Modelling Rust's Reference Ownership Analysis Declaratively in Datalog

Albin Stjerna

July 16, 2019

Abstract

This thesis is about something.

Contents

Chapter 1

Introduction

Rust is a young systems language originally developed at Mozilla Research [1]. Its stated intention is to combine high-level programming language features like automatic memory management and strong safety guarantees, in particular in the presence of concurrency or parallellism, with predictable performance and pay-as-you-go abstractions in the style of C++ and similar systems languages.

One of its core features is the memory ownership model, which enables compile-time safety guarantees against data races, unsafe pointer dereferencing, and runtime-free automatic memory management, including for dynamic memory allocated on the heap.

This report describes the implementation of Rust's memory safety checker, called the borrow checker, in an embedded Datalog engine, as well as its optimisation. In practice, the full analysis encompasses a variable liveness analysis, initialisation and deinitialisation tracking, and may-reference analysis for validation of Rust's memory safety guarantees.

mention results.

Chapter 2

Background

Whenever a reference to a resource is created in Rust, its borrowing rules described in Section 2.1 must be respected for as long as the reference is alive, including across function calls [2]. In order to enforce these rules, the Rust language treats the scope of a reference, called its lifetime, as part of its type, and also provides facilities for the programmer to name and reason about them as they would any other type.

Since its release, the Rust compiler has been extended through proposal RFC 2094 to add support for so-called non-lexical lifetimes (NLLs), allowing the compiler to calculate lifetimes of references based on the control-flow graph rather than the lexical scopes of variables [3]. During the spring of 2018, Nicholas Matsakis began experimenting with a new formulation of the borrow checker, called Polonius, using rules written in Datalog [4]. The intention was to use Datalog to allow for a more advanced analysis while also allowing for better compile-time performance through the advances done centrally to the fixpoint solving provided by the Datalog engine used for the computations [5].

Datalog, and other types of logic programming has been previously employed for program analysis, in particular pointer analyses such as may-point-to and must-point-to analysis, both similar to what is described in this report in that they require fix-point solving and graph traversal, often with a context sensitive analysis (i.e. respecting function boundaries) like the one described here[6]–[17]. These systems employ a wide variety of solver technologies and storage back-ends for fact storage, from Binary Decision Diagrams (BDDs) to explicit tuple storage, as used in this study. Some of them, like Flix, also extends Datalog specifically for static program analysis[16].

In addition to being context-sensitive, Rust's borrow checker is also flow-sensitive (i.e. performs analysis for each program point), like the system described by Hardekopf and Lin, and whose form is very similar to the analysis performed in practice by Polonius [18].

A 2016 study uses the Soufflé system, which synthesizes performant C++ code from the Datalog specifications, similar to how Datafrog embeds a minimal solver as a Rust library, to show promising performance for analysis of large programs [19]. The Doop system, developed by Smaragdakis and Bravenboer, also shows that explicit tuple storage sometimes vastly outperforms BDDs in terms of execution time [14], as do sparse bitmaps [12].

Formally, the semantics of Rust's lifetime rules have been captured in the language Oxide, described by Weiss, Patterson, Matsakis, *et al.* in a draft paper which describes a minimal Rust-like language called Oxide along with its type system [20]. Oxide is notable in that it shares Polonius' view of variables as sets of possible loans that would give rise to the reference. However, as the Oxide paper already covers the formalisms of such a type system, this report will not concern itself with the semantics of the borrow rules except where necessary.

The contributions made within the scope of the thesis project specifically includes the implementation of liveness and initialisation calculations (Sections 3.1.3 and 3.1.2 respectively), as well as work on region inference for higher-ranked types. Finally, the report also evaluates the runtime performance of the system and suggests some potential optimisations in Section 3.4, and performs a field study of the shape of input data in Section 3.3. The core rules of Polonius for the region constraints were already written when the project started, but are described in Section 3.1.4 for completeness. For clarity, sections detailing components not developed by me are marked with (†).

2.1 The Borrowing Rules

Most of these examples are borrowed from Weiss, Patterson, Matsakis, *et al.* [20].

