Specificaton and Verification of Heap Access Policies

Immutability, Non-Nullness and Lexically Scoped Regions for Object Initialization

Christian Haack

Radboud University, Nijmegen

ESF Winter School on Verification of Object-Oriented Programs
Viinistu, Estonia, Jan 25-29, 2009

What is This Course About?

Today: Pluggable Type Systems

- a type system for object immutability (Haack/Poll 2009)
- a type system for non-nullness (Fähndrich/Xia 2007)
- lexically scoped regions for object initialization

Wednesday: Typestate Protocols

- basis: a fragment of separation logic

Today

First half: a type system for immutability

Based on [HP09].

Second half: lexically scoped regions for object initialization

- a type system for region-based memory management
- the immutability type system
- Fähndrich and Xia's non-nullness type system

Based on my notes [Haa09] (linked from the school web page), which are in turn based on [GMJ+02], [HP09] and [FX07].

Outline

1. A Type System for Immutability

2. Lexically Scoped Regions for Initialization of Object Types
   - Safe Deallocation with Lexically Scoped Regions
   - Lexically Scoped Regions for Immutability
   - A Type System for Non-Nullness

3. References

Class Immutability

Immutable Classes

An immutable class is a class whose instances cannot be modified.

Examples

- Java's `String` class
- Java's wrappers for primitive types, e.g., `Integer`, `Boolean`

Object Immutability

Immutable Objects

An object is immutable if its state cannot be modified.

- Immutable objects need not be instances of immutable classes.
- Their classes may provide mutator methods and non-final public fields, but immutable objects don't use these.
- They are not necessarily initialized inside object constructors.

Examples

- immutable arrays
- immutable collections implemented as instances of mutable collection classes from Java's collection library

Immutable Objects — Why are They Useful?

- Thread safety: immutable objects can be thread-shared without synchronization.
  (object immutability is enough)
- Easy maintenance of object invariants: invariants of initialized immutable objects are never invalid, not even temporarily.
  (object immutability is enough)
- Security: even unchecked code (e.g., legacy code or untrusted Java applets) cannot mutate immutable data.
  (class immutability is needed)

Reference immutability (a.k.a. Read-only References)

Immutable References

A reference is immutable if the state of the object it refers to cannot be modified through this reference.

Examples

- The following Java library method creates an immutable reference to a collection `c`:

```
Collection unmodifiableCollection(Collection c)
```

- `Iterator` references to collection internals are often immutable.

Java's Final Fields: You Can Do a Lot With Them

- Immutable records.

```
@Immutable class Point {
    final int x;
    final int y;
    Point(int x, int y) { this.x = x; this.y = y; }
}
```

- Immutable recursive data structures (in functional style).

```
@Immutable interface ConsList<E> {
    E head();
    ConsList<E> tail();
}
@Immutable class Cons<E> implements ConsList<E> {
    final E hd;
    final ConsList<E> tl;
    public Cons(E hd, Cons<E> tl) { this.hd = hd; this.tl = tl; }
    public E head() { return hd; }
    public ConsList<E> tail() { return tl; }
}
@Immutable class Nil<E> implements ConsList<E> { ... }
```

So Why are Final Fields Not Enough?

So Why are Final Fields Not Enough?

- They don't prevent methods that mutate representation objects.

```
@Immutable class String {
    private final char[] a; // part of the string's representation
    public void bad() {
        a[0] = 'z';         // mutation not prevented
    }
}
```

So Why are Final Fields Not Enough?

- They don't prevent methods that mutate representation objects.

```
@Immutable class String {
    private final char[] a; // part of the string's representation
    public void bad() {
        a[0] = 'z';         // mutation not prevented
    }
}
```

- They don't enforce encapsulation of representation objects.

```
@Immutable class String {
    private final char[] a;
    public String(char[] a) {
        this.a = a;         // failure to make a defensive copy
    }                       // string can be mutated from outside
```

So Why are Final Fields Not Enough?

