`Flux`: Liquid types for Rust

Nico Lehmann, Adam Geller, Gilles Barthe, Niki Vazou, Ranjit Jhala

Motivation

Types vs. Floyd-Hoare logic

Demonstration

Flux - Liquid Types for Rust

Evaluation

Flux v. Prusti for Memory safety

Types vs. Floyd-Hoare logic

Liquid/Refinements 101

```
type Nat = {v: Int | 0 <= v}
```

- `Int` is the *base type* of the value
- `v` names the *value* being described
- `0 <=v` is a *predicate* constraint

Liquid/Refinements 101

Generate the sequence of values between `lo` and `hi`

```
range :: lo:Int -> {hi:Int | lo <= hi} -> List {v:Int|lo <= v && v < hi}
```

Liquid/Refinements 101

Generate the sequence of values between `lo` and `hi`

```
range :: lo:Int -> {hi:Int | lo <= hi} -> List {v:Int|lo <= v && v < hi}
```

Input Type is a Precondition

`lo <= hi`

Liquid/Refinements 101

Generate the sequence of values between `lo` and `hi`

```
range :: lo:Int -> {hi:Int | lo <= hi} -> List {v:Int|lo <= v && v < hi}
```

Output Type is a Postcondition

Every element in sequence is between `lo` and `hi`

Types vs. Floyd-Hoare logic

> Types *decompose* (quantified) assertions to *quantifier-free* refinements

Lists in Dafny vs LiquidHaskell

```
datatype List<a>
  = Nil
  | Cons(head: a, tail: List<a>)
```

```
data List a
  = Nil
  | Cons {head :: a, tail :: List a}
```

Accessing the `i`-th List Element

```
function elements<a>(xs: List<a>): set<a>
{
  if xs == Nil
    then {}
    else {xs.head} + elements(xs.tail)
}

function method ith<a>(xs: List<a>, i:int, def:a): (res: a)
ensures res in elements(xs) + {def} {
  match xs
  case Cons(h, t) => if i == 0 then x else ith(t, i-1, def)
  case Nil        => def
}
```

```
ith :: List a -> Int -> a -> a

ith xs i def = case xs of
  Cons h t -> if i == 0 then h else ith t (i-1) def
  Nil      -> def
```

Floyd-Hoare requires `elements` and *quantified* axioms

Liquid Parametric *polymorphism* yields spec for free

Building and Using Lists

```
function method mkList(n: int, k: int) : (res: List<int>)
ensures forall v :: v in elements(res) ==> k <= v
{
  if (0 < n)
    then Cons(k, mkList(n - 1, k + 1))
    else Nil
}

method testPosN(i : int, n: nat, pos: List<int>)
{
  var pos := mkList(n, 100);
  if (0 <= i < 100) {
    assert (ith(pos, i, 1) > 0);
  }
}
```

```
mkList :: Int -> Int -> List Int

mkList n k =
   if 0 <  n
     then Cons k (mkList (n - 1) (k + 1))
     else Nil


testPosN :: Int -> Int -> ()
testPosN i n =
   let pos = mkList n 100 in
   if (0 < i && i < 100)
     then assert (ith pos i 1 > 0) ()
     else ()
```

Floyd-Hoare *Quantified* postcondition (hard to infer)

Building and Using Lists

```
function method mkList(n: int, k: int) : (res: List<int>)
ensures forall v :: v in elements(res) ==> k <= v
{
  if (0 < n)
    then Cons(k, mkList(n - 1, k + 1))
    else Nil
}

method testPosN(i : int, n: nat, pos: List<int>)
{
  var pos := mkList(n, 100);
  if (0 <= i < 100) {
    assert (ith(pos, i, 1) > 0);
  }
}
```

```
mkList :: Int -> Int -> List Int

mkList n k =
   if 0 <  n
     then Cons k (mkList (n - 1) (k + 1))
     else Nil


testPosN :: Int -> Int -> ()
testPosN i n =
   let pos = mkList n 100 in
   if (0 < i && i < 100)
     then assert (ith pos i 1 > 0) ()
     else ()
```

Liquid *Quantifier-free* type (easy to infer)

```
Int -> k:Int -> List {v:Int| k <= v}
```

Types vs. Floyd-Hoare logic

Types decompose assertions to quantif-free refinements ...

