The Ownership Problem

Tom.Divers@bristol.ac.uk

1 Introduction

1.1 C to Rust Translation

[3]

1.2 The Borrow Checker

1.3 Problem Statement

2 A Language with Pointers and Mutation

2.1 Syntax and Types

We begin by defining a simple STLC variant with references and mutation. 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.

Our syntax and types are given in BNF as follows (where $k \in \mathbb{Z}$ is an integer, and $x \in V$ is a variable):

$$\begin{aligned} M \ \ N ::= {} & () \mid k \mid M + N \mid \cdots \mid \\ & \lambda x.M \mid M\ N \mid (M, N) \mid \\ & \textbf{let } x \Leftarrow M \textbf{ then } N \mid \\ & \textbf{mut } x \Rightarrow M \textbf{ then } N \mid \\ & \textbf{ref } M \mid \textbf{deref } M \end{aligned}$$

$$\tau \ \ \sigma ::= \textbf{Unit} \mid \textbf{Int} \mid \tau \to \sigma \mid M \times N \mid \&\tau$$

We include a standard set of STLC typing judgements, as well as the following rules for references and mutation:

$$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \textbf{ref } M : \&\tau}\textsc{Ref}$$

$$\frac{\Gamma \vdash M : \&\tau}{\Gamma \vdash \textbf{deref } M : \tau}\textsc{Deref}$$

$$\frac{x : \tau \in \Gamma \quad \Gamma \vdash M : \tau \to \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash \textbf{mut } x \Rightarrow M \textbf{ then } N : \sigma}\textsc{Mutate}$$

Note that, for the sake of clarity, we will typically annotate function arguments with their intended types (e.g $\lambda x^{\sigma}.M$).

Observe that the following program (of type $\textbf{Int} \to \textbf{Int}$) typechecks in this language:

$$\begin{aligned} \mathit{Double} := \lambda x^{\textbf{Int}} . \ & \textbf{let } y \Leftarrow \textbf{ref } x \textbf{ then} \\ & \textbf{let } z \Leftarrow \textbf{ref } x \textbf{ then} \\ & (\textbf{deref } y) + (\textbf{deref } z) \end{aligned}$$

A C analogue of this program would compile without any errors. In Rust, however, this would be flagged by the borrow checker.

Initially, y and z both *borrow* x's value. Because y and z both both borrow x simultaneously, this borrow must be *immutable*. When y is dereferenced, the value of x is *moved* into **deref** y,[1] meaning that x is no longer owning at that point. Hence, when we try and move x into z, the borrow checker complains.

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

We now specify semantic rules for our language, sufficient to encapsulate the concept of mutation.

We define a heap $H : V\partial$

3 Memory-safe Type Systems

Plenty of literature exists on type systems that meaningfully abstract the notions of ownership and borrowing present in Rust [2, 1], which we take inspiration from.

The main issue with systems such as these (for our purposes) is that many of the decisions regarding ownership and borrowing are specified by the programmer at the syntactic level. In our context, we need these to be specified by the transpiler by encoding them solely at the type level.

3.1 Fractional Grades

3.2 Our System

Our new memory-safe types are given as follows (where $p \in (0, 1]_{\mathbb{Q}} \cup \{*\}$ is a permission grade):

$$\tau \ \ \sigma ::= \mathbf{Unit} \mid \mathbf{Int} \mid \tau \multimap \sigma \mid M \otimes N \mid \&_p \tau$$

We now introduce the following typing rules:

References

[1] Daniel Marshall and Dominic Orchard. "Functional Ownership through Fractional Uniqueness". In: *Proc. ACM Program. Lang.* 8.OOPSLA1 (Apr. 2024). DOI: 10.1145/3649848.

[2] Daniel Marshall, Michael Vollmer, and Dominic Orchard. "Linearity and Uniqueness: An Entente Cordiale". In: *Programming Languages and Systems*. 2022, pp. 346–375. DOI: 10.1007/978-3-030-99336-8_13.

[3] 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.