



Rustbelt

Rick Koenders, Mees Meuwissen

December 17th, 2020



Outline

Semantic Model

Lifetime Logic

Interior Mutability: Cell and Mutex





Semantic Model





Semantic typing

- What is a type?





Semantic typing

- What is a type?
- Fixed size $\llbracket \tau \rrbracket.size$
- $\llbracket \tau \rrbracket.own(t, \bar{v}) \Rightarrow |\bar{v}| = \llbracket \tau \rrbracket.size$





Semantic typing

- What is a type?
- Fixed size $\llbracket \tau \rrbracket.size$
- $\llbracket \tau \rrbracket.own(t, \bar{v}) \Rightarrow |\bar{v}| = \llbracket \tau \rrbracket.size$
- Example: **bool**
 - $\llbracket \mathbf{bool} \rrbracket.size := 1$
 - $\llbracket \mathbf{bool} \rrbracket.own(t, \bar{v}) := \bar{v} = [\mathbf{true}] \vee \bar{v} = [\mathbf{false}]$





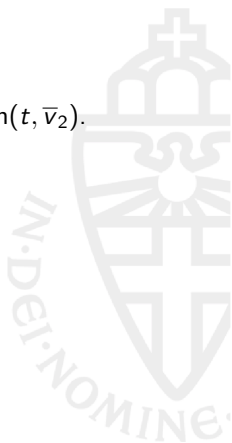
Semantic typing

- Product type

- $\llbracket \tau_1 \times \tau_2 \rrbracket . \text{size} := \llbracket \tau_1 \rrbracket . \text{size} + \llbracket \tau_2 \rrbracket . \text{size}$

- $\llbracket \tau_1 \times \tau_2 \rrbracket . \text{own}(t, \bar{v}) :=$

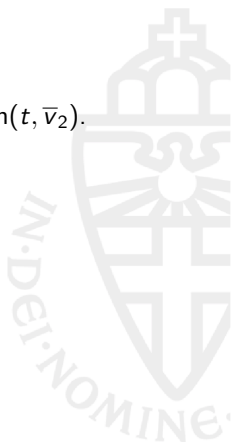
$$\exists \bar{v}_1, \bar{v}_2. \bar{v} = \bar{v}_1 \# \bar{v}_2 * \llbracket \tau_1 \rrbracket . \text{own}(t, \bar{v}_1) * \llbracket \tau_2 \rrbracket . \text{own}(t, \bar{v}_2).$$





Semantic typing

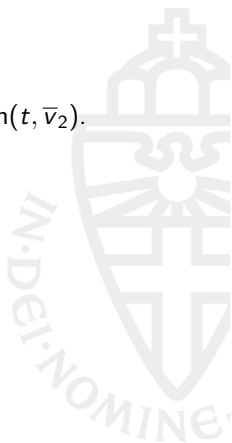
- Product type
 - $\llbracket \tau_1 \times \tau_2 \rrbracket . \text{size} := \llbracket \tau_1 \rrbracket . \text{size} + \llbracket \tau_2 \rrbracket . \text{size}$
 - $\llbracket \tau_1 \times \tau_2 \rrbracket . \text{own}(t, \bar{v}) :=$
 $\exists \bar{v}_1, \bar{v}_2. \bar{v} = \bar{v}_1 \# \bar{v}_2 * \llbracket \tau_1 \rrbracket . \text{own}(t, \bar{v}_1) * \llbracket \tau_2 \rrbracket . \text{own}(t, \bar{v}_2).$
- Owned pointers
 - $\llbracket \text{own}_n \tau \rrbracket . \text{size} = 1$
 - $\llbracket \text{own}_n \tau \rrbracket . \text{own}(t, \bar{v}) :=$
 $\exists \ell. \bar{v} = [\ell] * \exists \bar{w}. \ell \mapsto \bar{w} * \triangleright \llbracket \tau \rrbracket . \text{own}(t, \bar{w}) *$
 $\triangleright \text{DeallocSize}(\ell, n, \llbracket \tau \rrbracket . \text{size}).$