Variables must be provably initialised before use Whenever a variable is used, the compiler must be able to tell that it is guaranteed to be initialised:

```
let x: u32;
let y = x + 1; // ERROR: x is not initialised
```

A move deinitialises a variable Whenever ownership of a variable is passed on (*moved* in Rust parlance), e.g. by a method call or reassignment, the variable becomes deinitialised:

```
struct Point(u32, u32);

let mut pt = Point(6, 9);
let x = pt;
let y = pt; // ERROR: pt was already moved to x
```

There can be any number of shared references A shared reference, also called a *borrow* of a variable, is created with the `&` operator, and there can be any number of simultaneously live shared references to a variable:

```
struct Point(u32, u32);

let mut pt = Point(6, 9);
let x = &pt;
let y = &pt; // This is fine
```

There can only be one simultaneous live unique reference Whenever a unique reference is created, with `&mut`, it must be unique:

```
struct Point(u32, u32);

let mut pt = Point(6, 9);
let x = &mut pt;
let y = &mut pt; // ERROR: pt is already borrowed

// code that uses x and y
```

This error happens even if the first borrow is shared, but not if either `x` or `y` are dead (not used).

A reference must not outlive its referent A reference must go out of scope at the very latest at the same time as its referent, which protects aganst use-after-frees:

```
struct Point(u32, u32);

let x = {
    let mut pt = Point(6, 9);
    &pt
};

let z = x.0; // ERROR: pt does not live long enough
```

In this example, we try to set `x` to point to the variable `pt` inside of a block that has gone out of scope before `x` does.

2.1.1 Variables, Places, and Paths

A notable detail of the borrow check is what is meant by a "variable". In Rust, some data structures, such as `struct`s, and tuples, are analysed at the granularity of the individual components, which may have arbitrarily deep nesting (known at compile-time). This means that the following code, for example, *does* pass the borrow check, as the loans don't 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 don't overlap!
```

In our instance, the root variable `pt` contains the *paths* `pt.1` and `pt.2`. Such paths constitute a tree with its root in the variable itself. Both the core borrow check and the initialisation tracking that we will discuss reasons about variables on the path level, with some notable exceptions including vectors and arrays.

2.2 The Borrow Check in the Rust Compiler

The logic of the borrow check as described in Section 2.3 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. The input data to the Polonius solver 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.

2.3 From Lifetimes to Provenance Variables

As the lifetime of its value is a part of a reference variable's type, it can be referred to by name like any other type using the syntax `&'lifetime`. In the literature, the terms "region" [4], "(named) lifetime" , and "reference provenance" [20] (provenance) are all employed. As the section heading suggests, I will use the last one of them as I believe it best captures the concept. Named provenances (such as `'lifetime` above) are referred to as "provenance variables".

From a type system perspective, the provenance is part of the type of any reference and corresponds to the borrow expressions that might have generated it in the Polonius formulation of the borrow check. For example, if a reference `r` has the type `&'a Point`, `r` is only valid as long as the terms of the loans in `'a` are upheld. Take for example the code of Listing 2.1, where `p` would have the type `&'a i32` where `a` is the set $\{L_0, L_1\}$.

Listing 2.1: An example of a multi-path borrow.

```
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 assigned 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 regions establishes a set membership constraint between the regions, as seen in Rule 2.3 of Section 2.4, which gives a brief introduction to the reference ownership analysis of Polonius from a type system perspective.

Finally, when talking about the *liveness* of a provenance variable r at some point in the control-flow graph p, we will mean that r occurs in the type of at least one variable which is live at p. This has the semantic implication that any of the loans in r might be dereferenced at control-flow points reachable from p, and thus that the terms of the loans in r must be respected at that point. For a more formal explanation of this, including an expansion of how provenance variables are populated, please see the following section.

2.4 Reference Ownership as a Type System

Summing up the ongoing work of Weiss, Patterson, Matsakis, *et al.* on formalising the reference ownership rules of Rust [20]. At the heart of the type system lies the flow-sensitive typing judgments seen in Rules 2.1 and 2.2, both of Weiss, Patterson, Matsakis, *et al.*'s paper (Figure 1), with slight modifications.