- They don't prevent methods that mutate representation objects.

```
@Immutable class String {
    private final char[] a; // part of the string's representation
    public void bad() {
        a[0] = 'z';         // mutation not prevented
    }
}
```

- They don't enforce encapsulation of representation objects.

```
@Immutable class String {
    private final char[] a;
    public String(char[] a) {
        this.a = a;         // failure to make a defensive copy
    }                       // string can be mutated from outside
```

- They don't support initialization outside constructors.

So Why are Final Fields Not Enough?

- They don't prevent methods that mutate representation objects.

```
@Immutable class String {
    private final char[] a; // part of the string's representation
    public void bad() {
        a[0] = 'z';         // mutation not prevented
    }
}
```

- They don't enforce encapsulation of representation objects.

```
@Immutable class String {
    private final char[] a;
    public String(char[] a) {
        this.a = a;         // failure to make a defensive copy
    }                       // string can be mutated from outside
```

- They don't support initialization outside constructors.
- They don't support reference immutability.

A Type System for Immutability

I will now present a simple type system for specifying and verifying:

- object immutability
- reference immutability
- class immutability

Type Qualifiers: RdWr and Rd

Qualifiers.

$$q \quad ::= \quad \begin{array}{ll} \texttt{RdWr} & \text{read-write access (default)} \\ \texttt{Rd} & \text{read-only access} \end{array}$$

Types.

$$T \ ::= \ q \ C$$

If an object has type `Rd` C then its fields may only be read.

Type Qualifiers: RdWr and Rd

Qualifiers.

$$q \quad ::= \quad \texttt{RdWr} \quad \text{read-write access (default)}$$
$$\texttt{Rd} \quad \text{read-only access}$$

Types.

$$T \ ::= \ q\ C$$

If an object has type `Rd` C then its fields may only be read.

```
class C { int f; }

static void m(Rd C x) {
    x.f = 42; // TYPE ERROR
}

static void m(RdWr C x) {
    x.f = 42; // OK
}
```

Safety Property.

Well-typed programs never write to `Rd`-objects.

Type Qualifiers: Any for Reference Immutability

Qualifiers.

$q \quad ::= \quad \cdots$

`Any` "the referred object is either `Rd` or `RdWr`"

Subqualifying.

`Rd` <: `Any` `RdWr` <: `Any`

Subtyping.

$$\frac{p <: q \qquad C <: D}{p\ C <: q\ D}$$

Writes through `Any`-references are prohibited.

Type Qualifiers: Any for Reference Immutability

Qualifiers.

q ::= ⋯

`Any` "the referred object is either `Rd` or `RdWr`"

Subqualifying.

`Rd` <: `Any` `RdWr` <: `Any`

Subtyping.

$$\frac{p <: q \qquad C <: D}{p\ C <: q\ D}$$

Writes through `Any`-references are prohibited.

```
interface Util {
    void foo(int Any [] x);
}

static void m(Util util) {
    int[] a = new int RdWr [] {42,43,44};
    util.foo(a);
    assert a[0] == 42;
}
```

The Access Qualifier is a Class Parameter

- Classes have a special class parameter `MyAccess`.
- `MyAccess` refers to the access qualifier for `this`.

```
class Square {
    MyAccess Point upperleft;
    MyAccess Point lowerright;
}

class Point {
    int x;
    int y;
}

static void m(Rd Square s) {
    s.upperleft = s.lowerright; // TYPE ERROR
    s.upperleft.x = 42; // TYPE ERROR
}
```

The Access Qualifier is a Class Parameter

- A covariant class parameter? Is that sound?
- Yes it is, because writeable qualifiers are minimal.
- So whenever one upcasts a qualifier, the associated reference becomes immutable.

Object Initialization

- Object initialization inside constructors is often too restrictive, e.g.:
  - immutable arrays
  - immutable instances of mutable collections
  - immutable cyclic structures
- Moreover, restricting to initialization inside constructors does not make things simpler, because arbitrary code is allowed in constructor bodies.

Initialization Inside Stack-Local Regions

Initialization Token.

$n \in$ Token — token for initializing a set of related objects

Qualifiers.

$q ::= \cdots$

`Fresh(`n`)` — fresh object under initialization

`Fresh` objects are writeable, even if they later turn immutable.

