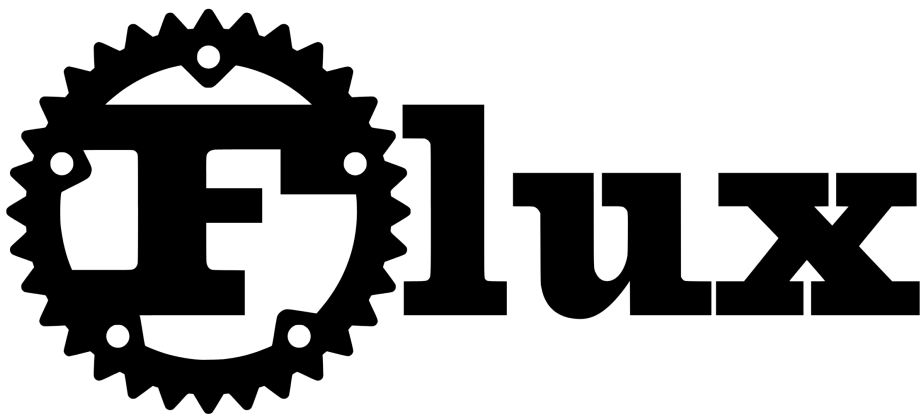
Liquid Types for Rust



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

What *is* Programming Languages Research?

“We shall make

thoughtcrime

literally impossible:

there will be no words

to express it.”

— George Orwell (1984)

“We shall make

~~thoughtcrime~~ bugs

literally impossible:

there will be no words

to express it.”

— George Orwell (1984)

What *is* Programming Languages Research?

"We shall make

~~thoughtcrime~~ bugs

literally impossible:

there will be no words

to express it."

— George Orwell (1984)

Null Derefs

What *is* Programming Languages Research?

"We shall make

~~thoughtcrime~~ bugs

literally impossible:

there will be no words

to express it."

— George Orwell (1984)

Null Derefs

Array Overflows

What *is* Programming Languages Research?

"We shall make

~~thoughtcrime~~ bugs

literally impossible:

there will be no words

to express it."

— George Orwell (1984)

Null Derefs

Array Overflows

Integer Overlflows

What *is* Programming Languages Research?

"We shall make

~~thoughtcrime~~ bugs

literally impossible:

there will be no words

to express it."

— George Orwell (1984)

Null Derefs

Array Overflows

Integer Overlflows

User def. invariants

What *is* Programming Languages Research?

"We shall make

~~thoughtcrime~~ bugs

literally impossible:

there will be no words

to express it."

— George Orwell (1984)

Null Derefs

Array Overflows

Integer Overlflows

User def. invariants

Security Requirements

What *is* Programming Languages Research?

"We shall make

~~thoughtcrime~~ bugs

literally impossible:

there will be no words

to express it."

— George Orwell (1984)

Null Derefs

Array Overflows

Integer Overlflows

User def. invariants

Security Requirements

Functional Correctness

What *is* Programming Languages Research?

But … *how*?

Null Derefs

Array Overflows

Integer Overlflows

User def. invariants

Security Requirements

Functional Correctness

Refinement Types

Constable & Smith 1987, Rushby et al. 1997

Refinement Types

Constable & Smith 1987, Rushby et al. 1997

$$B\{x : p\}$$

Base-type Value name Refinement

Refinement Types

Constable & Smith 1987, Rushby et al. 1997

$$B\{x : p\}$$

Base-type Value name Refinement

"Set of values x of type B such that p is true"

Refinement Types

Constable & Smith 1987, Rushby et al. 1997

```
Int{x : 0 < x}
```

Base-type Value name Refinement

"Set of *positive integers*"

Refinement Types for *Functional* Code

LiquidHaskell

Vazou et al. 2014

F⋆

Swamy et al. 2016

SAIL

Sewell et al. 2019

ASL

Reid 2019

Refinement Types for *Imperative* Code ?

Refinement Types for *Imperative* Code ?

```
fn foo(mut x: i32, y: i32{v: x < v}) {
  x += 1;
  ...
}
```

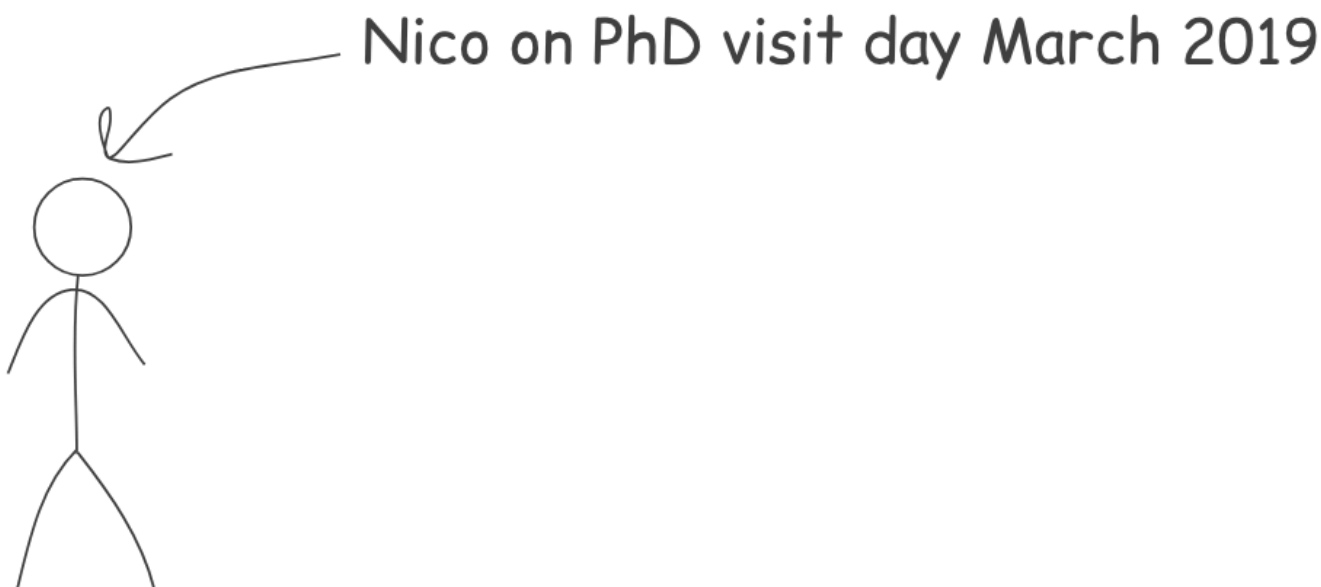
Refinement Types for *Imperative* Code ?

```
fn foo(mut x: i32, y: i32{v: x < v}) {
  x += 1;
  ...
}
```

Problem: Dependency on *mutable* variables!

