Corten: Refinement Types for Imperative Languages with Ownership

Abschlusspräsentation Masterarbeit

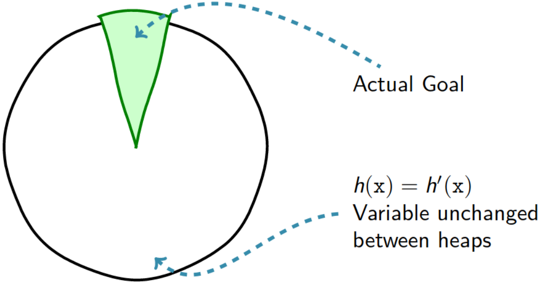
Carsten Csiky | 26th Oktober 2022

Inhaltsverzeichnis

1. Motivation
2. Type System
3. Soundness Justification
4. Related Work
5. Conclusion / Future Work

Motivation

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


Actual Goal
$h(\mathrm{x}) = h'(\mathrm{x})$
Variable unchanged between heaps

Motivation

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

Motivation

```rust
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 $\Gamma = (a : \{v : \texttt{i32} \mid \text{true}\}, b : \{v : \texttt{i32} \mid \text{true}\})$ and $\tau = \{v : \texttt{i32} \mid v \geq a \land v \geq b\}$

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

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 }
}
```

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

$$\dfrac{\dfrac{}{\Gamma, a > b \vdash a : \tau} \qquad \dfrac{}{\Gamma, \neg(a > b) \vdash b : \tau}}{\Gamma \vdash \texttt{if } a > b\ \{a\}\ \texttt{else }\ \{b\} : \tau}$$

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

$$\dfrac{\dfrac{\dfrac{}{\Gamma, a > b \vdash \{v : \texttt{i32} \mid v \doteq a\} \preceq \tau}}{\Gamma, a > b \vdash a : \tau} \qquad \dfrac{}{\Gamma, \neg(a > b) \vdash b : \tau}}{\Gamma \vdash \texttt{if}\ a > b\ \{a\}\ \texttt{else}\ \{b\} : \tau}$$

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

$$\dfrac{\dfrac{\dfrac{\star}{\Gamma, a > b \vdash a : \{v : \texttt{i32} \mid v \doteq a\}} \qquad \dfrac{}{\Gamma, a > b \vdash \{v : \texttt{i32} \mid v \doteq a\} \preceq \tau}}{\Gamma, a > b \vdash a : \tau} \qquad \dfrac{}{\Gamma, \neg(a > b) \vdash b : \tau}}{\Gamma \vdash \texttt{if}\ a > b\ \{a\}\ \texttt{else}\ \{b\} : \tau}$$

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 }
}
```

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

$$\dfrac{\dfrac{\dfrac{\star}{\Gamma, a > b \vdash a : \{v : \texttt{i32} \mid v \doteq a\}} \qquad \dfrac{\text{SMT-VALID}\left(\begin{array}{l} \text{true} \wedge \text{true} \wedge a > b \\ \ \wedge v \doteq a \\ \qquad \Longrightarrow \ (v \geq a \wedge v \geq b) \end{array}\right)}{\Gamma, a > b \vdash \{v : \texttt{i32} \mid v \doteq a\} \preceq \tau}}{\Gamma, a > b \vdash a : \tau} \qquad \dfrac{}{\Gamma, \neg(a > b) \vdash b : \tau}}{\Gamma \vdash \texttt{if } a > b\ \{a\}\ \texttt{else}\ \{b\} : \tau}$$

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

$$\dfrac{\dfrac{\dfrac{\star}{\Gamma, a > b \vdash a : \{v : \texttt{i32} \mid v \doteq a\}} \qquad \dfrac{\text{SMT-VALID}\begin{pmatrix}\text{true} \land \text{true} \land a > b \\ \land\, v \doteq a \\ \implies (v \geq a \land v \geq b)\end{pmatrix}}{\Gamma, a > b \vdash \{v : \texttt{i32} \mid v \doteq a\} \preceq \tau}}{\Gamma, a > b \vdash a : \tau} \qquad \dfrac{\vdots}{\Gamma, \neg(a > b) \vdash b : \tau}}{\Gamma \vdash \texttt{if}\ a > b\ \{a\}\ \texttt{else}\ \{b\} : \tau}$$

Motivation

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

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

```
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 it `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 it `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
- . . .
- ⇒ 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