Semantic typing

- Product type
 - $\llbracket \tau_1 \times \tau_2 \rrbracket . \text{size} := \llbracket \tau_1 \rrbracket . \text{size} + \llbracket \tau_2 \rrbracket . \text{size}$
 - $\llbracket \tau_1 \times \tau_2 \rrbracket . \text{own}(t, \bar{v}) := \exists \bar{v}_1, \bar{v}_2. \bar{v} = \bar{v}_1 \uparrow\uparrow \bar{v}_2 * \llbracket \tau_1 \rrbracket . \text{own}(t, \bar{v}_1) * \llbracket \tau_2 \rrbracket . \text{own}(t, \bar{v}_2)$.
- Owned pointers
 - $\llbracket \text{own}_n \tau \rrbracket . \text{size} = 1$
 - $\llbracket \text{own}_n \tau \rrbracket . \text{own}(t, \bar{v}) := \exists \ell. \bar{v} = [\ell] * \exists \bar{w}. \ell \mapsto \bar{w} * \triangleright \llbracket \tau \rrbracket . \text{own}(t, \bar{w}) * \triangleright \text{DeallocSize}(\ell, n, \llbracket \tau \rrbracket . \text{size})$.
- Mutable references
 - $\llbracket \&_{\text{mut}}^{\kappa} \tau \rrbracket . \text{size} := 1$.
 - $\llbracket \&_{\text{mut}}^{\kappa} \tau \rrbracket . \text{own}(t, \bar{v}) := \exists \ell. \bar{v} = [\ell] * \&_{\text{full}}^{\llbracket \kappa \rrbracket} (\exists \bar{w}. \ell \mapsto \bar{w} * \llbracket \tau \rrbracket . \text{own}(t, \bar{w}))$.





Interpreting shared references

- Cell, Mutex create problems





Interpreting shared references

- Cell, Mutex create problems
- Introduce $\llbracket \tau \rrbracket . \text{shr}(\llbracket \kappa \rrbracket, t, \ell)$ predicate





Interpreting shared references

- Cell, Mutex create problems
- Introduce $\llbracket \tau \rrbracket. \text{shr}(\llbracket \kappa \rrbracket, t, \ell)$ predicate
 - $\llbracket \&_{\text{shr}}^{\kappa} \tau \rrbracket. \text{size} := 1$
 - $\llbracket \&_{\text{shr}}^{\kappa} \tau \rrbracket. \text{own}(t, \bar{v}) := \exists \ell. \bar{v} = [\ell] * \llbracket \tau \rrbracket. \text{shr}(\llbracket \kappa \rrbracket, t, \ell)$





Interpreting shared references

- Cell, Mutex create problems
- Introduce $\llbracket \tau \rrbracket. \text{shr}(\llbracket \kappa \rrbracket, t, \ell)$ predicate
 - $\llbracket \&_{\text{shr}}^{\kappa} \tau \rrbracket. \text{size} := 1$
 - $\llbracket \&_{\text{shr}}^{\kappa} \tau \rrbracket. \text{own}(t, \bar{v}) := \exists \ell. \bar{v} = [\ell] * \llbracket \tau \rrbracket. \text{shr}(\llbracket \kappa \rrbracket, t, \ell)$
- $\text{persistent}(\llbracket \tau \rrbracket. \text{shr}(\kappa, t, \ell))$





Interpreting shared references

- Cell, Mutex create problems
- Introduce $\llbracket \tau \rrbracket. \text{shr}(\llbracket \kappa \rrbracket, t, \ell)$ predicate
 - $\llbracket \&_{\text{shr}}^{\kappa} \tau \rrbracket. \text{size} := 1$
 - $\llbracket \&_{\text{shr}}^{\kappa} \tau \rrbracket. \text{own}(t, \bar{v}) := \exists \ell. \bar{v} = [\ell] * \llbracket \tau \rrbracket. \text{shr}(\llbracket \kappa \rrbracket, t, \ell)$
- $\text{persistent}(\llbracket \tau \rrbracket. \text{shr}(\kappa, t, \ell))$
- $\&_{\text{full}}^{\kappa}(\exists \bar{w}. \ell \mapsto \bar{w} * \llbracket \tau \rrbracket. \text{own}(t, \bar{w})) * [\kappa]_q \equiv * (\llbracket \tau \rrbracket. \text{shr}(\kappa, t, \ell) * [\kappa]_q)$