... but what about **imperative programs**

Motivation

Types vs. Floyd-Hoare logic

Demonstration

`Flux` *Liquid Types for Rust*

Evaluation

`Flux` v. `Prusti` for Memory Safety

`Flux` Liquid Types for Rust

`flux` (/flʌks/)

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

`Flux` Liquid Types for Rust

1. `basics`
2. `borrows`
3. `rvec-api`
4. `vectors`

Verification via Horn Clauses (1/3)

```
#[lr::sig(fn(lo: i32, hi:i32{lo ⩽ hi})
          → RVec<i32{v:lo⩽v<hi}>[hi - lo])]
pub fn range(lo: i32, hi: i32) → RVec<i32> {
    let mut i = lo;
    let mut res = RVec::new();
    while i < hi {
        res.push(i);
        i += 1;
    }
    res
}
```

```
∀lo: int.
  true ⇒
    ∀ hi: int.
      lo ≤ hi ⇒
        $k_i(lo)
        $k_size(0)
        ∀ v: int{false}. $k_val(v)
        ∀ n: int{$k_size(n)}, i: int{$k_i(i)}.
          ¬(i < hi) ⇒
            ∀ v: int{$k_val(v)}. (lo ≤ v ∧ v < hi)
            (n = hi - lo)
          i < hi ⇒
            $k_val(i)
            $k_size(n + 1)
            $k_i(i + 1)
```

Verification via Horn Clauses (2/3)

```
#[lr::sig(fn(lo: i32, hi:i32{lo ⩽ hi})
        → RVec<i32{v:lo⩽v<hi}>[hi - lo])]
pub fn range(lo: i32, hi: i32) → RVec<i32> {
    let mut i = lo;
    let mut res = RVec::new();
    while i < hi {
        res.push(i);
        i += 1;
    }
    res
}
```

```
∀lo: int.
  true ⇒
    ∀ hi: int.
      lo ≤ hi ⇒
        $k_i(lo)
        $k_size(0)
        ∀ v: int{false}. $k_val(v)
        ∀ n: int{$k_size(n)}, i: int{$k_i(i)}.
          ¬(i < hi) ⇒
            ∀ v: int{$k_val(v)}. (lo ≤ v ∧ v < hi)
            (n = hi - lo)
          i < hi ⇒
            $k_val(i)
            $k_size(n + 1)
            $k_i(i + 1)
```

Verification via Horn Clauses (3/3)

```
#[lr::sig(fn(lo: i32, hi:i32{lo ⩽ hi})
        → RVec<i32{v:lo⩽v<hi}>[hi - lo])]
pub fn range(lo: i32, hi: i32) → RVec<i32> {
    let mut i = lo;
    let mut res = RVec::new();
    while i < hi {
        res.push(i);
        i += 1;
    }
    res
}
```

```
∀lo: int.
  true ⇒
    ∀ hi: int.
      lo ≤ hi ⇒
        $k_i(lo)
        $k_size(0)
        ∀ v: int{false}. $k_val(v)
        ∀ n: int{$k_size(n)}, i: int{$k_i(i)}.
          ¬(i < hi) ⇒
            ∀ v: int{$k_val(v)}. (lo ≤ v ∧ v < hi)
            (n = hi - lo)
          i < hi ⇒
            $k_val(i)
            $k_size(n + 1)
            $k_i(i + 1)
```

```
SOLUTION  $k_i(i)    := lo <= i && i <= hi
          $k_val(i)  := lo <= i && i <  hi
          $k_size(n) := n  == i - lo
```