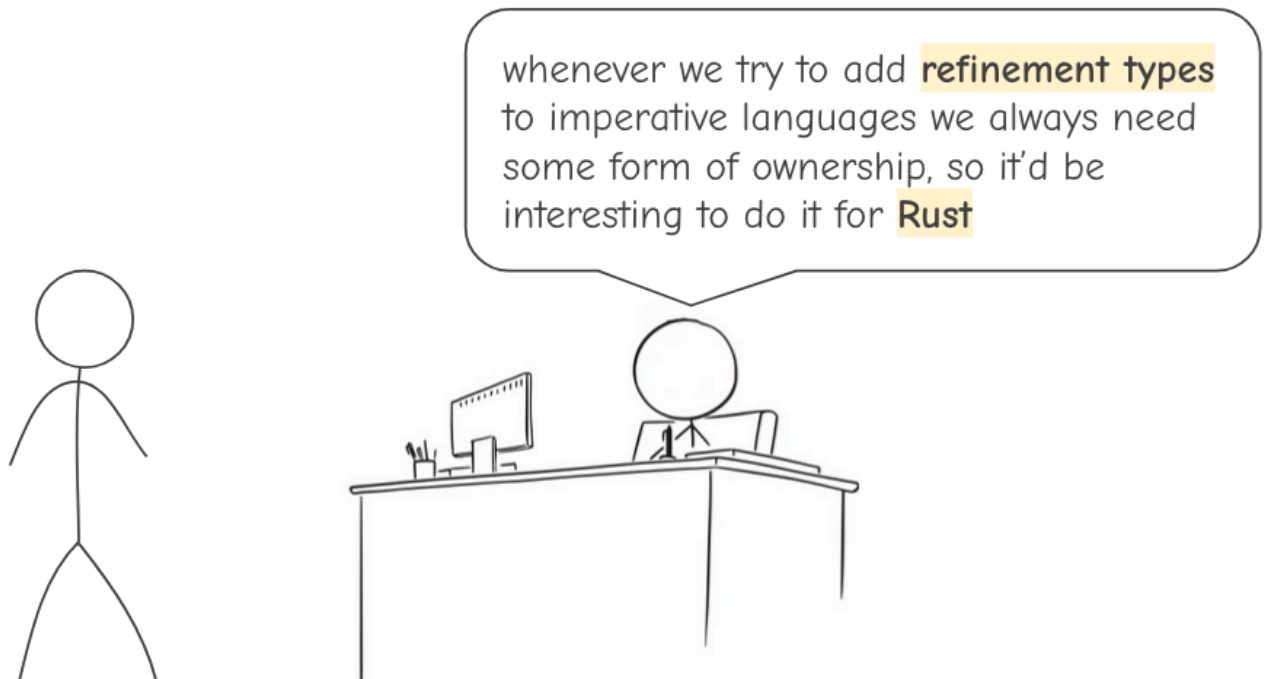
March 2019

March 2019

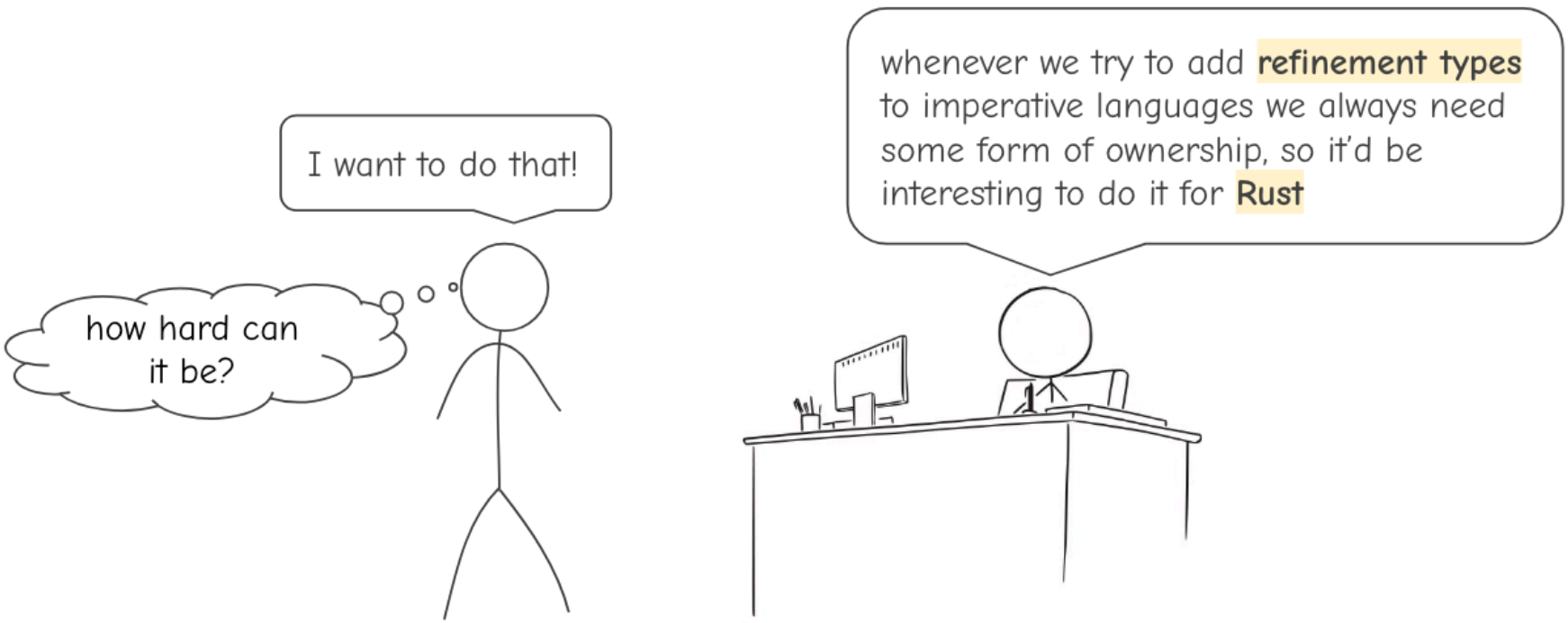

Nico on PhD visit day March 2019


Ranjit


are burritos
better in
San Diego?


whenever we try to add refinement types to imperative languages we always need some form of ownership, so it'd be interesting to do it for Rust

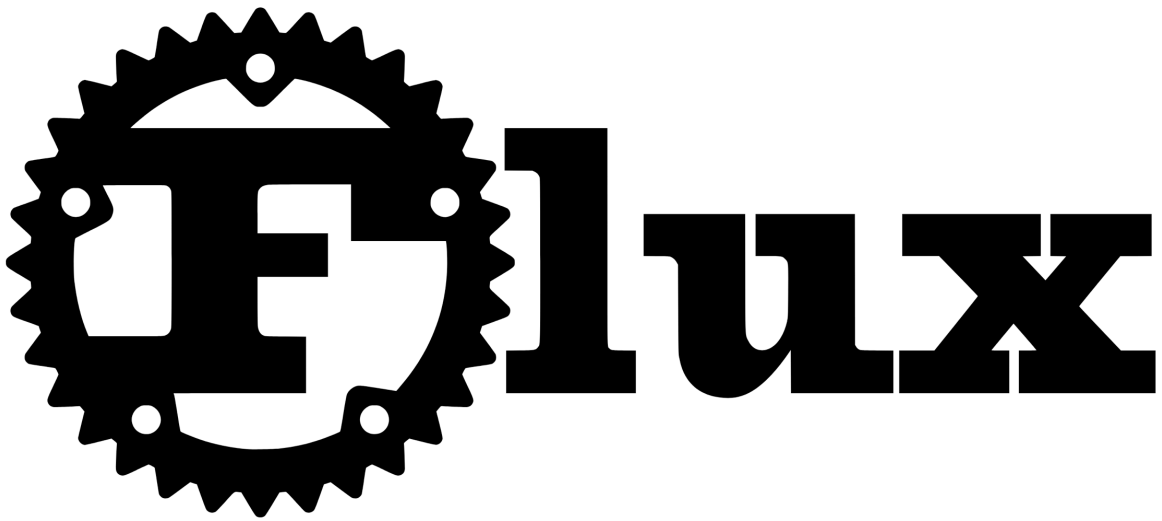

I want to do that!
whenever we try to add refinement types to imperative languages we always need some form of ownership, so it'd be interesting to do it for Rust


I want to do that!
how hard can it be?
whenever we try to add refinement types to imperative languages we always need some form of ownership, so it'd be interesting to do it for Rust

… 6 years† later

† and 62KLoc and 1,556 commits…


What is one thing you would change about
SEIZE THE MEANS OF COMPUTATION!



(/flʌks/)

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

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`, `&`, `&mut`, ...
3. *Datatypes* `struct`, `enum`, ...
4. *Interfaces* `trait`, `impl`, ...

1. Refinements

1. Refinements

Index specifies *single value*

1. Refinements

Index specifies *single value*

Parameters abstract over *input values*

1. Refinements

Index specifies *single value*

Parameters abstract over *input values*

Existential specifies unknown *sets of values*

1. Refinements

Index specifies *single value*

Parameters abstract over *input values*

Existential specifies unknown *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*!

Parameters abstract over *input values*

Parameters abstract over *input values*

Refinement parameters

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

Parameters abstract over *input values*

Refinement parameters

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

Declare with @-syntax

```
fn (i32[@n]) → bool[n > 0]
```

Parameters abstract over *input values*

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]
```

Parameters abstract over *input values*

Output's type *depends on input*

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

Parameters abstract over *input values*

Output's type *depends on input*

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

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

Parameters abstract over *input values*

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
```

Parameters abstract over *input values*

Output's type *depends on input*

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

Parameters abstract over *input values*

Output's type *depends on input*

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

...
assert(incr(5 - 5) > 0);  // ok
```

Parameters abstract over *input values*

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
```

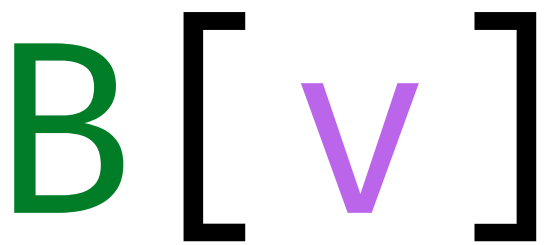
1. Refinements

Index specifies *single value*

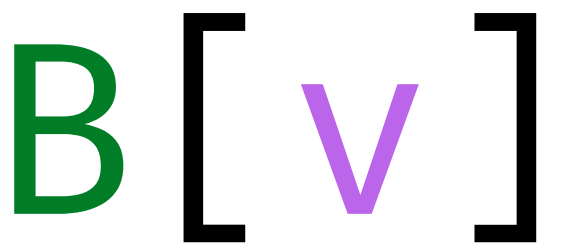
Parameters abstract over *input values*

Existential specifies unknown *sets of values*

Index specifies *single value*



Index specifies *single value*



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

1. Refinements

Index specifies *single value*

Parameters abstract over *input values*

Existential specifies unknown *sets of values*

Existential specifies *sets of values*

Existential specifies *sets of values*

$$B\{v : p(v)\}$$

Base Type Constraint

Existential specifies *sets of values*

$$B\{v : p(v)\}$$

Set of `B` values whose index `v` satisfies `p(v)`

Existential specifies *sets of values*

`i32{v:0 ⩽ v}`

`i32` values that are *non-negative*

Existential specifies *sets of values*

`usize{v:v < n}`

`usize` values *less than* `n`

Existential specifies *sets of values*

abs *returns* a non-negative `i32`

```
fn abs(n:i32) → i32{v:0⩽v} {
  if n < 0 {
    0 - n
  } else {
    n
  }
}
...
assert(abs(n) ⩾ 0);
assert(abs(n) >= n); // EX: How to fix?
```

Existential specifies *sets of values*

get *requires* a valid index!

```
fn get<T>(x: &[T], i: usize) → &T
{
  &x[i] // EX: How to fix?
}
```

Existential specifies *sets of values*

`get` *requires* a valid index!

```
fn get<T>(x: &[T], i: usize) → &T
{
  &x[ i ] // EX: How to fix?
}
```

What is a suitable type for the input `i`?

Existential specifies *sets of values*

`get` *requires* a valid index!

```
fn get<T>(x: &[T][@n], i: usize{v:v<n}) → &T
{
  &x[ i ]
}
```

Precondition: `i` less than *size of slice* `n`

1. Refinements

Index specifies *single value*

Parameters abstract over *input values*

Existential specifies unknown *sets of values*

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`, `&`, `&mut`, ...

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

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

2. *Ownership*

2. Ownership

Update types at *owned locations*

2. Ownership

Update types at *owned locations*

Preserve refinements in *borrows*

2. Ownership

Update types at *owned locations*

Preserve refinements in *borrows*

Strong updates at *mutable borrows*

2. *Ownership*

Update types at *owned locations*

Preserve refinements in *borrows*

Strong updates at *mutable borrows*

Update types at *owned locations*

Update types at *owned locations*

```
let mut x = 0;     // x : i32[0]
assert(x == 0);
```

Exclusive ownership allows strong updates

Update types at *owned locations*

```
let mut x = 0;      // x : i32[0]
assert(x == 0);
x += 10;            // x : i32[10]
assert(x == 10);
```

Exclusive ownership allows strong updates

Update types at *owned locations*

```
let mut x = 0;      // x : i32[0]
assert(x == 0);
x += 10;            // x : i32[10]
assert(x == 10);
x += 10;            // x : i32[20]
assert(x == 20);
```

Exclusive ownership allows strong updates

2. *Ownership*

Update types at *owned locations*

Preserve refinements in *borrows*

Strong updates at *mutable borrows*

Preserve refinements in *borrows*

Preserve refinements in *borrows*

```
fn read(x: &i32{v: 0 < v}) {
  let n = *x;
  assert(0 < n);
}
```

`&` reference = aliasing but *no mutation*

Value *read* through `&` reference satisfies refinements

Preserve refinements in *borrows*

```
fn incr(x: &mut i32{v: 0 < v}) {
  *x += 1;
}
```

`&mut` reference = mutation but *no aliasing*

Value *written* through `&mut` *must preserve* refinements

Preserve refinements in *borrows*

```
fn decr(x: &mut i32{v: 0 < v}) {
    *x -= 1; // EX: how to fix?
}
```

Value *written* through `&mut` *must preserve* refinements

Exercise: How to *fix* the error in `decr` ?

