On the Termination of Borrow Checking in Featherweight Rust

Étienne Payet[1] David J. Pearce[2] Fausto Spoto[3]

LIM, Université de La Réunion, France

Victoria University of Wellington, New Zealand

Dipartimento di Informatica, Università di Verona, Italy

@WhileyDave

Rust: History

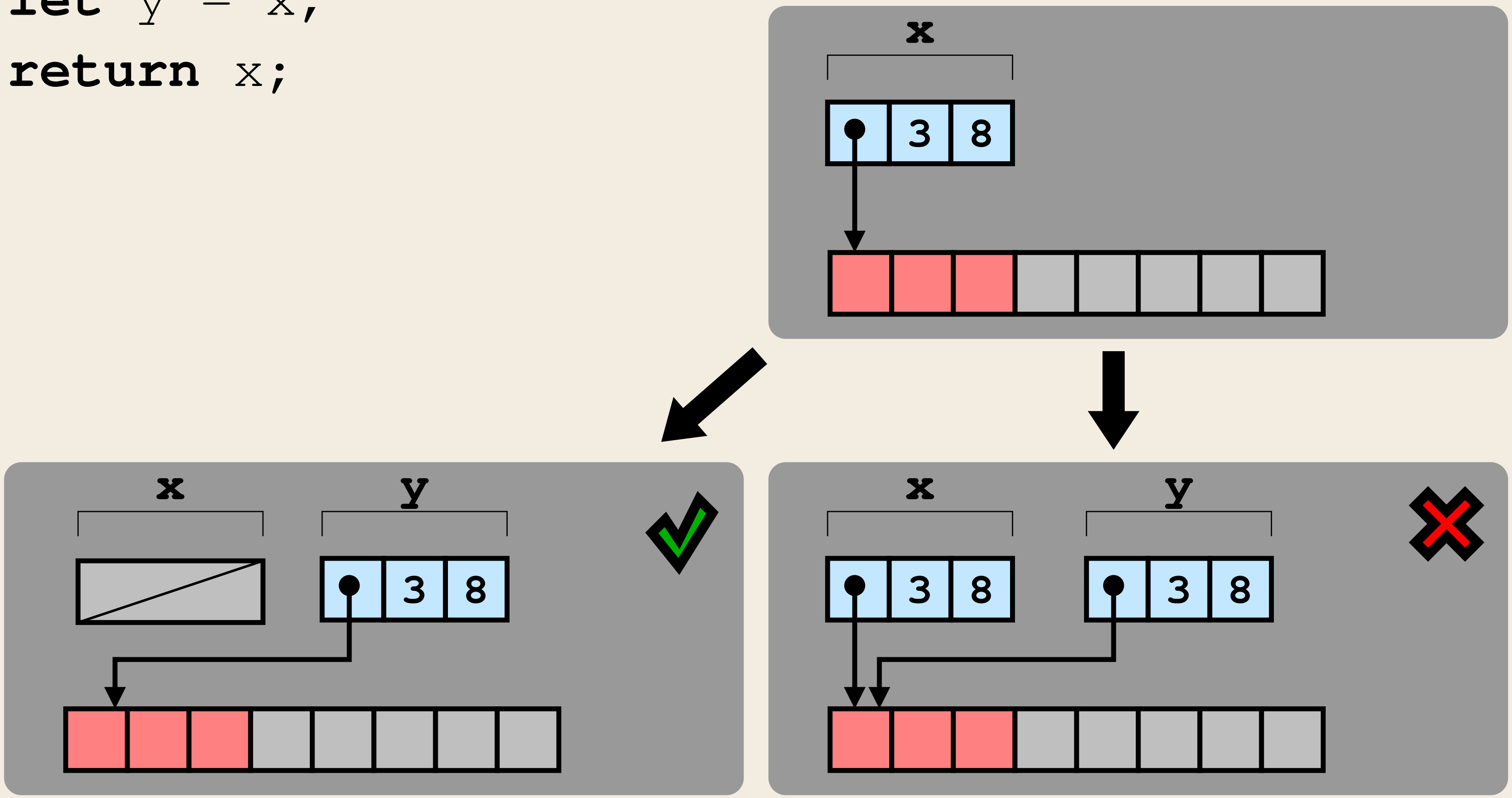
> *"A* **systems** *programming language that runs* **blazingly fast***, prevents* **segmentation faults***, and guarantees* **thread safety***"*
>
> `rust-lang.org`

