The Ownership Problem

Tom.Divers@bristol.ac.uk

1 Introduction

1.1 C to Rust Translation

[1]

1.2 The Borrow Checker

1.3 Problem Statement

2 Our Language and Type System(s)

2.1 Our C Analogue

We begin by defining a simple STLC variant with references and simple data structures. 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 is given in BNF as follows (where $k \in \mathbb{Z}$ is an integer, and $x \in V$ is a variable, and $f_i \in \mathcal{F} \forall i \in [n]$, where $\mathcal{F}$ is some set of **field identifiers**):

$$\begin{aligned} M \;\; N \;\; \{F_i\}_{i\in[n]} ::= \;& k \mid M + N \mid M \times N \mid \cdots \mid \\ & \lambda x.M \mid M\ N \mid \textbf{let } x \Leftarrow N \textbf{ in } M \\ & \textbf{ref } M \mid *M \mid \\ & \textbf{mkStruct } \{f_1 : F_1, \cdots, f_n : F_n\} \mid M.f \end{aligned}$$

Those types are given as follows:

$$\begin{aligned} \tau \;\; \sigma \;\; \{\gamma_i\}_{i\in[n]} ::= & \\ & \textbf{Int} \mid \tau \to \sigma \mid \&\tau \mid \textbf{struct } \{f_1 : \gamma_1, \cdots, f_n : \gamma_n\} \end{aligned}$$

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

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

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

$$\frac{\Gamma \vdash F_i : \gamma_i \quad \forall i \in [n]}{\Gamma \vdash \textbf{mkStruct } \{f_i : F_i\}_{i\in[n]} : \textbf{struct } \{f_i : \gamma_i\}_{i\in[n]}}\textsc{MkStruct}$$

$$\frac{\Gamma \vdash M : \textbf{struct } \{\cdots, f : \gamma, \cdots\}}{\Gamma \vdash M.f : \gamma}\textsc{Index}$$

As in C, we define $M \!\to\! f := (*M).f$. Also 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:

$$\begin{aligned} \mathit{Double} := \lambda x^{\textbf{Int}} \;.\; & \textbf{let } y \Leftarrow \textbf{ref } x \textbf{ in} \\ & \textbf{let } z \Leftarrow \textbf{ref } x \textbf{ in} \\ & *y + *z \end{aligned}$$

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 $*y$,[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 is difficult to nail down. 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 A Memory-safe Type 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.