The first rule (2.1) shows that for a given environment Γ, a move of a given variable π (occurring if π cannot be copied) is only valid if π is the only name for that value and removes that value from the environment, effectively deallocating it.

$$\frac{\Gamma \vdash_{\text{mut}} \pi : \tau^s \quad \text{noncopyable } \tau^s}{\Sigma; \Delta; \Gamma \vdash \boxed{\pi} : \tau^s \implies \Gamma - \pi} \tag{2.1}$$

The second rule, Rule 2.2, tells that we may create an ω-reference to any variable where ω-use is safe (either sharedly or uniquely), and produce a reference to that variable of the type "reference to a value of π's type, τ, with its provenance variable being the set containing only that loan, denoted ${}^{\omega}\pi$".

$$\frac{\Gamma \vdash_{\omega} \pi : \tau}{\Sigma; \Delta; \Gamma \vdash \boxed{\&\omega\pi} : \{{}^{\omega}\pi\}\,\omega\tau \implies \Gamma} \tag{2.2}$$

fudge the ω-safety rules.

Both of these rules 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, I 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, I will also describe how provenance variables come into relation with each other through type unification.

describe the branching rules

describe the sequencing rules

describe type unification

describe assignment rules

$$\frac{a \subseteq b}{\texttt{\&'a u32} <: \texttt{\&'b u32}} \tag{2.3}$$

Under these rules, a Rust function is valid from the perspective of the borrow check if, for each point in the control flow graph, the borrow rules are respected. In logic terms, this means that we cannot find a point p in the control-flow graph such that a loan l is live there while its conditions are being validated, or more formally $\neg\exists(l, p) : \text{Live}(l, p) \wedge \text{Invalidated}(l, p))$. These relations are then defined as follows:

the $\text{Live}(l, p)$ **relation for a loan** l A loan is live at a given position p if there exists a live provenance variable R such that $l \in R$, or formally: $\exists(l, R, p, \Gamma_p) : \text{Live}(R, p) \wedge l \in \Gamma_p(R)$, where I by Γ_p mean the environment at p, and $\Gamma_p(R)$ mean the loans in R at point p. This is not how Weiss uses gamma!

the $\text{Live}(R, p)$ **relation for a provenance variable** R A provenance variable is live at p if and only if a variable whose type it is part of is live there. type-system describe "there exists a variable v with type τ such that R is τ's provenance variable and $\text{Live}(v, p)$"

the $\text{Live}(v, p)$ **relation for a variable** v A variable is live at p if there exists a point q reachable from p where v is accessed either directly or by a `drop()` (see Section 2.5 for a discussion of when a variable is considered to be used in a deallocation), without having been overwritten ("killed") somewhere between q and p. Notably, a variable is only used by a Drop if it might be partially initialised when the drop happens. Formalise this.

the $\text{MayBeInitialised}(v, q)$ **relation for a variable** v A variable may be (partially) initialised if there is a path through the CFG from a point p, where some part π of v is initialised, to q without π getting deallocated along that path. Formalise this.

$l \in R$ **for loan** l A loan is a member of R if and only if is was originally part of the loan that created R, or if $\exists R' : l \in R' \wedge R' \subseteq R$. Describe relationship to gamma! How to point-qualify these relationships?

$R' \subseteq R$ **for provenance variables** R, R' write this

2.5 Deallocation As a Special Case of Variable Use

When Rust's variables go out of scope, they are implicitly deallocated, or dropped in Rust parlance. Explicit deallocation is also possible by calling the function `drop()`, which takes ownership of a variable (that is, deinitialises it) and performs deallocation, or, for complex objects, calls the `drop()` method. For some types such as integers, deallocation is not necessary and the compiler generates no actual `drop()`:s in the MIR.