Contributions

- Empirical Use-Case Analysis
- Refinement Type System
  - Automatic & Decidable Type Checking
  - Path Sensitivity
  - Mutable Data & References
  - Modularity
  - Partial, Mechanized Proof of Soundness
- Implementation
  - Accessible Interface
  - Type-Error Messages with Source Code Locations
  - Counter-Example Generation
- Evaluation
  - Automatic Verification of non-trivial Programs
  - Comparison to other tools

Contributions

- Empirical Use-Case Analysis
- Refinement Type System
  - Automatic & Decidable Type Checking
  - Path Sensitivity
  - Mutable Data & References
  - Modularity
  - Partial, Mechanized Proof of Soundness
- Implementation
  - Accessible Interface
  - Type-Error Messages with Source Code Locations
  - Counter-Example Generation
- Evaluation
  - Automatic Verification of non-trivial Programs
  - Comparison to other tools

Restrictions

- No Inference System
- Datatypes: Integers, Booleans and References

Overview

- Common Refinement Types
- Mutable Values
- Mutable References
- Function Calls
- Verification of `clamp` Example
- Demonstration

```
fn clamp(
  a: &mut ty!{ a1 : i32 | true => a2 | a2 <= b1 },
  b: ty!{ b1: i32 }
) -> ty!{ v: () } {
  if *a > b {
      *a = b as ty!{ r | (r <= b1) }; ()
  } else {};
  ()
}

fn client() -> ty!{ v: () } {
  ...
  let m = 42;
  clamp(&mut x, m);
  x as ty!{ v : i32 | v < 43 };
}
```

Refinement Types for Rust – Syntax

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

- Embedding using a Macro
- `ty!`$\{l : b \mid \varphi\}$ in place of a type

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 }
}
```

- Embedding using a Macro
- `ty!`$\{l : b \mid \varphi\}$ in place of a type

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 }
}
```

- Embedding using a Macro
- `ty!`$\{l : b \mid \varphi\}$ in place of a type

Type Updates

```
fn incr() -> 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
  - $\Gamma \vdash s \Rightarrow \Gamma'$ (Statement Type Checking)
  - $\Gamma \vdash e : \tau$ (Expression Typing)

Type Updates

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

- Type of `i` after decrementing?
  - Naïve: `ty!{ v : i32 | v = v + 1 }`
- How to keep type context consistent?
  - separation of program-variables and logic-variables
  - Γ: association of program- to logic-variables and predicate
  - on assignment: *replace* association, *append* predicate
  - observation: assignments can not invalidate existing predicates

Type Updates

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

- Type of `i` after decrementing?
  - Naïve: `ty!{ v : i32 | v = v + 1 }`
- How to keep type context consistent?
  - separation of program-variables and logic-variables
  - Γ: association of program- to logic-variables and predicate
  - on assignment: *replace* association, *append* predicate
  - observation: assignments can not invalidate existing predicates

Type Updates

```
fn incr() -> 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;
  // $\Gamma_3 = (\{i \mapsto v_2\}, v > 0 \land v_2 \doteq v + 1)$
  i }
```

$$\textsc{Intro-Sub}\ \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau \preceq \tau'}{\Gamma \vdash e \ \texttt{as}\ \tau' : \tau'}$$

$$\textsc{Decl}\ \frac{\Gamma \vdash e : \{\beta : b \mid \varphi\}}{\Gamma \vdash \texttt{let}\ x = e \Rightarrow \Gamma[x \mapsto \beta], \varphi}$$

Type Updates

```
fn incr() -> ty!{ w : i32 | w > 0 } {
  // $\Gamma_1 = (\{\}, \mathrm{true})$
  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 }
```

$$\textsc{BinOp}\ \frac{\Gamma \vdash \alpha \text{ fresh}}{\Gamma \vdash x_1 \odot x_2 : \{\alpha : b \mid \alpha \simeq [\![x_1 \odot x_2]\!]\Gamma\}}$$

$$\textsc{Assign}\ \frac{\Gamma \vdash e : \{\beta : b \mid \varphi\}}{\Gamma \vdash x = e \Rightarrow \Gamma[x \mapsto \beta], \varphi}$$

Type Updates

```
fn incr() -> 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 }
```

$$\textsc{Seq}\ \frac{\Gamma \vdash s_1 \Rightarrow \Gamma' \qquad \Gamma' \vdash s_2 \Rightarrow \Gamma''}{\Gamma \vdash s_1 ; s_2 \Rightarrow \Gamma''}$$

References – Strong Updates

