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

$$\begin{aligned} P \ni M, N ::= {} & k \mid M + N \mid \cdots \mid \\ & \lambda x.M \mid M\ N \mid (M, N) \mid \mathbf{fst}\ M \mid \mathbf{snd}\ M \mid \\ & \mathbf{let}\ x \Leftarrow M\ \mathbf{then}\ N \mid \\ & \mathbf{mut}\ x \Leftarrow M\ \mathbf{then}\ N \mid \mathbf{mutAt}\ x \Leftarrow M\ \mathbf{then}\ N \mid \\ & \mathbf{ref}\ x \mid \mathbf{deref}\ M \end{aligned}$$

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

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

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

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

$$\frac{x : \tau \in \Gamma \quad \Gamma \vdash M : \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathbf{mut}\ x \Leftarrow M\ \mathbf{then}\ N : \sigma}\textsc{Mut}$$

$$\frac{\Gamma \vdash x : \&\tau \quad \Gamma \vdash M : \tau \quad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathbf{mutAt}\ x \Leftarrow M\ \mathbf{then}\ N : \sigma}\textsc{MutPtr}$$

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*.

Observe that the following program $\mathit{Double} : \mathbf{Int} \to \mathbf{Int}$ typechecks in this language:

$$\begin{aligned} \mathit{Double} := \lambda x^{\mathbf{Int}}\ .\ & \mathbf{let}\ y \Leftarrow \mathbf{ref}\ x\ \mathbf{then} \\ & \mathbf{let}\ z \Leftarrow \mathbf{ref}\ x\ \mathbf{then} \\ & \mathbf{mutAt}\ y \Leftarrow \lambda i^{\mathbf{Int}}\ .\ 1 + i\ \mathbf{then} \\ & \mathbf{deref}\ y + \mathbf{deref}\ z \end{aligned}$$

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 $\mathbf{deref}\ y$,[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 big-step semantic rules for our language, sufficient to encapsulate the concepts of mutation and referencing. Note that this is included for the sake of completeness, and is not our principal focus.

We define a heap $H : V \rightharpoonup \Omega$ to be a partial function from variables to a set of concrete values Ω, which is defined as follows:

- Ω contains integer literals (i.e. $\mathbb{Z} \subseteq \Omega$)
- Ω contains any function which has not been applied to all its arguments (i.e. a program $M \in \Omega$ if $M : \tau \to \sigma$ for any types τ and σ).
- All variable are also concrete values (i.e. for any $x \in V$, $\mathbf{ref}\ x \in \Omega$)

We also define the operation $\triangleright$, which overwrites the value of a variable in the heap:

$$H \triangleright x \mapsto v := (H \setminus \{x \mapsto z \mid z \in \Omega\}) \cup \{x \mapsto v\}$$

We define a big-step reduction $\rightsquigarrow \subseteq P \times \Omega$ over our programs P as a set of inference rules. Our judgements are of the form $H \vdash M \rightsquigarrow m$, where H is some heap context.

$$\frac{H(x) = v}{H \vdash x \rightsquigarrow v}\textsc{RedVar}$$

$$\frac{k \in \mathbb{Z}}{H \vdash k \rightsquigarrow k}\textsc{RedLit}$$

$$\frac{}{H \vdash \lambda x.M \rightsquigarrow \lambda x.M}\textsc{RedFunc}$$

$$\frac{H \vdash N \rightsquigarrow n \quad H \triangleright x \mapsto n \vdash M \rightsquigarrow m}{H \vdash (\lambda x.M)\ N \rightsquigarrow m}\textsc{RedFuncApp}$$

$$\frac{x \in V}{H \vdash \mathbf{ref}\ x \rightsquigarrow \&x}\textsc{RedRef}$$

$$\frac{H \vdash M \rightsquigarrow m \quad H(m) = \&n}{H \vdash \mathbf{deref}\ M \rightsquigarrow n}\textsc{RedDeref}$$

$$\frac{H \vdash M \rightsquigarrow m \quad H \triangleright x \mapsto m \vdash N \rightsquigarrow n}{H \vdash \mathbf{let}\ x \Leftarrow M\ \mathbf{then}\ N \rightsquigarrow n}\textsc{RedLet}$$

$$\frac{H \vdash M \rightsquigarrow m \quad H \triangleright x \mapsto m \vdash N \rightsquigarrow n}{H \vdash \mathbf{mut}\ x \Leftarrow M\ \mathbf{then}\ N \rightsquigarrow n}\textsc{RedMut}$$

$$\frac{H \vdash M \rightsquigarrow m \quad H(x) = \&y \quad H \triangleright y \mapsto m \vdash N \rightsquigarrow n}{H \vdash \mathbf{mutAt}\ x \Leftarrow M\ \mathbf{then}\ N \rightsquigarrow n}\textsc{RedMutPtr}$$

[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.

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 $p \in (0,1]_{\mathbb{Q}} \cup \{*, l\}$ is a permission):

$$T^* \ni \tau, \sigma ::= \mathbf{Int} \mid \tau \multimap \sigma \mid \tau \otimes \sigma \mid \&_p \tau$$

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.