Preserve refinements in *borrows*

```
fn incr(x: &mut i32{v: 0 < v}) {
  *x += 1;
}
```

Mutable borrow `&mut i32`: *refinement-preserving* writes

Preserve refinements in *borrows*

```
fn incr(x: &mut i32{v: 0 < v}) {
  *x += 1;
}
fn test() {
  let mut z = 1;   // z: i32[1]
```

Mutable borrow `&mut i32`: *refinement-preserving* writes

Preserve refinements in *borrows*

```
fn incr(x: &mut i32{v: 0 < v}) {
  *x += 1;
}
fn test() {
  let mut z = 1;   // z: i32[1]
  incr(&z);
  assert(z == 2); // z: i32{v:0 < v}
}
```

Mutable borrow `&mut i32`: *refinement-preserving* writes

Preserve refinements in *borrows*

```
fn incr(x: &mut i32{v: 0 < v}) {
  *x += 1;
}
fn test() {
  let mut z = 1;    // z: i32[1]
  incr(&z);
  assert(z == 2); // z: i32{v:0 < v}
}
```

Need to specify `incr` *updates* the type of `x`

2. *Ownership*

Update types at *owned locations*

Preserve refinements in *borrows*

Strong updates at *mutable borrows*

Strong updates at *mutable borrows*

Strong updates at *mutable borrows*

```
fn incr(x: &mut i32[@n]) ensures x: i32[n+1] {
  *x += 1;
}
```

Strong updates at *mutable borrows*

```
fn incr(x: &mut i32[@n]) ensures x: i32[n+1] {
  *x += 1;
}
fn test() {
  let mut z = 1;  // z: i32[1]
```

Strong updates at *mutable borrows*

```
fn incr(x: &mut i32[@n]) ensures x: i32[n+1] {
  *x += 1;
}
fn test() {
  let mut z = 1;  // z: i32[1]
  incr(&z);
  assert(z == 2); // z: i32[2]
}
```

Strong updates at *mutable borrows*

```
fn incr(x: &mut i32[@n]) ensures x: i32[n+1] {
  *x += 1;
}
fn test() {
  let mut z = 1;  // z: i32[1]
  incr(&z);
  assert(z == 2); // z: i32[2]
}
```

Sound as `&mut` is *unique* (no aliasing)

2. Ownership

Update types at *owned locations*

Preserve refinements in *borrows*

Strong updates at *mutable borrows*

Refinements for Rust

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

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

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

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

3. Datatypes

Compositional specification & verification

3. Datatypes

Compositional specification & verification

"Make illegal states unrepresentable"

3. Datatypes

`struct`

"Product" types

`enum`

"Sum" types

3. Datatypes

`struct`

"Product" types

`enum`

"Sum" types

`struct`

Example: *Refined Vectors*

`struct`: *Refined Vectors*

```
struct RVec<T> {
  inner: Vec<T>;
}
```

RVec is a *wrapper* around *built-in* Vec

`struct`: *Refined Vectors*

```
#[refined_by(len: int)]
struct RVec<T> {
  inner: Vec<T>;
}
```

`refined_by`: *refinement value(s)* tracked for `RVec<T>`

Refined Vectors: Specification

Refined Vectors: *Specification*

```
fn new() → RVec<T>[{len: 0}]
```

Create a Refined Vector

Newly *returned* vector has size `0`

Refined Vectors: *Specification*

```
fn push(self: &mut RVec<T>[@v], val:T)
   ensures
     self: RVec<T>[{len: v.len + 1}]
```

Push value into a Refined Vector

Pushing *increases size* by 1

Refined Vectors: *Specification*

```
fn len(&RVec<T>[@v]) → usize[v.len]
```

Compute the *length* of a Refined Vector

Output `usize` indexed by *input* vector's size

Refined Vectors: *Verification*

Refined Vectors: *Verification*

```
let mut v = RVec::new();  // v: RVec<i32>[0]
```

Strong update *changes type* of `v` after each `push`

Refined Vectors: *Verification*

```
let mut v = RVec::new();  // v: RVec<i32>[0]
v.push(10);               // v: RVec<i32>[1]
```

Strong update *changes type* of `v` after each `push`

Refined Vectors: *Verification*

```
let mut v = RVec::new();  // v: RVec<i32>[0]
v.push(10);               // v: RVec<i32>[1]
v.push(20);               // v: RVec<i32>[2]
```

Strong update *changes type* of `v` after each `push`

Refined Vectors: *Verification*

```
let mut v = RVec::new();  // v: RVec<i32>[0]
v.push(10);               // v: RVec<i32>[1]
v.push(20);               // v: RVec<i32>[2]
assert(v.len() == 2);
```

Strong update *changes type* of `v` after each `push`

Refined Vectors: *Verification*

```
let mut v = RVec::new();   // v: RVec<i32>[0]
v.push(10);                // v: RVec<i32>[1]
v.push(20);                // v: RVec<i32>[2]
assert(v.len() == 2);
v.push(30);                // v: RVec<i32>[3]
assert(v.len() == 2);
```

Strong update *changes type* of `v` after each `push`

Refined Vectors: *Verification*

```
fn init<F, A>(n: usize, mut f: F) → RVec<A>[n]
where
  F: FnMut(usize{v:v < n}) → A,
```

Refined Vectors: *Verification*

```
fn init<F, A>(n: usize, mut f: F) → RVec<A>[n]
where
  F: FnMut(usize{v:v < n}) → A,
{
  let mut i = 0;
  let mut res = RVec::new();  // res: ?
```

Refined Vectors: *Verification*

```
fn init<F, A>(n: usize, mut f: F) → RVec<A>[n]
where
  F: FnMut(usize{v:v < n}) → A,
{
  let mut i = 0;
  let mut res = RVec::new();  // res: ?
  while i < n  {
    res.push(f(i));
    i += 1;                   // res: ?
  }
```

Refined Vectors: *Verification*

```
fn init<F, A>(n: usize, mut f: F) → RVec<A>[n]
where
  F: FnMut(usize{v:v < n}) → A,
{
  let mut i = 0;
  let mut res = RVec::new();  // res: ?
  while i < n  {
    res.push(f(i));
    i += 1;                   // res: ?
  }
  res                         // res: ?
}
```

Refined Vectors: *Verification*

```
fn init<F, A>(n: usize, mut f: F) → RVec<A>[n]
where
  F: FnMut(usize{v:v < n}) → A,
{
  let mut i = 0;
  let mut res = RVec::new();  // res: ?
  while i < n  {
    res.push(f(i));
    i += 1;                   // res: ?
  }
  res                         // res: ?
}
```

Refined Vectors: *Specification*

```
// get `i`-th element of vector
fn get(&RVec<T>[@v], usize{i: i < v.len}) → &T

// set `i`-th element of vector
fn set(&mut RVec<T>[@n], usize{i: i < v.len}, val:T)
```

Access elements of a Refined Vector

Require index `i` to be within vector `v` bounds

Refined Vectors: *Verification*

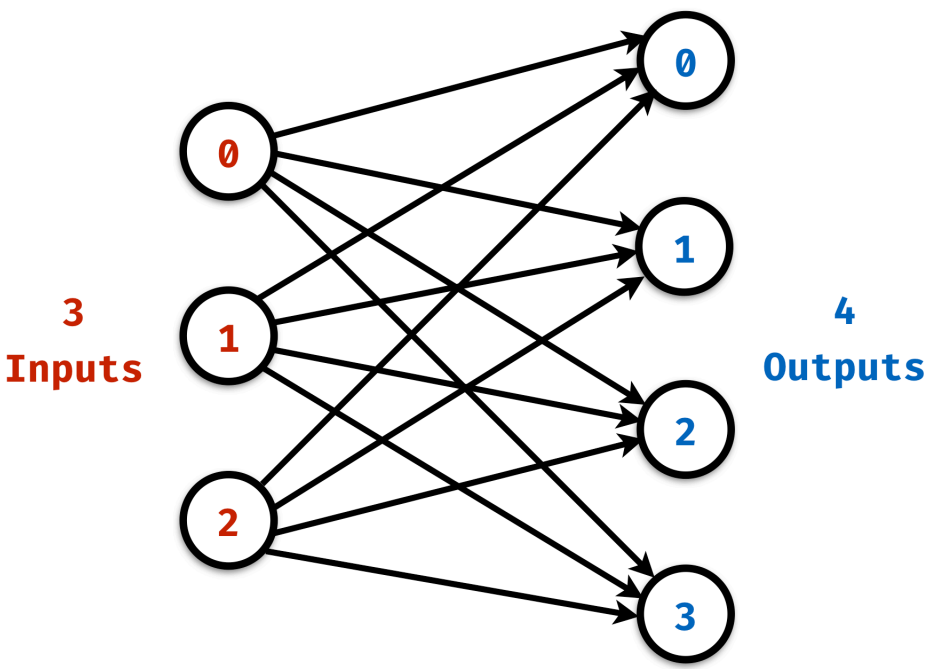
```
fn dot(x:&RVec<f64>, y:&RVec<f64>) → f64 {
  let mut res = 0.0;
  let mut i = 0;
  while (i < xs.len()) {
    res += xs[i] + ys[ i ];
    i += 1;
  }
  res
}
```

How can we *fix* the error?

