

Corten: Refinement Types for Imperative Languages with Ownership

Abschlusspräsentation Masterarbeit

Carsten Csiky | 26th Oktober 2022

Inhaltsverzeichnis

1. Motivation

2. Empirical Analysis

3. Type System

4. Soundness Justification

5. Related Work

6. Conclusion / Future Work

Motivation
○○○○○○○

Empirical Analysis
○

Type System
○○○○○○○○○○○○○

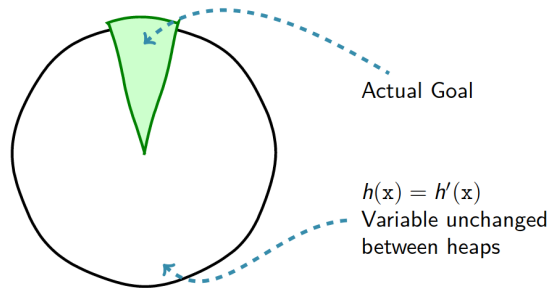
Soundness Justification
○○○

Related Work
○○

Conclusion / Future Work
○○

Motivation

```
public IntList square(IntList list) {
    return list.map(x -> x*x);
}
```



Motivation

```
fn max(a: i32, b: i32) {
  if a > b { a } else { b }
}
```

Motivation
 ●○○○○○

Empirical Analysis
 ○

Type System
 ○○○○○○○○○○○

Soundness Justification
 ○○○

Related Work
 ○○

Conclusion / Future Work
 ○○

Motivation

```
fn max(a: i32, b: i32) {
  if a > b { a } else { b }
}
```

■ Return Value (v) : $v \geq a \wedge v \geq b$

Motivation

```
fn max(a: i32, b: i32) {
  if a > b { a } else { b }
}
```

- Return Value (v) : $v \geq a \wedge v \geq b$
- Rondon et al. [RKJ08]: Refinement Types for Functional Programming Languages

Motivation

```
//@ max(a: i32, b: i32) -> {v:i32 | v >= a && v >= b }
fn max(a: i32, b: i32) -> i32 {
  if a > b { a } else { b }
}
```

Motivation

```
//@ max(a: i32, b: i32) -> {v:i32 | v >= a && v >= b }
fn max(a: i32, b: i32) -> i32 {
  if a > b { a } else { b }
}
```

let $\Gamma = (a : \{v : i32 \mid \text{true}\}, b : \{v : i32 \mid \text{true}\})$ and $\tau = \{v : i32 \mid v \geq a \wedge v \geq b\}$

$$\Gamma \vdash \text{if } a > b \{a\} \text{ else } \{b\} : \tau$$

Motivation

```
//@ max(a: i32, b: i32) -> {v:i32 | v >= a && v >= b }
fn max(a: i32, b: i32) -> i32 {
  if a > b { a } else { b }
}
```

let $\Gamma = (a : \{v : i32 \mid \text{true}\}, b : \{v : i32 \mid \text{true}\})$ and $\tau = \{v : i32 \mid v \geq a \wedge v \geq b\}$

$$\frac{\Gamma, a > b \vdash a : \tau \quad \Gamma, \neg(a > b) \vdash b : \tau}{\Gamma \vdash \text{if } a > b \{a\} \text{ else } \{b\} : \tau}$$

Motivation

```
//@ max(a: i32, b: i32) -> {v:i32 | v >= a && v >= b }
fn max(a: i32, b: i32) -> i32 {
  if a > b { a } else { b }
}
```

let $\Gamma = (a : \{v : i32 \mid \text{true}\}, b : \{v : i32 \mid \text{true}\})$ and $\tau = \{v : i32 \mid v \geq a \wedge v \geq b\}$

$$\frac{\frac{\Gamma, a > b \vdash \{v : i32 \mid v \doteq a\} \preceq \tau}{\Gamma, a > b \vdash a : \tau} \quad \Gamma, \neg(a > b) \vdash b : \tau}{\Gamma \vdash \text{if } a > b \{a\} \text{ else } \{b\} : \tau}$$

