RustBelt: A Quick Dive Into the Abyss

Ralf Jung, Michael Sammler
MPI-SWS, Germany

Rust Verification Workshop 2021
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
- Mutex
- Other libraries
Mapping the Abyss: RustBelt

Extensible safety proof for Rust
The $\lambda_{\text{Rust}}$ type system

$\tau ::= \text{bool} \mid \text{int} \mid \text{own}_n \tau \mid &^\kappa \text{mut} \tau \mid &^\kappa \text{shr} \tau \mid \Pi \bar{\tau} \mid \Sigma \bar{\tau} \mid \ldots$
The \( \lambda_{Rust} \) type system

\[ \tau ::= \text{bool} \mid \text{int} \mid \text{own}_n \tau \mid \&^\kappa \text{mut} \tau \mid \&^\kappa \text{shr} \tau \mid \prod \tau \mid \Sigma \tau \mid \ldots \]
The $\lambda$Rust type system

$$\tau ::= \text{bool} \mid \text{int} \mid \text{own}_n \tau \mid \&^\kappa \text{mut} \tau \mid \&^\kappa \text{shr} \tau \mid \prod \tau \mid \Sigma \tau \mid \ldots$$
The \( \lambda_{\text{Rust}} \) type system

\[ \tau ::= \text{bool} \mid \text{int} \mid \text{own}_n \tau \mid \&^{\mu} \tau \mid \&^{\kappa} \tau \mid \Pi \tau \mid \Sigma \tau \mid \ldots \]
The $\lambda_{\text{Rust}}$ type system

\[ \tau ::= \text{bool} \mid \text{int} \mid \text{own}_n \tau \mid \&_{\text{mut}}^\kappa \tau \mid \&_{\text{shr}}^\kappa \tau \mid \prod \bar{\tau} \mid \Sigma \bar{\tau} \mid \ldots \]

\[ T ::= \emptyset \mid T, p <\triangleleft \tau \mid \ldots \]

Typing context assigns types to paths $p$ (denoting fields of structures)
The $\lambda_{\text{Rust}}$ type system

$\tau ::= \text{bool} | \text{int} | \text{own}_n \tau | \&^\kappa \text{mut} \tau | \&^\kappa \text{shr} \tau | \Pi \tau | \Sigma \tau | \ldots$

$T ::= \emptyset | T, p \triangleleft \tau | \ldots$

Core substructural typing judgments:

$E, L; T_1 \vdash I \vdash x. T_2$

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

$E, L; K, T \vdash F$

Typing whole functions $F$ ($K$ tracks continuations)
Syntactic type safety

\[ E, L; K, T \vdash F \implies F \text{ is safe} \]

Well-typed programs can’t go wrong:

- No data race
- No invalid memory access
But what about unsafe code?

Unsafe code is essentially untyped.
Syntactic type safety

\[ E, L; K, T \vdash F \quad \implies \quad F \text{ is safe} \]

**Logical relations:** "semantic everything"
1. Semantic interpretation of types (\(\llbracket \tau \rrbracket\))
2. Semantic interpretation of judgments (\(\vdash\))
1. Semantic interpretation of types

Define **ownership invariant** for every type $\tau$: 

$$[\tau].\text{own}(t,v)$$
1. Semantic interpretation of types

Define **ownership invariant** for every type $\tau$:

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

Owing thread’s ID
1. Semantic interpretation of types

Define **ownership invariant** for every type $\tau$:

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

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 $\tau$:

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

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 $\tau$:

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

Lift to semantic contexts $[T](t)$:

$$[p_1 \triangleleft \tau_1, p_2 \triangleleft \tau_2](t) := [\tau_1].\text{own}(t, p_1) * [\tau_2].\text{own}(t, p_2)$$
2. Lift to all judgments

Define **ownership invariant** for every type $\tau$:

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

Lift to semantic contexts $[T](t)$:

$$[[p_1 \triangleleft \tau_1, p_2 \triangleleft \tau_2]](t) := [[\tau_1]].\text{own}(t,p_1) \ast [[\tau_2]].\text{own}(t,p_2)$$

Separating conjunction
2. Lift to all judgments

Define **ownership invariant** for every type \( \tau \):

\[
[\tau].\text{own}(t,v)
\]