Rust provides a default deallocator for data structures, but it can be overridden. This has repercussions on liveness calculations, because while the default deallocator for an object never needs to access its fields except to deallocate them, a custom deallocator might access any of them in arbitrary ways. This means that any conditions of a loan that resulted in a reference r stored in a `struct` s instance a must only be respected as far as `a.drop()` is concerned if s implements a custom deallocator. Otherwise the loan of r may be safely violated, as the default deallocator never dereferences r and thus doesn't require r to be valid. An illustration of this can be seen in Listing 2.2.

Listing 2.2: The custom deallocator for `OwnDrop` enforces the loan of `data`, but the loan in `OwnDrop` is effectively dead and thus can be violated.

```
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
}
```

2.6 Datalog and Datafrog

Datalog is a derivative of the logic programming language Prolog, with x guarantees about execution. It describes fixpoint calculations over logical relations as predicates, described as fixed input *facts*, computed *relations*, or *rules* describing how to populate the relations based on facts or other relations. For example, defining a fact describing that an individual is another individual's parent might look like `parent(mary, john).`, while computing the `ancestor` relation could use the two rules `ancestor(X, Y) :- parent(X, Y).` and `ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y)` reflecting the fact that ancestry is respectively either parenthood or transitive parenthood (example from the Wikipedia article on Datalog [21]). This thesis uses the notation of Soufflé [19].

Datafrog [5] is a minimalist Datalog implementation embedded in Rust, providing an implementation of a worst-case optimal join algorithm as described in [22]. The fact that Datafrog is embedded in Rust means that standard Rust language abstractions are used to describe the computation. Static facts are described as `Relations`, while `Variables` are used to capture the results of computations, both of which are essentially sets of tuples. Rules are described using a join with either a `Variable` or a `Relation`, with an optimised join method for joins with the variable itself. 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.

Listing 2.3: The implementation of `var_live(V, P)` in Datafrog

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

As an example, the Datafrog code for `var_live(V, P)` of Listing 3.1 becomes the code in Listing 2.3, and the corresponding join used for the first half of `region_live_at(R, P)` of Listing 3.2 can be seen in Listing 2.4.

Listing 2.4: 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 2.4 `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.

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_live_var` of Listing 2.3), 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.

Chapter 3

A Declarative Model for the Rust Borrow Checker

3.1 The Borrow Checker in Datalog

3.1.1 Starting Facts

The following short-hand names are used:

R is a provenance.

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

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

V is a variable.

`borrow_region(R, L, P)` the provenance R may refer to data from loan L starting at the point P (this is usually the point *after* the right-hand-side of a borrow expression).

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

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

`killed(L, P)` when some prefix of the path (explain prefixes and paths) borrowed in L is assigned at point P.

`outlives(R1, R2, P)` when $R_1 \subseteq R_2$ must hold at point P, a consequence of subtyping relationships as described in Rule (2.3).

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

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

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

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

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

`var_drops_region(V, R)` when the type of V includes the provenance R, and V also implements a custom drop method which might need all of V's data, as discussed in Section 2.5.

3.1.2 Variable Initialisation

Reporting Initialisation Errors

3.1.3 Reference Liveness

The basic liveness of a variable (Listing 3.1) is defined 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; in essence liveness is propagated backwards through the CFG.

Listing 3.1: The Datalog rules for variable use-liveness.

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

describe drop-liveness

There is also a very similar relation for drop-liveness with an identical shape and different input relations corresponding to drop-uses. An example of the output from this calculation can be seen in Figure 3.1.

Both of these are then used to calculate the reference liveness relation (Listing 3.2), which serves as input for the rest of the borrow checker. A given provenance r is live at some point p if it is in the type of a variable v which is either drop-live or use-live at p, with some notable caveats for drop-liveness (discussed in Section 2.5) embedded in the `var_drops_region` relation.

Listing 3.2: The Datalog rules for provenance liveness.