```
fn client() -> ty!{ v: i32 | v == 4 } {
  let mut a = 2;       // a : $\{v_1 : i32 \mid v_1 == 2\}$
  let mut q = 3;       // q : $\{v_2 : i32 \mid v_2 == 3\}$
  let mut b = &mut a;  // b : $\{v_3 : \&i32 \mid v_3 == \&a\}$
  *b = 0;              // changes a's value and type
  b = &mut q;          // b : $\{v_4 : \&i32 \mid v_4 == \&q\}$
  *b = 4;              // changes q's value and type
  a
}
```

References – Strong Updates

```
fn client() -> ty!{ v: i32 | v == 4 } {
  let mut a = 2;          // a : {v_1 : i32 | v_1 == 2}
  let mut q = 3;          // q : {v_2 : i32 | v_2 == 3}
  let mut b = &mut a;     // b : {v_3 : &i32 | v_3 == &a}
  *b = 0;                 // changes a's value and type
  b = &mut q;             // b : {v_4 : &i32 | v_4 == &q}
  *b = 4;                 // changes q's value and type
  a
}
```

$$\textsc{Lit}\ \frac{\Gamma \vdash \alpha \text{ fresh}}{\Gamma \vdash v : \{\alpha : b \mid \alpha \simeq [\![v]\!] \Gamma\}}$$

References – Strong Updates

```
fn client() -> ty!{ v: i32 | v == 4 } {
  let mut a = 2;        // a : {v_1 : i32 | v_1 == 2}
  let mut q = 3;        // q : {v_2 : i32 | v_2 == 3}
  let mut b = &mut a;   // b : {v_3 : &i32 | v_3 == &a}
  *b = 0;               // changes a's value and type
  b = &mut q;           // b : {v_4 : &i32 | v_4 == &q}
  *b = 4;               // changes q's value and type
  a
}
```

$$\textsc{Ref}\ \frac{\Gamma \vdash \alpha \text{ fresh}}{\Gamma \vdash \&x : \{\alpha : \&b \mid \alpha \simeq [\![\&x]\!]\Gamma\}}$$

References – Strong Updates

```
fn client() -> ty!{ v: i32 | v == 4 } {
  let mut a = 2;         // a : {v_1 : i32 | v_1 == 2}
  let mut q = 3;         // q : {v_2 : i32 | v_2 == 3}
  let mut b = &mut a;    // b : {v_3 : &i32 | v_3 == &a}
  *b = 0;                // changes a's value and type
  b = &mut q;            // b : {v_4 : &i32 | v_4 == &q}
  *b = 4;                // changes q's value and type
  a
}
```

$$\textsc{Assign-Strong}\ \frac{\begin{array}{c}\Gamma(z) = \beta \\ \Gamma \vdash x \in \{\&y\} \qquad \Gamma \vdash \gamma \text{ fresh}\end{array}}{\Gamma \vdash *x = z \Rightarrow \Gamma[y \mapsto \gamma], \gamma \doteq \beta}$$

(Also Assign-Weak)

References – Strong Updates

```
fn client() -> ty!{ v: i32 | v == 4 } {
  let mut a = 2;        // a : $\{v_1 : i32 \mid v_1 == 2\}$
  let mut q = 3;        // q : $\{v_2 : i32 \mid v_2 == 3\}$
  let mut b = &mut a;   // b : $\{v_3 : \&i32 \mid v_3 == \&a\}$
  *b = 0;               // changes a's value and type
  b = &mut q;           // b : $\{v_4 : \&i32 \mid v_4 == \&q\}$
  *b = 4;               // changes q's value and type
  a
}
```

References – Strong Updates

```
fn client() -> ty!{ v: i32 | v == 4 } {
  let mut a = 2;         // a : {v1 : i32 | v1 == 2}
  let mut q = 3;         // q : {v2 : i32 | v2 == 3}
  let mut b = &mut a;    // b : {v3 : &i32 | v3 == &a}
  *b = 0;                // changes a's value and type
  b = &mut q;            // b : {v4 : &i32 | v4 == &q}
  *b = 4;                // changes q's value and type
  a
}
```

$$\textsc{Var}\ \frac{\Gamma \vdash \alpha \text{ fresh}}{\Gamma \vdash x : \{\alpha : b \mid \alpha \simeq [\![x]\!]\Gamma\}}$$

Mutable Arguments