> *"I made a prototype, then my employer threw millions of dollars at it and hired dozens of researchers and programmers (and tireless interns, hi!) and a giant community of thousands of volunteers showed up and* **then** *the book arrived."*
>
> *–Graydon Hoare, 2018*

- Designed by **Graydon Hoare** at Mozilla around 2006
- Automatic memory management **without** garbage collection
- Influenced by **Cyclone** and C++ **smart pointers**, amongst others

Rust: Ownership

```
fn f(x: Vec<i32>) -> Vec<i32> {
    let y = x;
    return x;
}
```



Rust: Borrowing

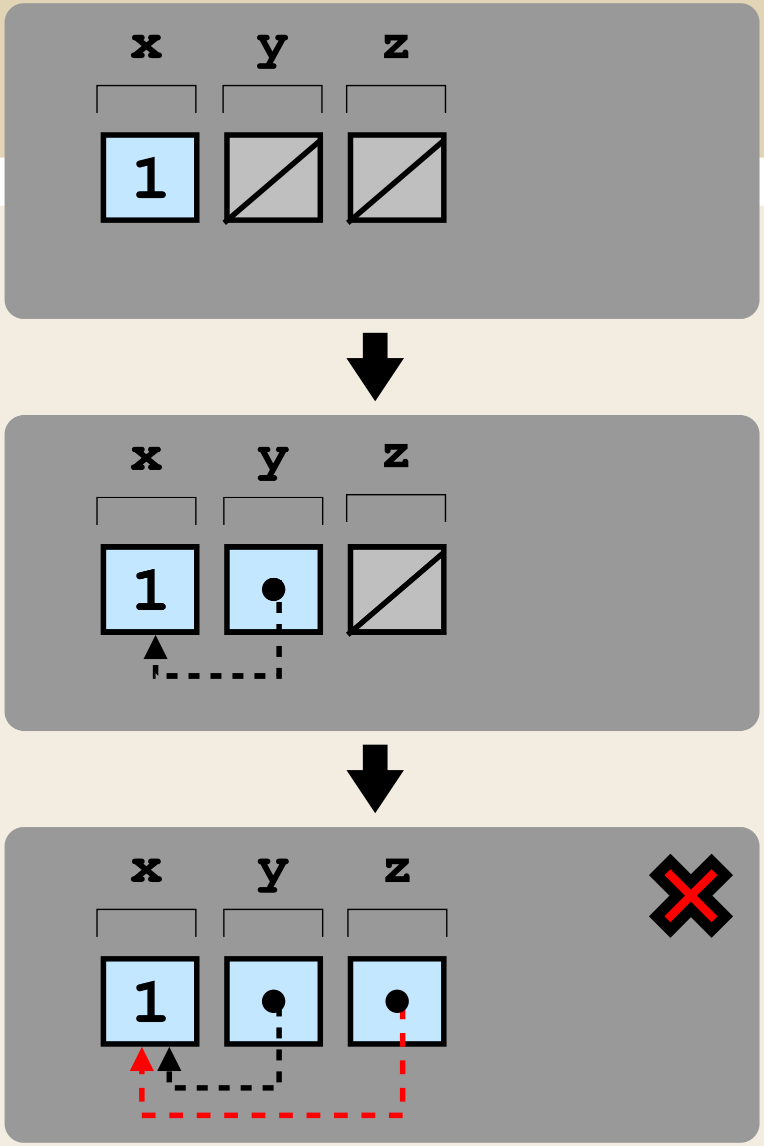
```
fn is_nat(x : &i32) -> bool {
  if *x >= 0 { return true; }
  else { return false;}
}

fn f() -> (i32,bool) {
  let x = 0;
  let y = is_nat(&x);
  return (x,y);
}
```

- Borrowing enables **controlled breakages** of ownership invariant
- Borrowing give access without **responsibility** for memory management

Rust: Borrow Checking

```
fn f() -> i32 {
   let mut x = 1;
   let y = &x;
   let z = &mut x;
   return x + *y + *z;
}
```