Motivation

```
//@ max(a: i32, b: i32) -> {v:i32 | v >= a && v >= b }
fn max(a: i32, b: i32) -> i32 {
  if a > b { a } else { b }
}
```

let $\Gamma = (a : \{v : i32 \mid \text{true}\}, b : \{v : i32 \mid \text{true}\})$ and $\tau = \{v : i32 \mid v \geq a \wedge v \geq b\}$

$$\frac{\star \quad \frac{\Gamma, a > b \vdash a : \{v : i32 \mid v \doteq a\}}{\Gamma, a > b \vdash a : \tau} \quad \frac{\Gamma, a > b \vdash \{v : i32 \mid v \doteq a\} \preceq \tau}{\Gamma, a > b \vdash a : \tau}}{\Gamma \vdash \text{if } a > b \{a\} \text{ else } \{b\} : \tau} \quad \frac{}{\Gamma, \neg(a > b) \vdash b : \tau}$$

Motivation

```
//@ max(a: i32, b: i32) -> {v:i32 | v >= a && v >= b }
fn max(a: i32, b: i32) -> i32 {
  if a > b { a } else { b }
}
```

let $\Gamma = (a : \{v : i32 \mid \text{true}\}, b : \{v : i32 \mid \text{true}\})$ and $\tau = \{v : i32 \mid v \geq a \wedge v \geq b\}$

$$\frac{\star \quad \text{SMT-VALID} \left(\begin{array}{l} \text{true} \wedge \text{true} \wedge a > b \\ \wedge v \doteq a \\ \implies (v \geq a \wedge v \geq b) \end{array} \right)}{\frac{\Gamma, a > b \vdash a : \{v : i32 \mid v \doteq a\} \quad \Gamma, a > b \vdash \{v : i32 \mid v \doteq a\} \preceq \tau}{\Gamma, a > b \vdash a : \tau}} \quad \frac{}{\Gamma, \neg(a > b) \vdash b : \tau}$$

$$\Gamma \vdash \text{if } a > b \{a\} \text{ else } \{b\} : \tau$$

Motivation

```
//@ max(a: i32, b: i32) -> {v:i32 | v >= a && v >= b }
fn max(a: i32, b: i32) -> i32 {
  if a > b { a } else { b }
}
```

let $\Gamma = (a : \{v : i32 \mid \text{true}\}, b : \{v : i32 \mid \text{true}\})$ and $\tau = \{v : i32 \mid v \geq a \wedge v \geq b\}$

$$\frac{\star \quad \frac{\Gamma, a > b \vdash a : \{v : i32 \mid v \doteq a\}}{\Gamma, a > b \vdash a : \tau} \quad \frac{\text{SMT-VALID} \left(\begin{array}{l} \text{true} \wedge \text{true} \wedge a > b \\ \wedge v \doteq a \\ \implies (v \geq a \wedge v \geq b) \end{array} \right) \quad \Gamma, a > b \vdash \{v : i32 \mid v \doteq a\} \preceq \tau}{\Gamma, a > b \vdash a : \tau} \quad \vdots}{\Gamma \vdash \text{if } a > b \{a\} \text{ else } \{b\} : \tau}$$

Motivation

```
fn clamp(a: &mut i32, b: i32) {
    if *a > b { *a = b }
}
```

Motivation
○○○●○○

Empirical Analysis
○

Type System
○○○○○○○○○○○○

Soundness Justification
○○○

Related Work
○○

Conclusion / Future Work
○○

Motivation

```

fn clamp(a: &mut i32, b: i32) {
    if *a > b { *a = b }
}

fn client(...) {
    ...
    clamp(&mut x, 5);
    clamp(&mut y, 6);
    print!(x);
    ...
}

```

Motivation
○○○●○○

Empirical Analysis
○

Type System
○○○○○○○○○○

Soundness Justification
○○○

Related Work
○○

Conclusion / Future Work
○○

Motivation

```
fn clamp(a: &mut i32, b: i32) {
    if *a > b { *a = b }
}

fn client(...) {
    ...
    clamp(&mut x, 5);
    clamp(&mut y, 6);
    print!(x);
    ...
}
```

What does this it print(x) output?

