RustBelt:
A Quick Dive Into the Abyss

Ralf Jung, Michael Sammler
MPI-SWS, Germany

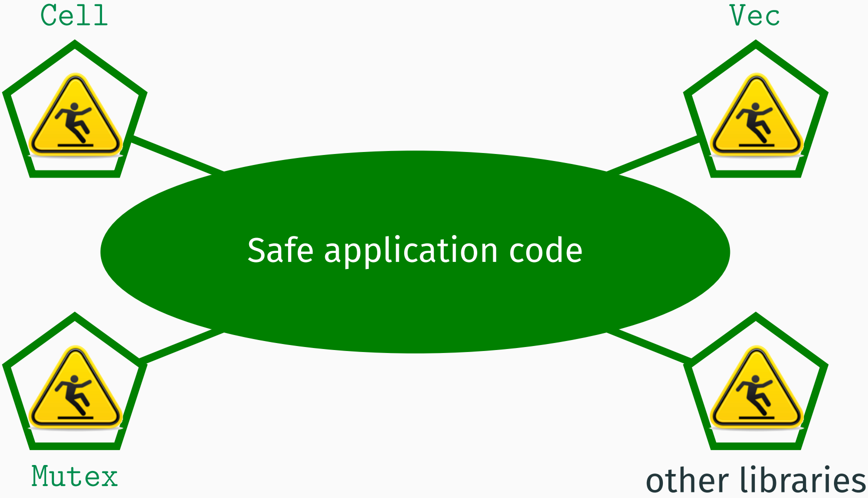
Rust Verification Workshop 2021

RustBelt – formalizing Rust's safety story

The safety of Rust rests on two main pillars:

- A sophisticated type system based on the ideas of ownership and borrowing
- Safe encapsulation of unsafe code

Safely wrapped unsafe code is used pervasively in the Rust ecosystem:



`Cell`

`Vec`

Safe application code

`Mutex`

other libraries

Mapping the Abyss: RustBelt

Extensible safety proof for Rust

The λ_{Rust} type system

$$\tau ::= \textbf{bool} \mid \textbf{int} \mid \textbf{own}_{n}\, \tau \mid \&^{\kappa}_{\textbf{mut}}\, \tau \mid \&^{\kappa}_{\textbf{shr}}\, \tau \mid \Pi \overline{\tau} \mid \Sigma \overline{\tau} \mid \ldots$$

The λ_{Rust} type system

$$\tau ::= \mathbf{bool} \mid \mathbf{int} \mid \mathbf{own}_{n}\,\tau \mid \&^{\kappa}_{\mathbf{mut}}\,\tau \mid \&^{\kappa}_{\mathbf{shr}}\,\tau \mid \Pi\overline{\tau} \mid \Sigma\overline{\tau} \mid \ldots$$

The λ_{Rust} type system

$\tau ::= \textbf{bool} \mid \textbf{int} \mid \textbf{own}_n\,\tau \mid \&^{\kappa}_{\textbf{mut}}\,\tau \mid \&^{\kappa}_{\textbf{shr}}\,\tau \mid \Pi\overline{\tau} \mid \Sigma\overline{\tau} \mid \ldots$

The λ_{Rust} type system

$$\tau ::= \textbf{bool} \mid \textbf{int} \mid \textbf{own}_n\,\tau \mid \&^{\kappa}_{\textbf{mut}}\,\tau \mid \&^{\kappa}_{\textbf{shr}}\,\tau \mid \Pi\overline{\tau} \mid \Sigma\overline{\tau} \mid \ldots$$

The λ_{Rust} type system

$$\tau ::= \textbf{bool} \mid \textbf{int} \mid \textbf{own}_n\,\tau \mid \&^{\kappa}_{\textbf{mut}}\,\tau \mid \&^{\kappa}_{\textbf{shr}}\,\tau \mid \Pi\overline{\tau} \mid \Sigma\overline{\tau} \mid \dots$$

$$\mathbf{T} ::= \emptyset \mid \mathbf{T}, p \lhd \tau \mid \dots$$

Typing context assigns types to paths p (denoting fields of structures)