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

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

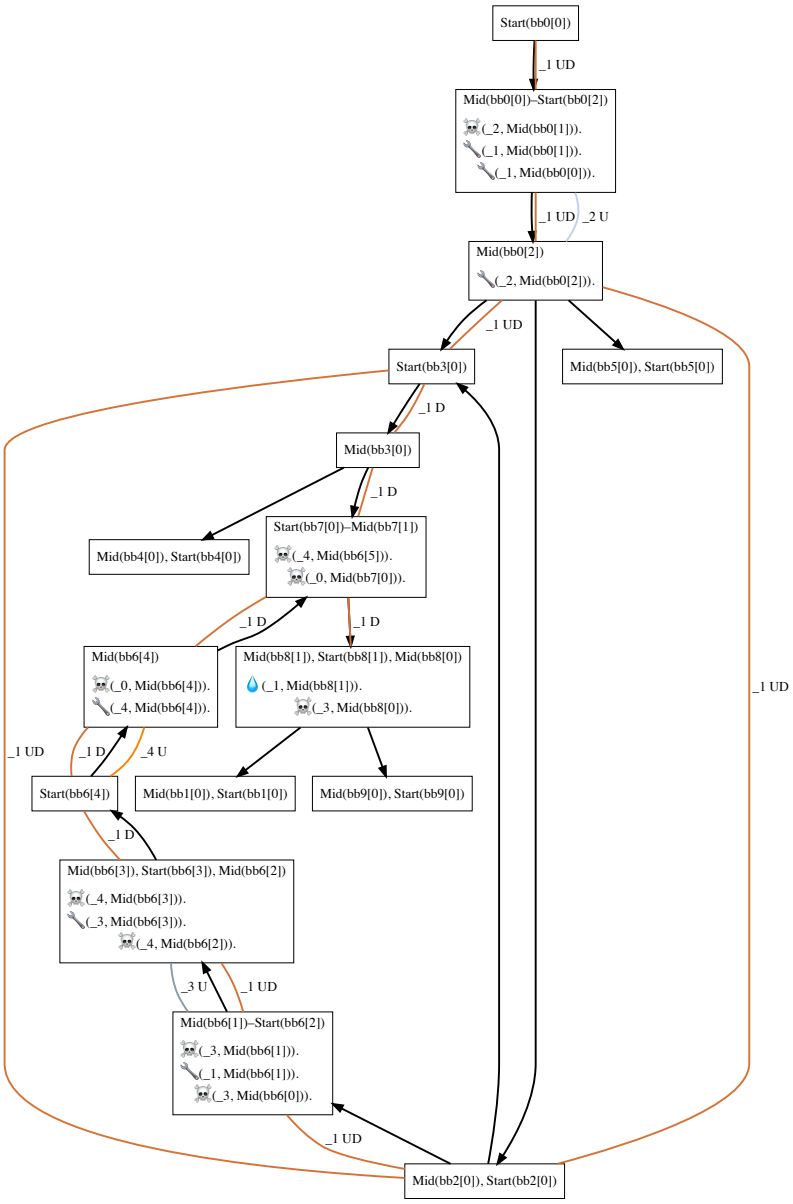

Figure 3.1: A graph representation of the the variable liveness calculation results, with relevant Polonius facts as they occur (a droplet symbolising `var_drop_used`*, a wrench* `var_used`*, and a skull and crossbones symbolising* `var_defined`*). Variables are named by prefixing underscores, and edges annotated with the propagated live variable and its liveness type(s) (**D**rop or **U**se).*

3.1.4 Loan Constraint Propagation†

The first relation used in Polonius is the `subset(R1, R2, P)` relation, which states that $R_1 \subseteq R_2$ for two provenance variables R_1, R_2 at point p in the CFG. Initially, these have to hold at the points where the constraints are generated by the Rust compiler, as seen by the input parameter `outlives`. The brief one-liner in Listing 3.3 captures this fact, providing a "base case" for the computation. Additionally, subset relations are transitive, which is captured in Listing 3.4.

Listing 3.3: Subset relations hold at the point where they are introduced.

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

Listing 3.4: Subset relations are transitive.

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

Finally, Polonius needs logic to carry these subset relations across program flow. Recalling the rules of which ones? refer back to the theory here, a subset relation should be propagated across an edge of the control-flow graph if and only if its provenance variables are live, otherwise we are in a "if a tree falls in the woods" situation where the conditions of the loans can be safely violated as there is no live reference to be affected. Therefore, the rule for propagating the subset constraint across a CFG edge $P \rightarrow Q$ becomes the formulation seen in Listing 3.5, which notably uses the output of the liveness calculations described in Section 3.1.3.