- In most imperative programming languages:
 - Could be: old x or 5

Motivation

```
fn clamp(a: &mut i32, b: i32) {
    if *a > b { *a = b }
}

fn client(...) {
    ...
    clamp(&mut x, 5);
    clamp(&mut y, 6);
    print!(x);
    ...
}
```

What does this `print(x)` output?

- In most imperative programming languages:
 - Could be: old `x` or 5
 - But also 6 (if `x` aliases with `y`)!

Motivation

```
fn clamp(a: &mut i32, b: i32) {
    if *a > b { *a = b }
}

fn client(...) {
    ...
    clamp(&mut x, 5);
    clamp(&mut y, 6);
    print!(x);
    ...
}
```

What does this `print(x)` output?

- In most imperative programming languages:
 - Could be: old x or 5
 - But also 6 (if x aliases with y)!
- In Rust:
 - Just old x or 5
 - And nothing else!

Motivation

```
fn clamp(a: &mut i32, b: i32) {
    // borrows a
    // owns b
    if *a > b { *a = b }
    // "returns" the borrow of a
}

fn client(...) { // owns x, y
    ...
    clamp(&mut x, 5); // lend x mutably
    clamp(&mut y, 6); // lend y mutably
    print!(x);
    ...
}
```

Ownership in Rust: Mutability XOR Aliasing

Each lexical scope tracks permissions for visible memory objects. Possible Permission Levels:

- Owner (e.g. b)
 - can: read, write
 - transfer ownership (if no outstanding borrows)
- Mutable Reference (e.g. &mut x)
 - can: read, write
 - guarantee: no aliasing
- Immutable Reference (e.g. &v)
 - can: read, alias
 - guarantee: no mutation

Motivation

Consequences:

- unique data owner
- no global, mutable state
- no cycles in memory structure

Ownership in Rust: Mutability XOR Aliasing

Each lexical scope tracks permissions for visible memory objects. Possible Permission Levels:

- Owner (e.g. `b`)
 - can: read, write
 - transfer ownership (if no outstanding borrows)
- Mutable Reference (e.g. `&mut x`)
 - can: read, write
 - guarantee: no aliasing
- Immutable Reference (e.g. `&v`)
 - can: read, alias
 - guarantee: no mutation

Motivation

Consequences:

- unique data owner
- no global, mutable state
- no cycles in memory structure

Used for:

- safe non-gc memory management
- safe concurrency
- safe low-level hardware access
- ...

Ownership in Rust: Mutability XOR Aliasing

Each lexical scope tracks permissions for visible memory objects. Possible Permission Levels:

- Owner (e.g. `b`)
 - can: read, write
 - transfer ownership (if no outstanding borrows)
- Mutable Reference (e.g. `&mut x`)
 - can: read, write
 - guarantee: no aliasing
- Immutable Reference (e.g. `&v`)
 - can: read, alias
 - guarantee: no mutation

Motivation

Consequences:

- unique data owner
- no global, mutable state
- no cycles in memory structure

Used for:

- safe non-gc memory management
- safe concurrency
- safe low-level hardware access
- ...
- \Rightarrow show: program verification as well

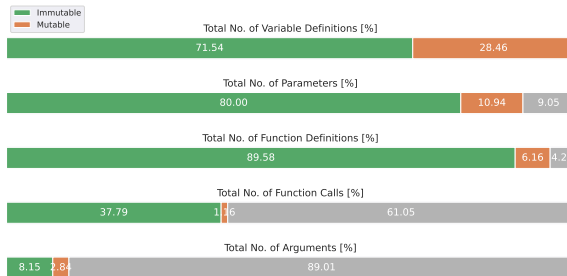
Ownership in Rust: Mutability XOR Aliasing

Each lexical scope tracks permissions for visible memory objects. Possible Permission Levels:

- Owner (e.g. `b`)
 - can: read, write
 - transfer ownership (if no outstanding borrows)
- Mutable Reference (e.g. `&mut x`)
 - can: read, write
 - guarantee: no aliasing
- Immutable Reference (e.g. `&v`)
 - can: read, alias
 - guarantee: no mutation