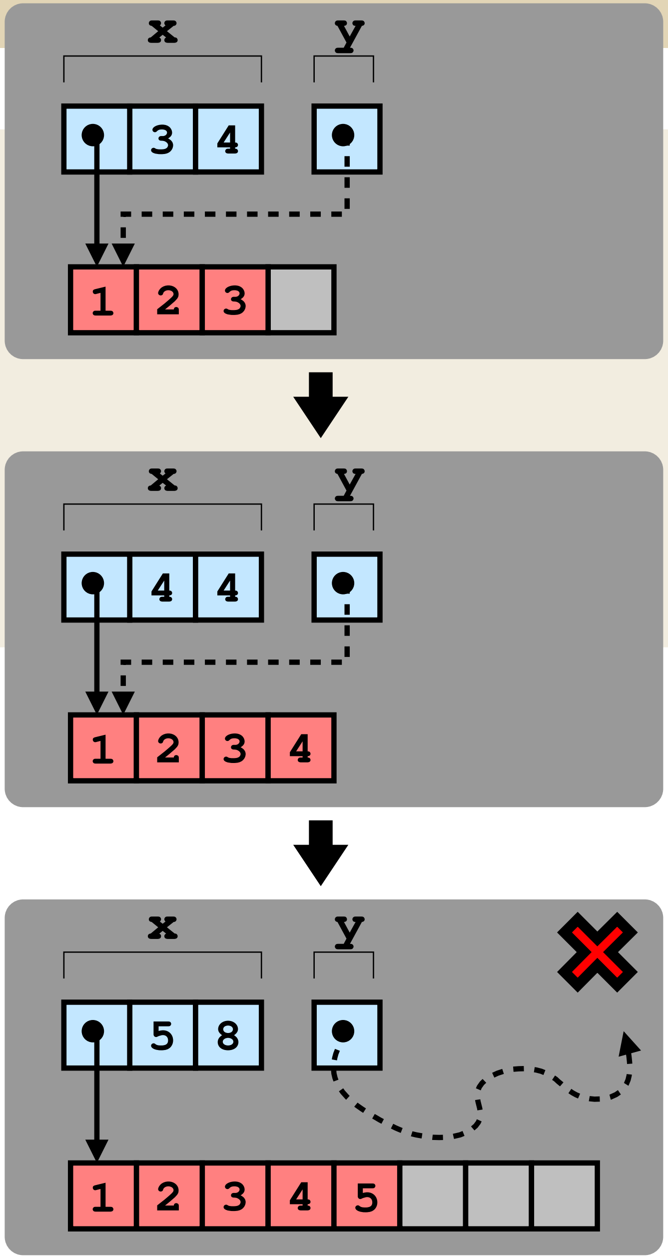


- Multiple **immutable** borrows can coexist for same location
- At most one **mutable** borrow can exist for a location

Rust: Single Writer, Multiple Readers

```
let mut x = vec![1,2,3];
let y = &x[0];
//
x.push(4);
x.push(5);
```

- Can take reference of **array element**!



Featherweight Rust

Featherweight Rust: Syntax

$$
\begin{aligned}
\mathtt{t} \coloneqq\ & \{\,\overline{\mathtt{t}}\,\}^{\mathtt{l}} \\
& \mathtt{let\ mut\ x = t} \\
& \mathtt{w = t} \\
& \mathtt{box\ t} \\
& \&[\mathtt{mut}]\ \mathtt{w} \\
& \mathtt{w} \\
& \hat{\mathtt{w}} \\
& \mathtt{v}
\end{aligned}
$$