Lift to **semantic typing judgments**:

\[
E, L; \; T_1 \models I \models T_2 \quad := \\
\forall t. \{[E] \times [L] \times [T_1](t)\} \models I \{[E] \times [L] \times [T_2](t)\}
\]
2. Lift to all judgments

Define **ownership invariant** for every type $\tau$:

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

Lift to **semantic typing judgments**:

$$E, L; T_1 \models I \models T_2 :=$$

$$\forall t. \{[[E] \ast [L] \ast [T_1](t)] \mid I \{[[E] \ast [L] \ast [T_2](t)] \}$$

**Hoare triple**
Composition with unsafe code

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

⊢ implies ⊨

⊨ implies safety

Cell ⊨ implies $F_{\text{safe}} \Rightarrow F$ is safe is not enough!

Use a semantic approach based on logical relations.

Logical relations in three steps:

1. Semantic interpretation of types ($\vdash \tau$)
2. Lift that to all judgments (⊨)
3. Prove compatibility (⊢ implies ⊨)

4. ...

5. Profit! Publish!

The whole program is safe if the unsafe pieces are safe!
Composition with unsafe code

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

Safe application code

- **Cell**
- **Vec**
- **Mutex**
- Other libraries
Depth 1m: How do we define $\tau$.own$(t,v)$?
\[ [\text{own}_n \tau]\cdot \text{own}(t, \ell) := \]
\[ \triangleright (\exists w. \ell \leftrightarrow w \ast [\tau]\cdot \text{own}(t, w)) \ast \ldots \]
\[
\left[ \text{own}_n \; \tau \right] . \text{own}(t, \ell) :=
\triangleright (\exists w. \; \ell \rightarrow w \ast \left[ \tau \right] . \text{own}(t, w)) \ast \ldots
\]
\[ [\text{own}_n \tau] . \text{own}(t, \ell) := \]
\[ \triangleright (\exists w. \ell \mapsto w \ast [\tau] . \text{own}(t, w)) \ast \ldots \]

\[ [\&^\kappa_\text{mut} \tau] . \text{own}(t, \ell) := \]
\[ \&^\kappa_{\text{full}} (\exists w. \ell \mapsto w \ast [\tau] . \text{own}(t, w)) \]
\[
[\text{own}_n \tau].\text{own}(t, \ell) := \left\langle \left( \exists w. \ell \mapsto w * [\tau].\text{own}(t, w) \right) \right\rangle * \ldots
\]

\[
[\text{&}^{\kappa}_{\text{mut}} \tau].\text{own}(t, \ell) := \&^{\kappa}_{\text{full}} \left( \exists w. \ell \mapsto w * [\tau].\text{own}(t, w) \right)
\]

**Lifetime logic connective**
An extension of separation logic adding support for borrowing:
Lifetime logic

An extension of separation logic adding support for **borrowing**:

- \( \&_{\text{full}}^{\kappa} P: P \) borrowed for lifetime \( \kappa \)
Lifetime logic

An extension of separation logic adding support for **borrowing**:

- $\&_{\text{full}}^\kappa P$: $P$ borrowed for lifetime $\kappa$
- $[\kappa]$: Witnessing and owning the fact that $\kappa$ is still ongoing
Depth 10m:

Cell<T>
Verification steps in RustBelt:

1. Define type invariants: \([\text{Cell}(\tau)]\)
Verification steps in RustBelt:

1. Define type invariants: \([\text{Cell}(\tau)]\)
2. Verify semantic well-typedness: \(\models\)
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()
    }
}
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()
    }
}

\[
[\text{Cell}(\tau)] \cdot \text{own}(\text{t}, \text{v}) := [\tau] \cdot \text{own}(\text{t}, \text{v})
\]
Semantic well-typedness of Cell::new: ⊨

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

}
```
Semantic well-typedness of Cell::new: \[ \models \]

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

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

}

\[
\{ [[Cell(\tau)]].own(t, return) \}
\]
Semantic well-typedness of Cell::new: $\models$

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

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