Empirical Use-Case Analysis

- public open-source code (crates.io)
- about 64 million lines of Rust code
- syntactical analysis



Refinement Types for Rust – Syntax

```
fn max(a: i32, b: i32) -> i32 {
  if a > b { a } else { b }
}
```

Addition of two macros

- $\text{ty!}\{l : b \mid \varphi\}$ in place of a type
- $\text{relax_ctx!}\{ \dots \}$ in place of a statement

Refinement Types for Rust – Syntax

```
fn max(
  a: ty!{ av: i32 | true },
  b: ty!{ bv : i32 | true }
) -> ty!{ v : i32 | v >= av && v >= bv } {
  if a > b { a } else { b }
}
```

Addition of two macros

- $\text{ty!}\{l : b \mid \varphi\}$ in place of a type
- $\text{relax_ctx!}\{ \dots \}$ in place of a statement

Refinement Types for Rust – Syntax

```
fn max(
  a: ty!{ av: i32 },
  b: ty!{ bv : i32 }
) -> ty!{ v : i32 | v >= av && v >= bv } {
  if a > b { a } else { b }
}
```

Addition of two macros

- `ty!{ $l : b \mid \varphi$ }` in place of a type
- `relax_ctx!{ ... }` in place of a statement

Type Updates

```
fn decr() -> ty!{ w : i32 | w >= 0 } {
  let mut i = ... as ty!{ v: i32 | v > 0};
  i = i - 1;
  i
}
```

■ Types need to change through execution

- \Rightarrow type updates
- Separation of program-variables and logic-variables
- Γ association of program- to logic-variables and predicate
- $\Gamma \vdash s \Rightarrow \Gamma'$ (Statement Type Checking)
- $\Gamma \vdash e : \tau$ (Expression Typing)
- On assignment: *replace* association, *append* predicate

Type Updates

```

fn decr() -> ty!{ w : i32 | w >= 0 } {
  //  $\Gamma_1 = (\{\}, \text{true})$ 
  let mut i = ... as ty!{ v: i32 | v > 0 };
  //  $\Gamma_2 = (\{i \mapsto v\}, v > 0)$ 
  i = i - 1;
  //  $\Gamma_3 = (\{i \mapsto v_2\}, v > 0 \wedge v_2 \doteq v - 1)$ 
  i }
  
```

■ Types need to change through execution

- \Rightarrow type updates
- Separation of program-variables and logic-variables
- Γ association of program- to logic-variables and predicate
- $\Gamma \vdash s \Rightarrow \Gamma'$ (Statement Type Checking)
- $\Gamma \vdash e : \tau$ (Expression Typing)
- On assignment: *replace* association, *append* predicate

Type Updates

```
fn decr() -> ty!{ w : i32 | w >= 0 } {
  //  $\Gamma_1 = (\{\}, \text{true})$ 
  let mut i = ... as ty!{ v : i32 | v > 0 };
  //  $\Gamma_2 = (i \mapsto v, v > 0)$ 
  i = i - 1;
  //  $\Gamma_3 = (i \mapsto v_2, v > 0 \wedge v_2 \doteq v - 1)$ 
  i }
```

let $\Gamma_2 = \Gamma[i \mapsto v_1], v > 0$ and $\tau = \{v : i32 \mid v > 0\}$

$$\begin{array}{c}
\text{INTRO-SUB} \frac{\Gamma_1 \vdash \dots : \tau' \quad \Gamma_1 \vdash \tau' \preceq \tau}{\Gamma_1 \vdash \dots \text{ as } \tau : \tau} \\
\text{DECL} \frac{\Gamma_1 \vdash \dots \text{ as } \tau : \tau}{\Gamma_1 \vdash \text{let } i = \dots \text{ as } \tau \Rightarrow \Gamma_2} \\
\text{SEQ} \frac{\Gamma_1 \vdash \text{let } i = \dots \text{ as } \tau \Rightarrow \Gamma_2 \quad \text{ASS} \frac{\text{BINOP} \frac{\Gamma_1 \vdash v_2 \text{ fresh}}{\Gamma_1 \vdash i - 1 : \{v_2 : i32 \mid v_2 \doteq v - 1\}}}{\Gamma_2 \vdash i = i - 1 \Rightarrow \Gamma[i \mapsto v_2], v > 0, v_2 \doteq v - 1}}{\Gamma_1 \vdash \text{let } i = \dots \text{ as } \tau; i = i - 1 \Rightarrow \Gamma[i \mapsto v_2], v > 0, v_2 \doteq v - 1}
\end{array}$$