Motivation

Types vs. Floyd-Hoare logic

Demonstration

`Flux` *Liquid Types for Rust*

Evaluation

`Flux` v. `Prusti` for Memory Safety

Evaluation

`Flux` v. `Prusti` for Memory Safety

`Flux` v. `Prusti` by the numbers

	Flux			Prusti			
	LOC	Spec	Time (s)	LOC	Spec	Annot	Time (s)
Library							
`RVec`	47	22	-	45	29	-	-
`RMat`	30	8	1.4	41	16	-	-
Total	77	30	1.4	86	45	-	-
Benchmark							
`bsearch`	25	1	0.8	25	0	1	2.3
`dotprod`	12	1	0.7	12	1	1	2.1
`fft`	180	17	3.1	188	22	24	240.6
`heapsort`	39	2	1.1	37	5	9	7.7
`simplex`	124	10	2.7	125	25	8	21.6
`kmeans`	94	12	2.7	87	37	10	21.1
`kmp`	48	2	1.9	49	4	7	9.9
Total	522	45	13	523	94	60	304.2

`Flux` v. `Prusti` : Types Simplify Specifications

```
// Rust
fn store(&mut self, idx: usize, value: T)

// Flux
fn store(self: &mut RVec<T>[@n], idx: usize{idx < n}, value: T)

// Prusti
requires(index < self.len())
ensures(self.len() == old(self.len()))
ensures(forall(|i:usize| (i < self.len() && i != index) ==>
                      self.lookup(i) == old(self.lookup(i))))
ensures(self.lookup(index) == value)
```

Quantifiers make SMT *slow*!

`Flux` v. `Prusti` : Types Enable Code Reuse

Example: `kmeans.rs` in `flux`

```
fn kmeans(n:usize, cs: k@RVec<RVec<f32>[n]>, ps: &RVec<RVec<f32>[n]>, iters: i32) -> RVec<RVec<f32>[n]>[k] where 0 < k
```

- **Point** is an `n` dimensional float-vec `RVec<f32>[n]`
- **Centers** are a vector of `k` points `RVec<RVec<f32>[n]>[k]`

`Flux` v. `Prusti` : Types Enable Code Reuse

Example: `kmeans.rs` in `prusti`

```
ensures(forall(|i:usize| (i < self.len() && i != index) ==>
                    self.lookup(i) == old(self.lookup(i))))
```

Value equality *prevents vector nesting!*

Have to duplicate code in new (untrusted) wrapper library

`Flux` v. `Prusti` : Types Simplify Invariants & Inference

Dimension preservation obfuscated by `Prusti` spec

```
#[requires(i < self.rows() && j < self.cols())]
#[ensures(self.cols() == old(self.cols()) && self.rows() == old(self.rows()))]
pub fn set(&mut self, i: usize, j: usize, value: T) {
  self.inner[i][j] = value;
}
```

Hassle programmer for dimension preservation invariants

- `kmeans::normalize_centers` in prusti vs. flux
- `fft::loop_a` in prusti vs. flux