```rust
fn new(val: T) -> Cell<T> {
    return unsafe { mem::transmute(val) };
}

$$\{[[\text{Cell}(\tau)]].\text{own}(t, return)\}$$
```
Semantic well-typedness of \texttt{Cell::new}: $\models$

\begin{align*}
\llbracket \text{Cell}(\tau) \rrbracket.\text{own}(t, v) &:= \llbracket \tau \rrbracket.\text{own}(t, v) \\
\{ \llbracket \tau \rrbracket.\text{own}(t, \text{val}) \} &
\end{align*}

\begin{verbatim}
fn new(val: T) -> Cell<T> {
    \{ \llbracket \tau \rrbracket.\text{own}(t, \text{val}) \}
    return unsafe { mem::transmute(val) }
}
\end{verbatim}

\begin{align*}
\{ \llbracket \text{Cell}(\tau) \rrbracket.\text{own}(t, \text{return}) \}
\end{align*}
Semantic well-typedness of Cell::new: \(\models\)

\[
[\text{Cell}(\tau)].\text{own}(t, v) := [\tau].\text{own}(t, v)
\]

\[
\{[\tau].\text{own}(t, val)\}
\]

```rust
fn new(val: T) -> Cell<T> {
    \{[\tau].\text{own}(t, val)\}
    return unsafe { mem::transmute(val) };
    \{[Cell(\tau)].\text{own}(t, return)\}
}
\}
\{[Cell(\tau)].\text{own}(t, return)\}
```
Depth 100m:

&Cell<T>
Sharing predicates

We have seen:

\[ T \equiv \text{Cell}\langle T \rangle \]
Sharing predicates

We have seen:

\[ T \equiv \text{Cell}\langle T \rangle \]

But we also know:

\[ \&T \neq \&\text{Cell}\langle T \rangle \]
Sharing predicates

We have seen:

\[ T \equiv \text{Cell} \langle T \rangle \]

But we also know:

\[ \&T \not\equiv \&\text{Cell} \langle T \rangle \]

Needed: separate invariant for shared Cell
Sharing predicates

We have seen:

\[ T \equiv \text{Cell}^< T > \]

But we also know:

\[ T \not\equiv \& \text{Cell}^< T > \]

Needed: separate invariant for shared `Cell`

Semantic type consists of:

1. Ownership invariant: \([\tau].\text{own}(t,v)\)
Sharing predicates

We have seen:

\[ T \equiv \text{Cell} < T > \]

But we also know:

\[ T \not\equiv \& \text{Cell} < T > \]

Needed: separate invariant for shared Cell

Semantic type consists of:

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

We have seen:

\[ \tau \equiv \text{Cell} < \tau > \]

But we also know:

\[ \tau \not\equiv \& \text{Cell} < \tau > \]

Needed: separate invariant for shared Cell

Semantic type consists of:

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

What is \( [\text{Cell}(\tau)].\text{shr}(\kappa, t, \ell) \)?
impl<T> Cell<T> {
    pub fn set(&self, val: T) {
        unsafe {
            let value_ptr : *mut T = self.value.get();
            *value_ptr = val;
        }
    }
}
impl<T> Cell<T> {
    pub fn set(&self, val: T) {
        unsafe {
            let value_ptr : *mut T = self.value.get();
            *value_ptr = val;
        }
    }
}
impl<T> Cell<T> {
    pub fn set(&self, val: T) {
        unsafe {
            let value_ptr : *mut T = self.value.get();
            *value_ptr = val;
        }
    }
}

Why is \texttt{Cell::set} safe?

- No concurrent access (\texttt{Cell} is not \texttt{Sync})
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
Remember: $\lceil \text{Cell}(\tau) \rceil . \text{shr}(\kappa, t, \ell)$ should not allow concurrent accesses.

$$
\lceil \text{Cell}(\tau) \rceil . \text{shr}(\kappa, t, \ell) := \text{???}
$$
Remember: $\llbracket \text{Cell}(\tau) \rrbracket \cdot \text{shr}(\kappa, t, \ell)$ should not allow concurrent accesses.

$$\llbracket \text{Cell}(\tau) \rrbracket \cdot \text{shr}(\kappa, t, \ell) := \&_{na}^{\kappa/t} (\exists v. \ell. \text{value} \mapsto v * \llbracket \tau \rrbracket \cdot \text{own}(t,v))$$
$[\text{Cell}(\tau)].\text{shr}(\kappa, t, \ell)$

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

$[\text{Cell}(\tau)].\text{shr}(\kappa, t, \ell) := \&^{\kappa/t}_{\text{na}} (\exists v. \ell.\text{value} \mapsto v \ast [\tau].\text{own}(t,v))$

Non-atomic borrow
Semantic well-typedness of Cell::set: \( \models \)

\[
\{ [\text{Cell}(\tau)].\text{shr}(\alpha, t, \text{self}) \ast [\tau].\text{own}(t, \text{val}) \} \\
\text{fn set(&'a self, val: T) {}

\}

\{ [\text{Cell}(\tau)].\text{shr}(\alpha, t, \text{self}) \}

\]
Semantic well-typedness of Cell::set: $\models$