Expression Typing $\Gamma \vdash e : \tau$

$$\begin{array}{c}
 \text{LIT} \frac{\Gamma \vdash \alpha \text{ fresh}}{\Gamma \vdash v : \{\alpha : b \mid \alpha \simeq \llbracket v \rrbracket \Gamma\}} \quad \text{BINOP} \frac{\Gamma \vdash \alpha \text{ fresh}}{\Gamma \vdash x_1 \odot x_2 : \{\alpha : b \mid \alpha \simeq \llbracket x_1 \odot x_2 \rrbracket \Gamma\}} \\
 \text{VAR} \frac{\Gamma \vdash \alpha \text{ fresh}}{\Gamma \vdash x : \{\alpha : b \mid \alpha \simeq \llbracket x \rrbracket \Gamma\}} \quad \text{INTRO-SUB} \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \preceq \tau'}{\Gamma \vdash e \text{ as } \tau' : \tau'}
 \end{array}$$

Statement Type Checking $\Gamma \vdash s \Rightarrow \Gamma'$

$$\begin{array}{c}
 \text{IF} \frac{\Gamma, \Gamma(x) \doteq \text{true} \vdash s_t \Rightarrow \Gamma' \quad \Gamma, \Gamma(x) \doteq \text{false} \vdash s_e \Rightarrow \Gamma'}{\Gamma \vdash \text{if } x \text{ then } s_t \text{ else } s_e \Rightarrow \Gamma'} \\
 \text{SEQ} \frac{\Gamma \vdash s_1 \Rightarrow \Gamma' \quad \Gamma' \vdash s_2 \Rightarrow \Gamma''}{\Gamma \vdash s_1; s_2 \Rightarrow \Gamma''} \\
 \text{DECL} \frac{\Gamma \vdash e : \{\beta : b \mid \varphi\}}{\Gamma \vdash \text{let } x = e \Rightarrow \Gamma[x \mapsto \beta], \varphi} \quad \text{ASSIGN} \frac{\Gamma \vdash e : \{\beta : b \mid \varphi\}}{\Gamma \vdash x = e \Rightarrow \Gamma[x \mapsto \beta], \varphi}
 \end{array}$$

References – Strong Updates

```

fn client() -> ty!{ v: i32 | v == 4 } {
  let a = 2;           // a : { v1 : i32 | v1 == 2 }
  let b = &mut a;      // b : { v2 : &i32 | v2 == &a }
  *b = 0; // changes a's value and type
  let c = &mut b;      // c : { v3 : &i32 | v3 == &b }
  **c = 4; // changes a's value and type
  a
}

```

Expression Typing $\Gamma \vdash e : \tau$

$$\text{REF} \frac{\Gamma \vdash \alpha \text{ fresh}}{\Gamma \vdash \&x : \{\alpha : \&b \mid \alpha \simeq \llbracket \&x \rrbracket \Gamma\}}$$

$$\text{VAR-DEREF} \frac{\Gamma \vdash *x \in \{y\} \quad \Gamma \vdash y : \tau}{\Gamma \vdash *x : \tau}$$

Statement Type Checking $\Gamma \vdash s \Rightarrow \Gamma'$

$$\text{ASSIGN-STRONG} \frac{\Gamma(z) = \beta \quad \Gamma \vdash *x \in \{y\} \quad \Gamma \vdash \gamma \text{ fresh}}{\Gamma \vdash *x = z \Rightarrow \Gamma[y \mapsto \gamma], \gamma \dot{=} \beta}$$

