Making aliasing safe again

An exploratory comparison of linear types and deny-capabilities.

Christopher J. Hall

Supervised by
Dr. Benjamin Lippmeier
Dr. Gabriele Keller

Programming Languages and Systems Group
School of Computer Science and Engineering
UNSW

Google Australia

SAPLING 2017-10-20

Pre(r)amble

The usual disclaimer: Any views, thoughts, and opinions expressed belong solely to the author, and do not necessarily reflect those of my employer or any organization I belong to.

I would like to thank the Australian government for supporting this work through their Research Training Program (RTP) scholarship.

My background

I am a (Slowly) recovering C programmer

I learned just enough Haskell and Prolog to be dangerous

(but not enough to be very useful).

My interest areas includes:

Mutation and Aliasing,
Programming language design and implementation,
Static verification,
Proof theory,
Runtime systems, Garbage collectors, Hash tables.

Motivation

Motivation:

I want to be able to write statically safe efficient data structures
e.g. Robin-hood Linear-probing hash tables, B+ trees

This often means I need mutation and aliasing.

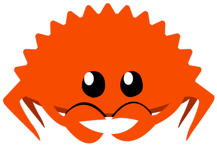
Relying on a compiler to convert pure code into code using mutation is often sub-optimal.

Rust

"Rust is a systems programming language that runs blazingly fast, prevents segfaults, and guarantees thread safety."

Promises:

- zero-cost abstractions
- guaranteed memory safety
- threads without data races
- minimal runtime



RAII

Resource Acquisition Is Initialization (‘RAII’) is a C++ design pattern whereby resources are bound to object lifetimes.

RAII

Resource Acquisition Is Initialization (‘RAII’) is a C++ design pattern whereby resources are bound to object lifetimes.

The simplest case of this is

- object allocating some resources within constructor
- possibly changing these resources throughout object’s lifetime
- finally releasing all those resources in object’s destructor

C++ RAII unique_ptr

```
struct MyResources {
  void *data;
  MyResources() { // Constructor
    puts("Allocate my resources");
    data = malloc(sizeof(int) * 100);
  }
  ~MyResources() { // Destructor
    puts("Free my resources");
    free(data);
  }
};

int main(void) {
  MyResources r;
  puts("Do some work");
}
```

```
Allocate my resources
Do some work
Free my resources
```

Rust RAII

```
struct MyResources {
  void *data;
  MyResources() { // Constructor
    puts("Allocate my resources");
    data = malloc(sizeof(int) * 100);
  }
  ~MyResources() { // Destructor
    puts("Free my resources");
    free(data);
  }
};

int main(void) {
  MyResources r;
  puts("Do some work");
}
```

```
Allocate my resources
Do some work
Free my resources
```

RAII example: smart pointer

```
struct MyResources {
  std::unique_ptr<int[]> data;
  MyResources() { // Constructor
    puts("Allocate my resources");
    data = std::unique_ptr<int[]>(new int [100]);
  }
};

int main(void) {
  MyResources r;
  puts("Do some work");
}
```

```
Allocate my resources
Do some work
```

RAII example: smart pointer

```
struct MyResources {
  std::unique_ptr<int[]> data;
  MyResources() { // Constructor
    puts("Allocate my resources");
    data = std::unique_ptr<int[]>( new int [100] );
  }
};

int main(void) {
  MyResources r;
  puts("Do some work");
}
```

```
Allocate my resources
Do some work
```

RAII example: rust

```rust
pub struct MyResources {
    pub data: Vec<i8>,
}

impl MyResources {
    pub fn new() -> MyResources {
        println!("Allocate my resources");
        MyResources {data: vec!(0, 100)}
    }
}

fn main() {
    let a = MyResources::new();
    println!("Do some work");
}
```

```
Allocate my resources
Do some work
```

RAII example: rust

```rust
pub struct MyResources {
    pub data: Vec<i8>,
}

impl MyResources {
    pub fn new() -> MyResources {
        println!("Allocate my resources");
        MyResources {data: vec!(0, 100) }
    }
}

fn main() {
    let a = MyResources::new();
    println!("Do some work");
}
```

```
Allocate my resources
Do some work
```

Limitations of linearity

```
fn main() {
  let data = RefCell::new(0);
  {
    let mut r1 = data.borrow_mut();
    *r1 += 1;
  }
  {
    let mut r2 = data.borrow_mut();
    *r2 += 1;
  }
  println!("{}", data.borrow());
}
```

```
2
```

Limitations of linearity

```
fn main() {
  let data = RefCell::new(0);
  let mut r1 = data.borrow\_mut();
  let mut r2 = data.borrow_mut();
  println!("{}", data.borrow());
}
```

```
thread 'main' panicked at 'already borrowed: BorrowMutError' ...
```

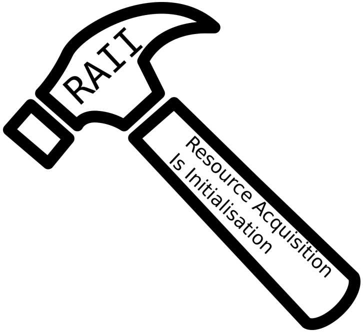
Doubly linked lists are not linear

```
pub struct MyDoublyLinkedList<T> {
  head: Option<Shared<Node<T>>>,
  tail: Option<Shared<Node<T>>>,
  len: usize,
  /* magic incantation to avoid error ... */
  marker: PhantomData<Box<Node<T>>>,
}

struct Node<T> {
  next: Option<Shared<Node<T>>>,
  prev: Option<Shared<Node<T>>>,
  element: T,
}
```

