The Ownership Problem and its Algorithmic Complexity

Tom.Divers@bristol.ac.uk

1 Introduction

1.1 C to Rust Translation

[1]

1.2 The Borrow Checker

1.3 Problem Statement

2 A Language with Pointers and Mutation

We begin by defining a simple imperative language with references. This language is intended to encapsulate C's pointer system, without enforcing any of the memory safety rules enforced by Rust. Later on, we will define an alternate type system for this language which *does* enforce Rust's ownership and borrowing constraints.

2.1 Syntax and Types

We define our access paths, types and programs as follows (assuming the existence of some infinite set of variable identifiers V):

$$P ::= x \in V \mid *P \mid P.\mathbf{fst} \mid P.\mathbf{snd}$$

$$\tau, \sigma ::= \mathbf{Int} \mid \tau \times \sigma \mid \tau \to \sigma$$
$$M ::=$$

Observe that the following program typechecks in this language:

$$=$$

A C analogue of this program would compile without any errors. In Rust, however, the borrow checker complains about two problems that violate memory safety.

Initially, y and z both *borrow* x's value. Because y and z both both borrow x simultaneously, this borrow must be *immutable*. The first issue follows: Rust complains when the value pointed to by y is mutated. The second issue concerns y's dereference: when y is dereferenced, the value of x is *moved* into $\mathbf{deref}\ y$,[1] meaning that x is no longer owning at that point. Hence, the borrow checker complains when we try and move x when z is dereferenced.

Our problem is therefore to decide whether or not a program in this language makes the borrow checker complain. Obviously, this is a nebulous question, with a precise definition that we nail down in the next section. In order to do this, we introduce an alternate type system for our language, which enforces that a well-typed program satisfies some of Rust's borrow-checking constraints.

2.2 Semantics

3 Memory-safe Type Systems

3.1 Permissions as Fractional Grades

3.2 Our System

References

[1] Hanliang Zhang, Cristina David, Yijun Yu, and Meng Wang. "Ownership Guided C to Rust Translation". In: *Computer Aided Verification*. 2023, pp. 459–482. DOI: 10.1007/978-3-031-37709-9_22.

[1] We assume that the **Int** type we are using does not implement the `Clone` trait, meaning that the `.clone()` method is not called implicitly here.