Rustbelt

Rick Koenders, Mees Meuwissen

December 17th, 2020

Outline

Semantic Model

Semantic typing

- What is a type?

Semantic typing

- What is a type?
- Fixed size $[\![\tau]\!].\mathsf{size}$
- $[\![\tau]\!].\mathsf{own}(t, \overline{v}) \Rightarrow |\overline{v}| = [\![\tau]\!].\mathsf{size}$

Semantic typing

- What is a type?
- Fixed size $[\![\tau]\!].\text{size}$
- $[\![\tau]\!].\text{own}(t, \overline{v}) \Rightarrow |\overline{v}| = [\![\tau]\!].\text{size}$
- Example: **bool**
  - $[\![\textbf{bool}]\!].\text{size} := 1$
  - $[\![\textbf{bool}]\!].\text{own}(t, \overline{v}) := \overline{v} = [\textbf{true}] \lor \overline{v} = [\textbf{false}]$

Semantic typing

- Product type
  - $[\![\tau_1 \times \tau_2]\!].\mathsf{size} := [\![\tau_1]\!].\,\mathsf{size} + [\![\tau_2]\!].\,\mathsf{size}$
  - $[\![\tau_1 \times \tau_2]\!].\,\mathsf{own}(t, \overline{v}) :=$
    $\exists \overline{v}_1, \overline{v}_2. \overline{v} = \overline{v}_1 +\!\!+ \overline{v}_2 * [\![\tau_1]\!].\,\mathsf{own}(t, \overline{v}_1) * [\![\tau_2]\!].\,\mathsf{own}(t, \overline{v}_2).$

Semantic typing

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

Semantic typing

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

Interpreting shared references

- Cell, Mutex create problems

Interpreting shared references

- Cell, Mutex create problems
- Introduce $[\![\tau]\!].\,\mathsf{shr}([\![\kappa]\!], t, \ell)$ predicate

Interpreting shared references

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

Interpreting shared references

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

Interpreting shared references

- Cell, Mutex create problems
- Introduce $[\![\tau]\!].\,\mathsf{shr}([\![\kappa]\!], t, \ell)$ predicate
  - $[\![\&^{\kappa}_{\mathbf{shr}}\tau]\!].\,\mathsf{size} := 1$
  - $[\![\&^{\kappa}_{\mathbf{shr}}\tau]\!].\,\mathsf{own}(t, \overline{v}) := \exists \ell. \overline{v} = [\ell] * [\![\tau]\!].\,\mathsf{shr}([\![\kappa]\!], t, \ell)$
- $\mathsf{persistent}([\![\tau]\!].\,\mathsf{shr}(\kappa, t, \ell))$
- $\&^{\kappa}_{\mathsf{full}}(\exists \overline{w}. \ell \mapsto \overline{w} * [\![\tau]\!].\,\mathsf{own}(t, \overline{w})) * [\kappa]_q) \Rrightarrow$
  $([\![\tau]\!].\,\mathsf{shr}(\kappa, t, \ell) * [\kappa]_q)$

Interpreting shared references

- Cell, Mutex create problems
- Introduce $[\![\tau]\!].\,\mathsf{shr}([\![\kappa]\!], t, \ell)$ predicate
  - $[\![\&^{\kappa}_{\mathbf{shr}}\tau]\!].\,\mathsf{size} := 1$
  - $[\![\&^{\kappa}_{\mathbf{shr}}\tau]\!].\,\mathsf{own}(t, \overline{v}) := \exists \ell. \overline{v} = [\ell] * [\![\tau]\!].\,\mathsf{shr}([\![\kappa]\!], t, \ell)$
- $\mathsf{persistent}([\![\tau]\!].\,\mathsf{shr}(\kappa, t, \ell))$
- $\&^{\kappa}_{\mathsf{full}}(\exists \overline{w}. \ell \mapsto \overline{w} * [\![\tau]\!].\,\mathsf{own}(t, \overline{w})) * [\kappa]_q) \Rrightarrow$
  $([\![\tau]\!].\,\mathsf{shr}(\kappa, t, \ell) * [\kappa]_q)$
- $\kappa' \sqsubseteq \kappa \land [\![\tau]\!].\,\mathsf{shr}(\kappa, t, \ell) \Rightarrow [\![\tau]\!].\,\mathsf{shr}(\kappa', t, \ell)$