Initialization Inside Stack-Local Regions

Initialization Token.

n	$\in$	Token	token for initializing a set of related objects

Qualifiers.

q	$::=$	$\cdots$	
		`Fresh(`n`)`	fresh object under initialization

`Fresh` objects are writeable, even if they later turn immutable.

Ghost commands.

`newtoken` n	create a new initialization token
`commit Fresh(`n`) as` q	globally convert `Fresh(`n`)` to q

These have no runtime effect. They can be inferred automatically.

Example: Initializing an Immutable Array

```
static char Rd [] copy (char RdWr [] w) {
    newtoken n;
    char[] r = new char Fresh(n) [w.length];
    for (int i=0; i++; i < w.length) r[i] = w[i];
    commit Fresh(n) as Rd;
    return r;
}
```

Example: Initializing an Immutable Cyclic Structure

```
class Person {
    MyAccess Person partner;
}
class Couple {
    MyAccess Person husband;
    MyAccess Person wife;
}

newtoken n;
Person alice = new <Fresh(n)>Person();
Person bob = new <Fresh(n)>Person();
alice.partner = bob; bob.partner = alice;
Couple couple = new <Fresh(n)>Couple();
couple.husband = bob; couple.wife = alice;
commit Fresh(n) as Rd;
```

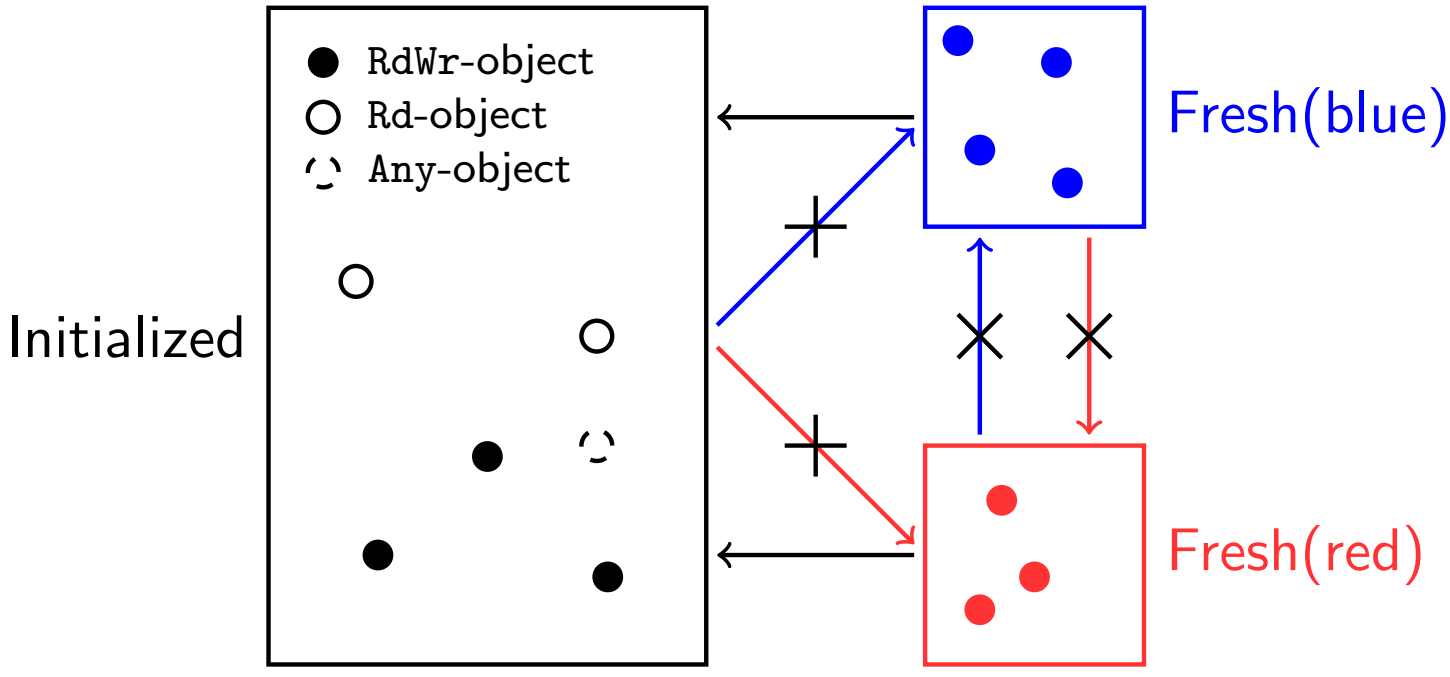
Soundness of Commit (Part 1)

