Albin Stjerna

A Dataflow Approach to Reference Ownership Analysis for the Rust Programming Language

This Page will be Replaced before Printing

Abstract

This thesis is about something.

Contents

1 Introduction 5

2 Background 6
- 2.1 The Borrowing Rules 6
- 2.2 The Borrow Check in the Rust Compiler 7
- 2.3 From Lifetimes to Reference Provenance 8
- 2.4 Deallocation As a Special Case of Variable Use 8
- 2.5 Datalog 10

3 A Datalog Model for the Rust Borrow Checker 11
- 3.1 The Borrow Checker in Datalog 11
  - 3.1.1 Starting Facts 11
  - 3.1.2 Reference Liveness 11
  - 3.1.3 Loan Constraint Propagation 12
  - 3.1.4 Loan Violations 12
  - 3.1.5 Illegal Subset Relations 12
- 3.2 A Field Study of Borrow Patterns 12
- 3.3 Optimising the Borrow Checker 12

4 Conclusions and Future Work 13

Bibliography 14

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.

mention results.

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 has been previously employed for program analysis, including in the Soufflé system, which is used to synthesise performant C++ from Datalog specifications, showing promising performance for analysis of large programs [6].

Formally, the semantics of Rust's lifetime rules has been captured in the language Oxide, described by Weiss, Patterson, Matsakis, *et al.* [7].

2.1 The Borrowing Rules

Most of these examples are borrowed from Weiss, Patterson, Matsakis, *et al.* [7].

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.

2.2 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. Describe what this means in terms of desugaring etc?

Describe Prefix Rules

2.3 From Lifetimes to Reference Provenance

As lifetimes are a part of a variable's types, they 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" [7] (RefProv) are all employed. As the section heading suggests, I will use the last one of them, as I believe it best captures the concept.

From a type system perspective, the RefProv is part of the type of any reference and corresponds to the borrow expressions that might have generated it. 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 Formula 2.1.

$$\frac{a \subseteq b}{\texttt{\&'a u32} <: \texttt{\&'b u32}} \tag{2.1}$$

How to incorporate the fact that this is with respect to the current point?

Finally, when talking about the *liveness* of a RefProv r, we will mean that r occurs (what does occurs mean? Does it encompass exploded subset-types etc?) in the type of a variable that is live at some point in the control-flow graph p, with the meaning 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.

2.4 Deallocation As a Special Case of Variable Use

describe the difference between drop-uses and normal uses and what is even a drop.

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 and performs deallocation, or, for complex objects, calls the `drop()` method. For some types of variables, such as integers, deallocation is not necessary and the compiler generates no actual `drop()`:s.

Rust provides a default deallocation method 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, a custom deallocator might access any of them. This means that any conditions of a loan that resulted in a reference r stored in a `struct` instance a must only be respected as far as `drop(a)` is concerned if s implements a custom deallocator. If it does not, 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.

```
#[derive(Debug)]
struct OwnDrop<'a> {
    data: &'a u32,
}

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

impl<'a> Drop for OwnDrop<'a> {
    fn drop(&mut self) {
        // the drop() method does something to the data
        println!("dropping, we had {:?}", 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;
    // Causes an error (if used): the loan of x must be respected...

    // ...but the loan of y need not be!
    let mutref = &mut y;
    *mutref = 17;
    println!("mutref {}", mutref);
```

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

2.5 Datalog

Describe datalog's syntax and execution model.

3. A Datalog Model for the Rust Borrow Checker

3.1 The Borrow Checker in Datalog

3.1.1 Starting Facts

Verify the terminology for this in Datalog parlance!

`borrow_region(R, B, P)` the region R may refer to data from borrow B starting at the point P (this is usually the point *after* a borrow rvalue).

`universal_region(R)` this is a "free region" within fn body

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

`killed(B,P)` when some prefix of the path borrowed at B is assigned at point P.

`outlives(R1, R2, P)` when we require 'R1@P: R2@P'

`invalidates(P, L)` when the loan L is invalidated 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 RefProv R.

`var_drops_region(V, R)` when the type of V includes the RefProf R, and V also implements a custom drop method which might need all of V's data, as discussed in Section 2.4.

3.1.2 Reference Liveness

The basic liveness of a variable 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.

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

There is also a very similar relation for drop-liveness with an identical shape and different input relations corresponding to drop-uses.

Both of these are then used to calculate the reference liveness relation, which serves as input for the rest of the borrow checker. A given RefProv 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 2.4.

```
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).
```

3.1.3 Loan Constraint Propagation

3.1.4 Loan Violations

3.1.5 Illegal Subset Relations

3.2 A Field Study of Borrow Patterns

3.3 Optimising the Borrow Checker

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. 5).

[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. 6).

[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. 6).

[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. 6, 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 p. 6).

[6] B. Scholz, H. Jordan, P. Subotić, and T. Westmann, "On fast large-scale program analysis in datalog," in *Proceedings of the 25th International Conference on Compiler Construction*, ACM, 2016, pp. 196–206 (cit. on p. 6).

[7] 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).