Expression Typing $\Gamma \vdash e : \tau$

$$\text{REF} \frac{\Gamma \vdash \alpha \text{ fresh}}{\Gamma \vdash \&x : \{\alpha : \&b \mid \alpha \simeq \llbracket \&x \rrbracket \Gamma\}}$$

$$\text{VAR-DEREF} \frac{\Gamma \vdash *x \in \{y\} \quad \Gamma \vdash y : \tau}{\Gamma \vdash *x : \tau}$$

Statement Type Checking $\Gamma \vdash s \Rightarrow \Gamma'$

$$\text{ASSIGN-STRONG} \frac{\Gamma(z) = \beta \quad \Gamma \vdash *x \in \{y\} \quad \Gamma \vdash \gamma \text{ fresh}}{\Gamma \vdash *x = z \Rightarrow \Gamma[y \mapsto \gamma], \gamma \dot{=} \beta}$$

$$\text{ASSIGN-WEAK} \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash *x \in \{y_1, \dots, y_n\}}{\Gamma \vdash y_i : \{\beta_i : b_i \mid \varphi_i\} \quad \Gamma \vdash \tau \preceq \{\beta_i : b_i \mid \varphi_i\}} \Gamma \vdash *x = e \Rightarrow \Gamma$$

Modularity & Mutation

```

fn clamp(a: &mut ty!{ a1 : i32 | true => a2 | a2 <= b1 }, b: ty!{ b1: i32 }) {
    if *a > b { *a = b }
}

fn client(...) {
    ...
    clamp(&mut x, 5);
    clamp(&mut y, 6);
    print!(x);
    ...
}

```

Modularity & Mutation

```

fn clamp(a: &mut ty!{ a1 : i32 | true => a2 | a2 <= b1 }, b: ty!{ b1: i32 }) {
    //  $\Gamma_1 = (\{a \mapsto v_1, arg_0 \mapsto a_1, b \mapsto b_1\}, v_1 \doteq \&arg_0 \wedge true \wedge true)$ 
    if *a > b { *a = b }
    //  $\Gamma_2 = (\{a \mapsto v_1, arg_0 \mapsto v_2, b \mapsto b_1\}, v_2 \leq b_1 \wedge v_1 \doteq \&arg_0 \wedge true \wedge true)$ 
}

fn client(...) {
    ...
    //  $\Gamma_1 = (\{x \mapsto v_1, y \mapsto v_2\}, \dots)$ 
    clamp(&mut x, 5);
    //  $\Gamma_2 = (\{x \mapsto v_3, y \mapsto v_2\}, \dots \wedge v_3 \leq 5)$ 
    clamp(&mut y, 6);
    //  $\Gamma_3 = (\{x \mapsto v_3, y \mapsto v_4\}, \dots \wedge v_3 \leq 5 \wedge v_4 \leq 6)$ 
    print!(x);
    ...
}

```

Motivation
○○○○○○○

Empirical Analysis
○

Type System
○○○○○○●○○○○○

Soundness Justification
○○○

Related Work
○○

Conclusion / Future Work
○○

Notion of Sub-Contexts

```
fn clamp(a: &mut ty!{ a1 : i32 | true => a2 | a2 <= b1 }, b: ty!{ b1: i32 }) {
    ...
    //  $\Gamma_2 = (\{a \mapsto v_1, arg_0 \mapsto v_2, b \mapsto b_1\}, v_2 \leq b_1 \wedge v_1 \doteq \&arg_0)$ 
}
```

- Question: Is Γ_2 a valid end-state for `clamp`?
- Question: Expected end-state?
- $\Rightarrow \Gamma_e = (\{a \mapsto v_1, arg_0 \mapsto a_2, b \mapsto b_1\}, v_1 \doteq \&arg_0 \wedge \text{true} \wedge \text{true} \wedge a_2 \leq b_1)$
- $\Rightarrow \Gamma_2$ must be a specialization of Γ_e . Written as $\Gamma_2 \preceq \Gamma_e$

$$\preceq\text{-CTX} \frac{\vdash v_1 \doteq \&arg_0 \wedge v_2 \leq b_1 \rightarrow v_2 \leq b_1 \wedge v_1 \doteq \&arg_0 \quad \text{dom}(\Gamma_e) \subseteq \text{dom}(\Gamma_2)}{\Gamma_2 \preceq \Gamma_e}$$

Use SMT-Solver

Motivation
○○○○○○○

Empirical Analysis
○

Type System
○○○○○○○●○○○

Soundness Justification
○○○

Related Work
○○

Conclusion / Future Work
○○

SMT Request

...

