Rusty Type Systems

Tom Divers

PLRG :: Bristol

12 November 2025

SEGMENTATION FAULT: (CORE DUMPED)

The plan:

- Rust and the Borrow Checker
- A toy language with some weird operational semantics
- (Some parts of) our type system
- An interesting type soundness statement

This work is in progress. Some of this might be slightly wrong!

Rust doesn't let you write code that does this (unless you ask it to let you).

Rust doesn't let you write code that does this (unless you ask it to let you).

Safe Rust ensures:

- No memory leaks
- No use-after-frees
- No race conditions

Rust doesn't let you write code that does this (unless you ask it to let you).

Safe Rust ensures:

- No memory leaks
- No use-after-frees
- No race conditions

How? *the borrow-checker*

[insert examples]

What I'm working on: A type system that can describe ownership and borrowing **(reasonably) concisely** in a higher-order setting.

Why: Denotational semantics, and (eventually) borrow inference algorithms.

Our Toy Language

$$\mathsf{Exp} \ni \mathbf{e}, \mathbf{f}, \mathbf{g} ::= n \in \mathbb{Z} \mid () \mid x \in \mathbb{V} \mid \mathbf{e}[\mathtt{OP}]\mathbf{f} \mid [\mathtt{OP}]\mathbf{f} \mid \mathsf{fix}\ \mathbf{e}$$
$$\lambda x.\mathbf{e} \mid \mathbf{e}\ \mathbf{f} \mid \mathsf{ifz}\ \mathbf{e}\ \mathsf{then}\ \mathbf{f}\ \mathsf{else}\ \mathbf{g} \mid \mathbf{let}\ x = \mathbf{e}; \mathbf{f}$$

Our Toy Language

$$\text{Exp} \ni \mathbf{e}, \mathbf{f}, \mathbf{g} ::= n \in \mathbb{Z} \mid () \mid x \in \mathbb{V} \mid \mathbf{e}[\texttt{OP}]\mathbf{f} \mid [\texttt{OP}]\mathbf{f} \mid \textsf{fix}\ \mathbf{e}$$

$$\lambda x.\mathbf{e} \mid \mathbf{e}\ \mathbf{f} \mid \textsf{ifz}\ \mathbf{e}\ \textsf{then}\ \mathbf{f}\ \textsf{else}\ \mathbf{g} \mid \mathbf{let}\ x = \mathbf{e}; \mathbf{f} \mid$$

$$\text{NEW STUFF:} \quad \mathbf{e}; \mathbf{f} \mid \ell \leftarrow \mathbf{e} \mid \&\ell \mid \mathcal{P}[\mathbf{e}]$$

$$\text{Path} \ni \mathcal{P} ::= \circ \mid *\mathcal{P}$$

$$\text{Loc} \ni \ell ::= \mathcal{P}[x]$$

Operational Semantics

Our language is **imperative**, so we need slightly fancier operational semantics than you've seen in ATiPL.

We define a relation $c_1 \longrightarrow c_2$, where c_1, c_2 are **configurations**.

Operational Semantics

Our language is **imperative**, so we need slightly fancier operational semantics than you've seen in ATiPL.

We define a relation $c_1 \longrightarrow c_2$, where c_1, c_2 are **configurations**. These look like $\langle \Omega \mid \mathbf{e} \mid \kappa \rangle$:

- $\mathbf{e} \in \mathsf{Exp}$
- Ω is a **heap** ($\Omega : \mathbb{V} \rightharpoonup_{\mathsf{fin}} \mathsf{Val} \times \{\mathbf{1}, \mathbf{w}, \mathbf{r}\}$)
- κ is a **continuation** (a list of commands)

Heaps

Ω tells us what each variable stores and how it can be used.

Heaps

Ω tells us what each variable stores and how it can be used.

- $x \mapsto^{\mathbf{1}} v \in \Omega$ means that x has **complete ownership** of v.

Heaps

Ω tells us what each variable stores and how it can be used.

- $x \mapsto^{\mathbf{1}} v \in \Omega$ means that x has **complete ownership** of v.
- $x \mapsto^{\mathbf{w}} v \in \Omega$ means that x has **read/write permissions** for v.

