Modelling Rust's Reference Ownership Analysis Declaratively in Datalog

Albin Stjerna

August 28, 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 3.5, and performs a field study of the shape of input data in Section 3.4. 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

This section will demonstrate the rules enforced by the borrow check. Most of these examples are borrowed directly or only slightly modified 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 structs, 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 do not overlap:

```rust
struct Point(u32, u32);

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

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

Finally, we sometimes talk of a path having *prefixes*, where a prefix is anything above a "leaf" in the tree spanning all paths. For example, the path `x.y` would have the prefixes `x` and `x.y`. This is useful when reasoning

2.2 Why a Reformulation?

Polonius' design is driven by shortcomings of the current borrow checker (NLL), which does not have the same degree of flow-sensitivity as Polonius provides. For this reason, NLL rejects certain desirable patterns of Rust, such as the code in Listing 2.1, that Polonius is designed to accept. Leaving out the flow-sensitive analysis of NLL was motivated by performance reasons and is one of the motivations for using the Datalog implementation, in addition to the intention of having a clearer formulation in a data-oriented language.

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`.[21]

```rust
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!()
}
```
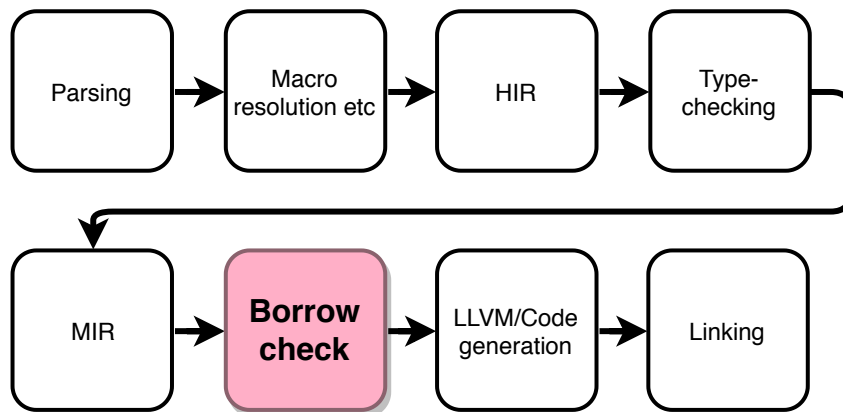
Figure 2.1: An overview of the Borrow Check's place in the process of compiling Rust code, as described in the Rust Developer's Guide [22].

2.3 The Borrow Check in the Rust Compiler

The logic of the borrow check as described in Section 2.4 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 2.1.

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.

The MIR consists of basic blocks in the traditional compilers sense, each containing a set of statements and usually ending with a *terminator*, an expression providing a branching to other basic blocks [22], [23]. A rendering of the MIR of the program in Listing 2.2 can be seen in Figure 2.2.

Listing 2.2: A minimal Rust program featuring branching and a function call. The MIR form of this program is shown in Figure 2.2.

```rust
fn main() {
    let x = 17;
    let z = if x == 3 {
        92
    } else {
        x
    };