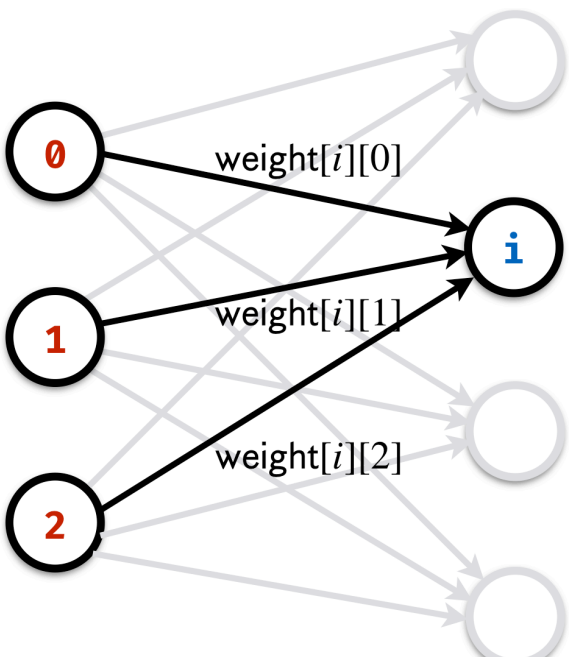
`struct`

Example: *Neuron Layer*

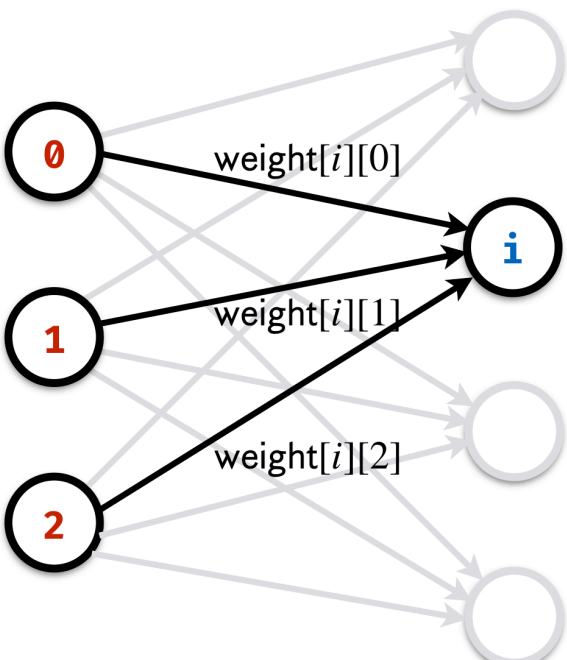
Example: *Neuron Layer*



Example: *Neuron Layer*



Neuron Layer: *Specification*

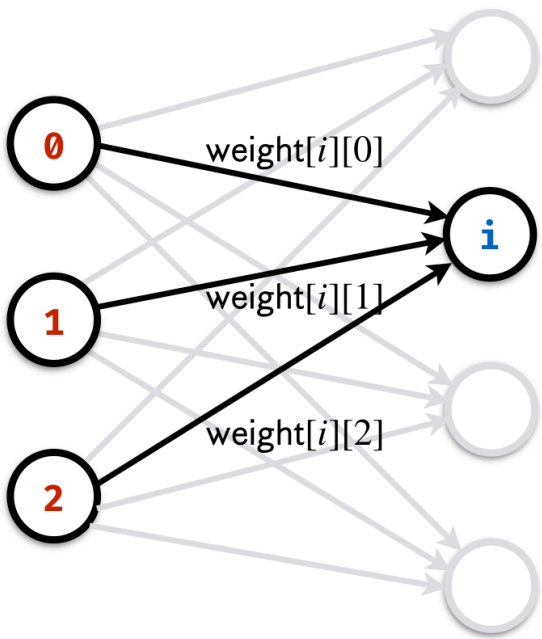



```
#[refined_by(i: int, o: int)]
struct Layer {
  num_inputs: usize[i],
  num_outputs: usize[o],
  weight: RVec<RVec<f64>[i]>[o],
  bias: RVec<f64>[o],
  outputs: RVec<f64>[o],
}
```

Neuron Layer: *Verification*

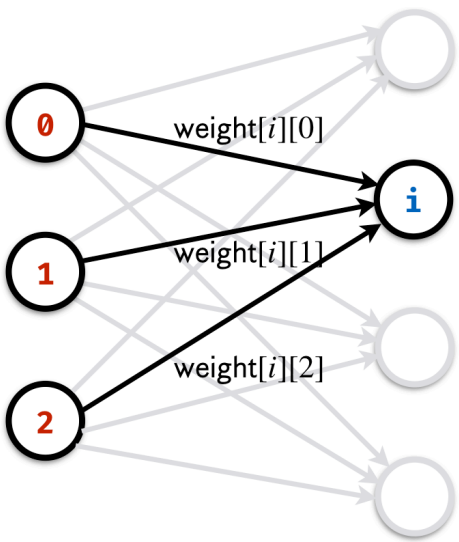
```
fn new(i: usize, o: usize) -> Layer[i, o] {
  let mut rng = rand::thread_rng();
  Layer {
    num_inputs: i,
    num_outputs: o,
    weight: init(o, |_| init(i, |_|
    rng.gen_range(-1.0..1.0))),
    bias: init(o, |_| rng.gen_range(-1.0..1.0)),
    outputs: init(o, |_| 0.0),
  }
}
```

Neuron Layer: *Forward Propagation*



$$\text{out}[i] \doteq \sigma(\text{in} \,.\, \text{weight}[i] + \text{bias}[i])$$

Neuron Layer: *Forward Propagation*



$$\text{out}[i] \doteq \sigma(\text{in} \,.\, \text{weight}[i] + \text{bias}[i])$$

```
fn forward(&mut self, input: &RVec<f64>) {
  (0..self.num_outputs).for_each(|i| {
    let in_wt = dot(&self.weight[i], input);
    let sum = in_wt + self.bias[i];
    self.outputs[i] = sigmoid(sum);
  })
}
```

3. Datatypes

`struct`

"Product" types

`enum`

"Sum" types

`enum`

Example: *Lists*

Lists: *Specification*

```
enum List<T> {
  Nil,
  Cons(T, Box<List<T>>),
}
```

Unrefined `List` specification

Lists: *Specification*

```
#[refined_by(size : int)]
enum List<T> {
  Nil → List[0],
  Cons(T, Box<List<T>[@n]>) → List[n+1],
}
```

Refined `List` indexed by *size* (or *set* or *seq* of values)

Lists: *Verification*

```
fn append<T>(l1: &mut List<T>[@n1], l2: List<T>[@n2])
   ensures l1: List<T>[n1+n2]
{
  match l1 {
    List::Nil ⇒ *l1 = l2,
    List::Cons(_, t1) ⇒ append(&mut *t1, l2),
  }
}
```

`l2` is *consumed* when *spliced into* `l1`

Lists: *Specification*

```
fn never<T>() requires false → T
{
    loop {}
}
```

A function that can *never* be called at run-time…

Lists: *Verification*

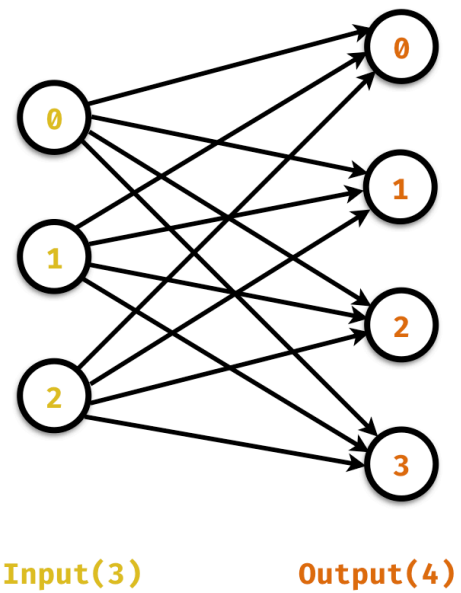
```
fn get_nth<T>(l: &List<T>, k: usize) → &T {
  match l {
    List::Cons(h, tl) if k == 0 ⇒ h,
    List::Cons(h, tl) ⇒ get_nth(tl, k - 1),
    List::Nil => never(),
  }
}
```

Exercise: *Fix* the specification for `get_nth`?