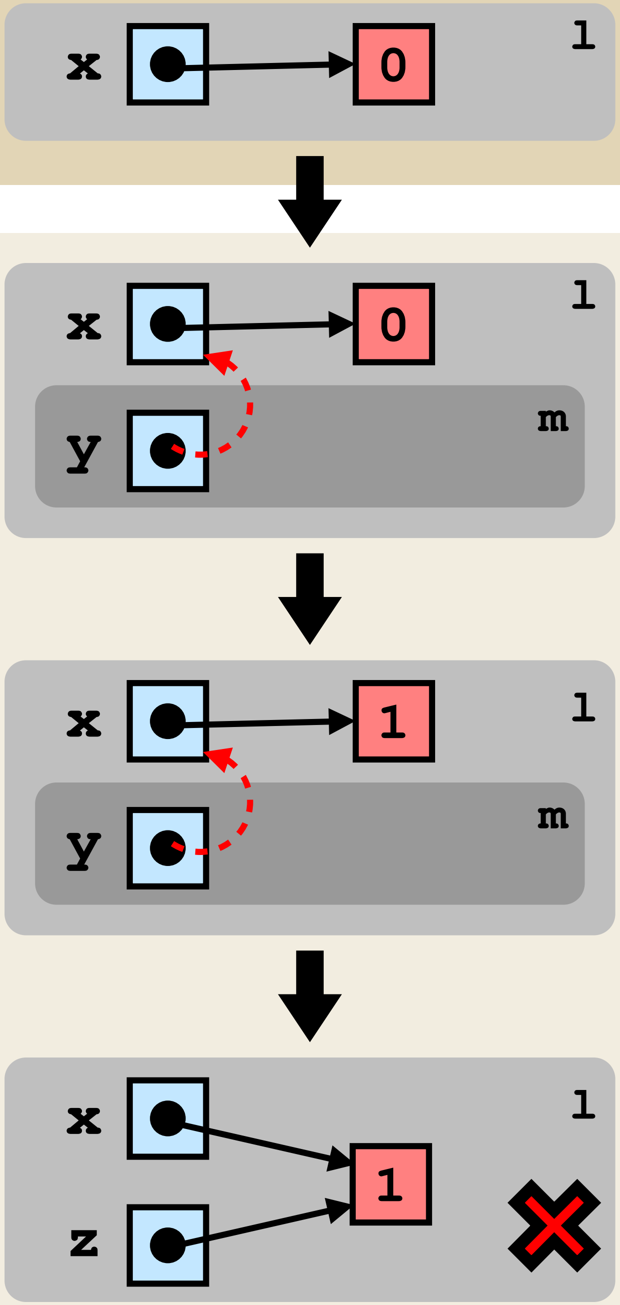
$$
\begin{aligned}
\mathtt{w} \coloneqq\ & \mathtt{x} \\
& \mathtt{*w}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{v} \coloneqq\ & \epsilon \\
& \mathtt{c} \\
& \ell^{\bullet}, \ell^{\circ}
\end{aligned}
$$

$$
\begin{aligned}
\mathtt{T} \coloneqq\ & \epsilon \\
& \mathtt{int} \\
& \&\mathtt{mut}\ \overline{\mathtt{w}} \\
& \&\overline{\mathtt{w}} \\
& \square\mathtt{T}
\end{aligned}
$$

Featherweight Rust: Example

```
{
  let mut x = box 0;
  {
    let mut y = &mut x;
    *y = box 1;
  }^m
  let mut z = x̂;
}^l
```



- Lifetimes form **partial order** and following nesting (hence $l \succeq m$)