    do_something(z);
}
```
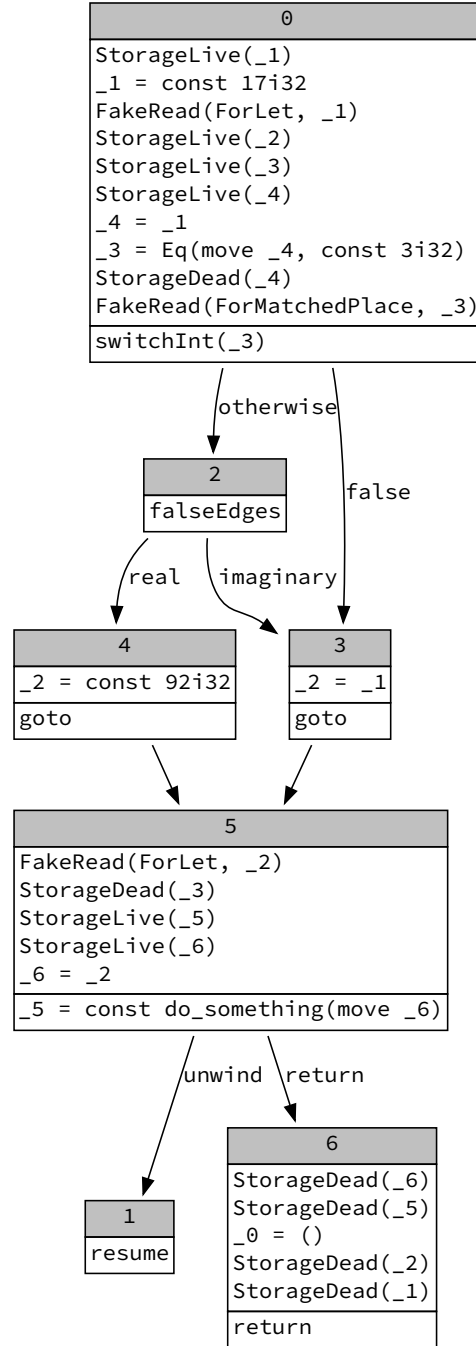
9

Figure 2.2: A graph rendering of the `main()` *function from the Rust program in Listing 2.2, illustrating branching (block 0), and a function call (5). Note the* **unwind** *arm of block 5's terminator (last line), which will be followed if the function call panics.*

2.4 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 2.3, where p would have the type `&'a i32` where a is the set $\{L_0, L_1\}$.

Listing 2.3: 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 **??** of Section 2.5, 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.

2.5 Reference Ownership as a Type System

In this section, I will summarise the ongoing work of Weiss, Patterson, Matsakis, *et al.* on formalising the reference ownership rules of Rust [20] into a type system as it relates to the Polonius formulation of the borrow checker. 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.

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

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 $^\omega\pi$".

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

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" [20]. 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 *unique* loans in the provenance part of τ.

Rules (2.1) and (2.2) constitute "base cases" for the ownership system, showing how variables get removed from the environment, and how provenance variables in reference types are created. In order to describe the full analysis, 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.

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 [20]. As in Oxide, the "type en-

vironment" 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. To capture this, I will use the name Γ_p to refer to the type-environment mapping places to their types just before evaluating an arbitrary point p of the CFG.

In Oxide, the typing rules for branch expressions provide for a type unification of the if expression such that its value unifies the provenance variables. The MIR produced by such a branching would 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. The same unification in Polonius has two aspects to it: approximation of reference origins, captured by transfer across CFG edges in the MIR, and type unification, captured on assignment.

Simply put, each assignment allows different types on each side of the assignment, as long as the types unify, as seen in Rule (2.3). This leads to the same type-unification for references as seen in Oxide (Rule (2.4)). In practice, this means that each assignment of a place π_1 from another place π_2 in the environment Γ updates Γ to have a new mapping such that π_2's provenance variable ρ_2 is a superset of π_1's.

Finally, in Oxide's T-BRANCH rule for branching, the type environments are unified into an approximate type environment that combines provenances in the variables from each branch. In Polonius, the corresponding logic is simply that a basic block's starting place-environment Γ is the transitive closure of the union of the place-environments at the terminators of the previous blocks.

$$\frac{\pi_1 : \tau_1 \ \ \pi_2 : \tau_2 \ \ \tau_1 \sim \tau_2 \implies \tau \ \ \Gamma \vdash_{uniq} \pi_1}{\Sigma; \Delta; \Gamma \vdash \boxed{\pi_1 = \pi_2} \ : \ \tau \implies \Gamma - \pi_1, \pi_1 \ : \ \tau} \qquad (2.3)$$

$$\frac{\tau_1 \sim \tau_2 \implies \tau \ \ \rho_1 \cup \rho_2 = \rho}{\&\rho_1\omega\tau_1 \sim \&\rho_2\omega\tau_2 \implies \&\rho\omega\tau} \qquad (2.4)$$

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 violated, 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!

13