```
fn clamp(a: &mut ty!{ a1 : i32 | true => a2 | a2 <= b1 }, b: ty!{ b1: i32 }) {
  if *a > b { *a = b }
}
fn client() -> ty!{ v: () } {
  ...
  let max = 42;
  clamp(&mut x, max);
  x as ty!{ v : i32 | v < 43 };
}
```

Mutable Arguments

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

Mutable Arguments

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

- `ty!` $\{\alpha : \mathsf{b} \mid \varphi \Rightarrow \beta \mid \psi\}$
- Callee requires φ for reference destination α
- Callee ensures ψ for reference destination β
- Of course, multiple arguments possible

Mutable Arguments

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

Mutable Arguments

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

Sub-Context

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

- still left: proof obligation from signature $a_2 \leq b_1$
- i.e. is Γ_2 a valid end-state?

Sub-Context

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

- still left: proof obligation from signature $a_2 \leq b_1$
- i.e. is Γ_2 a valid end-state?
- generalize notion of sub-types to context: sub-context

Sub-Context

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

- still left: proof obligation from signature $a_2 \leq b_1$
- i.e. is Γ_2 a valid end-state?
- generalize notion of sub-types to context: sub-context
- expected state: $\Gamma_e = (\{arg_0 \mapsto a_2, b \mapsto b_1\}, a_2 \leq b_1)$

Sub-Context

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

- still left: proof obligation from signature $a_2 \leq b_1$
- i.e. is Γ_2 a valid end-state?
- generalize notion of sub-types to context: sub-context
- expected state: $\Gamma_e = (\{arg_0 \mapsto a_2, b \mapsto b_1\}, a_2 \leq b_1)$
- show: $\Gamma_2 \preceq \Gamma_e$

Sub-Context

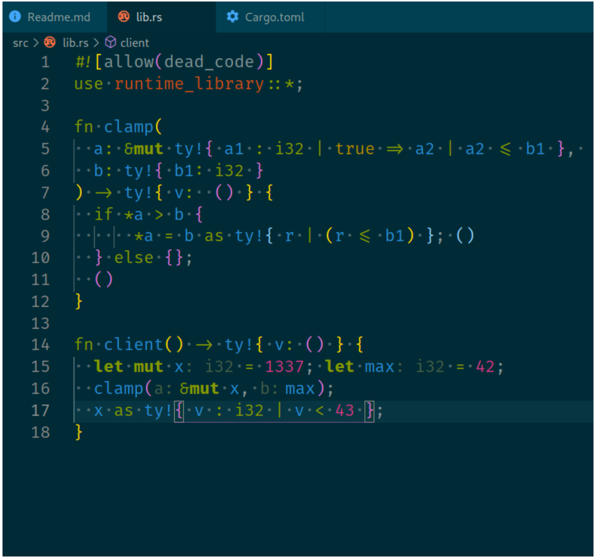
$$\preceq\text{-}\textsc{Ctx}\ \frac{\models \Phi \rightarrow \Phi'[\mu'(x) \triangleright \mu(x) \ \mid x \in \mathrm{dom}(\mu')] \qquad \mathrm{dom}(\mu') \subseteq \mathrm{dom}(\mu)}{(\mu, \Phi) \preceq (\mu', \Phi')}$$

- still left: proof obligation from signature $a_2 \leq b_1$
- i.e. is Γ_2 a valid end-state?
- generalize notion of sub-types to context: sub-context
- expected state: $\Gamma_e = (\{arg_0 \mapsto a_2, b \mapsto b_1\}, a_2 \leq b_1)$
- show: $\Gamma_2 \preceq \Gamma_e$

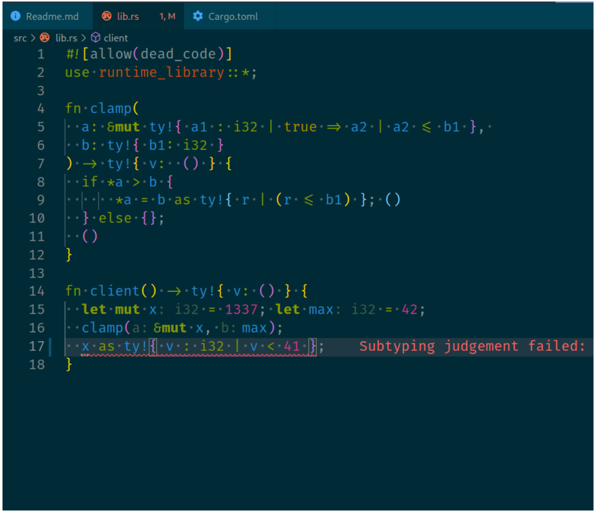
Mutable Calls