Burden programmer with dimension preservation invariants

- `fft::loop_a` in prusti vs. flux

```
113- #[lr::sig(fn (px: &mut n@RVec<f32>, py: &mut RVec<f32>[n]))]
114  fn loop_b(px: &mut RVec<f32>, py: &mut RVec<f32>) {
115      let n = px.len() - 1;
116      let mut is = 1;
117      let mut id = 4;
118      while is < n {
119          let mut i0 = is;
120          let mut i1 = is + 1;
121          while i1 ⩽ n {
122              let r1 = *px.get(i0);
123              *px.get_mut(i0) = r1 + *px.get(i1);
124              *px.get_mut(i1) = r1 - *px.get(i1);
125
126              let r1 = *py.get(i0);
127              *py.get_mut(i0) = r1 + *py.get(i1);
128              *py.get_mut(i1) = r1 - *py.get(i1);
129
130              i0 = i0 + id;
131              i1 = i1 + id;
132          }
133          is = 2 * id - 1;
134          id = 4 * id;
135      }
136  }
```

```
130+ #[requires(px.len() == py.len())]
131+ #[ensures(px.len() == old(px.len()))]
132+ #[ensures(py.len() == old(py.len()))]
133  fn loop_b(px: &mut RVec<f32>, py: &mut RVec<f32>) {
134      let n = px.len() - 1;
135+     let px_len = px.len();
136+     let py_len = py.len();
137+
138      let mut is = 1;
139      let mut id = 4;
140+
141      while is < n {
142+         body_invariant!(n < px.len() && n < py.len());
143+         body_invariant!(py.len() == py_len);
144+         body_invariant!(px.len() == px_len);
145          let mut i0 = is;
146          let mut i1 = is + 1;
147          while i1 ⩽ n {
148+             body_invariant!(n < px.len() && n < py.len());
149+             body_invariant!(i0 ⩽ i1 && i1 < px.len());
150+             body_invariant!(py.len() == py_len);
151+             body_invariant!(px.len() == px_len);
152              let r1 = *px.get(i0);
153              *px.get_mut(i0) = r1 + *px.get(i1);
154              *px.get_mut(i1) = r1 - *px.get(i1);
155
156              let r1 = *py.get(i0);
157              *py.get_mut(i0) = r1 + *py.get(i1);
158              *py.get_mut(i1) = r1 - *py.get(i1);
159
160              i0 = i0 + id;
161              i1 = i1 + id;
162          }
163          is = 2 * id - 1;
164          id = 4 * id;
165      }
166  }
```

`Flux` v. `Prusti` : Types Simplify Invariants & Inference

```
#[lr::sig(fn(lo: usize, hi:usize{lo ⩽ hi})
          → RVec<i32{v:lo⩽v && v<hi}>[hi - lo])]

pub fn range(lo: usize, hi: usize) → RVec<i32> {
    let mut i = lo;
    let mut res = RVec::new();
    while i < hi {
        res.push(i);
        i += 1;
    }
    res
}

fn _test_range(lo: usize, hi: usize) {
    if lo ⩽ hi {
        let mut rng = range(lo, hi);
        while !rng.is_empty() {
            let val = rng.pop();
            assert(lo ⩽ val);
        }
    }
}
```

```
#[requires(lo ⩽ hi)]
#[ensures(result.len() == hi - lo)]
#[ensures(forall(|x : usize| (0 ⩽ x && x < result.len())
                ⟹ (lo ⩽ result.lookup(x) && result.lookup(x) < hi)))]
pub fn range(lo: usize, hi: usize) → RVec {
    let mut i = lo;
    let mut res = RVec::new();
    while i < hi {
        body_invariant!(i ⩾ lo && i < hi);
        body_invariant!(res.len() == i - lo);
        body_invariant!(forall(|x: usize| (0 ⩽ x && x < res.len())
                                ⟹ (lo ⩽ res.lookup(x) && res.lookup(x) < hi)));
        res.push(i);
        i += 1;
    }
    res
}

fn _test_range(lo: usize, hi: usize) {
    if lo ⩽ hi {
        let mut rng = range(lo, hi);
        while !rng.is_empty() {
            body_invariant!(0 < rng.len());
            body_invariant!(forall(|x : usize| (0 ⩽ x && x < rng.len())
                                    ⟹ (lo ⩽ rng.lookup(x) && rng.lookup(x) < hi)));
            let val = rng.pop();
            assert(lo ⩽ val);
        }
    }
}
```

Types *decompose* quantified assertions to *quantifier-free* refinements

`flux` infers quantifier-free refinements via Horn-clauses/Liquid Typing

`Flux` v. `Prusti` : Types Simplify Invariants & Inference

`kmp_search` in prusti vs. flux

```
// Prusti
body_invariant!(forall(|x: usize| x < t.len() ==> t.lookup(x) < pat_len));

// Flux
t: RVec<{v:v < pat_len}>
```