The λ_{Rust} type system

$$\tau ::= \textbf{bool} \mid \textbf{int} \mid \textbf{own}_n\,\tau \mid \&^{\kappa}_{\textbf{mut}}\,\tau \mid \&^{\kappa}_{\textbf{shr}}\,\tau \mid \Pi\overline{\tau} \mid \Sigma\overline{\tau} \mid \ldots$$

$$\mathbf{T} ::= \emptyset \mid \mathbf{T}, p \lhd \tau \mid \ldots$$

Core substructural typing judgments:

$$\mathbf{E}, \mathbf{L};\ \mathbf{T}_1 \vdash I \dashv x.\,\mathbf{T}_2 \qquad\qquad \mathbf{E}, \mathbf{L};\ \mathbf{K}, \mathbf{T} \vdash F$$

Typing individual instructions I
(**E** and **L** track lifetimes)

Typing whole functions F
(**K** tracks continuations)

Syntactic type safety

$$\mathbf{E}, \mathbf{L}; \mathbf{K}, \mathbf{T} \vdash F \implies F \text{ is safe}$$

Well-typed programs can’t go wrong:

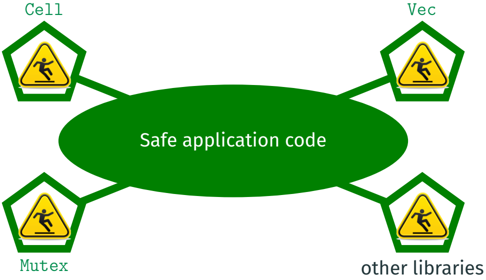
- No data race
- No invalid memory access

Syntactic type safety

Wel

- N
- N

But what about unsafe code?



Unsafe code is essentially untyped.

Syntactic type safety

$$\mathbf{E}, \mathbf{L}; \mathbf{K}, \mathbf{T} \vdash F \implies F \text{ is safe}$$

Logical relations: "semantic everything"

1. Semantic interpretation of types ($[\![\tau]\!]$)
2. Semantic interpretation of judgments ($\vDash$)

1. Semantic interpretation of types

Define ownership invariant for every type τ:

$$[\![\tau]\!].\mathrm{own}(t, v)$$

1. Semantic interpretation of types

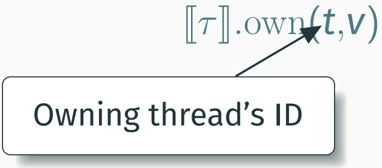
Define ownership invariant for every type τ:

$$[\![\tau]\!].\mathrm{own}(t, v)$$

Owning thread's ID

1. Semantic interpretation of types

Define ownership invariant for every type τ:



$$[\![\tau]\!].\mathrm{own}(t, v)$$

Owning thread's ID

What logic should we use to express the invariant?

Separation Logic to the Rescue!

1. Semantic interpretation of types

Define ownership invariant for every type τ:

$$[\![\tau]\!].\mathrm{own}(t, v)$$

Owning thread’s ID

We use a modern, higher-order, concurrent separation logic framework called Iris:

- Implemented in the Coq proof assistant
- Designed to derive new reasoning principles inside the logic

2. Lift to all judgments

Define ownership invariant for every type τ:

$$[\![\tau]\!].\mathrm{own}(t, v)$$

Lift to semantic contexts $[\![\mathbf{T}]\!](t)$:

$$[\![p_1 \lhd \tau_1, p_2 \lhd \tau_2]\!](t) \quad := \quad [\![\tau_1]\!].\mathrm{own}(t, p_1) * [\![\tau_2]\!].\mathrm{own}(t, p_2)$$

2. Lift to all judgments

Define ownership invariant for every type τ:

$$[\![\tau]\!].\mathrm{own}(t, v)$$

Lift to semantic contexts $[\![\mathbf{T}]\!](t)$:

$$[\![p_1 \lhd \tau_1, p_2 \lhd \tau_2]\!](t) \quad := $$

$$[\![\tau_1]\!].\mathrm{own}(t, p_1) * [\![\tau_2]\!].\mathrm{own}(t, p_2)$$

Separating conjunction

2. Lift to all judgments

Define ownership invariant for every type τ:

$$[\![\tau]\!].\mathrm{own}(t, v)$$

Lift to semantic typing judgments:

$$\mathbf{E}, \mathbf{L}; \ \mathbf{T}_1 \models I \dashv \mathbf{T}_2 \quad := \quad \forall t.\ \{[\![\mathbf{E}]\!] * [\![\mathbf{L}]\!] * [\![\mathbf{T}_1]\!](t)\}\ I\ \{[\![\mathbf{E}]\!] * [\![\mathbf{L}]\!] * [\![\mathbf{T}_2]\!](t)\}$$