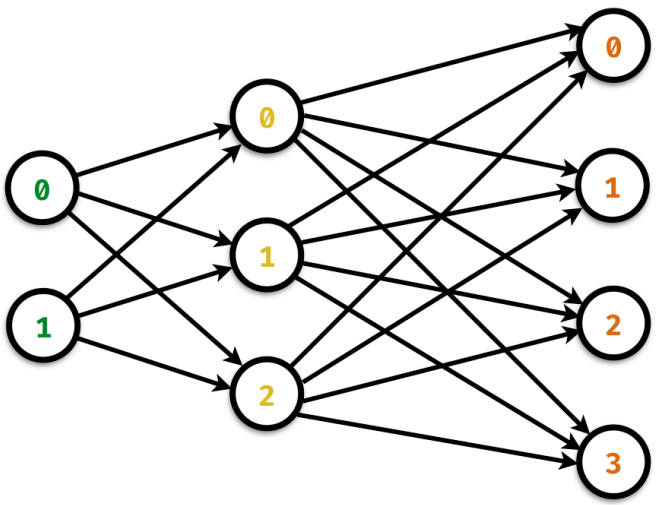
`enum`

Example: *Neural Network*

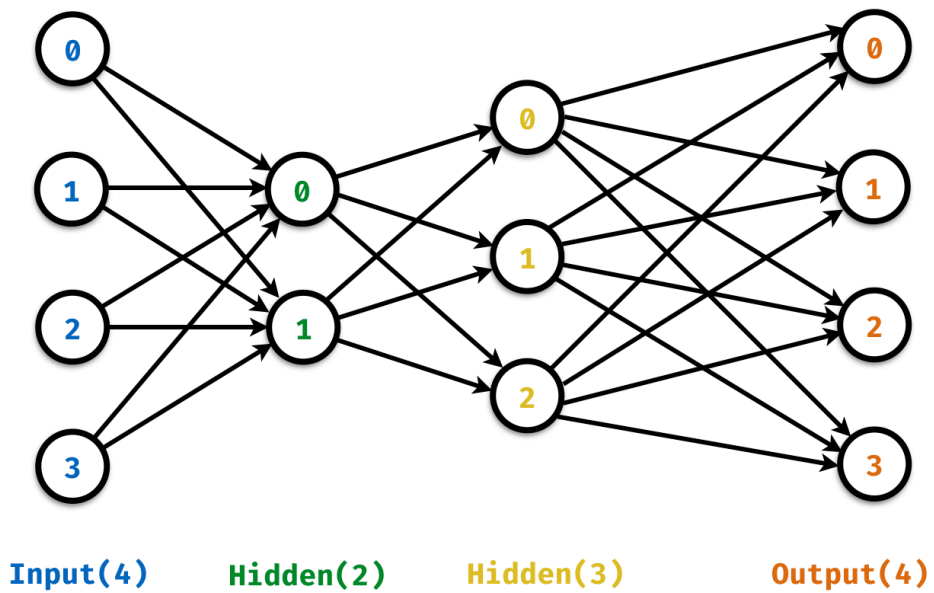
Neural Network has *Many* Layers



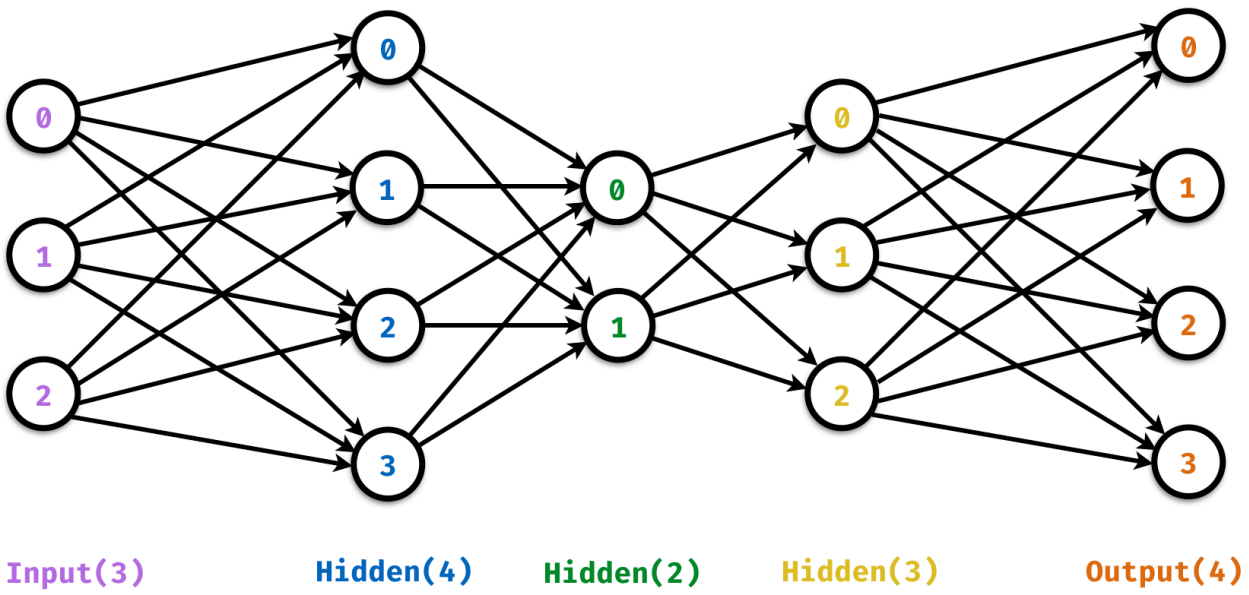
Neural Network has *Many* Layers



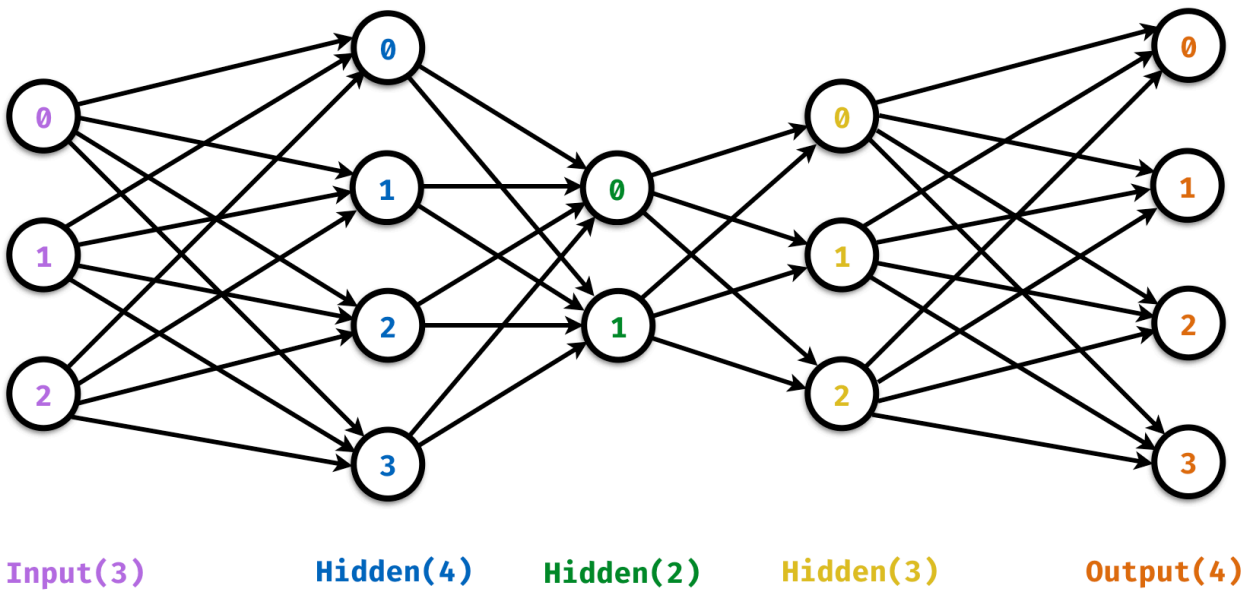
Neural Network has *Many* Layers



Neural Network has *Many* Layers



Neural Network has *Many* Layers



How to ensure layers *compose* correctly?

Neural Network: *Specification*

```
enum Network {
  Last(Layer),
  Next(Layer, Box<Network>),
}
```

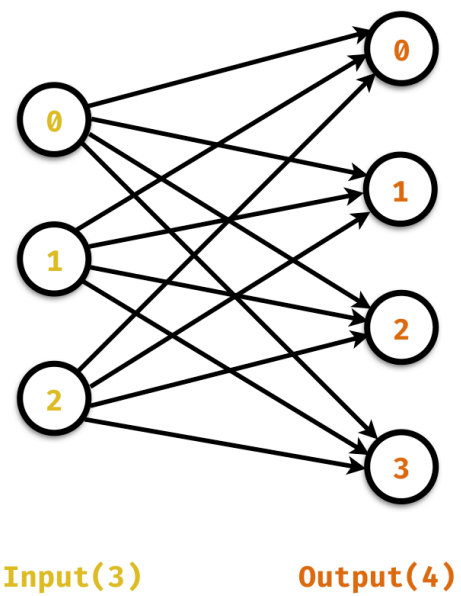
How to ensure layers *compose* correctly?

Lists: *Specification*

```
#[refined_by(i: int, o: int)]
enum Network {
  Last(Layer[@i, @o]) → Network[i, o],
  Next(Layer[@i, @n], Box<Network[n, @o]>) → Network[i, o],
}
```

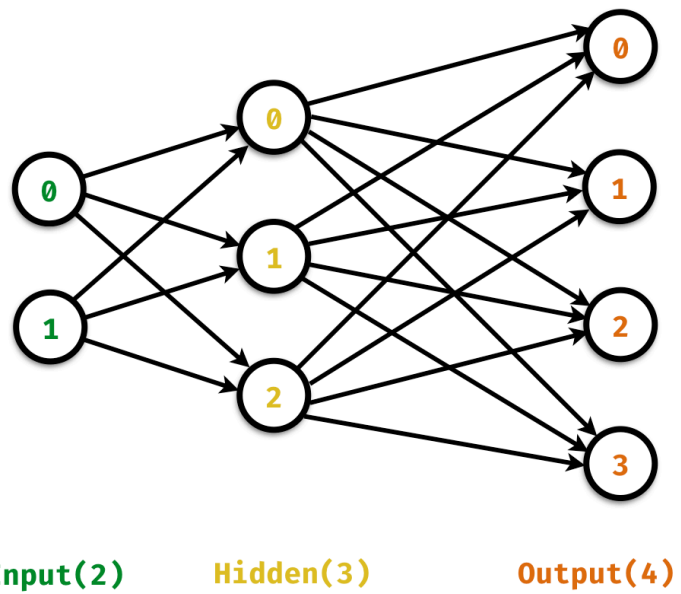
How to ensure layers *compose* correctly?

Refinements Ensure Correct *Composition*



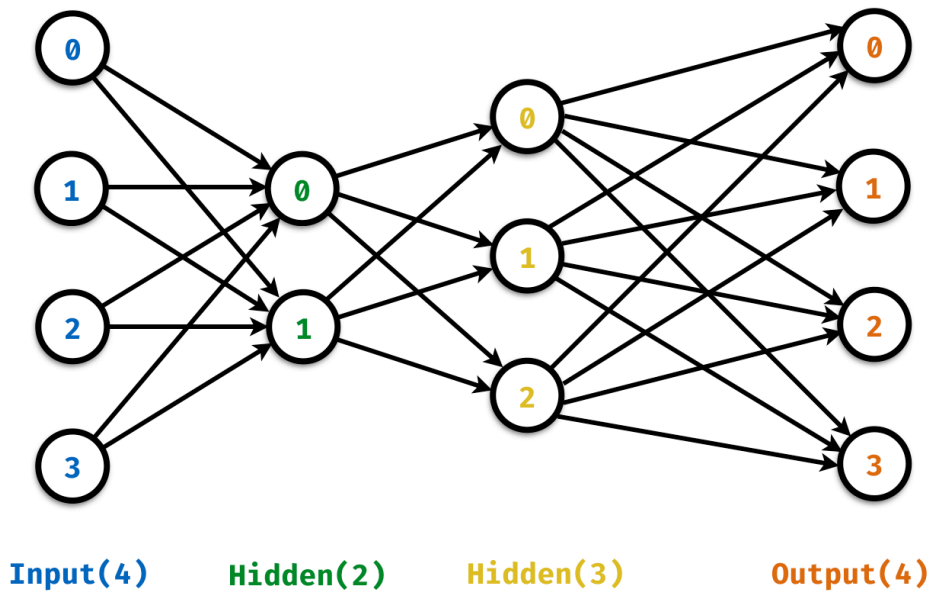

```
Last(Layer[3,4]) ⟶ Network[3, 4]
```