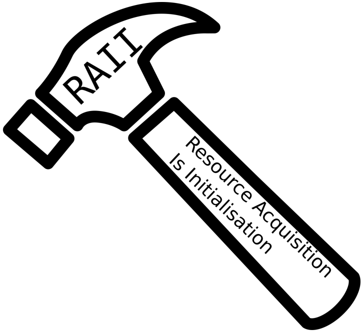
Doubly linked lists are not linear

```
fn push_front_node(&mut self, mut node: Box<Node<T>>) {
  node.next = self.head;
  node.prev = None;
  let node = Some(Shared::from(Box::into_unique(node)));

  unsafe {
    match self.head {
      None => self.tail = node,
      Some(mut head) => head.as_mut().prev = node,
    }
  }

  self.head = node;
  self.len += 1;
}
```

Rust toolbox



Rust toolbox

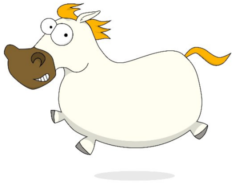


Pony

“Pony is an open-source, object-oriented, actor-model, capabilities-secure, high-performance programming language.”

Promises:

- type safe
- memory safe
- data race free
- deadlock free
- compiles to native code
- garbage collected



Deny capabilities

Capabilities:

A capability is an unforgeable token that (a) designates an object and (b) gives the program the authority to perform a specific set of actions on that object.

Deny capabilities

Capabilities:

A capability is an unforgeable token that (a) designates an object and (b) gives the program the authority to perform a specific set of actions on that object.

Deny capabilities:

Rather than indicate which operations are allowed on a reference, deny capabilities indicate what operations are denied on other references to the same object (aliases).

Pony principles

Pony distinguishes between what is denied to the actor that holds a reference (local aliases) from what is denied to all other actors (global aliases).

Some Pony principles:

- Every actor is single threaded
- Shared mutable data is hard
- Immutable data can be safely shared

(Some) Pony capabilities

Pony has 6 reference capabilities.
Some of the more interesting capabilities:

(Some) Pony capabilities

Pony has 6 reference capabilities.
Some of the more interesting capabilities:

- **Isolated** - the **only reference** (globally or locally) to this data.

(Some) Pony capabilities

Pony has 6 reference capabilities.
Some of the more interesting capabilities:

- **Isolated** - the **only reference** (globally or locally) to this data.
- **Value** - an **immutable** data structure.

(Some) Pony capabilities

Pony has 6 reference capabilities.
Some of the more interesting capabilities:

- **Isolated** - the **only reference** (globally or locally) to this data.
- **Value** - an **immutable** data structure.
- **Reference - mutable** non-isolated **thread-local** data.

(Some) Pony capabilities

Pony has 6 reference capabilities.
Some of the more interesting capabilities:

- **Isolated** - the **only reference** (globally or locally) to this data.
- **Value** - an **immutable** data structure.
- **Reference - mutable** non-isolated **thread-local** data.
- **Box - read-only** non-isolated **thread-local** data.

Pony capabilities example

```
class Cell
    var data: U64 val
    new create(d: U64 val) =>
        data = d

actor Main
  new create(env: Env) =>
    let a: Cell ref = Cell.create(U64(0))
    let b: Cell ref = a
    a.data = a.data + 1
    b.data = b.data + 1
    env.out.print(a.data.string())
    env.out.print(b.data.string())
```

```
2
2
```

Pony recover - easy isolated graph structures

```
class MyNode
    var prev: (MyNode | None) = None
    var next: (MyNode | None) = None
    fun ref set(pre: MyNode, nex: MyNode) =>
        prev = pre
        next = nex

actor Main
  new create(env: Env) =>
    let s: MyNode iso = recover
      let a: MyNode ref = MyNode.create()
      let b: MyNode ref = MyNode.create()
      let c: MyNode ref = MyNode.create()

      a.set(c, b)
      b.set(a, c)
      c.set(b, a)

      a
    end
```

Pony doubly linked list

```
class MyListNode[A]
  var item: (A | None)
  var prev: (MyListNode[A] | None) = None
  var next: (MyListNode[A] | None) = None

  new create(i: (A | None) = None) =>
    item = consume i
```

NB: MyListNode is polymorphic over both type and capability.

Pony doubly linked list

```
fun ref _push_front_node(node: MyListNode[A]) =>
  node.prev = None
  node.next = head

  match head
  | let head’: MyListNode[A] =>
    head’.prev = node
    head = node
    if tail is None then
      tail = node
      end
    else
      head = node
      tail = node
    end
```

Conclusion

Both approaches have a lot to offer, the suitability of each likely depends on **usecase**.

Rust's linear types and borrow system allow us to perform safe mutable updates in **limited** ways, but these restrictions allow us to (for the most part) **avoid a dependence on a runtime system**.

Pony's deny-capabilities allow us to combine **arbitrary thread-local mutation and aliasing** with restricted cross-thread aliasing, this combination allows us to more easily safely express **graph data structures**, but this freedom relies on leaving memory management to a **runtime garbage collector**.

Conclusion as a table

	Type/Memory safe	Datarace free	GC	Runtime locks
Rust(safe)	yes	yes	RC opt-*in*^{1}	*sometimes*^{2}
Pony	yes	yes	mark *sweep*^{3}	no

Caveats:

- 1 dropchk edge case
- 2 some types perform run-time checking of mut/immut rules
- 3 thread-local mark-sweep, cross-thread delayed reference counting

The end

Thank you for your time

Any questions?