2. Lift to all judgments

Define ownership invariant for every type τ:

$$[\![\tau]\!].\mathrm{own}(t, v)$$

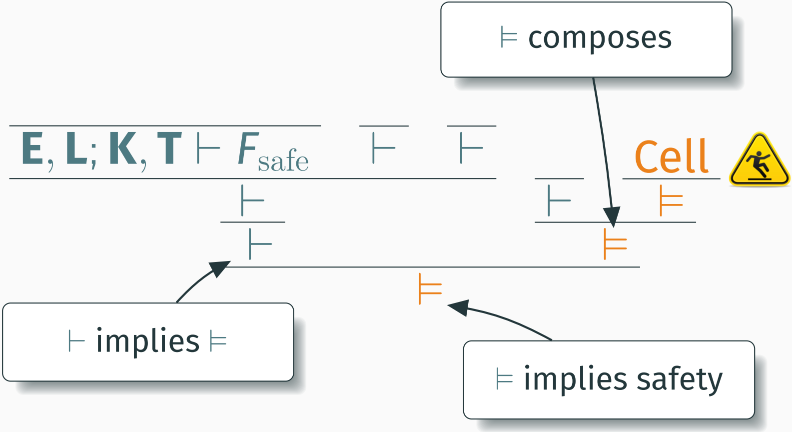
Lift to semantic typing judgments:

$$\mathbf{E}, \mathbf{L};\ \mathbf{T}_1 \models I \mathrel{=\!\!|} \mathbf{T}_2 \quad := $$

$$\forall t.\ \{[\![\mathbf{E}]\!] * [\![\mathbf{L}]\!] * [\![\mathbf{T}_1]\!](t)\}\ I\ \{[\![\mathbf{E}]\!] * [\![\mathbf{L}]\!] * [\![\mathbf{T}_2]\!](t)\}$$

Hoare triple

Composition with unsafe code



⊨ composes

$\mathbf{E}, \mathbf{L}; \mathbf{K}, \mathbf{T} \vdash F_{\text{safe}}$

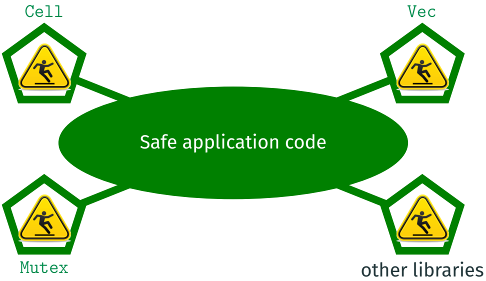
Cell

⊢ implies ⊨

⊨ implies safety

Composition with unsafe code

The whole program is safe if the unsafe pieces are safe!



Depth 1m:
How do we define $[\![\tau]\!].\mathrm{own}(t, v)$?

$$\llbracket \mathbf{own}_n\, \tau \rrbracket.\mathrm{own}(t, \ell) :=$$

$$\triangleright\big(\exists w.\ \ell \mapsto w * \llbracket \tau \rrbracket.\mathrm{own}(t, w)\big) * \ldots$$

$$[\![\mathbf{own}_n\, \tau]\!].\mathrm{own}(t, \ell) :=$$

$$\triangleright\big(\exists w.\ \ell \mapsto w * [\![\tau]\!].\mathrm{own}(t, w)\big) * \ldots$$

$$\llbracket \mathbf{own}_n\, \tau \rrbracket.\mathrm{own}(t, \ell) :=$$

$$\triangleright\big(\exists w.\ \ell \mapsto w * \llbracket \tau \rrbracket.\mathrm{own}(t, w)\big) * \ldots$$

$$\llbracket \&^{\kappa}_{\mathbf{mut}}\, \tau \rrbracket.\mathrm{own}(t, \ell) :=$$

$$\&^{\kappa}_{\mathbf{full}}\big(\exists w.\ \ell \mapsto w * \llbracket \tau \rrbracket.\mathrm{own}(t, w)\big)$$

$$\llbracket \mathbf{own}_n\, \tau \rrbracket.\mathrm{own}(t, \ell) :=$$
$$\triangleright\big(\exists w.\ \ell \mapsto w * \llbracket \tau \rrbracket.\mathrm{own}(t, w)\big) * \ldots$$

$$\llbracket \&^{\kappa}_{\mathbf{mut}}\, \tau \rrbracket.\mathrm{own}(t, \ell) :=$$
$$\&^{\kappa}_{\mathbf{full}}\big(\exists w.\ \ell \mapsto w * \llbracket \tau \rrbracket.\mathrm{own}(t, w)\big)$$