```
fn client(...) -> ty!{ v: () } {
  ...
  let m = 42;
  // $\Gamma_1 = (\{x \mapsto v_1, m \mapsto v_2\}, \ldots \wedge v_2 \doteq 42)$
  clamp(&mut x, m);
  // $\Gamma_2 = (\{x \mapsto v_3, m \mapsto v_2\}, \ldots \wedge v_2 \doteq 42 \wedge v_3 \leq 5)$
  x as ty!{ v : i32 | v < 43 };
}
```

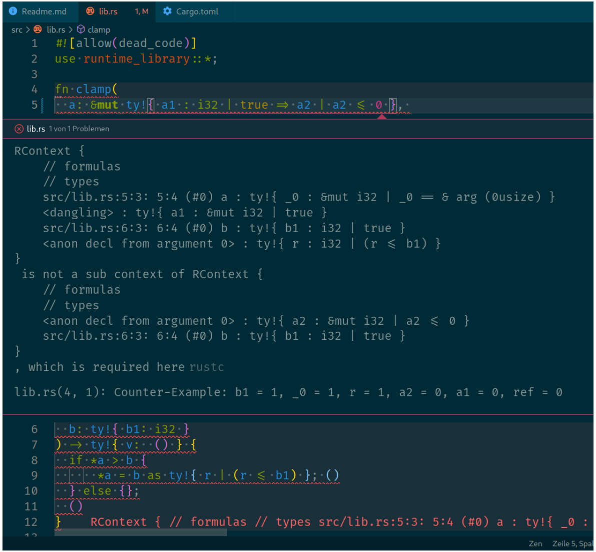
- append predicates from callee to context
- update association of logic variables



```rust
#![allow(dead_code)]
use runtime_library::*;

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

fn client() -> ty!{ v: () } {
  let mut x: i32 = 1337; let max: i32 = 42;
  clamp(a: &mut x, b: max);
  x as ty!{ v : i32 | v < 43 };
}
```



```
#![allow(dead_code)]
use runtime_library::*;

fn clamp(
  a: &mut ty!{ a1 : i32 | true ⇒ a2 | a2 ≤ b1 }, 
  b: ty!{ b1: i32 }
) → ty!{ v:  () } {
  if *a > b {
      *a = b as ty!{ r | (r ≤ b1) }; ()
  } else {};
  ()
}

fn client() → ty!{ v: () } {
  let mut x: i32 = 1337; let max: i32 = 42;
  clamp(a: &mut x, b: max);
  x as ty!{ v : i32 | v < 41 };    Subtyping judgement failed:
}
```



```
#![allow(dead_code)]
use runtime_library::*;

fn clamp(
  a: &mut ty!{ a1 : i32 | true ⇒ a2 | a2 ≤ 0 },
```

```
RContext {
    // formulas
    // types
    src/lib.rs:5:3: 5:4 (#0) a : ty!{ _0 : &mut i32 | _0 == & arg (0usize) }
    <dangling> : ty!{ a1 : &mut i32 | true }
    src/lib.rs:6:3: 6:4 (#0) b : ty!{ b1 : i32 | true }
    <anon decl from argument 0> : ty!{ r : i32 | (r ≤ b1) }
}
 is not a sub context of RContext {
    // formulas
    // types
    <anon decl from argument 0> : ty!{ a2 : &mut i32 | a2 ≤ 0 }
    src/lib.rs:6:3: 6:4 (#0) b : ty!{ b1 : i32 | true }
}
, which is required here rustc

lib.rs(4, 1): Counter-Example: b1 = 1, _0 = 1, r = 1, a2 = 0, a1 = 0, ref = 0
```

```
  b: ty!{ b1: i32 }
) → ty!{ v: () } {
  if *a > b {
      *a = b as ty!{ r | (r ≤ b1) }; ()
  } else {};
  ()
}    RContext { // formulas // types src/lib.rs:5:3: 5:4 (#0) a : ty!{ _0 :
```

SMT Request

```
; checking is_sub_context ...
(declare-datatypes () ((Unit unit)))
(declare-const |_0| Int)
(declare-const |r| Int)
(declare-const |a1| Int)
(declare-const |b1| Int)
(declare-const |a2| Int)

; ty!{ r : i32 | (r <= b1) }
(assert (<= |r| |b1|))

; ty!{ a1 : &mut i32 | true }
(assert true)

;     ty!{ b1 : i32 | true }
(assert true)
```

```
; SuperCtx:
(assert (not (and
        (<= |r| |b1|)
        true
    )))