Heaps

Ω tells us what each variable stores and how it can be used.

- $x \mapsto^{\mathbf{1}} v \in \Omega$ means that x has **complete ownership** of v.
- $x \mapsto^{\mathbf{w}} v \in \Omega$ means that x has **read/write permissions** for v.
- $x \mapsto^{\mathbf{r}} v \in \Omega$ means that x has **read-only permissions** for v.

Heaps

Ω tells us what each variable stores and how it can be used.

- $x \mapsto^{\mathbf{1}} v \in \Omega$ means that x has **complete ownership** of v.
- $x \mapsto^{\mathbf{w}} v \in \Omega$ means that x has **read/write permissions** for v.
- $x \mapsto^{\mathbf{r}} v \in \Omega$ means that x has **read-only permissions** for v.

We define two operations on heaps $\otimes_H, \gg_H$: Heap $\times$ Heap $\rightharpoonup$ Heap.

Heaps

Ω tells us what each variable stores and how it can be used.

- $x \mapsto^{\mathbf{1}} v \in \Omega$ means that x has **complete ownership** of v.
- $x \mapsto^{\mathbf{w}} v \in \Omega$ means that x has **read/write permissions** for v.
- $x \mapsto^{\mathbf{r}} v \in \Omega$ means that x has **read-only permissions** for v.

We define two operations on heaps $\otimes_H, \gg_H$: Heap $\times$ Heap $\rightharpoonup$ Heap.

- $\Omega = \Omega_1 \otimes_H \Omega_2$ means that Ω can be split into two heaps Ω_1, Ω_2 that can exist **at the same time**.

Heaps

Ω tells us what each variable stores and how it can be used.

- $x \mapsto^{\mathbf{1}} v \in \Omega$ means that x has **complete ownership** of v.
- $x \mapsto^{\mathbf{w}} v \in \Omega$ means that x has **read/write permissions** for v.
- $x \mapsto^{\mathbf{r}} v \in \Omega$ means that x has **read-only permissions** for v.

We define two operations on heaps $\otimes_H, \gg_H$: Heap $\times$ Heap $\rightharpoonup$ Heap.

- $\Omega = \Omega_1 \otimes_H \Omega_2$ means that Ω can be split into two heaps Ω_1, Ω_2 that can exist **at the same time**.
- $\Omega = \Omega_1 \gg_H \Omega_2$ means that Ω can be split into two heaps Ω_1, Ω_2 that can exist **one after the other**.

Let's see how the program $\mathbf{let}\ x = v; \mathbf{let}\ y = \&x; x$ ($v \in \mathsf{Val}$) executes.

Let's see how the program $\mathbf{let}\ x = v; \mathbf{let}\ y = \&x; x$ ($v \in \mathsf{Val}$) executes.

$$\begin{aligned} &\langle \emptyset \mid \mathbf{let}\ x = v; \mathbf{let}\ y = \&x; x \mid \varepsilon \rangle \\ &\longrightarrow \langle \emptyset \mid v \mid \mathsf{bind}_{\emptyset}(x.\mathbf{let}\ y = \&x; x) \rangle \\ &\longrightarrow \langle \{x \mapsto^{\mathbf{1}} v\} \mid \mathbf{let}\ y = \&x; x \mid \varepsilon \rangle \end{aligned}$$

Let's see how the program $\mathbf{let}\ x = v; \mathbf{let}\ y = \&x; x$ ($v \in \mathsf{Val}$) executes.

$$\begin{aligned} &\langle \emptyset \mid \mathbf{let}\ x = v; \mathbf{let}\ y = \&x; x \mid \varepsilon \rangle \\ &\longrightarrow \langle \emptyset \mid v \mid \mathsf{bind}_{\emptyset}(x.\mathbf{let}\ y = \&x; x) \rangle \\ &\longrightarrow \langle \{x \mapsto^{\mathbf{1}} v\} \mid \mathbf{let}\ y = \&x; x \mid \varepsilon \rangle \end{aligned}$$

Nondeterministically, we observe that $\Omega := x \mapsto^{\mathbf{1}} v = x \mapsto^{\mathbf{r}} v \gg_H \Omega$.