Classes cannot have field types with `Fresh(`n`)`-qualifiers.

```
class C {
    Fresh(n) D x; // TYPE ERROR: n out of scope
}
```

Soundness of Commit (The Heap Invariant)

(picture slightly inaccurate for Generics)



Heap Invariant.

There are no ingoing references into `Fresh`-regions.

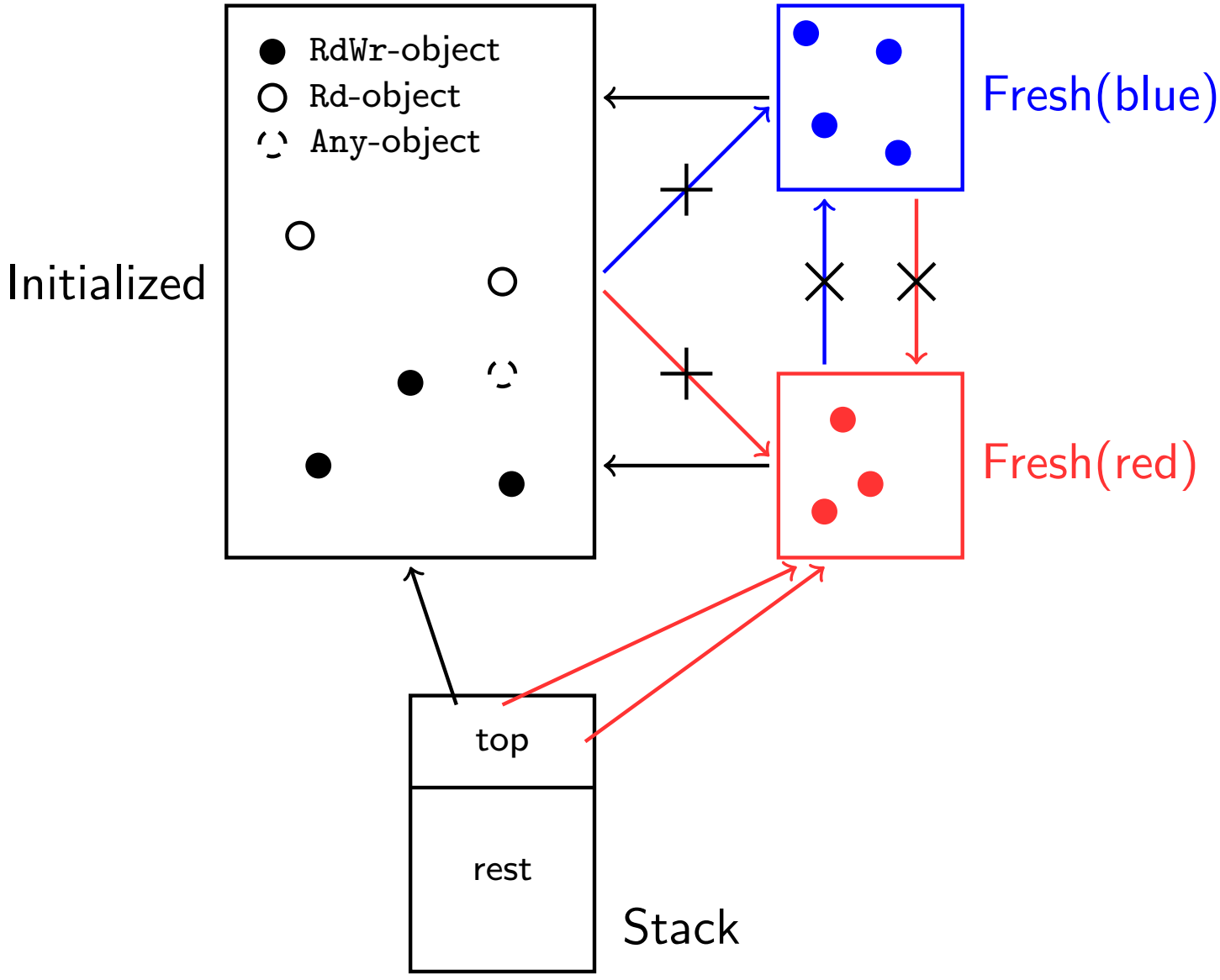
N.B.: references inside `Fresh` regions are possible, by `MyAccess` qualifier on fields.