Refinements Ensure Correct *Composition*



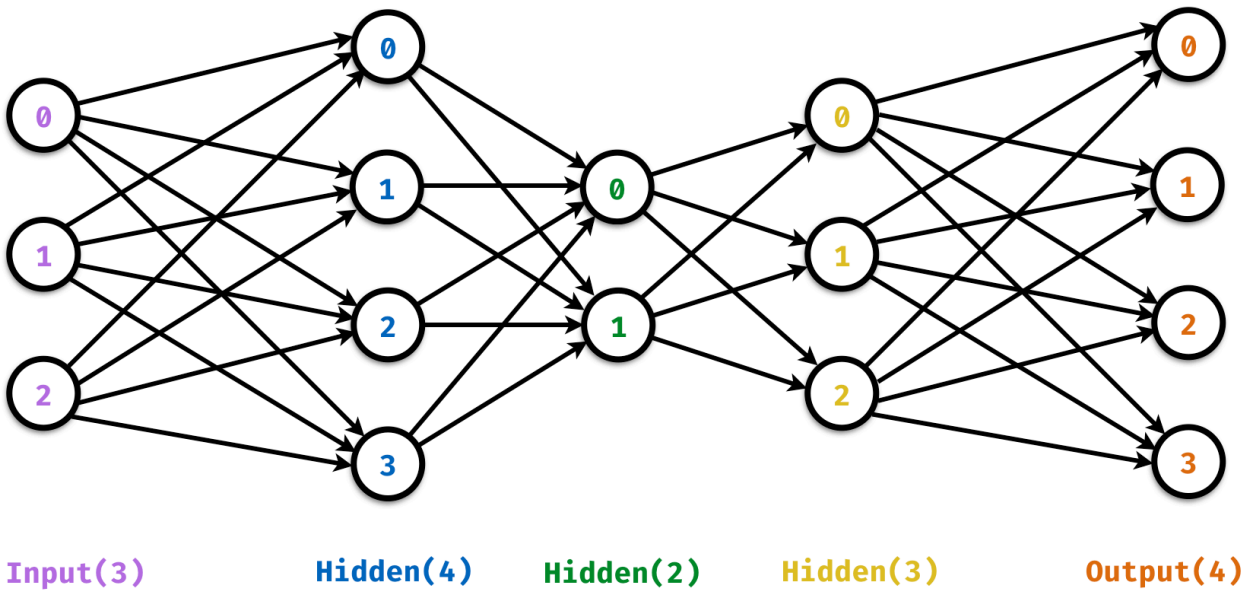

```
Next(Layer[2,3], Network[3,4]) ⟶ Network[2, 4]
```

Neural Network has *Many* Layers



`Next(Layer[4,2], Network[2,4])` ⟶ `Network[4, 4]`

Neural Network has *Many* Layers



`Next(Layer[3,4], Network[4,4])` ⟶ `Network[4, 4]`

Neural Network: *Verification*

```
fn new(input: usize, hidden: &[usize], output: usize)
   -> Network[input, output]
{
  if hidden_sizes.len() == 0 {
    Network::Last(Layer::new(input, output))
  } else {
    let n = hidden[0];
    let layer = Layer::new(input, n);
    let rest = Network::new(n, &hidden[1..], output);
    Network::Next(layer, Box::new(rest))
  }
}
```

Neural Network: *Verification*

```
fn forward(&mut Network, input: &RVec<f64>) → RVec<f64> {
  match self {
    NeuralNetwork::Last(layer) ⇒ {
      layer.forward(input);
      layer.outputs.clone()
    }
    NeuralNetwork::Next(layer, next) ⇒ {
      layer.forward(input);
      next.forward(&layer.outputs)
    }
  }
}
```

Exercise: *Fix* the specification for `forward`?

`enum`

Example: *Administrative Normal Form*

Administrative Normal Form

Sabry & Felleisen, 1992

Administrative Normal Form

Sabry & Felleisen, 1992

Expression

```
(1 + 2) * (4 - 3)
```

Administrative Normal Form

Sabry & Felleisen, 1992

Expression

```
(1 + 2) * (4 - 3)
```

ANF

```
let t1 = 1 + 2;
let t2 = 4 - 3;
t1 * t2
```

Administrative Normal Form

Sabry & Felleisen, 1992

Expression

```
(1 + 2) * (4 - 3)
```

ANF

```
let t1 = 1 + 2;
let t2 = 4 - 3;
t1 * t2
```

Calls/operations have ***immediate operands*** (i.e. vars or constants)

Administrative Normal Form: *Specification*

Calls/operations have ***immediate operands*** (i.e. vars or constants)

```
enum Exp {

  Var(String),

  Num(i32),

  Bin(Op, Box<Exp[@e1]>, Box<Exp[@e2]>),

  Let(Id, Box<Exp[@e1]>, Box<Exp[@e2]>),
}
```

Administrative Normal Form: *Specification*

Calls/operations have ***immediate operands*** (i.e. vars or constants)

```
#[refined_by(imm: bool)]
enum Exp {

  Var(String) → Exp[{imm: true}],

  Num(i32) → Exp[{imm: true}],

  Bin(Op, Box<Exp[@e1]>, Box<Exp[@e2]>) → Exp[{imm: false}],

  Let(Id, Box<Exp[@e1]>, Box<Exp[@e2]>) → Exp[{imm: false}])],
}
```

Administrative Normal Form: *Specification*

Calls/operations have immediate operands (i.e. vars or constants)

```
#[refined_by(imm: bool, anf: bool)]
enum Exp {

  Var(String) → Exp[{imm: true, anf: true}],

  Num(i32) → Exp[{imm: true, anf: true}],

  Bin(Op, Box<Exp[@e1]>, Box<Exp[@e2]>) → Exp[{imm: false, anf: e1.imm && e2.imm}],

  Let(Id, Box<Exp[@e1]>, Box<Exp[@e2]>) → Exp[{imm: false, anf: e1.anf && e2.anf}])],
}
```

Administrative Normal Form: *Verification*

```
fn is_imm(&Exp[@e]) → bool[e.imm] {
  match self {
    Exp::Var(_) ⇒ true,
    Exp::Num(_) ⇒ true,
    Exp::Bin(_, e1, e2) ⇒ false,
    Exp::Let(_, e1, e2) ⇒ false,
  }
}
```

Function to check if expression is ***immediate***

Administrative Normal Form: *Verification*

```
fn is_anf(&Exp[@e]) → bool[e.anf] {
  match self {
    Exp::Var(_) ⇒ true,
    Exp::Num(_) ⇒ true,
    Exp::Bin(_, e1, e2) ⇒ e1.is_imm() && e2.is_imm(),
    Exp::Let(_, e1, e2) ⇒ e1.is_anf() && e2.is_anf(),
  }
}
```

Function to check if expression is ***ANF***

Administrative Normal Form: *Conversion*

```
// Immediate subset of Exp
type Imm = Exp{e: e.imm};
```

Two helpful **Type Aliases**

Administrative Normal Form: *Conversion*

```
// Immediate subset of Exp
type Imm = Exp{e: e.imm};

// ANF subset of Exp
type Anf = Exp{e: e.anf};
```

Two helpful **Type Aliases**

Administrative Normal Form: *Conversion*

```
// Immediate subset of Exp
type Imm = Exp{e: e.imm};

// ANF subset of Exp
type Anf = Exp{e: e.anf};
```

Two helpful **Type Aliases**

Administrative Normal Form: *Conversion*

```
fn to_imm(&Exp, &mut usize, &mut RVec<(Id, Anf)>) → Imm
```

Convert `Exp` into Temp-`Anf` bindings + `Imm`

Administrative Normal Form: *Conversion*

```
fn to_imm(&Exp, &mut usize, &mut RVec<(Id, Anf)>) → Imm
```

Convert `Exp` into Temp-`Anf` bindings + `Imm`

`(1 + 2) * 4` $\Longrightarrow$

Administrative Normal Form: *Conversion*

```
fn to_imm(&Exp, &mut usize, &mut RVec<(Id, Anf)>) → Imm
```

Convert `Exp` into Temp-`Anf` bindings + `Imm`

`(1 + 2) * 4` ⟹ `[(t1, 1+2), (t2, t1*4)]` `t2`

Administrative Normal Form: *Conversion*

```
fn to_anf(&Exp, &mut usize) → Anf
```

Administrative Normal Form: *Conversion*

```
fn to_anf(&Exp, &mut usize) → Anf
```

Exercise: *Implement* `to_anf` so the following checks

```
fn prop_anf(e: &Exp) {
  assert(e.to_anf(&mut 0).is_anf())
}
```

3. Datatypes

Compositional specification & verification

`struct` `enum`

3. Datatypes

Compositional specification & verification

`struct` `enum`

"Make illegal states unrepresentable"

Refinements for Rust

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

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

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

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

4. Interfaces

Idiomatic Rust

Code like a Rustacean

Brenden Matthews

MANNING

Idiomatic Rust … *Not*

```
fn dot(xs: &RVec<f64>[@n], ys: &RVec<f64>[n]) → f64 {
  let mut res = 0.0;
  let mut i = 0;
  while (i < xs.len()) {
    res += xs[i] * ys[i];
    i += 1;
  }
  res
}
```

Idiomatic Rust … *Not*

Step 1: Use `std::vec::Vec`

```
fn dot(xs: &Vec<f64>[@n], ys: &Vec<f64>[n]) → f64 {
  let mut res = 0.0;
  let mut i = 0;
  while (i < xs.len()) {
    res += xs[i] * ys[i];
    i += 1;
  }
  res
}
```

Idiomatic Rust … *Not*

Step 2: Use `for` loop

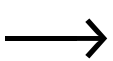
```
let mut res = 0.0;
let mut i = 0;
while i < xs.len() {
  res += xs[i] * ys[i];
  i += 1;
}
res
```

```
let mut res = 0.0;
for i in 0..xs.len() {
  res += xs[i] * ys[i];
}
res
```

Idiomatic Rust … *Not*

Step 3: Use `for_each`

```
let mut res = 0.0;
for i in 0..xs.len() {
  res += xs[i] * ys[i];
}
res
```

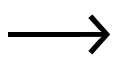


```
let mut res = 0.0;
(0..xs.len())
  .for_each(|i|
     res += xs[i] * ys[i]
   );
res
```

Idiomatic Rust … *Not*

Step 4: Use `map` and `sum`

```
let mut res = 0.0;
(0..xs.len())
 .for_each(|i|
   res += xs[i] * ys[i]
  );
res
```



```
(0..xs.len())
 .map(|i| xs[i] * ys[i])
 .sum()
```

Idiomatic Rust!

```
fn dot(xs: &Vec<f64>[@n], ys: &Vec<f64>[n]) → f64 {
  (0..xs.len())
    .map(|i| xs[i] * ys[i])
    .sum()
}
```

Idiomatic Rust!

```
fn dot(xs: &Vec<f64>[@n], ys: &Vec<f64>[n]) → f64 {
  (0..xs.len())
    .map(|i| xs[i] * ys[i])
    .sum()
}
```

A *Specification* Challenge…

A *Specification* Challenge…

Source

```
let mut res = 0.0;
for i in 0..xs.len() {
  res += xs[i] * ys[i];
}
res
```

A *Specification* Challenge…

Source

```
let mut res = 0.0;
for i in 0..xs.len() {
  res += xs[i] * ys[i];
}
res
```

⟹

Desugared using *Traits*

```
let mut res = 0.0;
let mut rng = 0..xs.len();
loop {
 match rng.next() {
  Some(i) => res += xs.index(i) * ys.index(i)
  None ⇒ break
 }
}
res
```

A *Specification* Challenge...