the $\mathrm{Live}(R, p)$ **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 $\mathrm{Live}(v, p)$"

the $\mathrm{Live}(v, p)$ **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 2.6 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 $\mathrm{MayBeInitialised}(v, q)$ **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

2.6   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 2.4.

Listing 2.4: The custom deallocator for `OwnDrop` enforces the loan of `data`, but the loan in `OwnDrop` is effectively dead and thus can be violated.

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

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

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

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

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

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

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

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

2.7   Datalog and 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 [24]. 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 then use the two rules, reflecting the fact that ancestry is respectively either direct parenthood or transitive parenthood (example from the Wikipedia article on Datalog [25]):

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

Datafrog [5] is a minimalist Datalog implementation embedded in Rust, providing an implementation of a worst-case optimal join algorithm as described in [26]. 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 2.5: 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 3.2 becomes the code in Listing 2.5, and the corresponding join used for the first half of `region_live_at(R, P)` of Listing 3.4 can be seen in Listing 2.6.

Listing 2.6: 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 2.6 `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 2.5), 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 this thesis, I will use the notation of Soufflé [19] for all examples for clarity and brevity, even though the actual code was written in Datafrog.

Chapter 3

A Declarative Model for the Rust Borrow Checker

3.1 The Borrow Checker in Datalog

An overview of Polonius can be seen in Figure 3.1: initialisation is calculated in order to calculate drop-liveness, which together with regular liveness is used to determine the actual liveness of variables. The liveness of variables is then used to determine the liveness of provenance variables in their types, and is used throughout the calculations. Subset relations between provenances 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.

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.

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

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).
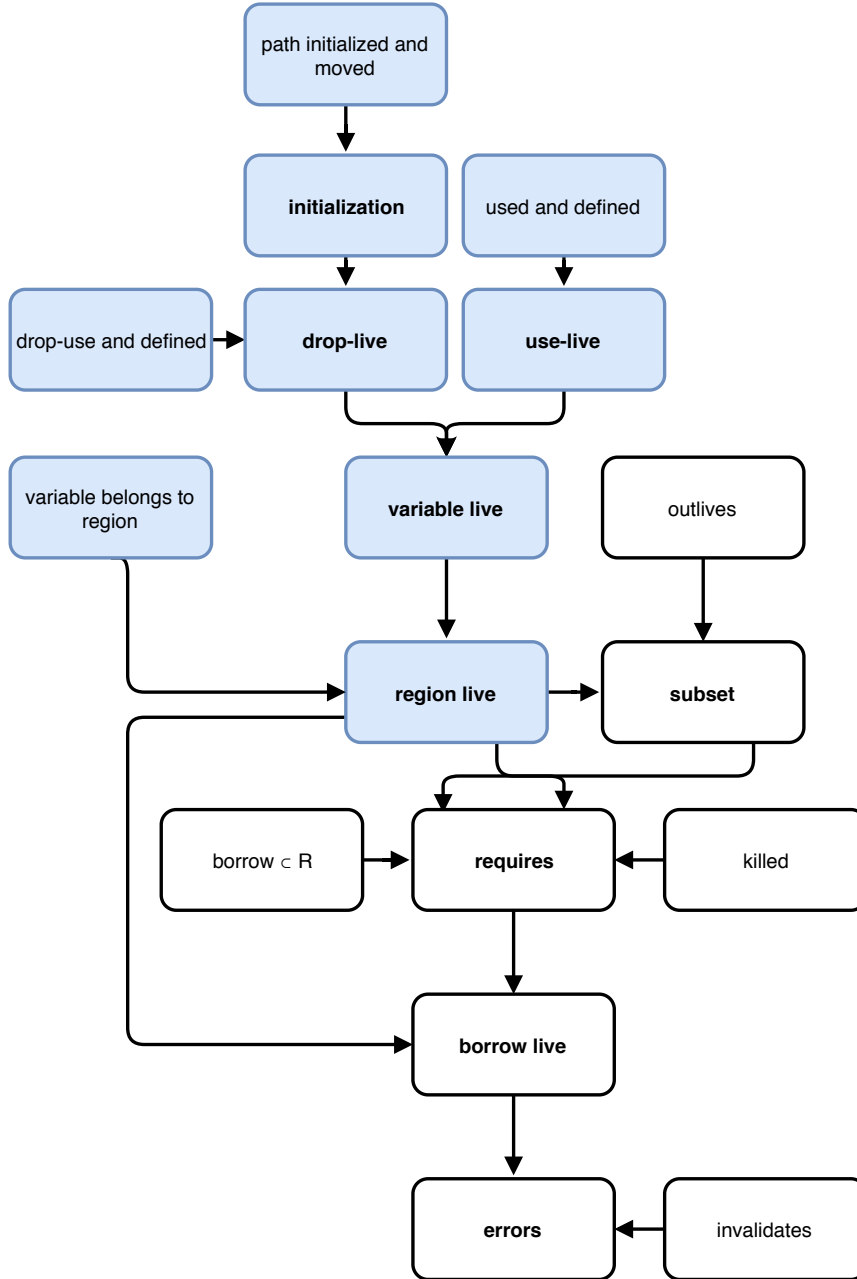
Figure 3.1: An overview of how the inputs and intermediate steps of Polonius combine into the final output. Blue boxes represent facts and relations implemented during the work on this thesis. Relations are shown using boldface, and facts in regular font.

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 $P \to Q$ in the control flow graph.

killed(L, P) when some prefix of the path 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 (**??**).

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 2.6.

child(M1, M2) when the move path M_1 is the child of M_2, That is, for example in the expression x.y.z, x.y.z is a child of x.y and x.

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

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

moved_out_at(M, P) when the move path M was moved out (deinitialised) at point P.

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