Soundness of Commit (Part 2)

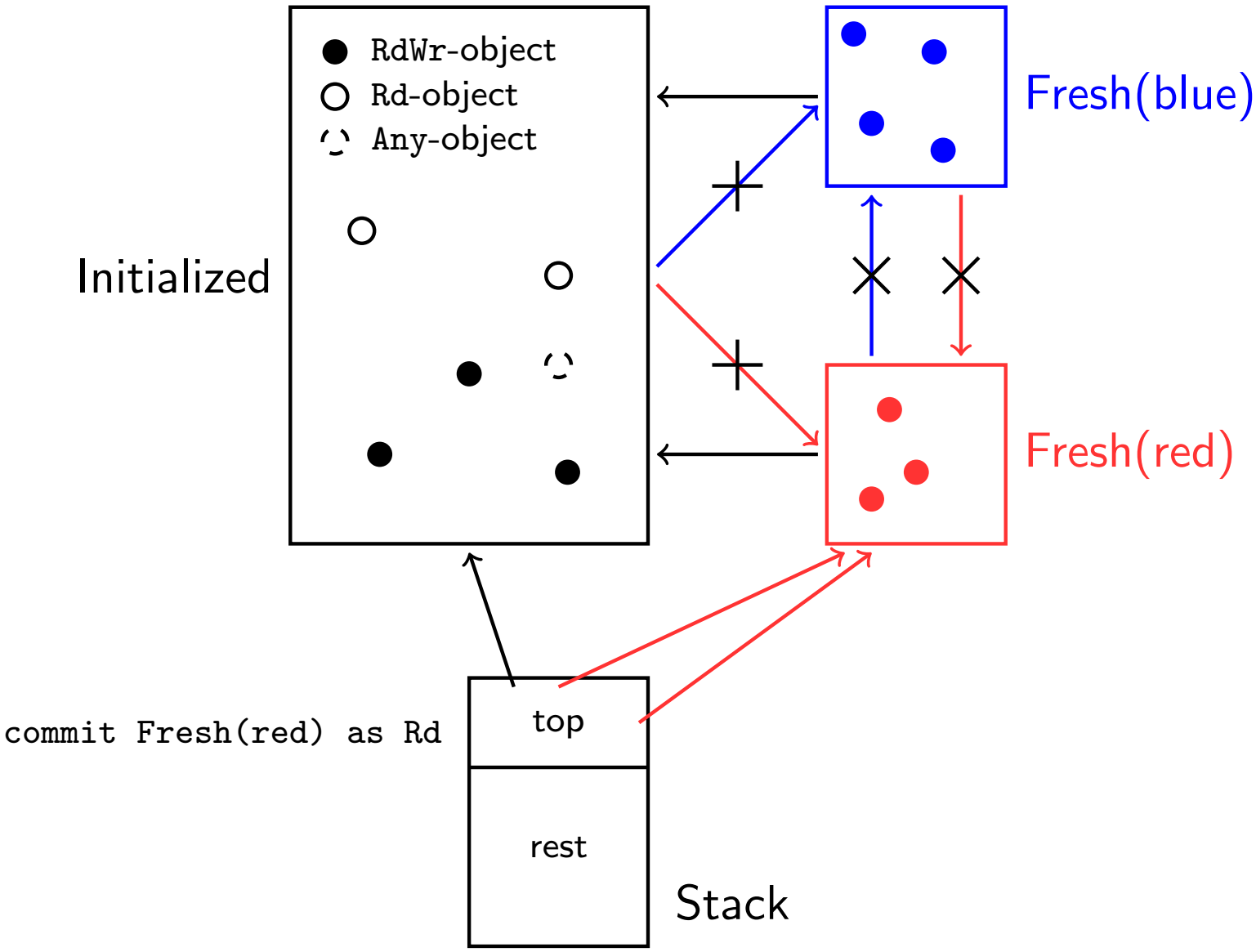
Only the method that generated *n* can commit `Fresh(`*n*`)`.

```
Rd C commit(Fresh(n) C x) { // TYPE ERROR: n out of scope
    commit Fresh(n) as Rd;
    return x;
}
```