Interpreting shared references

- Cell, Mutex create problems
- Introduce $\llbracket \tau \rrbracket. \text{shr}(\llbracket \kappa \rrbracket, t, \ell)$ predicate
 - $\llbracket \&_{\text{shr}}^{\kappa} \tau \rrbracket. \text{size} := 1$
 - $\llbracket \&_{\text{shr}}^{\kappa} \tau \rrbracket. \text{own}(t, \bar{v}) := \exists \ell. \bar{v} = [\ell] * \llbracket \tau \rrbracket. \text{shr}(\llbracket \kappa \rrbracket, t, \ell)$
- $\text{persistent}(\llbracket \tau \rrbracket. \text{shr}(\kappa, t, \ell))$
- $\&_{\text{full}}^{\kappa}(\exists \bar{w}. \ell \mapsto \bar{w} * \llbracket \tau \rrbracket. \text{own}(t, \bar{w})) * [\kappa]_q \equiv * (\llbracket \tau \rrbracket. \text{shr}(\kappa, t, \ell) * [\kappa]_q)$
- $\kappa' \sqsubseteq \kappa \wedge \llbracket \tau \rrbracket. \text{shr}(\kappa, t, \ell) \Rightarrow \llbracket \tau \rrbracket. \text{shr}(\kappa', t, \ell)$





Sharing predicates

- $\llbracket \tau \rrbracket = (\text{size}, \text{own}, \text{shr})$





Sharing predicates

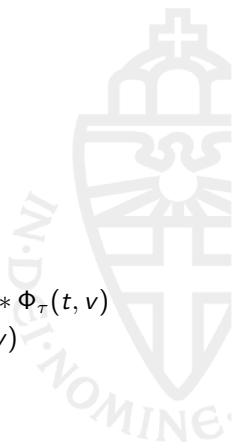
- $\llbracket \tau \rrbracket = (\text{size}, \text{own}, \text{shr})$
- Product Types
 - $\llbracket \tau_1 \times \tau_2 \rrbracket . \text{shr}(\kappa, t, \ell) :=$
 $\llbracket \tau_1 \rrbracket . \text{shr}(\kappa, t, \ell) * \llbracket \tau_2 \rrbracket . \text{shr}(\kappa, t, \ell + \llbracket \tau_1 \rrbracket . \text{size})$





Sharing predicates

- $\llbracket \tau \rrbracket = (\text{size}, \text{own}, \text{shr})$
- Product Types
 - $\llbracket \tau_1 \times \tau_2 \rrbracket. \text{shr}(\kappa, t, \ell) :=$
 $\llbracket \tau_1 \rrbracket. \text{shr}(\kappa, t, \ell) * \llbracket \tau_2 \rrbracket. \text{shr}(\kappa, t, \ell + \llbracket \tau_1 \rrbracket. \text{size})$
- Simple Types
 - Persistent predicate $\Phi_\tau(t, v)$
 - For types of size 1: $\llbracket \tau \rrbracket. \text{own}(t, \bar{v}) := \exists v. \bar{v} = [v] * \Phi_\tau(t, v)$
 - $\llbracket \tau \rrbracket. \text{shr}(\kappa, t, \ell) := \exists v. \&_{\text{frac}}^\kappa (\lambda q. \ell \xrightarrow{q} v) * \triangleright \Phi_\tau(t, v)$





Lifetime Logic





Lifetimes

- Variables have lifetimes
- Lifetime tokens: $[\kappa]_q$
- Reasoning about lifetime tokens: Lifetime logic





Example program

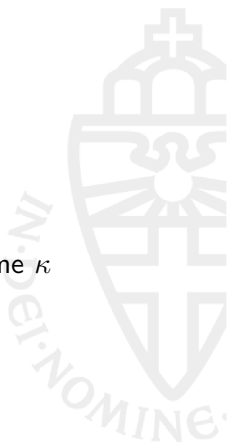
```
let mut v = Vec::new();
v.push(0);
{
    let mut head = v.index_mut(0);
    *head = 23;
}
println!("{:?}", v);
```





