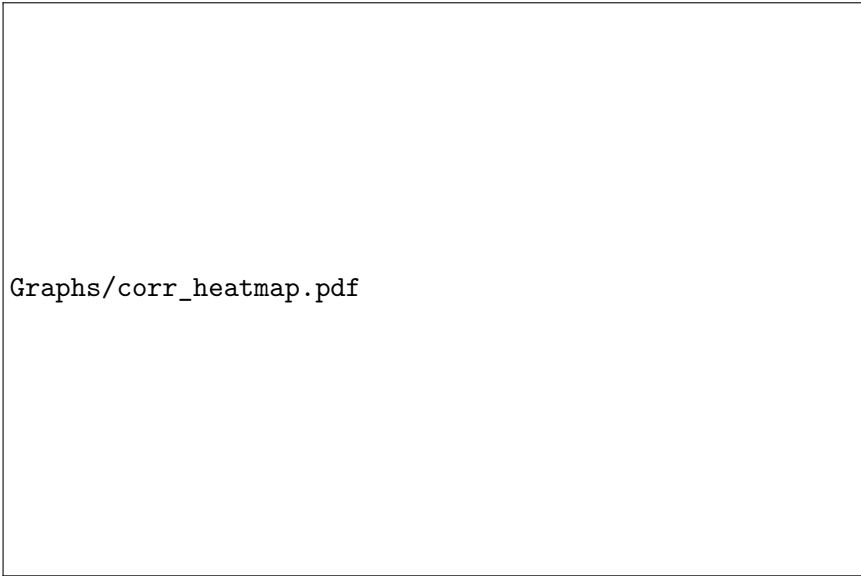*Figure 7.4: A box plot showing the distribution of the various input sizes.*

Graphs/var_drop_used_size_dist.pdf

Figure 7.5: A plot showing the distribution of sizes of `var_drop_used(V, P)`.

Figure 7.6: 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.

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.

7.3   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 7.6 and Table 7.1, while a scatter plot of the results with a linear regression for some interesting pairs of inputs can be seen in Figure 7.7.

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 7.1 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.

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

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 7.1).
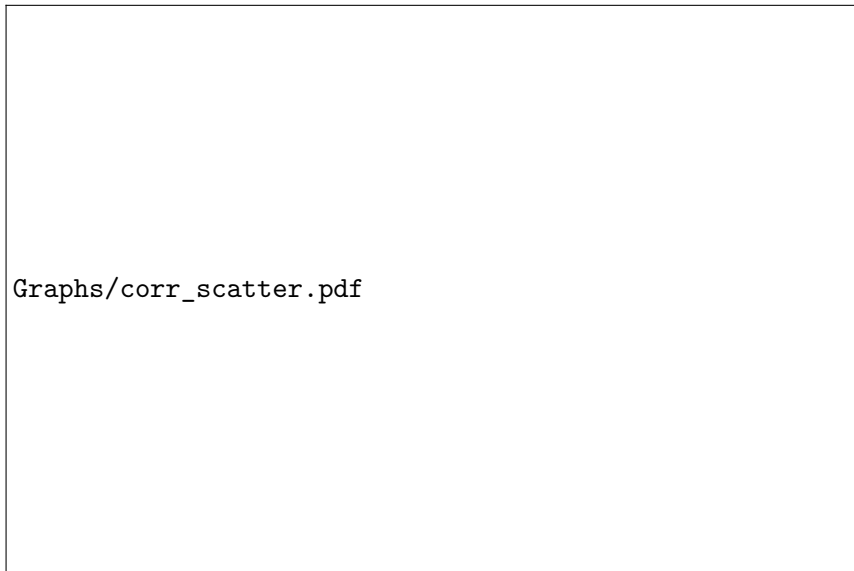
Figure 7.7: Scatter plot of runtimes under the naïve and optimised algorithms compared to variables and CFG edge count after having pruned extreme values (runtimes below 1 s or above 13 minutes). Y axis is runtime in seconds.

Chapter 8

Conclusions and Future Work

To sleep: perchance to dream: ay, there's the rub;
For in that sleep of death what dreams may come

Hamlet, Act-III, Scene-I

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 [11], 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.

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. There have been no comprehensive benchmarking against NLL, but the test suite for Rust shows more than one instance where a test crashes with an out-of-memory error under Polonius, suggesting that significant engineering for certain corner cases, at a minimum, would be needed before Polonius is even usable.

There are also several other sources of inefficiency in the fact generation code, which notably performs multiple walks across the CFG, needlessly increasing runtime. Additionally, many of the inputs are computed unnecessarily, and in many trivial cases the CFG could be compressed.

Returning to the analysis of Section 7 to inform our discussion on performance, 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 very early. 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 and using them as a starting point for Polonius could in many cases reduce the search space for Polonius significantly, as any other error has already been ruled out in the simpler analysis.

Regarding the supporting infrastructure, 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.

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 4.3. 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 8.1. 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.