Listing 3.5: Subset relations propagate across CFG edges iff their provenance variables are live.

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

These rules describe how provenance variables relate to each other. The other part of the logic describes which loans belong to which provenance variable. The trivial base case is shown in Listing 3.6, which just says that each provenance variable R contains the loan L that created it at point the point P where the borrow occurred.

Listing 3.6: A provenance variable trivially contains (`require`s) 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 3.7.

Listing 3.7: A subset relation between two provenance variables R_1, R_2 propagates the loans of $R1$ to $R2$.

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

Finally, Polonius performs the flow-sensitive propagation of these membership constraints across edges in the CFG. This is done using the rule in Listing 3.8, where the requirements propagate across CFG edges for every loan L as long as the reference corresponding to L is not overwritten (`killed`), and only for provenance variables that are still live. Recall (from where?) that loans are uniquely tied to one point in the CFG, and therefore to a single place. This is why a single loan is killed by a single assignment.

Listing 3.8: Propagate loans across CFG edges for live provenance variables and loans whose references are not overwritten.

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

Detecting Loan Violations

The compiler produces a set of points in the CFG where a loan could possibly be violated (e.g. by producing a reference to a value that already has a unique reference) in `invalidates`. All that remains for Polonius is to figure out which loans are live where (Listing 3.9), and see if any of those points intersect with an invalidation of that loan (Listing 3.10).

Listing 3.9: Loans are live when their provenance variables are.

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

Listing 3.10: It is an error to invalidate a live loan.

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

3.1.5 Illegal Subset Relations

this isn't implemented yet!!!

3.2 Generating Facts for Polonius in the Rust Compiler

say something about how and when the facts are generated in the Rust compiler.

3.3 A Field Study of Polonius Starting Facts

I selected 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, for analysis. [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. The source code of the packages was then translated to Polonius input files for a total of 400GBs of tab-separated tuples, which I used to measure Polonius solve-time 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 took more than 30 minutes, required more memory than what was available, or where fact generation took longer than 15 minutes. After this selection process, 10 000 repositories remained for the final study.

It is worth pointing out that Polonius' analysis as part of the Rust compiler will be performed on non-compiling Rust code as well, and that the requirement that all crates must compile is an entirely artificial one and might exclude interesting input cases. However, it is non-trivial to separate build failures due to syntax errors or missing dependencies from build failures at later stages of the build process that would involve Polonius. Therefore all non-compiling crates were excluded from the study.

The main metric of "performance" in this study is the time it would take Polonius to solve a given set of inputs. In this case, this also includes the time it takes to parse and intern the files of tab-separated input tuples to Polonius, which is assumed to be negligible compared to the runtime of the analysis itself, which includes both a liveness analysis and the borrow check itself. In practical scenarios the peak memory usage of the analysis would also be an interesting metric.

When studying inputs to Polonius, I am mainly interested in two properties; how large and how complex the function under analysis is. Neither of these can be measured directly, but useful proxy variables would be sizes of input tuples, the number of variables, loans, and regions, 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.

3.4 Optimising the Borrow Checker

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

Chapter 4

Conclusions and Future Work

Bibliography

[1] N. D. Matsakis and F. S. Klock II, "The rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '14, event-place: Portland, Oregon, USA, New York, NY, USA: ACM, 2014, pp. 103–104, isbn: 978-1-4503-3217-0. doi: 10.1145/2663171.2663188. [Online]. Available: http://doi.acm.org/10.1145/2663171.2663188 (visited on 03/29/2019) (cit. on p. 4).

[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 p. 5).

[3] *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) (cit. on p. 5).

[4] N. D. Matsakis. (Apr. 27, 2018). An alias-based formulation of the borrow checker, Baby Steps, [Online]. Available: http://smallcultfollowing.com/babysteps/blog/2018/04/27/an-alias-based-formulation-of-the-borrow-checker/ (visited on 04/01/2019) (cit. on pp. 5, 8).