Let's see how the program $\textbf{let}\ x = v; \textbf{let}\ y = \&x; x$ ($v \in \mathsf{Val}$) executes.

$$
\begin{aligned}
&\langle \emptyset \mid \textbf{let}\ x = v; \textbf{let}\ y = \&x; x \mid \varepsilon \rangle \\
&\longrightarrow \langle \emptyset \mid v \mid \mathsf{bind}_{\emptyset}(x.\textbf{let}\ y = \&x; x) \rangle \\
&\longrightarrow \langle \{x \mapsto^{\mathbf{1}} v\} \mid \textbf{let}\ y = \&x; x \mid \varepsilon \rangle
\end{aligned}
$$

Nondeterministically, we observe that $\Omega := x \mapsto^{\mathbf{1}} v = x \mapsto^{\mathbf{r}} v \gg_H \Omega$.

$$
\begin{aligned}
&\langle \{x \mapsto^{\mathbf{r}} v\} \gg_H \Omega \mid \textbf{let}\ y = \&x; x \mid \varepsilon \rangle \\
&\longrightarrow \langle \{x \mapsto^{\mathbf{r}} v\} \mid \&x \mid \mathsf{bind}_{\Omega}(y.x) \rangle \\
&\longrightarrow \langle \Omega \otimes \{y \mapsto^{\mathbf{1}} \&x\} \mid x \mid \varepsilon \rangle \\
&\longrightarrow \langle \Omega \mid x \mid \varepsilon \rangle \\
&\longrightarrow \langle \emptyset \mid v \mid \varepsilon \rangle
\end{aligned}
$$

Let's see how the program $\textbf{let } x = v; \textbf{let } y = \&x; x$ ($v \in \mathsf{Val}$) executes.

$$\begin{aligned} &\langle \emptyset \mid \textbf{let } x = v; \textbf{let } y = \&x; x \mid \varepsilon \rangle \\ &\longrightarrow \langle \emptyset \mid v \mid \mathsf{bind}_{\emptyset}(x.\textbf{let } y = \&x; x) \rangle \\ &\longrightarrow \langle \{x \mapsto^{\mathbf{1}} v\} \mid \textbf{let } y = \&x; x \mid \varepsilon \rangle \end{aligned}$$

Nondeterministically, we observe that $\Omega := x \mapsto^{\mathbf{1}} v = x \mapsto^{\mathbf{r}} v \gg_H \Omega$.

$$\begin{aligned} &\langle \{x \mapsto^{\mathbf{r}} v\} \gg_H \Omega \mid \textbf{let } y = \&x; x \mid \varepsilon \rangle \\ &\longrightarrow \langle \{x \mapsto^{\mathbf{r}} v\} \mid \&x \mid \mathsf{bind}_{\Omega}(y.x) \rangle \\ &\longrightarrow \langle \Omega \otimes \{y \mapsto^{\mathbf{1}} \&x\} \mid x \mid \varepsilon \rangle \\ &\longrightarrow \langle \Omega \mid x \mid \varepsilon \rangle \\ &\longrightarrow \langle \emptyset \mid v \mid \varepsilon \rangle \end{aligned}$$

These semantics are **nondeterministic**. We're using nondeterminism to decide how to split up our permissions.

Types

Ty is defined below (excluding lifetimes and nonrecursive data structures; contexts are TCtx):

$$\mathsf{Ty} \ni \tau, \sigma ::= \ () \mid \mathsf{Int} \mid \tau \longrightarrow^{\Gamma} \sigma \mid \&_{\mathbf{w}}\tau \mid \&_{\mathbf{r}}\tau$$

- $\tau \longrightarrow^{\Gamma} \sigma$ is the type of *closures* (functions from τ to σ that depend on free variables in $\Gamma \in \mathsf{TCtx}$)
- $\&_{\mathbf{w}}\tau$ is a **mutable (read-write) reference** to a τ
- $\&_{\mathbf{r}}\tau$ is an **immutable (read-only) reference** to a τ

Typing Contexts

Remember how our heaps needed to track permissions.

[1] Associative, total binary operations; associativity means that $a \circ (b \circ c) = (a \circ b) \circ c$.