3.1.2 Variable Initialisation

The idea behind variable initialisation calculations is fairly straightforward. Initialisation for a leaf path propagates exactly like use-liveness, except forwards across the CFG. The path-specific difference is that initialisation also travels upwards through the path tree, such that x is (partially) initialised whenever x.y is. An expression like `move` x would, in this example, only emit `moved_out_at(x, p)` as a starting fact. Finding the transitive closure given by projecting the move across all of the path's children would happen in Polonius.

Finally, paths are traced back to their root variables across the `path_belongs_to_var(P, V)` fact. It is worth noting here that this fact only needs to contain a mapping of the root path to a variable, as initialisation always bubbles up through the tree, eventually reaching the root path. The full Datalog code is shown in Listing 3.1.

Listing 3.1: The rules for computing possible partial variable initialisation. A path is trivially initialised where it is actually initialised. It is transitively initialised in all points reachable from a point where it is initialised, and where it has not been deinitialised (moved out). Initialisation propagates upwards in the move path tree, until it reaches the variable at the root of the path.

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

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

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

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

3.1.3 Reference Liveness

The basic liveness of a variable (Listing 3.2) 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 3.2: The rules for calculating use-liveness: a variable is use-live if it was used at a point P, or if it was live in Q, there is a transition $P \to Q$, and it was not defined (killed) in P.

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

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

Listing 3.3: The rules for calculating drop-liveness: the rules are similar to those for to use-liveness (Listing 3.2), but propagation of liveness only happens if the variable being dropped may be initialised. Note that the rule for calculating initialisation on entry is not transitive!

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

Drop-liveness is calculated in a similar fashion to use-liveness, with the exception that a deinitialised variable is never dropped, and therefore is not considered drop-live. The rules can be found in Listing 3.3. Note the use of the first rule, which is not transitive, to shift the point of the relation from the input's mid-point index (which is where a (de)initialisation would take effect) to the statement's starting-point. The reason for this transformation is mainly technical.

An example of the output from this calculation can be seen in Figure 3.2.

Both of these are then used to calculate the reference liveness relation (Listing 3.4), 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 2.6) embedded in the `var_drops_region` relation.

Listing 3.4: 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).
```
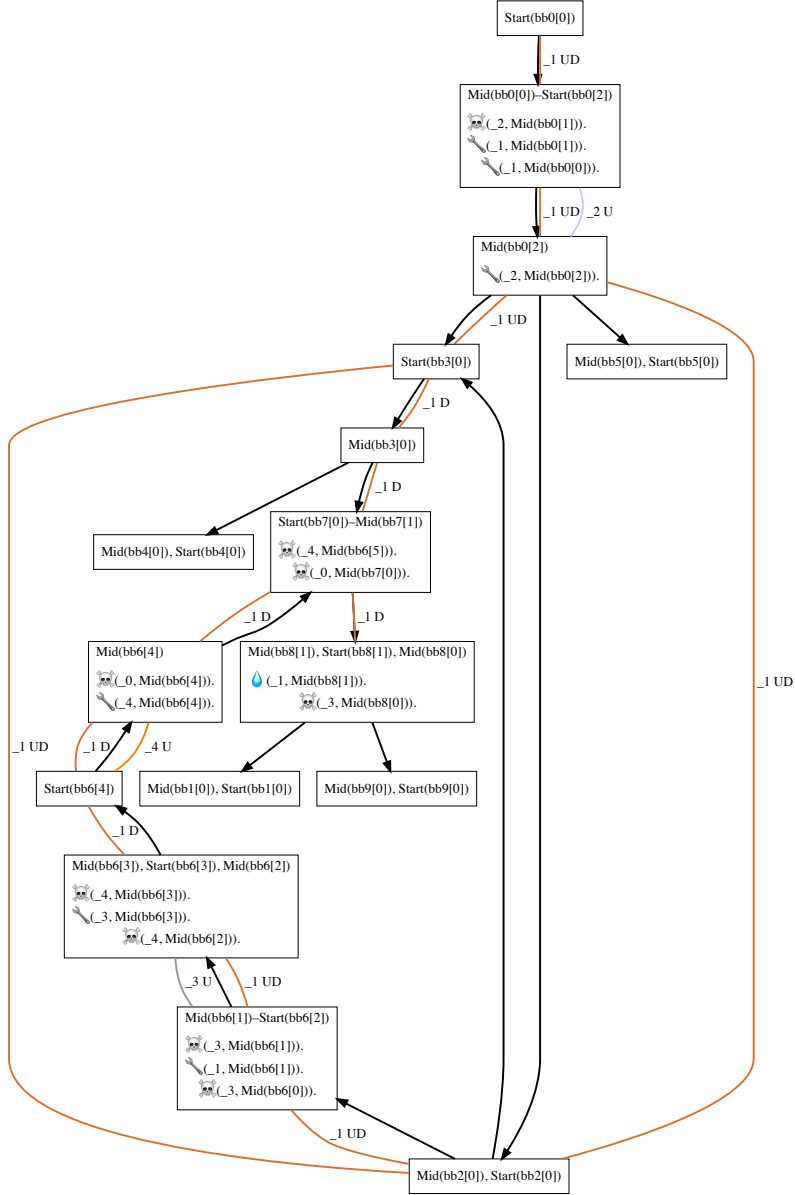
*Figure 3.2: A graph representation of the the variable liveness calculation results, with relevant Polonius facts as they occur (a droplet symbolising `var_drop_used`, a wrench `var_used`, and a skull and crossbones symbolising `var_defined`). Variables are named by prefixing underscores, and edges 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 3.5 captures this fact, providing a "base case" for the computation. Additionally, subset relations are transitive, which is captured in Listing 3.6.