Lifetime logic connective

Lifetime logic

An extension of separation logic adding support for borrowing:

Lifetime logic

An extension of separation logic adding support for borrowing:

- $\&_{\mathbf{full}}^{\kappa} P$: P borrowed for lifetime κ

Lifetime logic

An extension of separation logic adding support for borrowing:

- $\&_{\mathbf{full}}^{\kappa} P$: P borrowed for lifetime κ
- $[\kappa]$: Witnessing and owning the fact that κ is still ongoing

Depth 10m:

`Cell<T>`

Cell

Verification steps in RustBelt:

1. Define type invariants: $[\![\texttt{Cell}(\tau)]\!]$

Cell

Verification steps in RustBelt:

1. Define type invariants: $[\![\texttt{Cell}(\tau)]\!]$
2. Verify semantic well-typedness: $\vDash$

Cell

```
pub struct Cell<T> { value: UnsafeCell<T>, }
impl<T> Cell<T> {
  fn new(val: T) -> Cell<T> {
    // equivalent: unsafe { mem::transmute(val) }
    Cell { value: UnsafeCell::new(val) }
  }
  fn into_inner(self) -> T {
    // equivalent: unsafe { mem::transmute(self) }
    self.value.into_inner()
  }
}
```

Cell

```
pub struct Cell<T> { value: UnsafeCell<T>, }
impl<T> Cell<T> {
  fn new(val: T) -> Cell<T> {
    // equivalent: unsafe { mem::transmute(val) }
    Cell { value: UnsafeCell::new(val) }
  }
  fn into_inner(self) -> T {
    // equivalent: unsafe { mem::transmute(self) }
    self.value.into_inner()
  }
}
```

$$[\![\texttt{Cell}(\tau)]\!].\mathrm{own}(t, v) := [\![\tau]\!].\mathrm{own}(t, v)$$

Semantic well-typedness of Cell::new: $\vDash$

```
fn new(val: T) -> Cell<T> {



}
```

Semantic well-typedness of Cell::new: $\vDash$

$\{[\![\tau]\!].\mathrm{own}(t, \mathit{val})\}$

```
fn new(val: T) -> Cell<T> {



}
```

$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{own}(t, \mathit{return})\}$

Semantic well-typedness of Cell::new: $\vDash$

$$[\![\texttt{Cell}(\tau)]\!].\mathrm{own}(t,v) := [\![\tau]\!].\mathrm{own}(t,v)$$

$\{[\![\tau]\!].\mathrm{own}(t, \mathit{val})\}$

```
fn new(val: T) -> Cell<T> {

  return unsafe { mem::transmute(val) };

}
```

$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{own}(t, \mathit{return})\}$

Semantic well-typedness of Cell::new: $\vDash$

$$[\![\texttt{Cell}(\tau)]\!].\mathrm{own}(t, v) := [\![\tau]\!].\mathrm{own}(t, v)$$

$\{[\![\tau]\!].\mathrm{own}(t, val)\}$

```
fn new(val: T) -> Cell<T> {
```

$\quad\{[\![\tau]\!].\mathrm{own}(t, val)\}$

```
  return unsafe { mem::transmute(val) };


}
```

$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{own}(t, return)\}$

Semantic well-typedness of Cell::new: $\vDash$

$$[\![\texttt{Cell}(\tau)]\!].\mathrm{own}(t, v) := [\![\tau]\!].\mathrm{own}(t, v)$$

$\{[\![\tau]\!].\mathrm{own}(t, val)\}$

```
fn new(val: T) -> Cell<T> {
```

$\quad\{[\![\tau]\!].\mathrm{own}(t, val)\}$

```
  return unsafe { mem::transmute(val) };
```

$\quad\{[\![\texttt{Cell}(\tau)]\!].\mathrm{own}(t, return)\}$

```
}
```

$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{own}(t, return)\}$

Depth 100m:

`&Cell<T>`

Sharing predicates

We have seen:

$$T \equiv \texttt{Cell}< T >$$

Sharing predicates

We have seen:

$$T \equiv \texttt{Cell}< T >$$

But we also know:

$$\&T \not\equiv \&\texttt{Cell}< T >$$

Sharing predicates

We have seen:

$$T \equiv \texttt{Cell}< T >$$

But we also know:

$$\&T \not\equiv \&\texttt{Cell}< T >$$

Needed: separate invariant for shared `Cell`

Sharing predicates

We have seen:

Semantic type consists of:

1. Ownership invariant: $[\![\tau]\!].\mathrm{own}(t, v)$

But

Needed: separate invariant for shared `Cell`

Sharing predicates

We have seen:

Semantic type consists of:

1. Ownership invariant: $[\![\tau]\!].\mathrm{own}(t, v)$
2. Sharing invariant: $[\![\tau]\!].\mathrm{shr}(\kappa, t, \ell)$

But

Needed: separate invariant for shared `Cell`

Sharing predicates

We have seen:

Semantic type consists of:

1. Ownership invariant: $[\![\tau]\!].\mathrm{own}(t, v)$
2. Sharing invariant: $[\![\tau]\!].\mathrm{shr}(\kappa, t, \ell)$

What is $[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\kappa, t, \ell)$?

But

Needed: separate invariant for shared `Cell`

Cell::set

```
impl<T> Cell<T> {
  pub fn set(&self, val: T) {
    unsafe {
      let value_ptr : *mut T = self.value.get();
      *value_ptr = val;
    }
  }
}
```

Cell::set

```
impl<T> Cell<T> {
  pub fn set(&self, val: T) {
    unsafe {
      let value_ptr : *mut T = self.value.get();
      *value_ptr = val;
    }
  }
}
```

Why is `Cell::set` safe?

Cell::set

```
impl<T> Cell<T> {
  pub fn set(&self, val: T) {
    unsafe {
      let value_ptr : *mut T = self.value.get();
      *value_ptr = val;
    }
  }
}
```

Why is `Cell::set` safe?

- No concurrent access (`Cell` is not `Sync`)

Cell::set

```
impl<T> Cell<T> {
  pub fn set(&self, val: T) {
    unsafe {
      let value_ptr : *mut T = self.value.get();
      *value_ptr = val;
    }
  }
}
```

Why is `Cell::set` safe?

- No concurrent access (`Cell` is not `Sync`)
- No interior pointers

$[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\kappa, \boldsymbol{t}, \ell)$

Remember: $[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\kappa, \boldsymbol{t}, \ell)$ should not allow concurrent accesses.

$$[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\kappa, \boldsymbol{t}, \ell) :=$$

???

$[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\kappa, t, \ell)$

Remember: $[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\kappa, t, \ell)$ should not allow concurrent accesses.

$$[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\kappa, t, \ell) :=$$
$$\&_{\mathbf{na}}^{\kappa/t}\bigl(\exists v.\, \ell.\mathit{value} \mapsto v * [\![\tau]\!].\mathrm{own}(t, v)\bigr)$$

$[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\kappa, t, \ell)$

Remember: $[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\kappa, t, \ell)$ should not allow concurrent accesses.

$$[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\kappa, t, \ell) :=$$

$$\&_{\mathbf{na}}^{\kappa/t}(\exists v.\, \ell.\mathit{value} \mapsto v * [\![\tau]\!].\mathrm{own}(t, v))$$

Non-atomic borrow

Semantic well-typedness of Cell::set: $\vDash$

$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, t, \mathit{self}) * [\![\tau]\!].\mathrm{own}(t, \mathit{val}) \qquad \}$

```
fn set(&'a self, val: T) {
```

```
}
```

$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, t, \mathit{self}) \qquad \}$

Semantic well-typedness of Cell::set: $\vDash$

$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, \mathit{t}, \mathit{self}) * [\![\tau]\!].\mathrm{own}(\mathit{t}, \mathit{val}) * [\alpha]\}$

```
fn set(&'a self, val: T) {
```

```
}
```

$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, \mathit{t}, \mathit{self}) * [\alpha]\}$

Semantic well-typedness of Cell::set: $\vDash$

$$[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\kappa, \mathit{t}, \ell) := \&_{\mathbf{na}}^{\kappa/t}(\exists v.\, \ell.\mathit{value} \mapsto v * [\![\tau]\!].\mathrm{own}(t, v))$$

$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, t, \mathit{self}) * [\![\tau]\!].\mathrm{own}(t, \mathit{val}) * [\alpha]\}$

```
fn set(&'a self, val: T) {



    let value_ptr : *mut T = self.value.get();
    *value_ptr = val;



}
```