Lifetime rules: LFTL-BEGIN

- $\text{True} \equiv * \exists \kappa. [\kappa]_1 * ([\kappa]_1 \triangleright \equiv * [\dagger \kappa])$
- True: We can always do this
- After update: Create lifetime κ
- Separately we also get a method to end the lifetime κ





Lifetime rules: LFTL-BORROW

- $\triangleright P \equiv * \&_{\text{full}}^{\kappa} P * ([\uparrow \kappa] \equiv * \triangleright P)$
- Need ownership of P
- Get full borrow to P with lifetime κ
- We get P back once κ is dead





Lifetime rules: LFTL-BOR-SPLIT

- $\&_{\text{full}}^{\kappa}(P * Q) \equiv * \&_{\text{full}}^{\kappa}(P) * \&_{\text{full}}^{\kappa}(Q)$
- Split up a full borrow for separate parts of memory
- Get two separate full borrows





Lifetime rules: LFTL-BOR-ACC

- $\&_{\text{full}}^{\kappa} P * [\kappa]_q \Rightarrow * \triangleright P * (\triangleright P \Rightarrow * \&_{\text{full}}^{\kappa} P * [\kappa]_q)$
- Needed: full borrow and a valid lifetime token
- Get: Access and method to give up access
- Separating conjunction over time!





Example program: revisited

```
let mut v = Vec::new();
v.push(0);
{
    let mut head = v.index_mut(0);
    *head = 23;
}
println!("{:?}", v);
```

- $\text{True} \equiv * \exists \kappa. [\kappa]_1 * ([\kappa]_1 \triangleright \equiv * [\dagger \kappa])$





Example program: revisited

```
let mut v = Vec::new();
v.push(0);
{
    let mut head = v.index_mut(0);
    *head = 23;
}
println!("{:?}", v);
```

- $\triangleright P \equiv * \&_{\text{full}}^{\kappa} P * ([\dagger \kappa] \equiv * \triangleright P)$





Example program: revisited

```
let mut v = Vec::new();
v.push(0);
{
    let mut head = v.index_mut(0);
    *head = 23;
}
println!("{:?}", v);
```

- $\&_{\text{full}}^{\kappa}(P * Q) \Rightarrow * \&_{\text{full}}^{\kappa}(P) * \&_{\text{full}}^{\kappa}(Q)$





Example program: revisited

```
let mut v = Vec::new();
v.push(0);
{
    let mut head = v.index_mut(0);
    *head = 23;
}
println!("{:?}", v);
```

- $\&_{\text{full}}^{\kappa} P * [\kappa]_q \Rightarrow * \triangleright P * (\triangleright P \Rightarrow * \&_{\text{full}}^{\kappa} P * [\kappa]_q)$





Example program: revisited

```
let mut v = Vec::new();
v.push(0);
{
    let mut head = v.index_mut(0);
    *head = 23;
}
println!("{:?}", v);
```

- $\text{True} \equiv * \exists \kappa. [\kappa]_1 * ([\kappa]_1 \triangleright \equiv * [\dagger \kappa])$





Example program: revisited

```
let mut v = Vec::new();
v.push(0);
{
    let mut head = v.index_mut(0);
    *head = 23;
}
println!("{:?}", v);
```

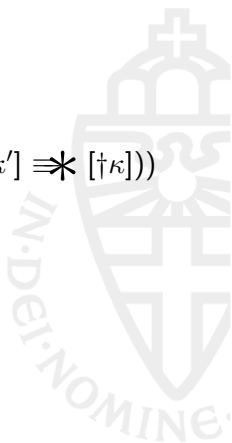
- $\triangleright P \equiv * \&_{\text{full}}^{\kappa} P * ([\dagger\kappa] \equiv * \triangleright P)$





Lifetime inclusion

- $\kappa \sqsubseteq \kappa' :=$
 $\Box((\forall q. [\kappa]_q \Rightarrow^* \exists q'. [\kappa']_{q'} * ([\kappa']_{q'} \Rightarrow^* [\kappa]_q)) * ([\dagger \kappa'] \Rightarrow^* [\dagger \kappa]))$
- Examples of rules with \sqsubseteq :
 - $\kappa \sqcap \kappa' \sqsubseteq \kappa$
 - $\kappa \sqsubseteq \kappa' * \kappa \sqsubseteq \kappa'' \implies \kappa \sqsubseteq \kappa' \sqcap \kappa''$