Listing 3.5: Subset relations hold at the point where they are introduced.

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

Listing 3.6: 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 $P \to Q$ becomes the formulation seen in Listing 3.7, which notably uses the output of the liveness calculations described in Section 3.1.3.

Listing 3.7: 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 3.8, 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 3.8: 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 3.9.

Listing 3.9: 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 3.10, 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 3.10: 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 3.11), and see if any of those points intersect with an invalidation of that loan (Listing 3.12).

Listing 3.11: Loans are live when their provenance variables are.

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

Listing 3.12: It is an error to invalidate a live loan.

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

25

3.2 What is Missing from Polonius?

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

3.2.1 Detecting Access to Deinitialised Paths

The current Polonius implementation only uses move data to calculate variables that may have components that are initialised in order to determine if they might be accessed by a deallocation. However, the full borrow check would also calculate paths that *may have been moved out* and emit errors on access, such as in this code:

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

All the necessary input facts are already collected but the actual implementation and testing of the logic depends on a re-designed interface to Polonius that would allow it to report errors of different kinds, and was not performed as part of the work for this thesis.

3.2.2 Illegal Subset Relations

Polonius currently does not verify that the subset relationships 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. In this instance, the subtyping rules would give that `'y ⊆ '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 regions, 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 does not [27].

3.2.3 Analysis of Higher Kinds

The final missing functionality in Polonius is interaction with higher-ranked subtyping arising from generic functions or trait-matching. The problem was described in a blog entry by Matsakis [28] and will require extensions in the Rust compiler, which would produce simpler constraints for Polonius to solve.

3.2.4 Addressing Provenance Variable Imprecision

During the work for this thesis, a shortcoming in both Polonius and (probably) Weiss, Patterson, Matsakis, *et al.*'s formal formulation of the borrow check was discovered, which would generate spurious errors in examples like Listing 3.13 where 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 3.1.4. At the conclusion of the work for this theses, the Polonius working group had not yet produced a final reformulation of Polonius that would address this issue.

Listing 3.13: 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: 'y` subset relation
}