[5] *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 pp. 5, 12).

[6] 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. [Online]. Available: http://doi.acm.org.ezproxy.its.uu.se/10.1145/231379.231399 (cit. on p. 5).

[7] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee, "Points-to analysis using bdds," *SIGPLAN Not.*, vol. 38, no. 5, pp. 103–114, May 2003, issn: 0362-1340. doi: 10.1145/780822.781144. [Online]. Available: http://doi.acm.org.ezproxy.its.uu.se/10.1145/780822.781144 (cit. on p. 5).

[8] E. Hajiyev, M. Verbaere, O. de Moor, and K. De Volder, "Codequest: Querying source code with datalog," in *OOPSLA Companion*, Citeseer, 2005, pp. 102–103 (cit. on p. 5).

[9] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," *SIGPLAN Not.*, vol. 39, no. 6, pp. 131–144, Jun. 2004, issn: 0362-1340. doi: 10.1145/996893.996859. [Online]. Available: http://doi.acm.org/10.1145/996893.996859 (cit. on p. 5).

[10] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel, "Context-sensitive program analysis as database queries," in *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, ACM, 2005, pp. 1–12 (cit. on p. 5).

[11] 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. [Online]. Available: http://doi.acm.org/10.1145/1273920.1273923 (cit. on p. 5).

[12] B. Hardekopf and C. Lin, "The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07, San Diego, California, USA: ACM, 2007, pp. 290–299, isbn: 978-1-59593-633-2. doi: 10.1145/1250734.1250767. [Online]. Available: http://doi.acm.org.ezproxy.its.uu.se/10.1145/1250734.1250767 (cit. on pp. 5, 6).

[13] Y. Smaragdakis, M. Bravenboer, and O. Lhoták, "Pick your contexts well: Understanding object-sensitivity," in *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '11, Austin, Texas, USA: ACM, 2011, pp. 17–30, isbn: 978-1-4503-0490-0. doi: 10.1145/1926385.1926390. [Online]. Available: http://doi.acm.org/10.1145/1926385.1926390 (cit. on p. 5).

[14] 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 pp. 5, 6).

[15] G. Balatsouras, K. Ferles, G. Kastrinis, and Y. Smaragdakis, "A datalog model of must-alias analysis," in *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ACM, 2017, pp. 7–12 (cit. on p. 5).

[16] M. Madsen, M.-H. Yee, and O. Lhoták, "From datalog to flix: A declarative language for fixed points on lattices," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16, Santa Barbara, CA, USA: ACM, 2016, pp. 194–208, isbn: 978-1-4503-4261-2. doi: 10.1145/2908080.2908096. [Online]. Available: http://doi.acm.org.ezproxy.its.uu.se/10.1145/2908080.2908096 (cit. on p. 5).

[17] M. Eichberg, S. Kloppenburg, K. Klose, and M. Mezini, "Defining and continuous checking of structural program dependencies," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08, Leipzig, Germany: ACM, 2008, pp. 391–400, isbn: 978-1-60558-079-1. doi: 10.1145/1368088.1368142. [Online]. Available: http://doi.acm.org.ezproxy.its.uu.se/10.1145/1368088.1368142 (cit. on p. 5).

[18] B. Hardekopf and C. Lin, "Semi-sparse flow-sensitive pointer analysis," in *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '09, Savannah, GA, USA: ACM, 2009, pp. 226–238, isbn: 978-1-60558-379-2. doi: 10.1145/1480881.1480911. [Online]. Available: http://doi.acm.org.ezproxy.its.uu.se/10.1145/1480881.1480911 (cit. on p. 5).

[19] 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 pp. 6, 12).

[20] 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. 6, 8, 9).

[21] Wikipedia contributors, *Datalog — Wikipedia, the free encyclopedia*, https://en.wikipedia.org/w/index.php?title=Datalog&oldid=899388108, [Online; accessed 1-June-2019], 2019 (cit. on p. 12).

[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. 12).