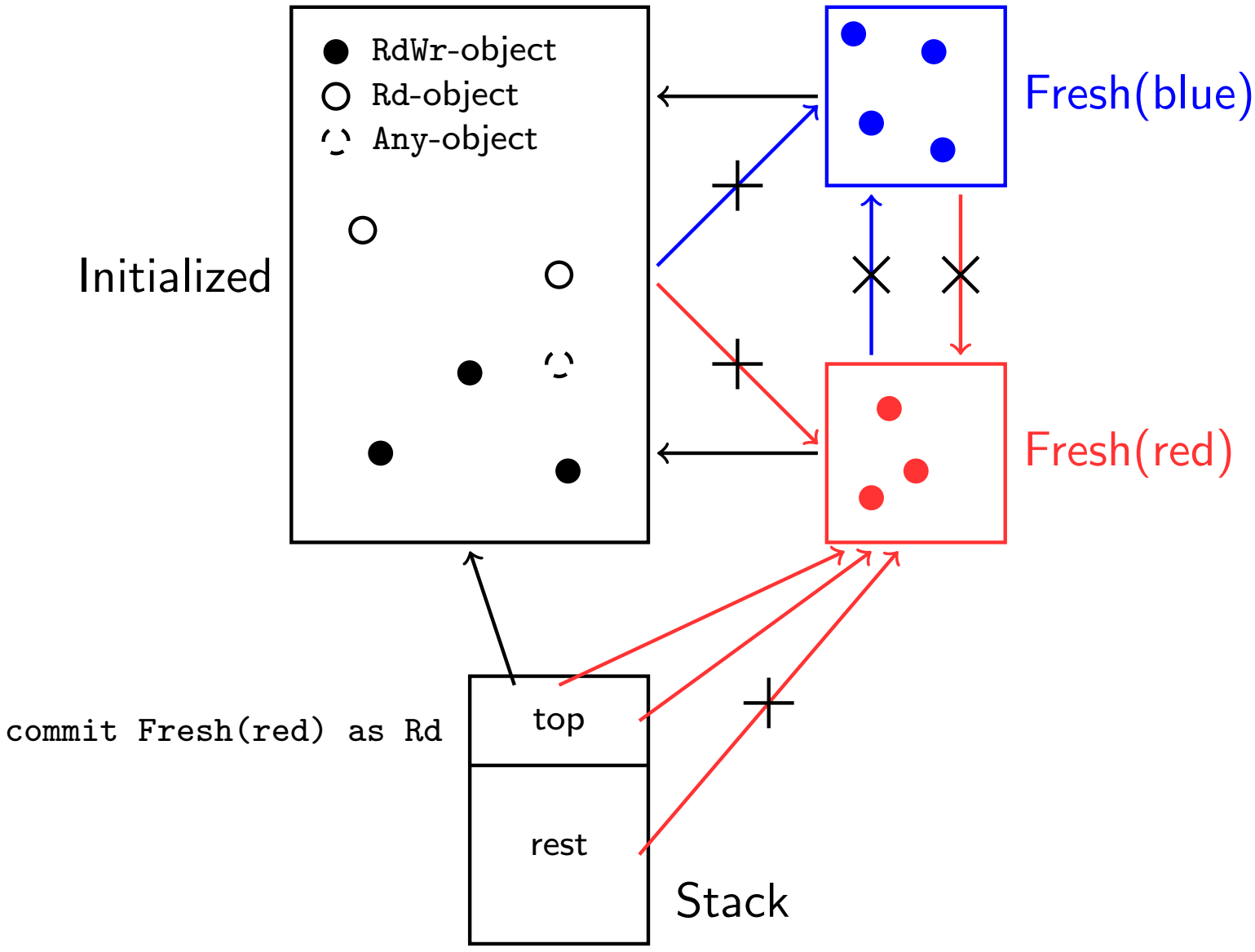
Soundness of Commit (Finale)