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

z += 1; // polonius flags an (unnecessary) error

drop(y);
```
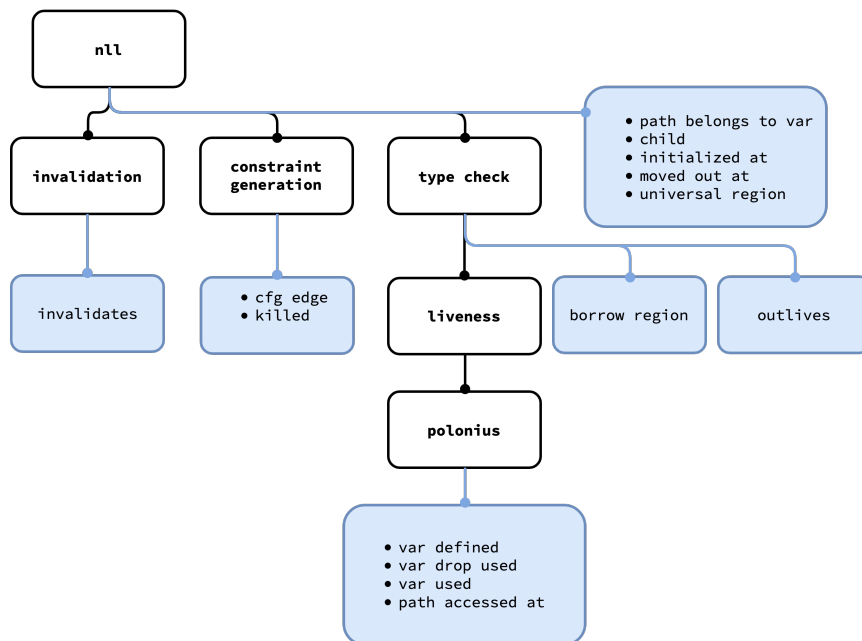
Figure 3.3: An illustration of where in the module hierarchy of the Rust compiler the various facts are emitted. Underscores are replaced with white space for readability. Blue boxes represent facts, and black boxes (sub-)modules.

3.3 Generating Facts for Polonius in the Rust Compiler

As stated above, the Polonius analysis is performed on the MIR, and the results are then mapped back onto the source code when generating user-facing errors. While Polonius is a self-contained package with a well-defined interface, however, the interface to the code performing the translation of compiler-internal data structures into input facts for Polonius has a much larger surface area. All the additions to the Rust compiler occurs in the `librustc_mir::borrow_check::nll` module, that is alongside the current borrow checker ("NLL"). The module hierarchy and the location of emission of the various facts is shown in Figure 3.3. It is worth noting that the Polonius analysis piggy-backs off of previous analysis, notably the `outlives` constraints generated by the previous borrow checker during type-checking.

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

All inputs based on provenance variables (that is, the ones with "region" in their names from the previous terminology); `path_belongs_to_var`, `universal_region`, `borrow_region`, `var_uses_region`, and `out-`

28

`lives`, 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 3.3 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 output by the type checker (i.e provenance variables and their relations to each other and to variables) under the `type_check` submodule.

Returning to one of the examples of the borrowing rules in Section 2.1, we can describe some of the facts that would be output on each line. An annotated example can be seen in Listing 3.14.

Listing 3.14: A minimal example of a violated loan in Rust and the Polonius input facts it would produce during compilation.