Graphs/polonius-refactor.png

Figure 8.1: 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] D. Haraway, "Situated knowledges: The science question in feminism and the privilege of partial perspective," *Feminist Studies*, vol. 14, no. 3, pp. 575–599, 1988, issn: 00463663. [Online]. Available: `http://www.jstor.org/stable/3178066` (cit. on p. 7).

[2] C. Nichols and S. Klabnik. (2019). The Rust programming language, [Online]. Available: `https://doc.rust-lang.org/book/foreword.html` (visited on 04/01/2019) (cit. on pp. 7, 11).

[3] 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, 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. 8).

[4] V. Astrauskas, P. Müller, F. Poli, and A. J. Summers, "Leveraging rust types for modular specification and verification," *Proc. ACM Program. Lang.*, vol. 3, no. OOPSLA, 147:1–147:30, Oct. 2019, issn: 2475-1421. doi: `10.1145/3360573`. [Online]. Available: `http://doi.acm.org.ezproxy.its.uu.se/10.1145/3360573` (cit. on p. 9).

[5] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *ACM SIGPLAN Notices*, vol. 44, ACM, 2009, pp. 243–262 (cit. on p. 9).

[6] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *Computer Aided Verification*, S. Chaudhuri and A. Farzan, Eds., Cham: Springer International Publishing, 2016, pp. 422–430, isbn: 978-3-319-41540-6 (cit. on p. 9).

[7] B. Anderson, L. Bergstrom, M. Goregaokar, J. Matthews, K. McAllister, J. Moffitt, and S. Sapin, "Engineering the servo web browser engine using rust," in *Proceedings of the 38th International Conference on Software Engineering Companion*, ACM, 2016, pp. 81–89 (cit. on p. 10).

[8] J. Soller and the Redox developers. (2019). Redox OS, [Online]. Available: `https://www.redox-os.org/` (visited on 11/14/2019) (cit. on p. 10).

[9] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black, "Traits: Composable units of behaviour," in *European Conference on Object-Oriented Programming*, Springer, 2003, pp. 248–274 (cit. on p. 10).

[10] *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. 12).

[11] 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. 13, 16–18, 52, 62).

[12] P. Wadler, "Linear types can change the world!" In *Programming concepts and methods*, Citeseer, vol. 3, 1990, p. 5 (cit. on p. 15).

[13] S. Clebsch, *The pony programming language*, 2015 (cit. on p. 15).

[14] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil, "Deny capabilities for safe, fast actors," in *Proceedings of the 5th International Workshop on Programming Based on Actors, Agents, and Decentralized Control*, ser. AGERE! 2015, Pittsburgh, PA, USA: ACM, 2015, pp. 1–12, isbn: 978-1-4503-3901-8. doi: `10.1145/2824815.2824816`. [Online]. Available: `http://doi.acm.org.ezproxy.its.uu.se/10.1145/2824815.2824816` (cit. on p. 15).

[15] E. Castegren, "Capability-based type systems for concurrency control," PhD thesis, Uppsala University, Computing Science, 2018, p. 106, isbn: 978-91-513-0187-7 (cit. on p. 15).

[16] 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. 16).

[17] *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).

[18] 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. 23).

[19] *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. 24).

[20] 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. 29).

[21] *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. 30).

[22] 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. 30).

[23] 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. 32).

[24] Y. Smaragdakis and M. Bravenboer, "Using Datalog for fast and easy program analysis," in *International Datalog 2.0 Workshop*, Springer, 2010, pp. 245–251 (cit. on p. 32).

[25] T. Antoniadis, K. Triantafyllou, and Y. Smaragdakis, "Porting doop to souffé: A tale of inter-engine portability for datalog-based analyses," in *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP 2017, Barcelona, Spain: ACM, 2017, pp. 25–30, isbn: 978-1-4503-5072-3. doi: `10.1145/3088515.3088522`. [Online]. Available: `http://doi.acm.org.ezproxy.its.uu.se/10.1145/3088515.3088522` (cit. on p. 32).

[26] M. Madsen, M.-H. Yee, and O. Lhoták, "From Datalog to Flix: A declarative language for fixed points on lattices," *SIGPLAN Not.*, vol. 51, no. 6, pp. 194–208, Jun. 2016, issn: 0362-1340. doi: `10.1145/2980983.2908096` (cit. on p. 32).

[27] W. C. Benton and C. N. Fischer, "Interactive, scalable, declarative program analysis: From prototype to implementation," in *Proceedings of the 9th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, ser. PPDP '07, Wroclaw, Poland: ACM, 2007, pp. 13–24, isbn: 978-1-59593-769-8. doi: `10.1145/1273920.1273923` (cit. on p. 32).

[28] S. Dawson, C. R. Ramakrishnan, and D. S. Warren, "Practical program analysis using general purpose logic programming systems—a case study," in *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, ser. PLDI '96, Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 117–126, isbn: 0-89791-795-2. doi: `10.1145/231379.231399` (cit. on p. 32).

[29] 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. 52).

[30] ——, (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. 52).