```
fn set(&'a self, val: T) {
    { [Cell(τ)].shr(α, t, self) * [τ].own(t, val) * [α] }

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

    { [Cell(τ)].shr(α, t, self) * [α] }
}
```
Semantic well-typedness of Cell::set: ⊨

$$[[\text{Cell}(\tau)]]\text{.shr}(\kappa, t, \ell) := \&_{\kappa/t}^{\text{na}} (\exists v. \ell. \text{value} \mapsto v \ast [[\tau]].\text{own}(t, v))$$

$$\{[[\text{Cell}(\tau)]]\text{.shr}(\alpha, t, \text{self}) \ast [[\tau]].\text{own}(t, \text{val}) \ast [\alpha]\}$$

fn set(&’a self, val: T) {

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

$$\{[[\text{Cell}(\tau)]]\text{.shr}(\alpha, t, \text{self}) \ast [\alpha]\}$$
Semantic well-typedness of `Cell::set`:

\[
\boxed{\text{Cell}(\tau)}\text{.shr}(\kappa, t, \ell) := \&_{\kappa/t}^\text{na} (\exists v. \ell. \text{value} \rightarrow v \star [\tau]\text{.own}(t, v))
\]

\[
\boxed{\text{fn set}(\&'a \text{self}, \text{val}: \text{T}) \{ \boxed{\text{let value_ptr : } *\text{mut T} = \text{self}.\text{value}.\text{get}(); \text{\star value_ptr = val;} } \}}
\]
Semantic well-typedness of Cell::set: $\models$

$\llbracket \text{Cell}(\tau) \rrbracket . \text{shr}(\kappa, t, \ell) := \&_{\kappa/t}^{\text{na}} (\exists v. \ell. \text{value} \mapsto v \ast \llbracket \tau \rrbracket . \text{own}(t, v))$

$\{ \llbracket \text{Cell}(\tau) \rrbracket . \text{shr}(\alpha, t, \text{self}) \ast \llbracket \tau \rrbracket . \text{own}(t, \text{val}) \ast [\alpha] \}$

fn set(&'a self, val: T) {
    $\{ \llbracket \text{Cell}(\tau) \rrbracket . \text{shr}(\alpha, t, \text{self}) \ast \llbracket \tau \rrbracket . \text{own}(t, \text{val}) \ast [\alpha] \}$
    $\{ \text{self.value} \mapsto v' \ast \llbracket \tau \rrbracket . \text{own}(t, v') \ast \llbracket \tau \rrbracket . \text{own}(t, \text{val}) \}$
    let value_ptr : *mut T = self.value.get();
    *value_ptr = val;
}

$\{ \llbracket \text{Cell}(\tau) \rrbracket . \text{shr}(\alpha, t, \text{self}) \ast [\alpha] \}$
Semantic well-typedness of Cell::set: $\models$

\[
[\text{Cell}(\tau)].\text{shr}(\kappa, t, \ell) := \&_{\kappa/t}(\exists v. \ell.\text{value} \mapsto v \ast [\tau].\text{own}(t, v))
\]

\{
[\text{Cell}(\tau)].\text{shr}(\alpha, t, \text{self}) \ast [\tau].\text{own}(t, \text{val}) \ast [\alpha] \}

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

\{
[\text{Cell}(\tau)].\text{shr}(\alpha, t, \text{self}) \ast [\tau].\text{own}(t, \text{val}) \ast [\alpha] \}

\{
\text{self}.\text{value} \mapsto v' \ast [\tau].\text{own}(t, v') \ast [\tau].\text{own}(t, \text{val})\}

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

*value_ptr = val;

\{
\text{self}.\text{value} \mapsto \text{val} \ast [\tau].\text{own}(t, \text{val})\}

\}

\{
[\text{Cell}(\tau)].\text{shr}(\alpha, t, \text{self}) \ast [\alpha] \}

}