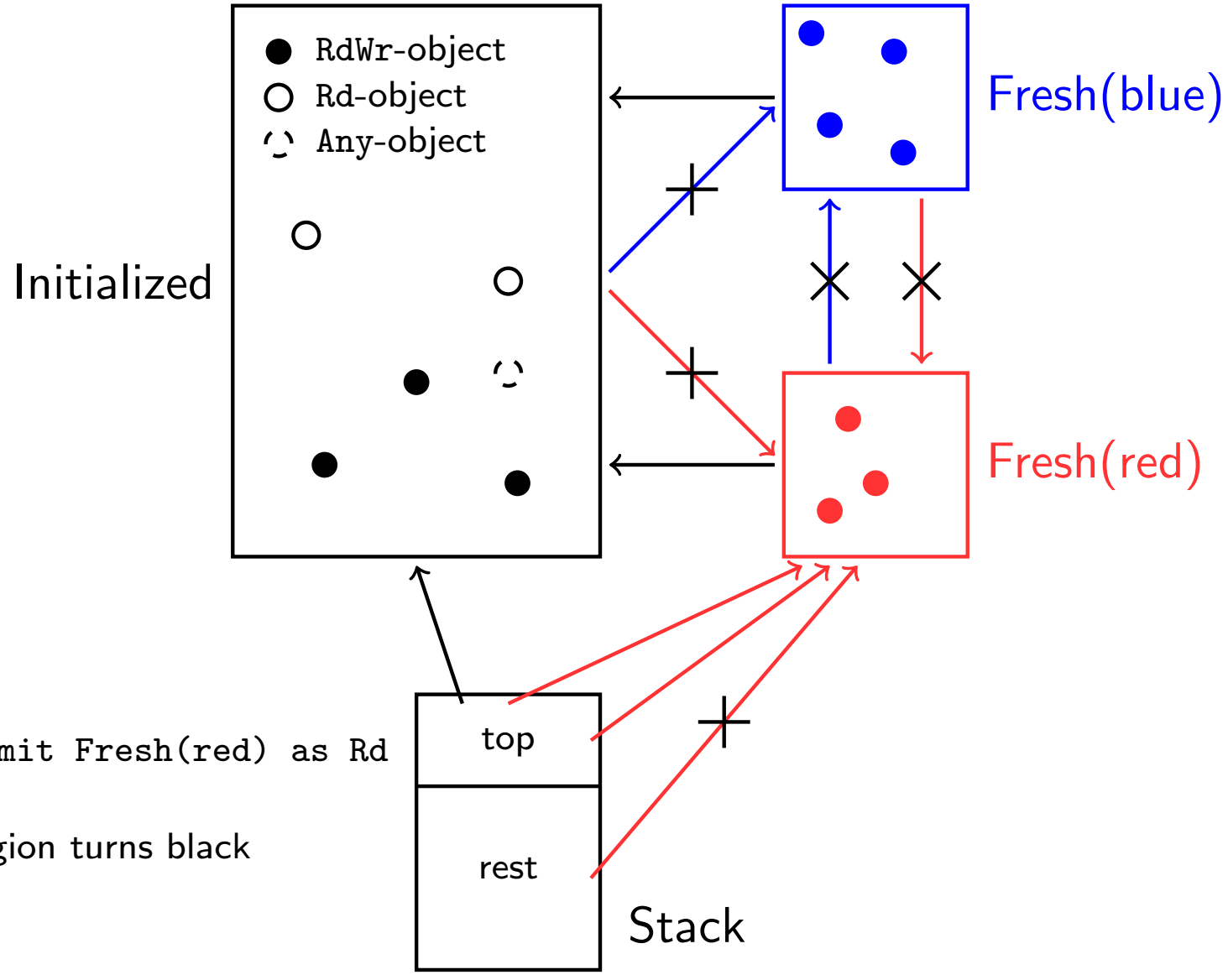
Soundness of Commit (Finale)