$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, t, \mathit{self}) * [\alpha]\}$

Semantic well-typedness of Cell::set: $\vDash$

$$[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\kappa, t, \ell) := \&_{\mathbf{na}}^{\kappa/t}(\exists v.\, \ell.\mathit{value} \mapsto v * [\![\tau]\!].\mathrm{own}(t, v))$$

$$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, t, \mathit{self}) * [\![\tau]\!].\mathrm{own}(t, \mathit{val}) * [\alpha]\}$$

```
fn set(&'a self, val: T) {
```

$$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, t, \mathit{self}) * [\![\tau]\!].\mathrm{own}(t, \mathit{val}) * [\alpha]\}$$

```
    let value_ptr : *mut T = self.value.get();
    *value_ptr = val;


}
```

$$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, t, \mathit{self}) * [\alpha]\}$$

Semantic well-typedness of Cell::set: $\vDash$

$$[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\kappa, t, \ell) := \&_{\mathbf{na}}^{\kappa/t}(\exists v.\, \ell.\mathit{value} \mapsto v * [\![\tau]\!].\mathrm{own}(t, v))$$

$$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, t, \mathit{self}) * [\![\tau]\!].\mathrm{own}(t, \mathit{val}) * [\alpha]\}$$

```
fn set(&'a self, val: T) {
```

$$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, t, \mathit{self}) * [\![\tau]\!].\mathrm{own}(t, \mathit{val}) * [\alpha]\}$$

$$\{\mathit{self}.\mathit{value} \mapsto v' * [\![\tau]\!].\mathrm{own}(t, v') * [\![\tau]\!].\mathrm{own}(t, \mathit{val})\}$$

```
    let value_ptr : *mut T = self.value.get();
    *value_ptr = val;



}
```

$$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, t, \mathit{self}) * [\alpha]\}$$

Semantic well-typedness of Cell::set: $\vDash$

$$[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\kappa, t, \ell) := \&_{\mathbf{na}}^{\kappa/t}(\exists v.\, \ell.\mathit{value} \mapsto v * [\![\tau]\!].\mathrm{own}(t, v))$$

$$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, t, \mathit{self}) * [\![\tau]\!].\mathrm{own}(t, \mathit{val}) * [\alpha]\}$$

```
fn set(&'a self, val: T) {
```

$$\quad\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, t, \mathit{self}) * [\![\tau]\!].\mathrm{own}(t, \mathit{val}) * [\alpha]\}$$

$$\qquad\{\mathit{self}.\mathit{value} \mapsto v' * [\![\tau]\!].\mathrm{own}(t, v') * [\![\tau]\!].\mathrm{own}(t, \mathit{val})\}$$

```
    let value_ptr : *mut T = self.value.get();
    *value_ptr = val;
```

$$\qquad\{\mathit{self}.\mathit{value} \mapsto \mathit{val} * [\![\tau]\!].\mathrm{own}(t, \mathit{val})\}$$

```
}
```

$$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, t, \mathit{self}) * [\alpha]\}$$

Semantic well-typedness of Cell::set: $\vDash$

$$[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\kappa, t, \ell) := \&_{\mathbf{na}}^{\kappa/t}(\exists v.\, \ell.\mathit{value} \mapsto v * [\![\tau]\!].\mathrm{own}(t,v))$$

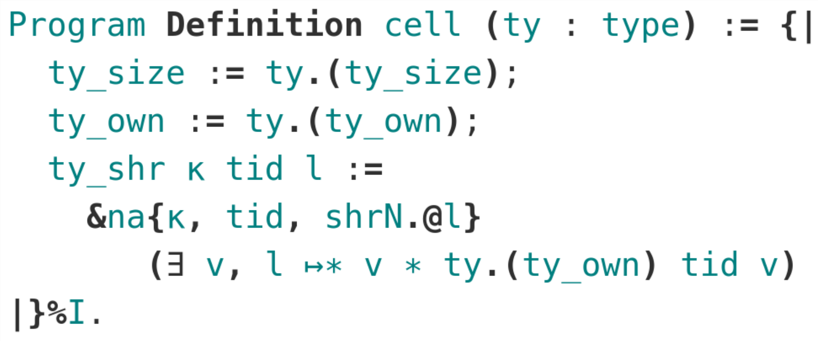
```
$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, t, \mathit{self}) * [\![\tau]\!].\mathrm{own}(t, \mathit{val}) * [\alpha]\}$
fn set(&'a self, val: T) {
  $\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, t, \mathit{self}) * [\![\tau]\!].\mathrm{own}(t, \mathit{val}) * [\alpha]\}$
    $\{\mathit{self}.\mathit{value} \mapsto v' * [\![\tau]\!].\mathrm{own}(t, v') * [\![\tau]\!].\mathrm{own}(t, \mathit{val})\}$
    let value_ptr : *mut T = self.value.get();
    *value_ptr = val;
    $\{\mathit{self}.\mathit{value} \mapsto \mathit{val} * [\![\tau]\!].\mathrm{own}(t, \mathit{val})\}$
  $\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, t, \mathit{self}) * [\alpha]\}$
}
$\{[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\alpha, t, \mathit{self}) * [\alpha]\}$
```