Semantic well-typedness of Cell::set: \( \implies \)

\[
\begin{align*}
\Cell(\tau) \cdot \shr(\kappa, t, \ell) &:= &\&_{na} (\exists v. \ell. value \mapsto v * \Cell(\tau) . \own(t, v)) \\
\{ \Cell(\tau) \cdot \shr(\alpha, t, self) * \Cell(\tau) . \own(t, val) * [\alpha] \}
\end{align*}
\]

fn set(&’a self, val: T) {
    \{ \Cell(\tau) \cdot \shr(\alpha, t, self) * \Cell(\tau) . \own(t, val) * [\alpha] \}
    \{ self.value \mapsto v’ * \Cell(\tau) . \own(t, v’) * \Cell(\tau) . \own(t, val) \}
    let value_ptr : *mut T = self.value.get();
    *value_ptr = val;
    \{ self.value \mapsto val * \Cell(\tau) . \own(t, val) \}
    \{ \Cell(\tau) \cdot \shr(\alpha, t, self) * [\alpha] \}
}\{ \Cell(\tau) \cdot \shr(\alpha, t, self) * [\alpha] \}
Depth 1000m: The deep end of the Abyss
Depth 1000m: The deep end of the Abyss RustBelt in Coq
Cell in Coq

\[
[\text{Cell}(\tau)].\text{own}(t, v) := \ [\tau].\text{own}(t, v) \\
[\text{Cell}(\tau)].\text{shr}(\kappa, t, \ell) := \ &\text{na} (\exists v. \ell. \text{value} \mapsto v \ast [\tau].\text{own}(t, v))
\]

Program Definition cell (ty : type) := |
    ty_size := ty.(ty_size);
    ty_own := ty.(ty_own);
    ty_shr \kappa tid \ell :=
        &\text{na}\{\kappa, \text{tid}, \text{shrN}@l\}
        (\exists v, l \mapsto v \ast ty.(ty\_own) tid v)
|} %I.
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 (_ f ret arg). inv_vec arg=>x.
    simpl_subst. iApply type_jump; [solve_typing..]].
    iIntros (???) "#LFT _ $ Hty".
    rewrite !tctx_interp_singleton /=. done.
Qed.
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"].
Lemma cell_replace_type ty ![TyWf ty]:
  typed_val (cell_replace ty) (fn(∀ α, @(α; &shr{α}{(cell ty), ty} → ty).

Proof.
  intros E L. iApply type_fn; ![solve_typing..]]. iIntros "=./ !#".
  iIntros (α f 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_lift alive tok α with "HE HL") as (q') "(Htok & HL & Hcclose1)"; ![solve_typing..]].
  iMod (na_bor_acc with "LFT Hc' Htok Htl") as "(Hc'→ & Htl & Hcclose2)"; ![solve_ndisj..]].
  iDestruct "Hc'→" as (vc') "[Hc'→ Hc'own]". iDestruct (ty_size_eq with "Hc'own") as "#>%".
  iDestruct "Hr" as "[Hr→ Hrt]". iDestruct "Hr→" as (vr) "[Hr→ Hrown]".
  iDestruct (ty_size_eq with "Hrown") as "->Heq". iDestruct "Heq" as %Heq.
  wp_apply (wp_memcmp with "[$Hr→ $Hc'→]"). { by rewrite Heq. } { f_equal. done. }
  iIntros "[Hr→ Hc'→]". wp_seq. iDestruct "Hx" as "[Hx→ Hxt]".
  iDestruct "Hx→" as (vx) "[Hx→ Hxown]". iDestruct (ty_size_eq with "Hxown") as "#%".
  wp_apply (wp_memcmp with "[$Hc'→ $Hx→]"); try by f_equal. iIntros "[Hc'→ Hx→]". wp_seq.
  iMod ("Hcclose2" with "[Hc'→ Hxown] Htl") as "[Htok Htl]"; first by auto with iFrame.
  iMod ("Hcclose1" 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→ Hxt".
    - 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!