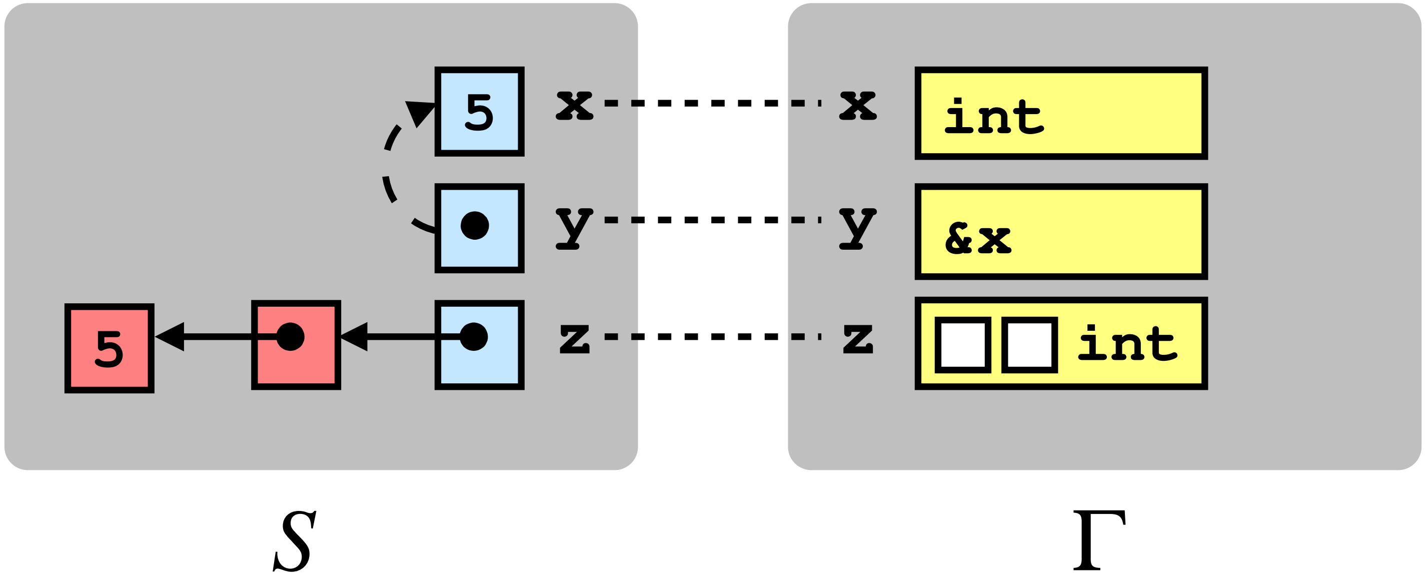
Featherweight Rust: Semantics & Typing

$$\langle\, \mathcal{S}_1 \triangleright \mathtt{t}_1 \longrightarrow \mathcal{S}_2 \triangleright \mathtt{t}_2 \,\rangle^{\mathtt{l}}$$

$$\Gamma_1 \vdash \langle\, \mathtt{t} : \mathtt{T} \,\rangle^{\mathtt{l}}_{\sigma} \dashv \Gamma_2$$

Featherweight Rust: Soundness

$$\mathcal{S} \sim \Gamma$$



Contribution

Featherweight Rust: LVal Typing

Definition (LVal Typing)

An lval $\mathtt{w}$ is said to be typed with respect to an environment Γ, denoted $\Gamma \vdash \mathtt{w} : \langle \tilde{\mathtt{T}} \rangle^{\mathtt{m}}$, according to the following rules:

$$\frac{\Gamma(\mathtt{x}) = \langle \mathtt{T} \rangle^{\mathtt{m}}}{\Gamma \vdash \mathtt{x} : \langle \mathtt{T} \rangle^{\mathtt{m}}} \text{ (T-LvVar)} \qquad \frac{\Gamma \vdash \mathtt{w} : \langle \square \mathtt{T} \rangle^{\mathtt{m}}}{\Gamma \vdash *\mathtt{w} : \langle \mathtt{T} \rangle^{\mathtt{m}}} \text{ (T-LvBox)}$$

$$\frac{\Gamma \vdash \mathtt{w} : \langle \&[\mathtt{mut}]\ \overline{\mathtt{u}} \rangle^{\mathtt{n}} \quad \overline{\Gamma \vdash \mathtt{u} : \langle \mathtt{T} \rangle^{\mathtt{m}}}}{\Gamma \vdash *\mathtt{w} : \langle \bigsqcup_{\mathtt{i}} \mathtt{T}_{\mathtt{i}} \rangle^{\sqcap_{\mathtt{i}} \mathtt{m}_{\mathtt{i}}}} \text{ (T-LvBor)}$$

- Not well founded!
- **Examples:** $\Gamma = \{\mathtt{x} \mapsto \langle \&\mathtt{x} \rangle^{\mathtt{n}}\}$, $\Gamma = \{\mathtt{x} \mapsto \langle \&\mathtt{y} \rangle^{\mathtt{n}}, \mathtt{y} \mapsto \langle \&\mathtt{x} \rangle^{\mathtt{n}}\}$, etc.

Featherweight Rust: Observation

Whilst cyclic typing environments exist, they do not arise when checking **well typed programs** *using the typing rules of* `FR`*.*

- *Hence, just need to prove this intuition holds!*

Featherweight Rust: Linearity

Linearizable

A typing is *linearizable* if each variable maps to a type that only contains variables of strictly lower rank.

- $\Gamma = \{\mathtt{x} \mapsto \langle \&\mathtt{y} \rangle^{\mathtt{n}}, \mathtt{y} \mapsto \langle \mathtt{int} \rangle^{\mathtt{n}}\}$ is linearizable.
- $\Gamma = \{\mathtt{x} \mapsto \langle \& * \mathtt{y} \rangle^{\mathtt{n}}, \mathtt{y} \mapsto \langle \&\mathtt{z} \rangle^{\mathtt{n}}, \mathtt{z} \mapsto \langle \mathtt{int} \rangle^{\mathtt{n}}\}$ is linearizable.
- $\Gamma = \{\mathtt{x} \mapsto \langle \&\mathtt{y} \rangle^{\mathtt{n}}, \mathtt{y} \mapsto \langle \&\mathtt{x} \rangle^{\mathtt{n}}\}$ is **not** linearizable.

Conclusion

- **Featherweight Rust** (FR) is a lightweight formalism of Rust.
- We discovered a source of **non-termination** within the calculus.
- We identified a **sufficient condition** which ensures borrow checking for FR terminates on well typed programs.
- This is a necessary step **towards mechanisation** of the calculus.