; checking: RContext {
;     a : ty!{ _0 : &mut i32 | _0 == & arg (0usize) }
;     <dangling> : ty!{ a1 : &mut i32 | true }
;     b : ty!{ b1 : i32 | true }
;     <argument 0> : ty!{ r : i32 | (r <= b1) }
; }
;  <: RContext {
;     <argument 0> : ty!{ a2 : &mut i32 | a2 <= b1 }
;     b : ty!{ b1 : i32 | true }
; }
(check-sat)
```

Soundness

Progress

If $\Gamma \vdash s_1, \sigma : \Gamma \Rightarrow \Gamma_2$ and $s_1 \neq \texttt{unit}$, 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

Partial, Mechanized Proof in Lean 4

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:

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

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 $\sigma : (\mu, \Phi_1)$.
- If $\sigma : (\mu, \Phi)$ and $\mathrm{FV}(\Phi) \subseteq \mathrm{dom}(\mu)$, then $\vDash \Phi[\mu(x) \triangleright [\![\sigma(x)]\!] \mid x \in \mathrm{dom}(\mu)]$

Intermediate Steps

Conformance of Symbolic Execution

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

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

Reference Predicates are Conservative

If $\sigma : \Gamma$ and $\Gamma \vdash *x \in \{y_1, \ldots, y_n\}$ then $[\![\sigma(x)]\!] = \&y_i$ for some $i \in 1, \ldots, n$

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
- Toman et al. [Tom+20] (ConSORT): Fractional Ownership, strong and weak updates

Rust verification

- Ullrich [Ull16]: 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] (RustHorn): constrained Horn clauses

Related Work: Flux – Refinement Types for Rust

- MIR vs. HIR
- specification in comments vs. embedding in types
- context inclusions vs. sub context
- distinction strong and weak references vs. dynamic choice by typ checking rules
- explicit introduction of logic variables vs. ad-hoc
- formalization based on RustBelt vs. formalization based on own language
- missing in Corten: records & inference
- otherwise: similar capabilities