Desugared using *Traits*

A *Specification* Challenge...

Desugared using *Traits*

```
trait Iterator {
 type Item;
 fn next(&mut self) -> Option<Item>;
}
```

A *Specification* Challenge...

Desugared using *Traits*

```
trait Iterator {
 type Item;
 fn next(&mut self) -> Option<Item>;
}
```

```
trait Index<Idx> {
 type Output;
 fn index(&self, i:Idx) -> &Output;
}
```

A *Specification* Challenge...

Desugared using *Traits*

```
trait Iterator {
 type Item;
 fn next(&mut self) -> Option<Item>;
}
```

```
trait Index<Idx> {
 type Output;
 fn index(&self, i:Idx) -> &Output;
}
```

Generic Interfaces implemented by many Types

Generic Interfaces

```
trait Index<Idx> {
 type Output;
 fn index(&self, i:Idx) -> &Output;
}
```

Implemented by Many Types

Generic Interfaces

```
trait Index<Idx> {
 type Output;
 fn index(&self, i:Idx) -> &Output;
}
```

Implemented by Many Types

```
impl Index<Range<usize>> for Vec<T> { ... }
```

`Self` ≐ `Vec<T>` `Idx` ≐ `usize` `Output` ≐ `&T`

Generic Interfaces

```
trait Index<Idx> {
 type Output;
 fn index(&self, i:Idx) -> &Output;
}
```

Implemented by Many Types

```
impl Index<Range<usize>> for Vec<T> { ... }
```

`Self` ≐ `Vec<T>` `Idx` ≐ `usize` `Output` ≐ `&T`

`char_vec[0]` ⇒ `'a'`

Generic Interfaces

```
trait Index<Idx> {
 type Output;
 fn index(&self, i:Idx) -> &Output;
}
```

Implemented by Many Types

```
impl Index<Range<usize>> for Vec<T> { ... }
```

`Self` ≐ `Vec<T>` `Idx` ≐ `Range<usize>` `Output` ≐ `&[T]`

Generic Interfaces

```
trait Index<Idx> {
 type Output;
 fn index(&self, i:Idx) -> &Output;
}
```

Implemented by Many Types

```
impl Index<Range<usize>> for Vec<T> { ... }
```

`Self` ≐ `Vec<T>` `Idx` ≐ `Range<usize>` `Output` ≐ `&[T]`

`char_vec[0..2]` ⇒ `&['a','b','c']`

Generic Interfaces

```
trait Index<Idx> {
 type Output;
 fn index(&self, i:Idx) -> &Output;
}
```

Implemented by Many Types

```
impl Index<&K> for Map<K, V> { ... }
```

`Self` ≐ `Map<K, V>` `Idx` ≐ `&K` `Output` ≐ `&V`

Generic Interfaces

```
trait Index<Idx> {
 type Output;
 fn index(&self, i:Idx) -> &Output;
}
```

Implemented by Many Types

```
impl Index<&K> for Map<K, V> { ... }
```

`Self` ≐ `Map<K, V>` `Idx` ≐ `&K` `Output` ≐ `&V`

```
age_map[&"ranjit"] ⇒ 48
```

A *Specification* Challenge...

```
trait Index<Idx> {
 type Output;
 fn index(&self, i:Idx) -> &Output;
}
```

Problem: How to specify *"index bounds"*?

Problem: How to specify *"index bounds"*?

```
fn head(v:&Vec<i32>) → i32
{
  *v.index(0)
}
```

Problem: How to specify *"index bounds"*?

```
fn head(v:&Vec<i32>) → i32
{
  *v.index(0)
}
```

Bounds Check

require `0 < v.len`

Problem: How to specify *"index bounds"*?

```
fn head(v:&Vec<i32>) → i32
{
  *v.index(0)
}
```

```
trait Index<Idx> {
  type Output;
  fn index(&self, i:Idx) → &Output;
}
```

Bounds Check

require `0 < v.len`

Trait

Problem: How to specify *"index bounds"*?

```
fn head(v:&Vec<i32>) → i32
{
  *v.index(0)
}
```

```
trait Index<Idx> {
  type Output;
  fn index(&self, i:Idx) → &Output;
}
```

Bounds Check

require `0 < v.len`

Trait is too *generic*!

`Self` not `Vec`! `Idx` not *number*!

Problem: How to specify *"index bounds"*?

```
fn head(v:&Vec<i32>) → i32
{
  *v.index(0)
}
```

```
impl<T> Index<usize> for Vec<T> {
  type Output = &T;
  fn index(&self, i:usize) → &T {..}
}
```

Bounds Check

require `0 < v.len`

Impl

Problem: How to specify *"index bounds"*?

```
fn head(v:&Vec<i32>) → i32
{
  *v.index(0)
}
```

```
impl<T> Index<usize> for Vec<T> {
  type Output = &T;
  fn index(&self, i:usize) → &T {..}
}
```

Bounds Check

require `0 < v.len`

Impl is too *specific*!

Unsoundly *misses* check!

Problem: How to specify *"index bounds"*?

```
fn head(v:&Vec<i32>) → i32
{
  *v.hack(0) // ok: no req on hack!
}

fn hack(c:C, i:usize) → &C::Output
where C: Index<usize>
{
  c.index(i) // ok: no req on trait!
}
```

```
trait Index<Idx> {
  type Output;
  fn index(&Self, i:Idx) → &T;
}
```

Impl is too *specific*!

Unsoundly *misses* check!

Generic Interfaces

Problem

How to specify *"index"* or *"iterator"* or …?

“All problems in computer science can be solved by another level of indirection“

— David Wheeler

Generic Interfaces

Problem

How to specify *"index"* or *"iterator"* or …?

Solution

Associated Refinements

Associated Refinements

Split specification across `trait` and `impl`

Associated Refinements

Split specification across `trait` and `impl`

Trait

```
trait Index<Idx> {
```

Impl

Associated Refinements

Split specification across `trait` and `impl`

Trait

```
trait Index<Idx> {
  type Out;
```

Impl

Associated Refinements

Split specification across `trait` and `impl`

Trait

```
trait Index<Idx> {
  type Out;
  reft in_bounds(&self, idx: Idx) → bool;
```

Impl

Associated Refinements

Split specification across `trait` and `impl`

Trait

```
trait Index<Idx> {
  type Out;
  reft in_bounds(&self, idx: Idx) → bool;
  fn index(&self, i:Idx{Self::in_bounds(self, i)}) -> &Out;
}
```

Impl

Associated Refinements

Split specification across `trait` and `impl`

Trait

```
trait Index<Idx> {
  type Out;
  reft in_bounds(&self, idx: Idx) → bool;
  fn index(&self, i:Idx{Self::in_bounds(self, i)}) → &Out;
}
```

Impl

```
impl<T> Index<usize> for Vec<T> {
  type Out = &T;
```

Associated Refinements

Split specification across `trait` and `impl`

Trait

```
trait Index<Idx> {
  type Out;
  reft in_bounds(&self, idx: Idx) → bool;
  fn index(&self, i:Idx{Self::in_bounds(self, i)}) → &Out;
}
```

Impl

```
impl<T> Index<usize> for Vec<T> {
  type Out = &T;
  reft in_bounds(&self, idx: Idx) -> bool { idx < self.len };
```

Associated Refinements

Split specification across `trait` and `impl`

Trait

```
trait Index<Idx> {
  type Out;
  reft in_bounds(&self, idx: Idx) → bool;
  fn index(&self, i:Idx{Self::in_bounds(self, i)}) → &Out;
}
```

Impl

```
impl<T> Index<usize> for Vec<T> {
  type Out = &T;
  reft in_bounds(&self, idx: Idx) → bool { idx < self.len };
  fn index(&self, i:Idx{Self::in_bounds(self, i)}) → &Out;
}
```

Associated Refinements

Split specification across `trait` and `impl`

Trait

```
trait Index<Idx> {
  type Output;
  reft in_bounds(&self, idx: Idx) → bool;
  fn index(&self, i:Idx{Self::in_bounds(self, i)}) → &T;
}
```

Impl

```
trait Index<Idx> {
  type Output;
  reft in_bounds(&self, idx: Idx) → bool { idx < self.len };
  fn index(&self, i:Idx{Self::in_bounds(self, i)}) → &T;
}
```

Associated Refinements

Can now *verify* *"index bounds"*

```
fn head(v:&Vec<i32>)
    → i32
{
   *v.index(0)
}
```

Associated Refinements

Can now *verify* *"index bounds"*

```
fn index(&self, i:Idx{Self::in_bounds(self, i)}) → &Out
```

```
fn head(v:&Vec<i32>)
    → i32
{
   *v.index(0)
}
```

Associated Refinements

Can now *verify* *"index bounds"*

```
fn index(&self, i:Idx{Self::in_bounds(self, i)}) → &Out
```

⇩ **Instantiation:** `Self` ≐ `Vec<i32>`, `Idx` ≐ `usize`, `Out` ≐ `i32`

```
fn index(&Vec<i32>[v], i:usize{Self::in_bounds(v,i)}) → &i32
```

```
fn head(v:&Vec<i32>)
    → i32
{
   *v.index(0)
}
```

Associated Refinements

Can now *verify* *"index bounds"*

```
fn index(&self, i:Idx{Self::in_bounds(self, i)}) → &Out
```

⇩ **Instantiation:** `Self ≐ Vec<i32>`, `Idx ≐ usize`, `Out ≐ i32`

```
fn index(&Vec<i32>[v], i:usize{Self::in_bounds(v,i)}) → &i32
```

⇩ **Projection:** `Self::in_bounds(self, i) ≐ i < self.len`

```
fn index(&Vec<i32>[v], i:usize{i < v.len}) → &i32
```

```
fn head(v:&Vec<i32>)
    → i32
{
   *v.index(0)
}
```

Associated Refinements

Can now *verify* *"index bounds"*

```
fn head(v:&Vec<i32>) → i32 {
  *v.index(0)
}
```

At call-site

```
fn index(&Vec<i32>[v], i:usize{i < v.len}) → &i32
```

Associated Refinements

Can now *verify* *"index bounds"*

```
fn head(v:&Vec<i32>) → i32 {
  *v.index(0)
}
```

At call-site

```
fn index(&Vec<i32>[v], i:usize{i < v.len}) → &i32
```

Exercise: Can you *fix* the specification for `head`?