Motivation
○○○○○○○

Empirical Analysis
○

Type System
○○○○○○○○●○○

Soundness Justification
○○○

Related Work
○○

Conclusion / Future Work
○○

Example Error Message

...

Motivation
○○○○○○○

Empirical Analysis
○

Type System
○○○○○○○○○●○

Soundness Justification
○○○

Related Work
○○

Conclusion / Future Work
○○

Ecosystem Integration

...

Motivation
○○○○○○○

Empirical Analysis
○

Type System
○○○○○○○○○○●

Soundness Justification
○○○

Related Work
○○

Conclusion / Future Work
○○

Progress

If $\Gamma \vdash s_1, \sigma : \Gamma \Rightarrow \Gamma_2$ and $s_1 \neq \text{unit}$, then there is a s_2 and σ_2 with $\langle s_1 \mid \sigma_1 \rangle \rightsquigarrow \langle s_2 \mid \sigma_2 \rangle$.

Corten strictly refines the base language, therefore progress depends on base type system.

Preservation

If $\Gamma \vdash s \Rightarrow \Gamma_2, \sigma : \Gamma$ and $\langle s \mid \sigma \rangle \rightsquigarrow \langle s_1 \mid \sigma_1 \rangle$, then there is a Γ_1 with $\Gamma_1 \vdash s_1 \Rightarrow \Gamma_2$ and $\sigma_2 : \Gamma_2$

Stronger property than base language preservation: Show that refined types are preserved

State Conformance

State Conformance $\sigma : \Gamma$

A state σ is conformant with respect to a typing context $\Gamma = (\mu, \Phi)$ (written as $\sigma : \Gamma$), iff:

$$\Phi[\mu(x) \triangleright \llbracket \sigma(x) \rrbracket \mid x \in \text{dom}(\mu)] \text{ is satisfiable}$$

I.e. a conformant type context does not contradict the execution state.

Examples:

- If $\sigma : (\emptyset, \Phi)$ then Φ is satisfiable
- If $\sigma : (\mu, \Phi_1 \wedge \Phi_2)$ then $\sigma : (\mu, \Phi_1)$ and $\sigma : (\mu, \Phi_2)$.
- If $\sigma : (\mu, \Phi)$ and $\text{FV}(\Phi) \subseteq \text{dom}(\mu)$, then $\models \Phi[\mu(x) \triangleright \llbracket \sigma(x) \rrbracket \mid x \in \text{dom}(\mu)]$

Intermediate Steps

Conformance of Symbolic Execution

If $\sigma : \Gamma, \Gamma \vdash \alpha$ fresh then $\sigma[x \mapsto \llbracket e \rrbracket \sigma] : \Gamma[x \mapsto \alpha], (\alpha \simeq \llbracket e \rrbracket \Gamma)$

where $(\alpha \simeq \llbracket e \rrbracket \Gamma)$ is the symbolic execution of e equated with α in context Γ

Reference Predicates are Conservative

If $\sigma : \Gamma$ and $\Gamma \vdash *x \in \{y_1, \dots, y_n\}$ then $\llbracket \sigma(x) \rrbracket = \&y_i$ for some $i \in 1, \dots, n$

Rare case where conservative typing requires

Sub-Context Relation is Conservative

If $\Gamma \preceq \Gamma'$ and $\sigma : \Gamma$ then $\sigma : \Gamma'$

Related Work

Refinement Types and Mutability

- Rondon et al. [RKJ10], Bakst and Jhala [BJ16]: Refinement Types for C subset. Lack of guarantees requires ad-hoc mechanisms to control aliasing
- Lanzinger [Lan21]: Property Types in Java (only immutable)
- Bachmeier [Bac22]: Extension using Ownership System