Depth 1000m: The deep end of the Abyss

Depth 1000m:
~~The deep end of the Abyss~~
RustBelt in Coq

Cell in Coq

$$
\begin{aligned}
[\![\texttt{Cell}(\tau)]\!].\mathrm{own}(t,v) &:= [\![\tau]\!].\mathrm{own}(t,v) \\
[\![\texttt{Cell}(\tau)]\!].\mathrm{shr}(\kappa,t,\ell) &:= \&_{\mathbf{na}}^{\kappa/t}(\exists v.\, \ell.\mathit{value} \mapsto v * [\![\tau]\!].\mathrm{own}(t,v))
\end{aligned}
$$

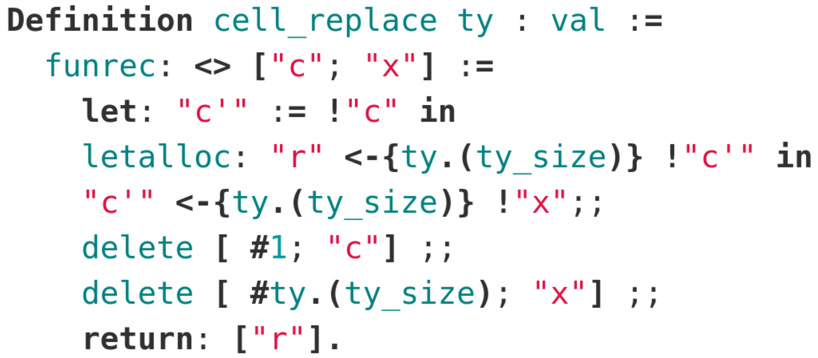


```
Program Definition cell (ty : type) := {|
  ty_size := ty.(ty_size);
  ty_own := ty.(ty_own);
  ty_shr κ tid l :=
    &na{κ, tid, shrN.@l}
      (∃ v, l ↦∗ v ∗ ty.(ty_own) tid v)
|}%I.
```

Cell::new in Coq

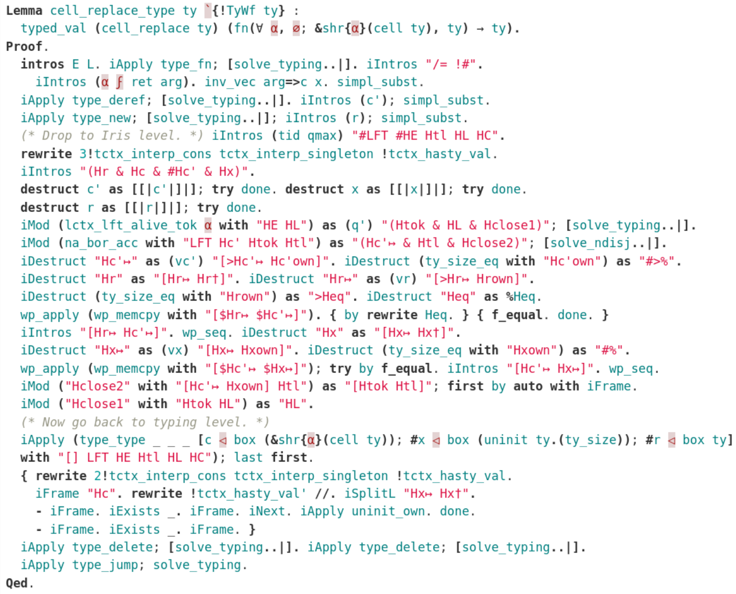
```
Definition cell_new : val :=
  funrec: <> ["x"] := return: ["x"].

Lemma cell_new_type ty `{!TyWf ty} :
  typed_val cell_new (fn(∅; ty) → cell ty).
Proof.
  intros E L. iApply type_fn; [solve_typing..|].
  iIntros "/= !#". iIntros (_ ϝ ret arg). inv_vec arg=>x.
  simpl_subst. iApply type_jump; [solve_typing..|].
  iIntros (???) "#LFT _ $ Hty".
  rewrite !tctx_interp_singleton /=. done.
Qed.
```