```
// Flux
//@ ensures *self: i32<n+1>;
fn increment(&strg v : i32<n>) -> ()
//@ requires n > 0
//@ ensures *self: i32<n-1>;
fn decrement(&strg v : i32<n>) -> ()
// Corten
fn increment(n: &mut ty!{
  n1: Nat => n1 | n1 == n1+1 }
) -> ();
fn decrement(n: &mut ty!{
  v1: Nat | v1 > 0 => v2 | v2 == v1-1 }
) -> ();
```

Future Work

- Records & ADTs
  - More Syntax, Nested Structures
  - Variant Distinction
- Predicate Generics (Abstract Predicates)
  - Uninterpreted Functions in Types
  - Syntactic Embedding?
- Concurrency using Predicate Generics?
  - Use Predicate Generics
  - Predicate describes Contract for Mutation
  - Interesting, because unusual guarantees in Rust

Conclusion

- Refinement Type System for Rust with Mutability
    - Decidable, Automatic
    - Complex Mutation Patterns
    - ...
- Minimal Interface
- Soundness Justification
- Practical Usability
    - Source Locations
    - Counter Example
    - IDE Integration

Conclusion

- Refinement Type System for Rust with Mutability
  - Decidable, Automatic
  - Complex Mutation Patterns
  - ...
- Minimal Interface
- Soundness Justification
- Practical Usability
  - Source Locations
  - Counter Example
  - IDE Integration

More Information:

- Implementation, Thesis, Mechanized Proof, Evaluation:
  `https://gitlab.com/csicar/liquidrust`
- Empirical Analysis:
  `https://gitlab.com/csicar/crates-analysis`

Literatur I

[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`.

Literatur II

[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: CHC-based verification for Rust programs“. In: *European Symposium on Programming*. Springer, Cham, 2020, S. 484–514.

[7] Patrick M. Rondon, Ming Kawaguci und Ranjit Jhala. „Liquid types“. In: *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI ’08*. the 2008 ACM SIGPLAN conference. Tucson, AZ, USA: ACM Press, 2008, S. 159. ISBN: 978-1-59593-860-2. DOI: `10.1145/1375581.1375602`. URL: `http://portal.acm.org/citation.cfm?doid=1375581.1375602` (besucht am 30. 06. 2022).

Literatur III

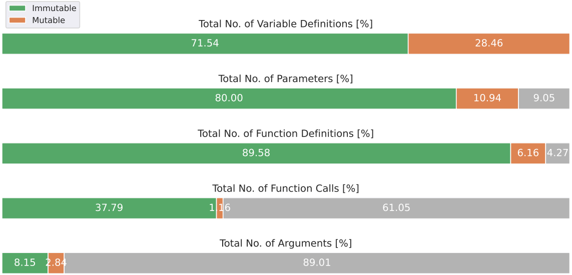
[8] Patrick Maxim Rondon, Ming Kawaguchi und Ranjit Jhala. „Low-level liquid types“. In: *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL ’10. New York, NY, USA: Association for Computing Machinery, 17. Jan. 2010, S. 131–144. ISBN: 978-1-60558-479-9. DOI: 10.1145/1706299.1706316. URL: https://doi.org/10.1145/1706299.1706316 (besucht am 16. 09. 2022).

[9] John Toman u. a. „ConSORT: Context- and Flow-Sensitive Ownership Refinement Types for Imperative Programs“. In: *Programming Languages and Systems*. Hrsg. von Peter Müller. Cham: Springer International Publishing, 2020, S. 684–714. ISBN: 978-3-030-44914-8. DOI: 10.1007/978-3-030-44914-8_25.

[10] Sebastian Ullrich. „Simple Verification of Rust Programs via Functional Purification“. In: (6. Dez. 2016), S. 65.

Empirical Use-Case Analysis

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



`decr` Typing Tree

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

$$\textsc{Seq}\ \dfrac{\textsc{Decl}\ \dfrac{\textsc{Intro-Sub}\ \dfrac{\Gamma_1 \vdash \ldots : \tau' \qquad \Gamma_1 \vdash \tau' \preceq \tau}{\Gamma_1 \vdash \texttt{... as } \tau : \tau}}{\Gamma_1 \vdash \texttt{let i = ... as } \tau \Rightarrow \Gamma_2} \qquad \textsc{Ass}\ \dfrac{\textsc{BinOp}\ \dfrac{\Gamma_1 \vdash v_2 \text{ fresh}}{\Gamma_1 \vdash \texttt{i - 1} : \{v_2 : \text{i32} \mid v_2 \doteq v - 1\}}}{\Gamma_2 \vdash \texttt{i = i - 1} \Rightarrow \Gamma[i \mapsto v_2], v > 0, v_2 \doteq v - 1}}{\Gamma_1 \vdash \texttt{let i = ... as } \tau\texttt{; i = i - 1} \Rightarrow \Gamma[i \mapsto v_2], v > 0, v_2 \doteq v - 1}$$

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

$$\textsc{Lit}\ \frac{\Gamma \vdash \alpha \text{ fresh}}{\Gamma \vdash v : \{\alpha : b \mid \alpha \simeq [\![v]\!]\Gamma\}} \qquad \textsc{BinOp}\ \frac{\Gamma \vdash \alpha \text{ fresh}}{\Gamma \vdash x_1 \odot x_2 : \{\alpha : b \mid \alpha \simeq [\![x_1 \odot x_2]\!]\Gamma\}}$$

$$\textsc{Var}\ \frac{\Gamma \vdash \alpha \text{ fresh}}{\Gamma \vdash x : \{\alpha : b \mid \alpha \simeq [\![x]\!]\Gamma\}} \qquad \textsc{Intro-Sub}\ \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau \preceq \tau'}{\Gamma \vdash e\ \texttt{as}\ \tau' : \tau'}$$

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

$$\textsc{If}\ \frac{\Gamma, \Gamma(x) \doteq \texttt{true} \vdash s_t \Rightarrow \Gamma' \qquad \Gamma, \Gamma(x) \doteq \texttt{false} \vdash s_e \Rightarrow \Gamma'}{\Gamma \vdash \text{if } x \text{ then } s_t \text{ else } s_e \Rightarrow \Gamma'}$$

$$\textsc{Seq}\ \frac{\Gamma \vdash s_1 \Rightarrow \Gamma' \qquad \Gamma' \vdash s_2 \Rightarrow \Gamma''}{\Gamma \vdash s_1 ; s_2 \Rightarrow \Gamma''}$$

$$\textsc{Decl}\ \frac{\Gamma \vdash e : \{\beta : b \mid \varphi\}}{\Gamma \vdash \texttt{let}\ x = e \Rightarrow \Gamma[x \mapsto \beta], \varphi} \qquad \textsc{Assign}\ \frac{\Gamma \vdash e : \{\beta : b \mid \varphi\}}{\Gamma \vdash x = e \Rightarrow \Gamma[x \mapsto \beta], \varphi}$$

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

$$\textsc{Ref}\ \frac{\Gamma \vdash \alpha \text{ fresh}}{\Gamma \vdash \&x : \{\alpha : \&b \mid \alpha \simeq [\![\&x]\!]\Gamma\}}$$

$$\textsc{Var-Deref}\ \frac{\Gamma \vdash x \in \{\&y\} \qquad \Gamma \vdash y : \tau}{\Gamma \vdash *x : \tau}$$

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

$$\textsc{Assign-Strong}\ \frac{\Gamma(z) = \beta \qquad \Gamma \vdash x \in \{\&y\} \qquad \Gamma \vdash \gamma \text{ fresh}}{\Gamma \vdash *x = z \Rightarrow \Gamma[y \mapsto \gamma], \gamma \doteq \beta}$$

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

$$\textsc{Ref}\ \dfrac{\Gamma \vdash \alpha \text{ fresh}}{\Gamma \vdash \&x : \{\alpha : \&b \mid \alpha \simeq [\![\&x]\!]\Gamma\}}$$

$$\textsc{Var-Deref}\ \dfrac{\Gamma \vdash x \in \{\&y\} \qquad \Gamma \vdash y : \tau}{\Gamma \vdash *x : \tau}$$

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

$$\textsc{Assign-Strong}\ \dfrac{\Gamma(z) = \beta \qquad \Gamma \vdash x \in \{\&y\} \qquad \Gamma \vdash \gamma \text{ fresh}}{\Gamma \vdash *x = z \Rightarrow \Gamma[y \mapsto \gamma], \gamma \doteq \beta}$$

$$\textsc{Assign-Weak}\ \dfrac{\dfrac{\Gamma \vdash e : \tau}{\Gamma \vdash y_i : \{\beta_i : b_i \mid \varphi_i\}} \qquad \dfrac{\Gamma \vdash x \in \{\&y_1, \dots, \&y_n\}}{\Gamma \vdash \tau \preceq \{\beta_i : b_i \mid \varphi_i\}}}{\Gamma \vdash *x = e \Rightarrow \Gamma}$$

Predicate Generics & Concurrecy