Typing Contexts

Remember how our heaps needed to track permissions. **So do contexts!**

Our contexts are of the form $\Gamma : \mathbb{V} \rightharpoonup_{\mathsf{fin}} \mathsf{Ty} \times \Theta$, where $\Theta := \{\boldsymbol{\omega}, \mathbf{1}, \mathbf{w}, \mathbf{r}, \mathbf{0}\}$.

Variables in our contexts are annotated with elements of Θ: e.g. $x :^{\mathbf{0}} \tau \in \Gamma$

[1] Associative, total binary operations; associativity means that $a \circ (b \circ c) = (a \circ b) \circ c$.

Typing Contexts

Remember how our heaps needed to track permissions. **So do contexts!**

Our contexts are of the form $\Gamma : \mathbb{V} \rightharpoonup_{\mathsf{fin}} \mathsf{Ty} \times \Theta$, where $\Theta := \{\boldsymbol{\omega}, \mathbf{1}, \mathbf{w}, \mathbf{r}, \mathbf{0}\}$.

Variables in our contexts are annotated with elements of Θ: e.g. $x :^{\mathbf{0}} \tau \in \Gamma$

We define two **monoids**[1] $\otimes_\Theta, \gg_\Theta : \Theta \times \Theta \to \Theta$.

These are lifted to $\otimes_C, \gg_C$: $\mathsf{TCtx} \times \mathsf{TCtx} \rightharpoonup \mathsf{TCtx}$; our derivations are built around these monoids.

[1] Associative, total binary operations; associativity means that $a \circ (b \circ c) = (a \circ b) \circ c$.

Typing Contexts

Remember how our heaps needed to track permissions. **So do contexts!**

Our contexts are of the form $\Gamma : \mathbb{V} \rightharpoonup_{\mathsf{fin}} \mathsf{Ty} \times \Theta$, where $\Theta := \{\boldsymbol{\omega}, \mathbf{1}, \mathbf{w}, \mathbf{r}, \mathbf{0}\}$.

Variables in our contexts are annotated with elements of Θ: e.g. $x :^{\mathbf{0}} \tau \in \Gamma$

We define two **monoids**[1] $\otimes_\Theta, \gg_\Theta$: $\Theta \times \Theta \to \Theta$.

These are lifted to $\otimes_C, \gg_C$: $\mathsf{TCtx} \times \mathsf{TCtx} \rightharpoonup \mathsf{TCtx}$; our derivations are built around these monoids.

We enforce a set of **consistency conditions** on our contexts.

[1]Associative, total binary operations; associativity means that $a \circ (b \circ c) = (a \circ b) \circ c$.

Typing Contexts

Remember how our heaps needed to track permissions. **So do contexts!**

Our contexts are of the form $\Gamma : \mathbb{V} \rightharpoonup_{\mathsf{fin}} \mathsf{Ty} \times \Theta$, where $\Theta := \{\boldsymbol{\omega}, \mathbf{1}, \mathbf{w}, \mathbf{r}, \mathbf{0}\}$.

Variables in our contexts are annotated with elements of Θ: e.g. $x :^{\mathbf{0}} \tau \in \Gamma$

We define two **monoids**[1] $\otimes_\Theta, \gg_\Theta$: $\Theta \times \Theta \to \Theta$.

These are lifted to $\otimes_C, \gg_C$: $\mathsf{TCtx} \times \mathsf{TCtx} \rightharpoonup \mathsf{TCtx}$; our derivations are built around these monoids.

We enforce a set of **consistency conditions** on our contexts.
We write $\Omega \vDash \Gamma$ if Ω and Γ have the same structure.

---

[1] Associative, total binary operations; associativity means that $a \circ (b \circ c) = (a \circ b) \circ c$.