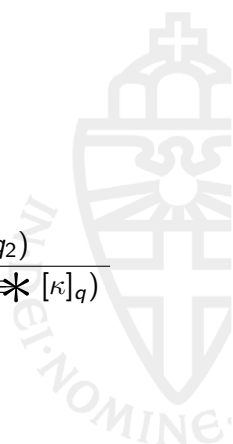
Fractured borrows

- LFTL-BOR-FRACTURE:

$$\&_{\text{full}}^{\kappa} \Phi(1) \equiv * \&_{\text{frac}}^{\kappa} \Phi$$

- LFTL-FRACT-ACC:

$$\frac{\forall q_1, q_2. \Phi(q_1 + q_2) \iff \Phi(q_1) * \Phi(q_2)}{\&_{\text{frac}}^{\kappa} \Phi * [\kappa]_q \equiv * \exists q'. \triangleright \Phi(q') * (\triangleright \Phi(q')) \equiv * [\kappa]_q}$$





Interior Mutability: Cell and Mutex



Interior mutability

- Libraries can provide mutable shared references
- Use the keyword `unsafe` internally
- Can still be safe to use!
- Examples: `Cell`, `Mutex`, `Rc`, `Arc`
- Different sharing predicate for every library





Cell

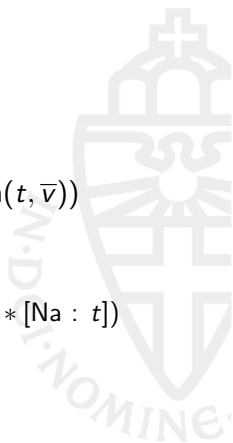
- `Cell<T>` stores a value of type `T`
- Provides `get` and `set` function
- Not thread safe





Encoding Cell

- $\llbracket \mathbf{cell}(\tau) \rrbracket . \text{size} := \llbracket \tau \rrbracket . \text{size}$
- $\llbracket \mathbf{cell}(\tau) \rrbracket . \text{own}(t, \bar{v}) := \llbracket \tau \rrbracket . \text{own}(t, \bar{v})$
- $\llbracket \mathbf{cell}(\tau) \rrbracket . \text{shr}(\kappa, t, \ell) := \&_{\text{na}}^{\kappa/t} (\exists \bar{v}. \ell \mapsto \bar{v} * \llbracket \tau \rrbracket . \text{own}(t, \bar{v}))$
- Rules for non-atomic persistent borrows:
 - $\&_{\text{full}}^{\kappa} P \equiv * \&_{\text{na}}^{\kappa/t} P$
 - $\&_{\text{na}}^{\kappa/t} P * [\kappa]_q * [\text{Na} : t] \equiv * \triangleright P * (\triangleright P \equiv * [\kappa]_q * [\text{Na} : t])$





Encoding Mutex

- $\llbracket \text{mutex}(\tau) \rrbracket . \text{size} := 1 + \llbracket \tau \rrbracket . \text{size}$
- $\llbracket \text{mutex}(\tau) \rrbracket . \text{own}(t, \bar{v}) := \llbracket \text{bool} \times \tau \rrbracket . \text{own}(t, \bar{v})$





Encoding Mutex

- $\llbracket \text{mutex}(\tau) \rrbracket. \text{size} := 1 + \llbracket \tau \rrbracket. \text{size}$
- $\llbracket \text{mutex}(\tau) \rrbracket. \text{own}(t, \bar{v}) := \llbracket \text{bool} \times \tau \rrbracket. \text{own}(t, \bar{v})$
- Atomic persistent borrow
- $\llbracket \text{mutex}(\tau) \rrbracket. \text{shr}(\kappa, t, \ell) := \exists \kappa'. \kappa \sqsubseteq \kappa' * \&_{\text{at}}^{\kappa}(\ell \mapsto \text{true} \vee \ell \mapsto \text{false} * \&_{\text{full}}^{\kappa'}(\exists \bar{v}. (\ell + 1) \mapsto \bar{v} * \llbracket \tau \rrbracket. \text{own}(t, \bar{v})))$
- Note the $\&_{\text{full}}^{\kappa'}$



Presentation has ended
Time to ask questions