Sharing predicates

- $[\![\tau]\!] = (\mathsf{size}, \mathsf{own}, \mathsf{shr})$

Sharing predicates

- $[\![\tau]\!] = (\mathsf{size}, \mathsf{own}, \mathsf{shr})$
- Product Types
  - $[\![\tau_1 \times \tau_2]\!].\,\mathsf{shr}(\kappa, t, \ell) :=$
    $[\![\tau_1]\!].\,\mathsf{shr}(\kappa, t, \ell) * [\![\tau_2]\!].\,\mathsf{shr}(\kappa, t, \ell + [\![\tau_1]\!].\,\mathsf{size})$

Sharing predicates

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

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

- True $\Rrightarrow \exists \kappa.[\kappa]_1 * ([\kappa]_1 \rhd \Rrightarrow [\dagger \kappa])$
- 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

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

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

Lifetime rules: LFTL-BOR-ACC

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

- True $\Rrightarrow \exists \kappa.[\kappa]_1 * ([\kappa]_1 \ \triangleright \Rrightarrow [\dagger \kappa])$

Example program: revisited

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

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

Example program: revisited

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

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

Example program: revisited

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

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

Example program: revisited

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

- True $\Rrightarrow \exists \kappa.[\kappa]_1 * ([\kappa]_1 \ \triangleright \Rrightarrow [\dagger \kappa])$

Example program: revisited

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

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

Lifetime inclusion

- $\kappa \sqsubseteq \kappa' :=$
  $\Box((\forall q.[\kappa]_q \Rrightarrow \exists q'.[\kappa']_{q'} * ([\kappa']_{q'} \Rrightarrow [\kappa]_q)) * ([\dagger\kappa'] \Rrightarrow [\dagger\kappa]))$
- 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:
  $\&^{\kappa}_{\mathsf{full}}\Phi(1) \Rrightarrow \&^{\kappa}_{\mathsf{frac}}\Phi$
- LFTL-FRACT-ACC:

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

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

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

Encoding Mutex

- $[\![\mathbf{mutex}(\tau)]\!].\,\mathsf{size} := 1 + [\![\tau]\!].\,\mathsf{size}$
- $[\![\mathbf{mutex}(\tau)]\!].\,\mathsf{own}(t, \overline{v}) := [\![\mathbf{bool} \times \tau]\!].\,\mathsf{own}(t, \overline{v})$

Encoding Mutex

- $[\![\textbf{mutex}(\tau)]\!].\,\mathsf{size} := 1 + [\![\tau]\!].\,\mathsf{size}$
- $[\![\textbf{mutex}(\tau)]\!].\,\mathsf{own}(t, \overline{v}) := [\![\textbf{bool} \times \tau]\!].\,\mathsf{own}(t, \overline{v})$
- Atomic persistent borrow
- $[\![\textbf{mutex}(\tau)]\!].\,\mathsf{shr}(\kappa, t, \ell) := \exists \kappa'.\kappa \sqsubseteq \kappa' *$
  $\&^{\kappa}_{\mathsf{at}}(\ell \mapsto \textbf{true} \lor$
  $\ell \mapsto \textbf{false} * \&^{\kappa'}_{\mathsf{full}}(\exists \overline{v}.(\ell + 1) \mapsto \overline{v} * [\![\tau]\!].\,\mathsf{own}(t, \overline{v})))$
- Note the $\&^{\kappa'}_{\mathsf{full}}$

Presentation has ended

Time to ask questions