Liquid Types for Rust

Flux

Nico Lehmann, Adam Geller, Niki Vazou, *Ranjit Jhala*

Flux

(/flʌks/)

n. 1 a flowing or flow. 2 a substance used to refine metals. v. 3 to melt; make fluid.

Programmer-Aided Analysis

I. Programs

Refinements for Rust

II. Analysis

Type-directed Abstract-Interpretation

Programmer-Aided Analysis

I. Programs

Refinements for Rust

II. Analysis

Type-directed Abstract-Interpretation

Refinements for Rust

1. *Refinements* `i32`, `bool`, ...

Refinements for Rust

1. *Refinements* `i32`, `bool`, ...
2. *Ownership* `mut`, `&`, `&mut`, ...

Refinements for Rust

1. *Refinements* `i32`, `bool`, …
2. *Ownership* `mut`, `&`, `&mut`, …
3. *Datatypes* `struct`, `enum`, …

Refinements for Rust

1. *Refinements* `i32`, `bool`, …
2. *Ownership* `mut`, `&`, `&mut`, …
3. *Datatypes* `struct`, `enum`, …
4. *Interfaces* `trait`, `impl`, …

Refinements for Rust

1. *Refinements* `i32`, `bool`, ...
2. *Ownership* `&`, `&mut`, ...
3. *Datatypes* `struct`, `enum`, ...
4. *Interfaces* `trait`, `impl`, ...

1. Refinements

1. Refinements

Index specifies *single value*

1. Refinements

Index specifies *single value*

Existential specifies *sets of values*

1. Refinements

Index specifies *single value*

Existential specifies *sets of values*

Index specifies *single value*

Index specifies *single value*

$$B[v]$$

Base Type Refine Index

Index specifies *single value*

`i32[5]`

The *singleton* `i32` that is equal to 5

Index specifies *single value*

`bool[true]`

The *singleton* `bool` that is equal to `true`

Index specifies *single value*

```
fn tt() → bool[true] {
   1 < 2
}
```

Output type specifies *Postcondition*

A function that *always returns* `true`

Index specifies *single value*

```
fn ff() → bool[false] {
  2 < 1
}
```

Output type specifies *Postcondition*

A function that *always returns* `false`

Index specifies *single value*

```
fn twelve() → i32[12] {
    4 + 8
}
```

Output type specifies *Postcondition*

A function that *always returns* 12

Index specifies *single value*

```
fn assert(b:bool[true]){}
```

Input type specifies *Precondition*

A function that *requires* input be `true`

Index specifies *single value*

```
fn assert(b:bool[true]){}
...
assert(1 < 2);
assert(10 < 2); // flux error!
```

Input type specifies ***Precondition***

A function that *requires* input be `true`

Index specifies *single value*

Constants are *boring*

Index specifies *single value*

Constants are *boring*

Parameterize signatures over refinements!

Index specifies *single value*

Index specifies *single value*

Refinement parameters

```
forall<n: int> fn (i32[n]) → bool[n > 0]
```

Index specifies *single value*

Refinement parameters

```
forall<n: int> fn (i32[n]) → bool[n > 0]
```

Declare with @-syntax

```
fn (i32[@n]) → bool[n > 0]
```

Index specifies *single value*

Refinement parameters

```
forall<n: int> fn (i32[n]) → bool[n > 0]
```

Declare with @-syntax

```
fn (i32[@n]) → bool[n > 0]
```

Or desugar from Rust

```
fn (n:i32) → bool[n > 0]
```

Refinement Parameters

Output's type *depends on* input

```
fn is_pos(n:i32) → bool[n>0] {
   n > 0
}
```

Refinement Parameters

Output's type *depends on* input

```
fn is_pos(n:i32) → bool[n>0] {
  n > 0
}

...
assert(is_pos(5));        // ok
```

Refinement Parameters

Output's type *depends on* input

```
fn is_pos(n:i32) → bool[n>0] {
  n > 0
}
…
assert(is_pos(5));       // ok
assert(is_pos(5 - 8)); // error
```

Refinement Parameters

Output's type *depends on* input

```
fn incr(n:i32) → i32[n+1] {
  n + 1
}
```

Refinement Parameters

Output's type *depends on* input

```
fn incr(n:i32) → i32[n+1] {
  n + 1
}
 ...
assert(incr(5 - 5) > 0);  // ok
```

Refinement Parameters

Output's type *depends on* input

```
fn incr(n:i32) → i32[n+1] {
  n + 1
}
...
assert(incr(5 - 5) > 0);  // ok
assert(incr(5 - 6) > 0); // error
```

Index specifies *single value*

B[v]

Index specifies ***single value***

$$B[v]$$

But what if we *don't know exact value?*

1. Refinements

Index specifies *single value*

Existential specifies *sets of values*

Existential specifies *sets of values*

HEREHERE

Refinements for Rust

1. *Refinements* `i32, bool, ...`
2. ***Ownership*** `mut, &, &mut, ...`
3. *Datatypes* `struct, enum, ...`
4. *Interfaces* `trait, impl, ...`

2. Ownership

Refinements for Rust

1. *Refinements* `i32`, `bool`, ...

2. *Ownership* `mut`, `&`, `&mut`, ...

3. ***Datatypes*** `struct`, `enum`, ...

4. *Interfaces* `trait`, `impl`, ...

3. Datatypes

Refinements for Rust

1. *Refinements* `i32`, `bool`, `...`
2. *Ownership* `mut`, `&`, `&mut`, `...`
3. *Datatypes* `struct`, `enum`, `...`
4. *Interfaces* `trait`, `impl`, `...`

4. Interfaces

END

Programmer-Aided Analysis

Programs

Refinements for Rust

Analysis

Type-directed Abstract-Interpretation

Programmer-Aided Analysis

Programs

Refinements for Rust

Analysis

Type-directed Abstract-Interpretation

Refinements for Rust

Refine using *Ownership*

Refine using *Ownership*

1. **Index** types with *pure values*

Refine using *Ownership*

1. **Index** types with *pure values*

2. **Update** refinements for *owned locations*

Refine using *Ownership*

1. **Index** types with *pure values*

2. **Update** refinements for *owned locations*

3. **Pack** invariants in *borrowed references*

Refine using *Ownership*

1. **Index** types with *pure values*

2. **Update** refinements for *owned locations*

3. **Pack** invariants in *borrowed references*

4. **Strong** updates using *strong references*

Refine using *Ownership*

1. **Index** types with *pure values*
2. **Update** refinements for *owned locations*
3. **Pack** invariants in *borrowed references*
4. **Strong** updates using *strong references*

Refine using *Ownership*

1. **Index** types with *pure values*

2. **Update** refinements for *owned locations*

3. **Pack** invariants in *borrowed references*

4. **Strong** updates using *strong references*

Refine using *Ownership*

1. **Index** types with *pure values*

2. **Update** refinements for *owned locations*

3. **Pack** invariants in *borrowed references*

4. **Strong** updates using *strong references*

Refine using *Ownership*

1. **Index** types with *pure values*
2. **Update** refinements for *owned locations*
3. **Pack** invariants in *borrowed references*
4. **Strong** updates using *strong references*

Programmer-Aided Analysis

Programs

Refinements for Rust

Analysis

Type-directed Abstract-Interpretation

Type-directed Abstract Interpretation