Contents

1 The Case for Rust as a Target Language

Rust is split into two languages: safe and unsafe Rust. Like most type systems, Rust's type and ownership system is conservative, meaning any (safe) Rust program that type checks, will not crash due to memory safety or type errors. Unsafe Rust gives the programmer the ability to expand the programs accepted by Rust.

In this thesis, we will only consider safe Rust, which is the subset of Rust most programmers interact with (See 2).

Rust features a few unusual design decisions that profoundly influence the design of verification systems for it. The following paragraphs will elaborate on this.

Aliasing XOR Mutabillity The key behind Rusts type system is, that for every variable at every point during the execution, that variable is either aliased or mutable, but never both at the same time.

Rust accomplished this by introducing three types of access permissions for a variable, which are associated with every scope[1]:

1. The scope *owns* the variable. This guarantees, that no other scope has access to any part of the variable and allows the scope to create references to (part of) the variable.

2. The scope (immutably) *borrows* the variable. This guarantees, that as long as the variable is used, its *value will not change* and allows the scope to further lend the variable to other scoped.

[1] In modern Rust, this model was generalized to cover more cases, but the idea is still valid

3. The scope *mutably borrows* the variable. This guarantees, that there are no other active references to any part of the memory of the variable.

A consequence of these rules it, that at any time, every piece of memory has a unique and compile-time known owner. There are no reference cycles.

This makes both aliasing as well as mutability quite tame: If a variable is aliased, it must be immutable and as a result, represents just a value (like in pure functional languages). If a variable is mutable, it can not be aliased and as a result, any effect of the mutation is well known and locally visible.

Opaque Generics Rust does not provide a way to check a generic parameter for its instantiation. This means that we cannot extract any additional information from a generic parameter `T`. A function from `T` to `T` can therefor only be the identity function. Wadler [**wadler_theorems_1989**] shows, that it is possible to derive facts about the behaviour of such polymorphic functions.

In contrast, languages with instance-of-checks allow an implementation of a polymorphic function to distinguish between different instances of the generic parameter, precluding this extensive reasoning.

Explicit Mutable Access A consequence of the ownership rules is, that a function can only mutate (part of) the state, that is passed to it as a parameter. There is no global state and there is no implicit access to an object instance. Thus any intention of mutating state must already be expressed in the function signature, which makes the specification of this mutation quite natural.

2 Usage Of Rust

2.1 Unsafe Rust

3 Subset of Rust

Rust's main disadvantage as a target language it its size: There is a lot of syntax and semantics that would need to be accounted.Some even incidental to the verification. To reduce the complexity and amount of work, that needs to be done, we will focus on a subset of Rust described in this section.

The goal is to remove as much incidental complexity as possible without compromising to the central topic of research: How to extend LiquidTypes to mutability under the presence of Rust's ownership model.

3.1 Syntax

<table>
<tr><th></th><th></th><th></th><th></th></tr>
<tr><td>decl</td><td>::=</td><td>func_decl *</td><td>function declaration</td></tr>
<tr><td>func_decl</td><td>::=</td><td>ident(param *) -> ty { stmt * }</td><td></td></tr>
<tr><td>param</td><td>::=</td><td>ident : ty</td><td></td></tr>
<tr><td>stmt</td><td>::=</td><td>expr</td><td>expression</td></tr>
<tr><td></td><td>|</td><td>let mut? ident = expr</td><td>declaration</td></tr>
<tr><td></td><td>|</td><td>ident = expr</td><td>assignment</td></tr>
<tr><td></td><td>|</td><td>while (expr) { stmt *}</td><td>while loop</td></tr>
<tr><td></td><td>|</td><td>relax_ctx!{ pred* ; (ident : ty)* }</td><td>context relaxation</td></tr>
<tr><td>expr</td><td>::=</td><td>ident</td><td>variable reference</td></tr>
<tr><td></td><td>|</td><td>lit</td><td>constant</td></tr>
<tr><td></td><td>|</td><td>expr + expr</td><td>addition</td></tr>
<tr><td></td><td>|</td><td>ident(ident *)</td><td>function call</td></tr>
<tr><td></td><td>|</td><td>if expr { stmt * } else { stmt * }</td><td>if expression</td></tr>
<tr><td></td><td>|</td><td>* ident</td><td>dereference</td></tr>
<tr><td></td><td>|</td><td>& ident</td><td>immutable reference</td></tr>
<tr><td></td><td>|</td><td>&mut ident</td><td>mutable reference</td></tr>
<tr><td></td><td>|</td><td>expr as ty</td><td>type relaxation</td></tr>
<tr><td>ty</td><td>::=</td><td>ty!{ logic_ident : base_ty | pred }</td><td>refinement type</td></tr>
<tr><td>pred</td><td>::=</td><td>logic_ident</td><td>variable</td></tr>
<tr><td></td><td>|</td><td>pred && pred</td><td>conjunction</td></tr>
<tr><td></td><td>|</td><td>pred | | pred</td><td>disjunction</td></tr>
<tr><td></td><td>|</td><td>! pred</td><td>negation</td></tr>
</table>

3.2 Semantics

4 Definitions

P is the set of program variables used in rust program. Common names a, b, c

L is the set of logic variables used in refinement types. Common names: α, β

$\Gamma = (\mu, \sigma)$ is a tuple containing a function $\mu : P \to L$ mapping all program variables to their (current) logic variable and a set of formulas σ over L. During execution of statements, the set increases monotonically

τ is a user defined type $\{\alpha : b \mid \varphi\}$. Where α is a logic variable from L, b is a base type from Rust (like `i32`) and φ is a formula over variables in L.