Typing Rules

Typing Rules

$$\textsc{Let}\ \frac{\Gamma_1 \vdash \mathbf{e}_1 : \tau \quad \Gamma_2, x :^{\mathbf{1}} \tau \vdash \mathbf{e}_2 : \sigma}{\Gamma_1 \gg \Gamma_2 \vdash \mathbf{let}\ x = \mathbf{e}_1 ; \mathbf{e}_2 : \sigma}$$

Typing Rules

$$\textsc{Let}\ \frac{\Gamma_1 \vdash \mathbf{e}_1 : \tau \quad \Gamma_2, x :^{\mathbf{1}} \tau \vdash \mathbf{e}_2 : \sigma}{\Gamma_1 \gg \Gamma_2 \vdash \mathbf{let}\ x = \mathbf{e}_1; \mathbf{e}_2 : \sigma}$$

$$\textsc{App}\ \frac{\Gamma_1 \vdash \mathbf{e}_1 : \&_p\tau \longrightarrow^{\Delta} \sigma \quad \Gamma_2 \vdash \mathbf{e}_2 : \&_p\tau}{\Gamma_1 \otimes \Gamma_2 \vdash \mathbf{e}_1\ \mathbf{e}_2 : \sigma}$$

We also give typing rules for our continuations:

$$\textsc{Bind}\ \frac{\Gamma, x : \sigma \vdash \mathbf{e} : \tau \quad \Omega \vDash \Gamma}{\mathsf{bind}_{\Omega}(x.\mathbf{e}) : \sigma \Longrightarrow^{\Gamma} \tau}$$

We also give typing rules for our continuations:

$$\textsc{Bind}\;\frac{\Gamma, x : \sigma \vdash \mathbf{e} : \tau \quad \Omega \vDash \Gamma}{\mathsf{bind}_{\Omega}(x.\mathbf{e}) : \sigma \Longrightarrow^{\Gamma} \tau}$$

This leads really nicely into a single rule for typing configurations:

$$\frac{\Gamma_1 \vdash \mathbf{e} : \sigma \quad \kappa : \sigma \Longrightarrow^{\Gamma_2} \tau \quad \Omega \vDash \Gamma_1 \quad \exists \Gamma = \Gamma_1 \gg \Gamma_2}{\langle \Omega \mid \mathbf{e} \mid \kappa \rangle : \tau}$$

Type Soundness

In ATiPL, we split type soundness into two statements:

- **Progress:** Well-typed terms step to something
- **Preservation:** Stepping to something leaves our types alone

Type Soundness

In ATiPL, we split type soundness into two statements:

- **Progress:** Well-typed terms step to something
- **Preservation:** Stepping to something leaves our types alone

Problem: Our semantics 'decide' typing information nondeterministically! So not every reduction step is well-typed!

Type Soundness

In ATiPL, we split type soundness into two statements:

- **Progress:** Well-typed terms step to something
- **Preservation:** Stepping to something leaves our types alone

Problem: Our semantics 'decide' typing information nondeterministically! So not every reduction step is well-typed!

How do we fix this?

Type Soundness

In ATiPL, we split type soundness into two statements:

- **Progress:** Well-typed terms step to something
- **Preservation:** Stepping to something leaves our types alone

Problem: Our semantics 'decide' typing information nondeterministically! So not every reduction step is well-typed!

How do we fix this? **We smoosh progress and preservation together!**

Type Soundness

In ATiPL, we split type soundness into two statements:

- **Progress:** Well-typed terms step to something
- **Preservation:** Stepping to something leaves our types alone

Problem: Our semantics 'decide' typing information nondeterministically!
So not every reduction step is well-typed!

How do we fix this? **We smoosh progress and preservation together!**

Theorem (Soundness)

If $\langle \Omega \mid \mathbf{e} \mid \kappa \rangle : \tau$*, then one of the following holds:*

1. $\exists \langle \Omega' \mid \mathbf{e}' \mid \kappa' \rangle : \tau$ *such that* $\langle \Omega \mid \mathbf{e} \mid \kappa \rangle \longrightarrow \langle \Omega' \mid \mathbf{e}' \mid \kappa' \rangle$
2. $\mathbf{e} \in \mathsf{Val}$ *and* κ *is empty*

Next steps:

- Denotational semantics ($\gg$ and $\otimes$ are intriguing)
- Type inference

Next steps:

- Denotational semantics ($\gg$ and $\otimes$ are intriguing)
- Type inference

Just a couple of closing remarks:

- This work is under review.
- I'll put what I've submitted on the unit Teams, but plz do not distribute.
- If you want to do research like this (or like what Charlie showed you), talk to Meng Wang.
- See you all tomorrow at 11!