```rust
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.
```

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

3.4 A Field Study of Polonius Inputs

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

---

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

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 for 3 077 887 functions, 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 30 minutes. After this selection process, 10 613 repositories remained for the final study, each of which contained at least one, but possibly multiple crates. It is worth mentioning that 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 47, including functions generated by desugaring as well as user-written functions.

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

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.

Additionally, I also excluded all functions that had no loans at all from the analysis, a surprisingly large portion; almost 47%. This is most likely due to code generation producing short "functions" that does not actually involve any borrowing at all.

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 the files of tab-separated input tuples, 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.
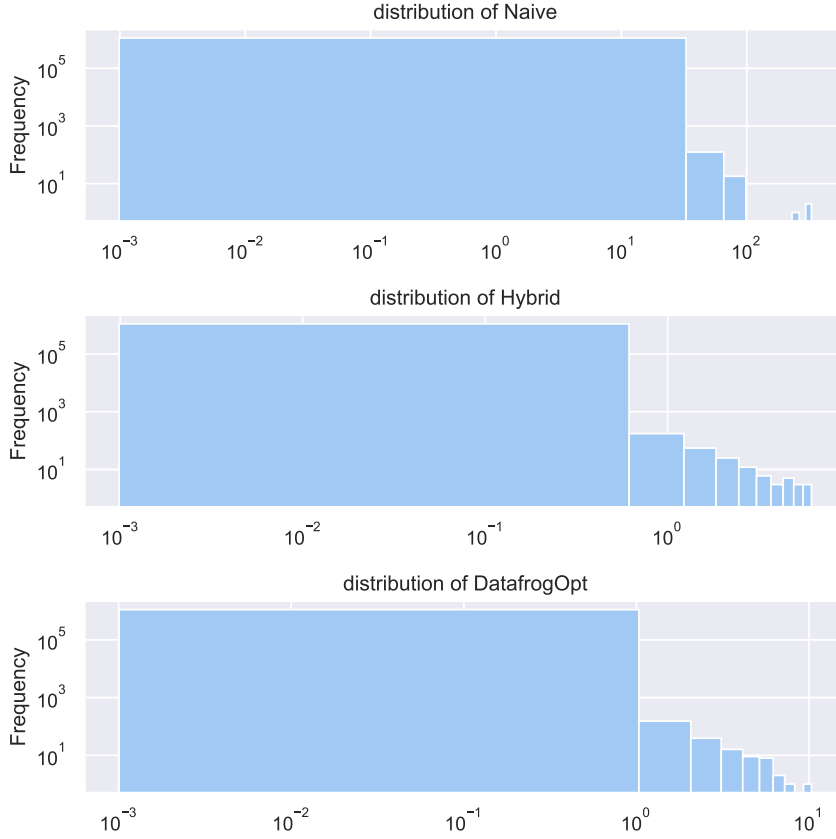
Figure 3.4: Distributions of solve-times per function for three implementations of Polonius. As can be seen here, the vast majority execute very quickly.

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.

Three variants of Polonius were included in the study; a NAIVE implementation, which is the one described in Section 3.1, 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.
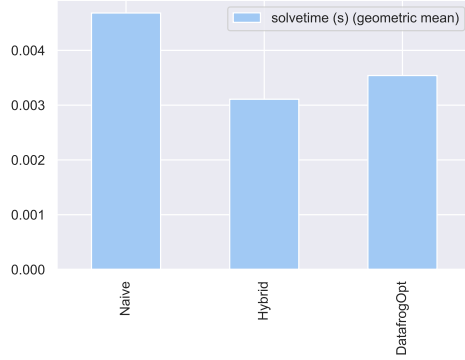
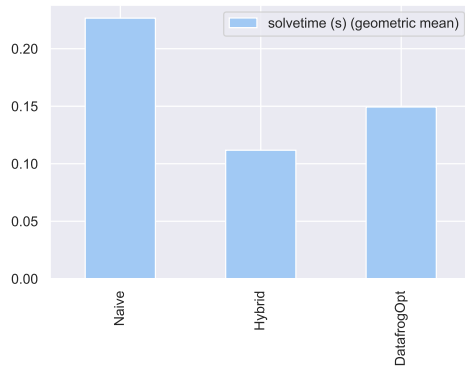*Figure 3.5: Geometric means of the solve-times per function and implementation.*

Figure 3.6: Geometric means of the solve-times per repositoriy and implementation.

3.4.1   Performance

In general, all three algorithms finished quickly for almost all functions, with both of the optimised algorithms already showing improvements in solve-times, as seen in Figure 3.4. Apparently, NAIVE has an even tighter spread of solve-times than the others. Additionally, geometric means of the observed solve-times show improvements from hybridisation (Figures 3.5 and 3.6), 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.
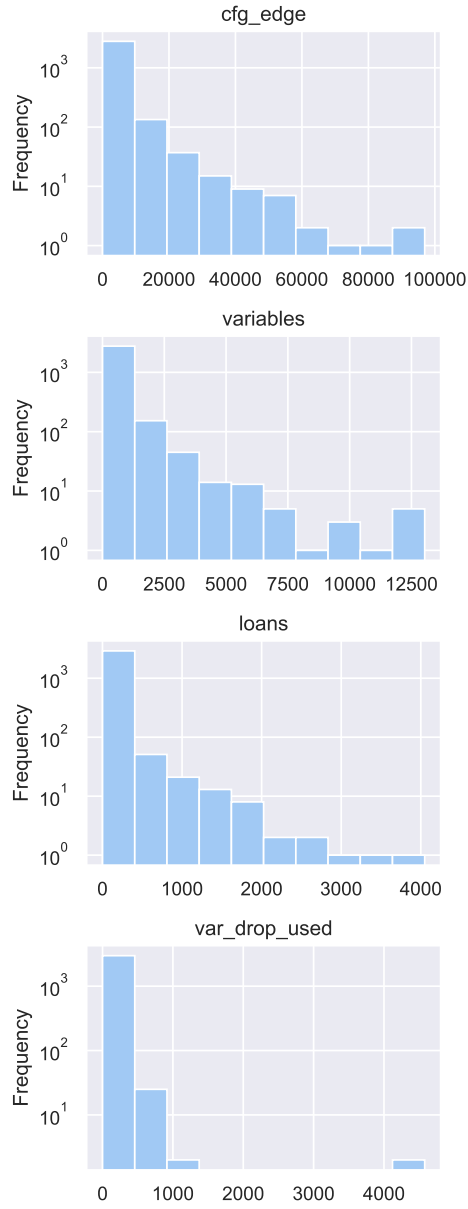
Figure 3.7: Distribution of the various input sizes.

3.4.2 What is a Typical Input?

A typical Polonius input consists of a small number of tuples for most relations, as seen in Figure 3.7. 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 71% of all studied functions having no (potential) drop-uses at all (0 median, 3.6 mean), and only very few loans (2 median, 5.4 mean).