5 Typing Rules

Abbreviations We write:

- Γ, c for $(\mu, \sigma \wedge c)$
- $\Gamma[a \mapsto \alpha]$ for $(\mu[a \mapsto \alpha], \sigma)$

5.1 Expression Typing: $\Gamma \vdash e : \tau$

$$\text{LIT}\;\frac{l \text{ fresh} \qquad \text{base\_ty}(v) = b}{\Gamma \vdash v : \{l : b \mid l \doteq v\} \Rightarrow \Gamma}$$

$$\text{VAR}\;\frac{\beta \text{ fresh} \qquad \mu(x) = \beta}{\Gamma = (\mu, \sigma) \vdash x : \{\alpha : b \mid \beta \doteq \alpha\} \Rightarrow \Gamma}$$

$$\text{VAR-REF}\;\frac{\Gamma \vdash y : \tau \qquad \Gamma \vdash x : \{\beta : \&b \mid \beta \doteq \&y\}}{\Gamma \vdash *x : \tau \Rightarrow \Gamma}$$

$$\text{IF}\;\frac{\Gamma \vdash c : \text{bool} \Rightarrow \Gamma_c \qquad \Gamma_c, c \vdash c_t : \tau \Rightarrow \Gamma' \qquad \Gamma_c, \neg c \vdash c_e : \tau \Rightarrow \Gamma'}{\Gamma \vdash \text{if } c \text{ then } c_t \text{ else } c_e : \tau \Rightarrow \Gamma'}$$

$$\text{WHILE}\;\frac{\Gamma_I, c \vdash s \Rightarrow \Gamma'_I \qquad \Gamma'_I \preceq \Gamma_I}{\Gamma_I \vdash \texttt{while}(c) s \Rightarrow \Gamma_I, \neg c}$$

$$\text{SEQ}\;\frac{\Gamma \vdash s_1 : \tau_1 \Rightarrow \Gamma' \qquad \Gamma' \vdash \bar{s} : \tau \Rightarrow \Gamma''}{\Gamma \vdash s_1; \bar{s} : \tau \Rightarrow \Gamma''}$$

$$\text{ADD}\;\frac{\Gamma \vdash e_1 : \{v_1 : b \mid \varphi_1\} \Rightarrow \Gamma' \qquad \Gamma' \vdash e_2 : \{v_2 : b \mid \varphi_2\} \Rightarrow \Gamma''}{\Gamma \vdash e_1 + e_2 : \{v : b \mid v \doteq [e_1] + [e_2]\} \Rightarrow \Gamma'', \varphi_1, \varphi_2}$$

$$\text{ASSIGN}\;\frac{\Gamma \vdash e : \{\beta \mid \varphi\} \Rightarrow \Gamma'}{\Gamma \vdash x = e : \tau \Rightarrow \Gamma'[x \mapsto \beta], \varphi}$$

$$\text{ASSIGN-STRONG}\;\frac{\Gamma \vdash e : \{\beta \mid \varphi\} \Rightarrow \Gamma' \qquad \Gamma \vdash x : \{\alpha : \&b \mid \alpha \doteq \&y\}}{\Gamma \vdash *x = e : \tau \Rightarrow \Gamma'[y \mapsto \beta], \varphi}$$

$$\text{ASSIGN-WEAK}\;\frac{\Gamma \vdash e : \tau \Rightarrow \Gamma' \qquad \Gamma \vdash x : \{\alpha : \&b \mid \alpha \doteq \&y \vee \alpha \doteq \&z \vee \dots\} \qquad \Gamma \vdash \tau \preceq \tau_y \qquad \Gamma \vdash y : \tau_y}{\Gamma \vdash *x = e : \tau \Rightarrow \Gamma'}$$

$$\text{FN-CALL}\;\frac{(\{a \mapsto \tau_a, \dots\}, \{\alpha \doteq \mu(a), \dots, \varphi_\alpha, \dots\}) \preceq \Gamma \qquad f : (\{\alpha \mid \varphi_\alpha\} \Rightarrow \{\alpha' \mid \varphi'_\alpha\}, \dots)}{\Gamma \vdash f(a, \dots) \Rightarrow (\mu[a \mapsto \alpha', \dots], \sigma \wedge \varphi'_\alpha \wedge \dots)}$$

$$\text{INTRO-SUB}\;\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau \preceq \tau'}{\Gamma \vdash e \;\texttt{as}\; \tau' : \tau'}$$

5.2 Sub-Typing Rules: $\Gamma \vdash \tau \preceq \tau'$

$$\preceq\text{-TY}\;\frac{\sigma \wedge \varphi'[\beta \triangleright \alpha] \vDash \varphi}{\Gamma = (\mu, \sigma) \vdash \{\alpha \mid \varphi\} \preceq \{\beta \mid \varphi'\}}$$

alternative (should be equivalent):

$$\preceq\text{-TY-ALT}\;\frac{\Gamma[f \mapsto \alpha], \varphi \preceq \Gamma[f \mapsto \beta], \varphi' \qquad f \text{ fresh}}{\Gamma \vdash \{\alpha \mid \varphi\} \preceq \{\beta \mid \varphi'\}}$$

5.3 Sub-Context Rules: $\Gamma \preceq \Gamma'$

$$\preceq\text{-CTX}\ \frac{\sigma'[\mu(\alpha) \triangleright \mu'(\alpha) \ \ \mid \alpha \in dom(\mu)] \vDash \sigma \qquad dom(\mu) \subseteq dom(\mu')}{(\sigma, \mu) \preceq (\sigma', \mu')}$$