Associated Refinements

Prevent *bypassing* checks e.g. `hack`

```
fn hack(c:C, i:usize) → &C::Output
where C: Index<usize>
{
  c.index(i)
}
```

Associated Refinements

Prevent *bypassing* checks e.g. `hack`

```
fn hack(c:C, i:usize) → &C::Output
where C: Index<usize>
{
  c.index(i)
}
```

At call-site

```
fn index(&C[c], i:usize{C::in_bounds(i, c)}) → &i32
```

Associated Refinements

Prevent *bypassing* checks e.g. `hack`

```
fn hack(c:C, i:usize) → &C::Output
where C: Index<usize>
{
  c.index(i)
}
```

At call-site

```
fn index(&C[c], i:usize{C::in_bounds(i, c)}) → &i32
```

Exercise: How to *fix* the specification for `hack`?

Beyond Indexing: *Iterators*

Beyond Indexing: *Iterators*

```
trait Iterator {
  type Item;
  fn next(&mut self) → Option<Item>;
}
```

Beyond Indexing: *Iterators*

```
trait Iterator {
  type Item;
  fn next(&mut self) → Option<Item>;
}
```

Used to desugar `for i in 0..n`

Beyond Indexing: *Iterators*

```
trait Iterator {
  type Item;
```

Iterators are *state machines*

Denis & Jourdan, 2023

Beyond Indexing: *Iterators*

```
trait Iterator {
  type Item;
  reft step(Self, Self) → bool; // relation between states
```

Iterators are *state machines*

Denis & Jourdan, 2023

Beyond Indexing: *Iterators*

```
trait Iterator {
  type Item;
  reft step(Self, Self) → bool; // relation between states
  reft done(Self) → bool;       // finished?
```

Iterators are *state machines*

Denis & Jourdan, 2023

Beyond Indexing: *Iterators*

```
trait Iterator {
  type Item;
  reft step(Self, Self) → bool; // relation between states
  reft done(Self) → bool;       // finished?
  fn next(&mut self[s]) → Option<Item>[!done(s)]
     ensures self: Self{t: next(s, t)};
}
```

Iterators are *state machines*

Denis & Jourdan, 2023

Beyond Indexing: *Iterators*

```
trait Iterator {
  type Item;
  reft step(Self, Self) → bool; // relation between states
  reft done(Self) → bool;       // finished?
  fn next(&mut self[s]) → Option<Item>[!done(s)]
     ensures self: Self{t: next(s, t)};
}
```

Iterators are *state machines*

Denis & Jourdan, 2023

Beyond Indexing: *Iterators*

```
trait Iterator {
  type Item;
  reft step(Self, Self) → bool; // relation between states
  reft done(Self) → bool;       // finished?
  fn next(&mut self[s]) → Option<Item>[!done(s)]
     ensures self: Self{t: next(s, t)};
}
```

Scales nicely to `Iter`, `Zip`, `Map`, `Enumerate` …

Denis & Jourdan, 2023

Problem: Specifications for *Generic Interfaces*

Implemented by many types

```
trait Iterator {
 type Item;
 fn next(&mut self) -> Option<Item>;
}
```

```
trait Index<Idx> {
 type Output;
 fn index(&self, i:Idx) -> &Output;
}
```

Problem: Specifications for *Generic Interfaces*

Implemented by many types

```
trait Iterator {
 type Item;
 fn next(&mut self) -> Option<Item>;
}
```

```
trait Index<Idx> {
 type Output;
 fn index(&self, i:Idx) -> &Output;
}
```

Solution: *Associated Refinements*

Split specification across `trait` and `impl`

Refinements for Rust

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

Flux in Practice

Flux in Practice

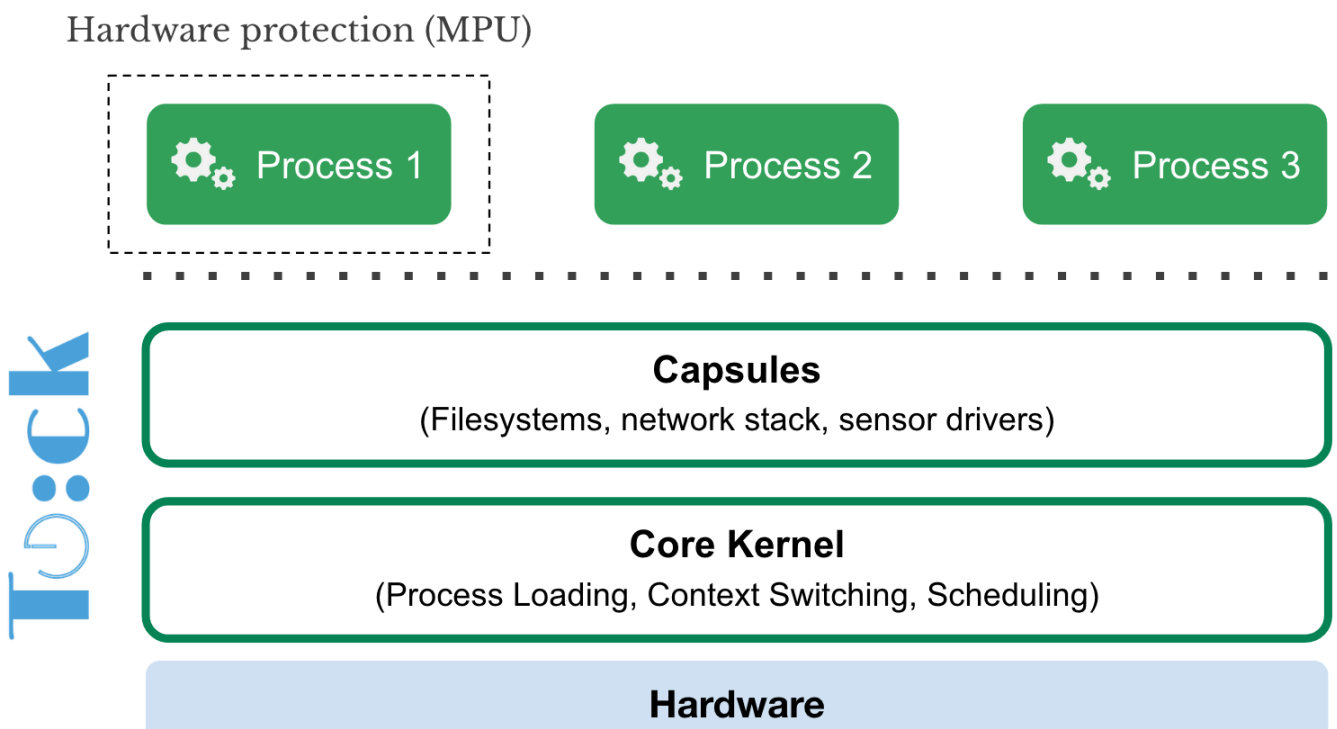
Verified *Process Isolation* in Tock

Verified *Process Isolation* in Tock

An *Embedded* OS Kernel

Used as firmware Google ChromeBooks, Microsoft Pluton Security,…

Verified *Process Isolation* in Tock



Verified *Process Isolation* in Tock

Cortex-M MPU: `allocate_app_memory_region` allows access to kernel grant memory #4366

Closed #4384

ARMv6m: Register clobbering in Generic ISR causes exception exit to return to process rather than Kernel #4245

Closed #4259

ARMv6m: Thread mode not set to privileged execution in certain ISRs #4246

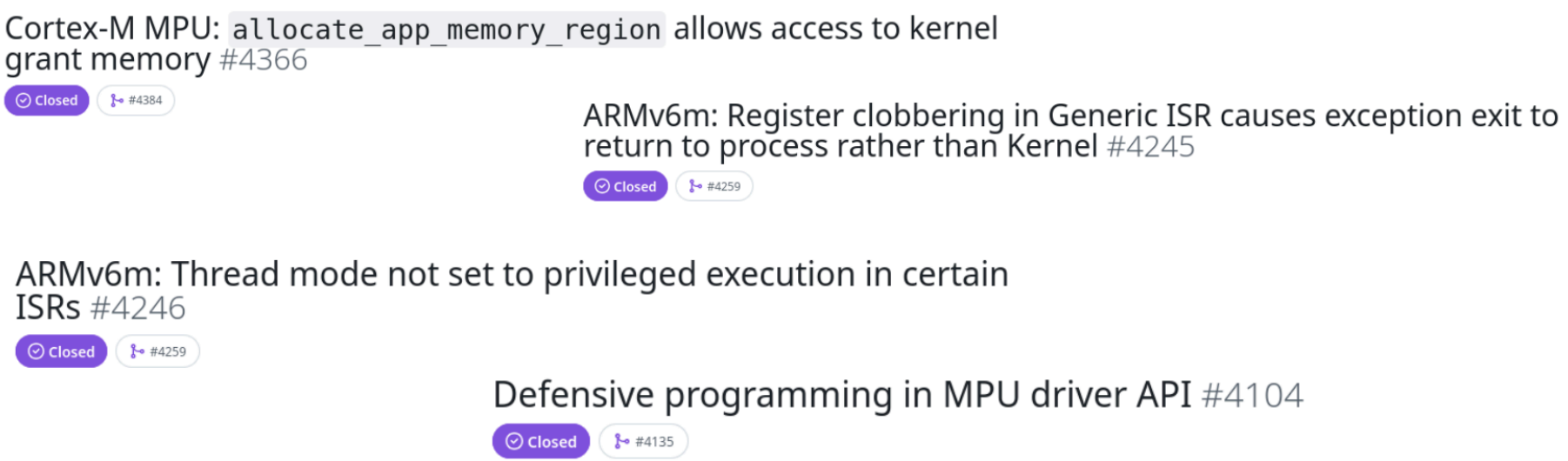
Closed #4259

Defensive programming in MPU driver API #4104

Closed #4135

Flux helped find five ***security vulnerabilities*** …

Verified *Process Isolation* in Tock



… and a clearer design yielding a ***faster*** OS kernel!

Verified *Process Isolation* in Tock

Component	Fns.	Total	Max	Mean	StdDev.
✓Tock (Monolithic)	660	5m19s	4m57s	0.48s	11.36s
✓Tock (Granular)	689	1m47s	24s	0.15s	1.44s

… and a clearer design yielding ***faster*** verification!

Liquid Types for Rust

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

https://flux-rs.github.io