The Ownership Problem and its Algorithmic Complexity

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

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.

2.1 Syntax and Types

We define a set of programs P, access paths A and types T using BNF as follows (where $k \in \mathbb{Z}$ is an integer, and $x \in V$ is a variable):

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

$$A \ni a ::= [\cdot] \mid *a \mid a.\textbf{fst} \mid a.\textbf{snd}$$

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

We include a standard set of STLC typing judgements, as well as the following rules for references, access paths, and type-preserving mutations:

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

$$\textsc{AccPathRoot}\frac{\Gamma \vdash x : \tau}{\Gamma \vdash [\cdot]x : \tau}$$

$$\textsc{PairAccFst}\frac{\Gamma \vdash x : \tau \times \sigma}{\Gamma \vdash x.\textbf{fst} : \tau}$$

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

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

Note that, for the sake of clarity, we will typically annotate function arguments with their intended types (e.g $\lambda x^{\sigma}.M$), but that these are not a part of our language. In other words, all the type signatures of our language are *inferred* (which will be of importance later).

Observe that the following program $Triple : \mathbf{Int} \to \mathbf{Int}$ typechecks under this type system:

$$\begin{aligned} Triple := \lambda x^{\mathbf{Int}} \,.\, & \mathbf{let}\ y \Leftarrow \mathbf{ref}\ x\ \mathbf{then} \\ & \mathbf{let}\ z \Leftarrow \mathbf{ref}\ x\ \mathbf{then} \\ & \mathbf{mut}\ *y \Rightarrow \lambda i^{\mathbf{Int}} \,.\, i + i\ \mathbf{then} \\ & *y + *z \end{aligned}$$

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 $*y$,[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.

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 Permissions as Fractional Grades

3.2 Our System

Our new memory-safe types are given as follows, where $p \in \{\mathcal{M}, \mathcal{I}\}$ is a **permission** and $l \in L$ is a **loan identifier** (used for the sake of avoiding lifetimes):

$$\begin{aligned} S \ni \tau, \sigma ::= & \mathbf{Int} \mid \tau \multimap \sigma \mid \tau \otimes \sigma \mid \\ & \mathbf{loan}^l_p \tau \mid \mathbf{borrow}^l_p \tau \end{aligned}$$

Informally, if a value is of type τ, then it is presumed to own its data, and it behaves linearly under any type derivation. When this value is borrowed, its type changes to $\mathbf{loan}^l_p \tau$. If it is borrowed mutably, then $p = \mathcal{M}$ (and vice versa). Any borrowed values are of type $\mathbf{borrow}^l_p \tau$.

We now introduce the following linear typing rules, which consist of a pre-context, the typing judgement of a term, and a post-context (written $\Gamma \vdash M : \tau \dashv \Delta$ for contexts Γ, Δ, type τ and program M):

$$\textsc{LinFunc}\frac{\Gamma, x : \tau \vdash M : \sigma \dashv \Delta}{\Gamma \vdash \lambda x.M : \tau \multimap \sigma \dashv \Delta}$$

$$\textsc{LinFuncApp}\frac{\Gamma \vdash M : \tau \multimap \sigma \dashv \Delta \quad \Delta \vdash N : \tau \dashv \Sigma}{\Gamma \vdash M\ N : \sigma \dashv \Sigma}$$

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.