```
struct Mutex<P, T> = ...;

impl Mutex<P, T> {
  fn lock() -> MutexLock<P, T> {
    ...
  }
}

struct MutexLock<P, T> = ...;

impl MutexLock<P, T> {
  fn drop(self : ty!{ v : T | P v }) {
    ...
  }
}
```

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

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

Farbpalette

kit-green100 kit-green90 kit-green80 kit-green70 kit-green60 kit-green50 kit-green40 kit-green30 kit-green25 kit-green20 kit-green15 kit-green10 kit-green5

kit-blue100 kit-blue90 kit-blue80 kit-blue70 kit-blue60 kit-blue50 kit-blue40 kit-blue30 kit-blue25 kit-blue20 kit-blue15 kit-blue10 kit-blue5

kit-red100 kit-red90 kit-red80 kit-red70 kit-red60 kit-red50 kit-red40 kit-red30 kit-red25 kit-red20 kit-red15 kit-red10 kit-red5

kit-gray100 kit-gray90 kit-gray80 kit-gray70 kit-gray60 kit-gray50 kit-gray40 kit-gray30 kit-gray25 kit-gray20 kit-gray15 kit-gray10 kit-gray5

kit-orange100 kit-orange90 kit-orange80 kit-orange70 kit-orange60 kit-orange50 kit-orange40 kit-orange30 kit-orange25 kit-orange20 kit-orange15 kit-orange10 kit-orange5

kit-lightgreen100 kit-lightgreen90 kit-lightgreen80 kit-lightgreen70 kit-lightgreen60 kit-lightgreen50 kit-lightgreen40 kit-lightgreen30 kit-lightgreen25 kit-lightgreen20 kit-lightgreen15
kit-lightgreen10 kit-lightgreen5

kit-brown100 kit-brown90 kit-brown80 kit-brown70 kit-brown60 kit-brown50 kit-brown40 kit-brown30 kit-brown25 kit-brown20 kit-brown15 kit-brown10 kit-brown5

kit-purple100 kit-purple90 kit-purple80 kit-purple70 kit-purple60 kit-purple50 kit-purple40 kit-purple30 kit-purple25 kit-purple20 kit-purple15 kit-purple10 kit-purple5

kit-cyan100 kit-cyan90 kit-cyan80 kit-cyan70 kit-cyan60 kit-cyan50 kit-cyan40 kit-cyan30 kit-cyan25 kit-cyan20 kit-cyan15 kit-cyan10 kit-cyan5