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 solve-time would be dominated by any 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 solve-time 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 20 (regular uses was, for comparison, 174).

I 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 "typical".

3.4.3 How Inputs Affect Solve-Time

A heatmap of the (Pearson) correlation between input size and solve-time for the various variants can be seen in Figure 3.8, while a scatter plot of the results with a linear regression for some interesting pairs of inputs can be seen in Figure 3.9.

Both results suggest only a very weak linear relation between input sizes and and the solve-time with the naive algorithm, while a clearer relation can be found between the two optimisations and input sizes respectively. In particular, the number of loans and number of nodes in the control-flow graph seems to affect runtime performance, which is hardly surprising: the number of CFG nodes (along with the number of variables) would affect how many times constraint propagation would happen, and the number of loans and variables would affect the size of the propagations for the borrow check and liveness analyses respectively.
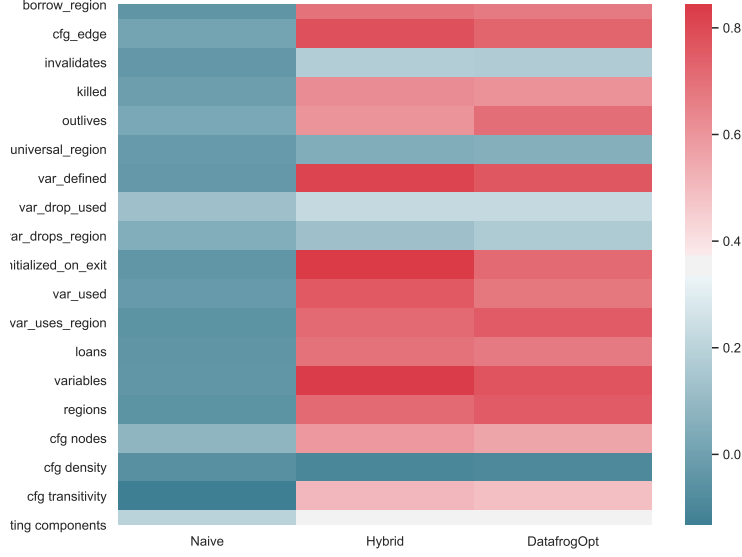
Figure 3.8: Heatmap of Pearson correlations between various input size metrics and solve-times for all three Polonius implementations, suggesting that in particular the number of variables and size of the CFG affect solve-time.
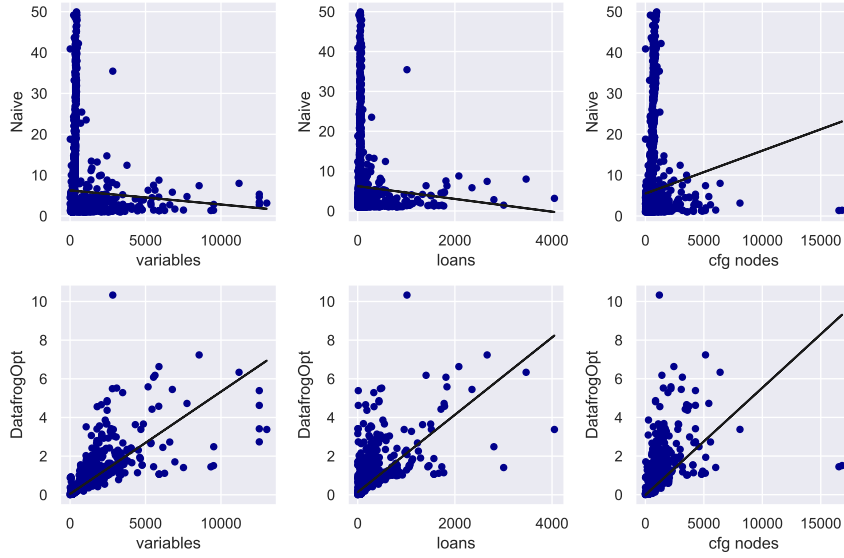

Figure 3.9: Scatter plot of solvetimes under the naive and optimised algorithms compared to variables and CFG edge count after having pruned extreme values (runtimes below 1 s or above 13 minutes).

3.5   Optimising the Borrow Checker

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

Finally, Datafrog itself could be optimised, including using faster vector instructions or parallelisation techniques.

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, 11).

[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, 15).

[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, 17).

[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, 11, 12, 27).

[21] *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. 8).

[22] Rustc developers. (). Guide to rustc development, [Online]. Available: https://rust-lang.github.io/rustc-guide/ (visited on 07/17/2019) (cit. on p. 9).

[23] *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. 9).

[24] 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. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0022000085710604 (visited on 03/29/2019) (cit. on p. 15).

[25] 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. 15).

[26] 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. 15).

[27] 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. 26).

[28] ——, (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. 27).