Rust verification

- Ullrich [Ull]: Translation to Lean; linear mutation chain. Denis et al [DJM21] similar, but to Why3
- Astrauskas et al. [Ast+19] (Prusti): heavy-weight verification, translation to separation logic (Viper)
- Matsushita et al. [MTK20]: constrained Horn clauses (RustHorn)

Motivation
○○○○○○○

Empirical Analysis
○

Type System
○○○○○○○○○○○

Soundness Justification
○○○

Related Work
●○

Conclusion / Future Work
○○

Flux – Refinement Types for Rust

- MIR vs. HIR
- Context-Inclusions vs. Sub-Context
-

Motivation
○○○○○○○

Empirical Analysis
○

Type System
○○○○○○○○○○○○○

Soundness Justification
○○○

Related Work
○●

Conclusion / Future Work
○○

Future Work

- Records & ADTs
- Predicate Generics (Abstract Predicates)
- Concurrency using Predicate Generics?

Motivation
○○○○○○○

Empirical Analysis
○

Type System
○○○○○○○○○○○○

Soundness Justification
○○○

Related Work
○○

Conclusion / Future Work
●○

Conclusion

- Working Refinement Type System for Rust with Mutability
- Minimal Interface
- Soundness Justification
- Evaluation

Motivation
○○○○○○○

Empirical Analysis
○

Type System
○○○○○○○○○○○○

Soundness Justification
○○○

Related Work
○○

Conclusion / Future Work
●

Literatur

- [1] Vytautas Astrauskas u. a. „Leveraging rust types for modular specification and verification“. In: *Proceedings of the ACM on Programming Languages* 3 (OOPSLA 10. Okt. 2019), S. 1–30. ISSN: 2475-1421. DOI: 10.1145/3360573. URL: <https://dl.acm.org/doi/10.1145/3360573> (besucht am 23. 02. 2022).
- [2] Joshua Bachmeier. *Property Types for Mutable Data Structures in Java*. 2022. DOI: 10.5445/IR/1000150318. URL: <https://publikationen.bibliothek.kit.edu/1000150318> (besucht am 03. 10. 2022).
- [3] Alexander Bakst und Ranjit Jhala. „Predicate Abstraction for Linked Data Structures“. In: *Verification, Model Checking, and Abstract Interpretation*. Hrsg. von Barbara Jobstmann und K. Rustan M. Leino. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2016, S. 65–84. ISBN: 978-3-662-49122-5. DOI: 10.1007/978-3-662-49122-5_3.
- [4] Xavier Denis, Jacques-Henri Jourdan und Claude Marché. „The Creusot Environment for the Deductive Verification of Rust Programs“. Diss. Inria Saclay-Île de France, 2021.

- [5] Florian Lanzinger. „Property Types in Java: Combining Type Systems and Deductive Verification“. Master Thesis. Karlsruher Institut für Technologie, Feb. 2021.

- [6] Yusuke Matsushita, Takeshi Tsukada und Naoki Kobayashi. „RustHorn: GHC-based verification for Rust programs“. In: *European Symposium on Programming*. Springer, Cham, 2020, S. 484–514.

Blöcke

in den KIT-Farben

Greenblock
Standard (block)

Blueblock
= exampleblock

Redblock
= alertblock

Brownblock

Purpleblock

Cyanblock

Yellowblock

Lightgreenblock

Orangeblock

Grayblock

Contentblock
(farblos)

Auflistungen

Text

- Auflistung
Umbruch
- Auflistung
 - Auflistung
 - Auflistung

Literatur
○●

Zweiter Abschnitt
○○○○

Farben
○

Bei Frames ohne Titel wird die Kopfzeile nicht angezeigt, und der freie Platz kann für Inhalte genutzt werden.

Bei Frames mit Option `[plain]` werden weder Kopf- noch Fußzeile angezeigt.

Beispielinhalt

Bei Frames mit Option [t] werden die Inhalte nicht vertikal zentriert, sondern an der Oberkante begonnen.

Beispielinhalt: Literatur

Literatur
○○

Zweiter Abschnitt
○○○●

Farben
○

Farbpalette