Types *decompose* quantified assertions to *quantifier-free* refinements

`flux` infers quantifier-free refinements via Horn-clauses/Liquid Typing

Types *decompose* quantified assertions to *quantifier-free* refinements

`kmp_search` in prusti vs. flux

```
33- #[lr::sig(fn(pat: RVec<u8>{0 < pat && pat ⩽ n}, target: &n@RVec<u8>{0 < n})
34  fn kmp_search(mut pat: RVec<u8>, target: &RVec<u8>) → usize {
35      let mut t_i = 0;
36      let mut p_i = 0;
37      let mut result_idx = 0;
38      let target_len = target.len();
39      let pat_len = pat.len();
40
41      let t = kmp_table(&mut pat);
42
43      while t_i < target_len && p_i < pat_len {
                ////////////////////////////////
                ////////////////////////////////
                ////////////////////////////////
                ////////////////////////////////
                ////////////////////////////////
44          if *target.get(t_i) == *pat.get(p_i) {
45              if result_idx == 0 {
46                  result_idx = t_i;
47              }
48              t_i = t_i + 1;
49              p_i = p_i + 1;
50              if p_i ⩾ pat_len {
51                  return result_idx;
52              }
53          } else {
54              if p_i == 0 {
55                  p_i = 0;
56              } else {
57                  p_i = *t.get(p_i - 1);
58              }
59              t_i = t_i + 1;
60              result_idx = 0;
61          }
62      }
63      target.len()
64  }
```

```
41+ #[requires((pat.len() > 0) && (target.len() > 0) && (target.len() ⩾ pat.len()))]
42  fn kmp_search(mut pat: RVec<u8>, target: &RVec<u8>) → usize {
43      let mut t_i = 0;
44      let mut p_i = 0;
45      let mut result_idx = 0;
46      let target_len = target.len();
47      let pat_len = pat.len();
48
49      let t = kmp_table(&mut pat);
50
51      while t_i < target_len && p_i < pat_len {
52+         body_invariant!(p_i < pat.len());
53+         body_invariant!(t.len() == pat.len());
54+         body_invariant!(forall(|x: usize| x < t.len() ⟹ t.lookup(x) < pat_len));
55+         body_invariant!(result_idx ⩽ t_i);
56          if *target.get(t_i) == *pat.get(p_i) {
57              if result_idx == 0 {
58                  result_idx = t_i;
59              }
60              t_i = t_i + 1;
61              p_i = p_i + 1;
62              if p_i ⩾ pat_len {
63                  return result_idx;
64              }
65          } else {
66              if p_i == 0 {
67                  p_i = 0;
68              } else {
69                  p_i = *t.get(p_i - 1);
70              }
71              t_i = t_i + 1;
72              result_idx = 0;
73          }
74      }
75      target.len()
76  }
```

Motivation

Types vs. Floyd-Hoare logic

Demonstration

`Flux` Liquid Types for Rust

Evaluation

`Flux` v. `Prusti` for Memory Safety

Conclusions

Refinements + Rust's Ownership = Ergonomic Imperative Verification...

- Specify complex invariants by *composing* type constructors & QF refinements
- Verify complex invariants by *decomposing* validity checks via syntactic subtyping

Conclusions

Refinements + Rust's Ownership = Ergonomic Imperative Verification...

- Specify complex invariants by *composing* type constructors & QF refinements
- Verify complex invariants by *decomposing* validity checks via syntactic subtyping

... But this is just the beginning

- `Flux` restricts specifications, `Prusti` allows *way* more ...
- ... how to stretch types to "full functional correctness"?

What are interesting application domains to focus on?

`flux` https://github.com/liquid-rust/flux/