Cell::replace in Coq



```
Definition cell_replace ty : val :=
  funrec: <> ["c"; "x"] :=
    let: "c'" := !"c" in
    letalloc: "r" <-{ty.(ty_size)} !"c'" in
    "c'" <-{ty.(ty_size)} !"x";;
    delete [ #1; "c"] ;;
    delete [ #ty.(ty_size); "x"] ;;
    return: ["r"].
```

Cell::replace in Coq



```
Lemma cell_replace_type ty `{!TyWf ty} :
  typed_val (cell_replace ty) (fn(∀ α, ∅; &shr{α}(cell ty), ty) → ty).
Proof.
  intros E L. iApply type_fn; [solve_typing..|]. iIntros "/= !#".
    iIntros (α ϝ ret arg). inv_vec arg=>c x. simpl_subst.
  iApply type_deref; [solve_typing..|]. iIntros (c'); simpl_subst.
  iApply type_new; [solve_typing..|]; iIntros (r); simpl_subst.
  (* Drop to Iris level. *) iIntros (tid qmax) "#LFT #HE Htl HL HC".
  rewrite 3!tctx_interp_cons tctx_interp_singleton !tctx_hasty_val.
  iIntros "(Hr & Hc & #Hc' & Hx)".
  destruct c' as [[|c'|]||]; try done. destruct x as [[|x|]||]; try done.
  destruct r as [[|r|]||]; try done.
  iMod (lctx_lft_alive_tok α with "HE HL") as (q') "(Htok & HL & Hclose1)"; [solve_typing..|].
  iMod (na_bor_acc with "LFT Hc' Htok Htl") as "(Hc'↦ & Htl & Hclose2)"; [solve_ndisj..|].
  iDestruct "Hc'↦" as (vc') "[>Hc'↦ Hc'own]". iDestruct (ty_size_eq with "Hc'own") as "#>%".
  iDestruct "Hr" as "[Hr↦ Hr†]". iDestruct "Hr↦" as (vr) "[>Hr↦ Hrown]".
  iDestruct (ty_size_eq with "Hrown") as ">Heq". iDestruct "Heq" as %Heq.
  wp_apply (wp_memcpy with "[$Hr↦ $Hc'↦]"). { by rewrite Heq. } { f_equal. done. }
  iIntros "[Hr↦ Hc'↦]". wp_seq. iDestruct "Hx" as "[Hx↦ Hx†]".
  iDestruct "Hx↦" as (vx) "[Hx↦ Hxown]". iDestruct (ty_size_eq with "Hxown") as "#%".
  wp_apply (wp_memcpy with "[$Hc'↦ $Hx↦]"); try by f_equal. iIntros "[Hc'↦ Hx↦]". wp_seq.
  iMod ("Hclose2" with "[Hc'↦ Hxown] Htl") as "[Htok Htl]"; first by auto with iFrame.
  iMod ("Hclose1" with "Htok HL") as "HL".
  (* Now go back to typing level. *)
  iApply (type_type _ _ _ [c ◁ box (&shr{α}(cell ty)); #x ◁ box (uninit ty.(ty_size)); #r ◁ box ty]
  with "[] LFT HE Htl HL HC"); last first.
  { rewrite 2!tctx_interp_cons tctx_interp_singleton !tctx_hasty_val.
    iFrame "Hc". rewrite !tctx_hasty_val' //. iSplitL "Hx↦ Hx†".
    - iFrame. iExists _. iFrame. iNext. iApply uninit_own. done.
    - iFrame. iExists _. iFrame. }
  iApply type_delete; [solve_typing..|]. iApply type_delete; [solve_typing..|].
  iApply type_jump; solve_typing.
Qed.
```

Semantic typing might look intimidating, but fundamentally it is just program verification!

Thanks for your attention!