Soundness of Commit (Finale)

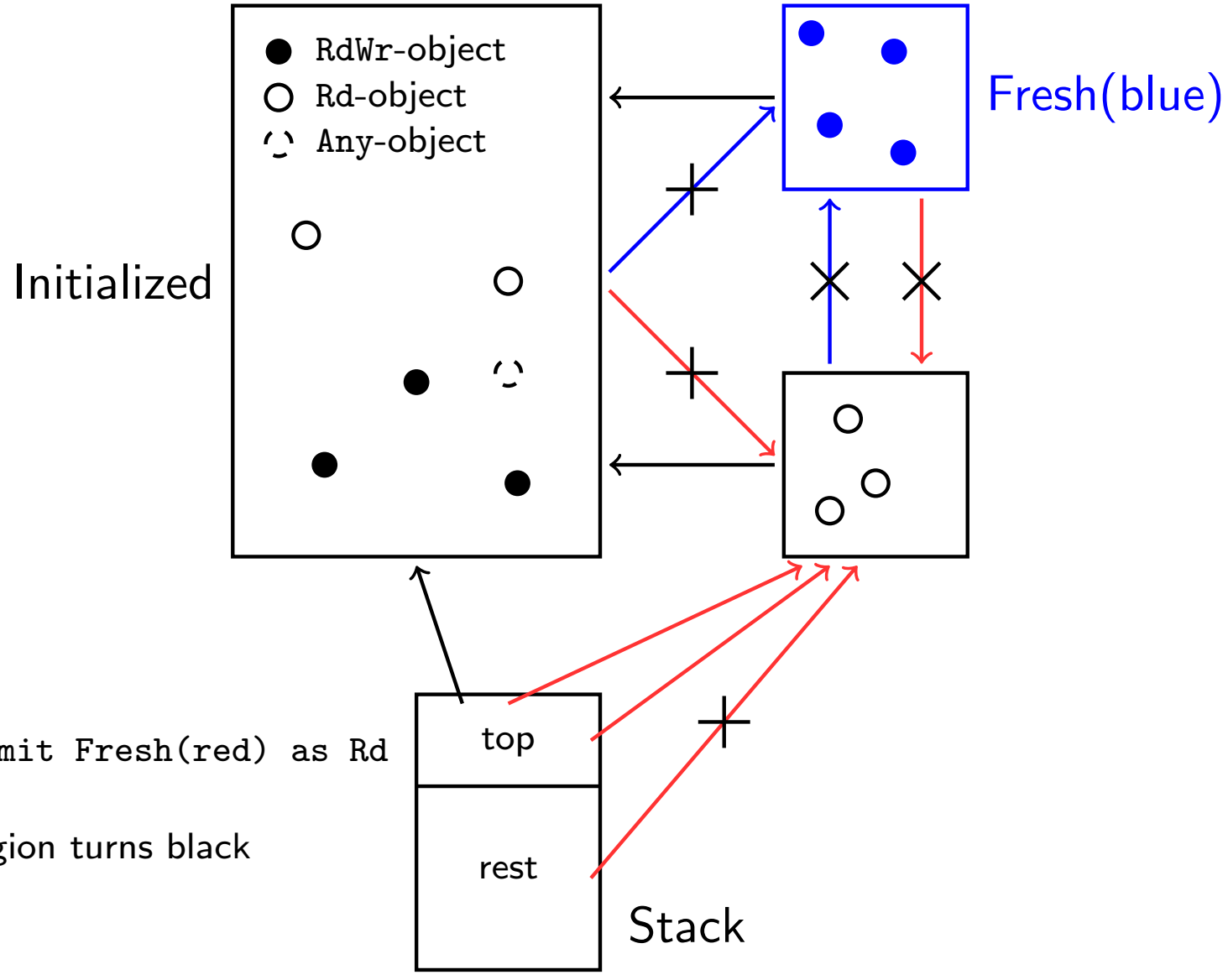


Soundness of Commit (Finale)



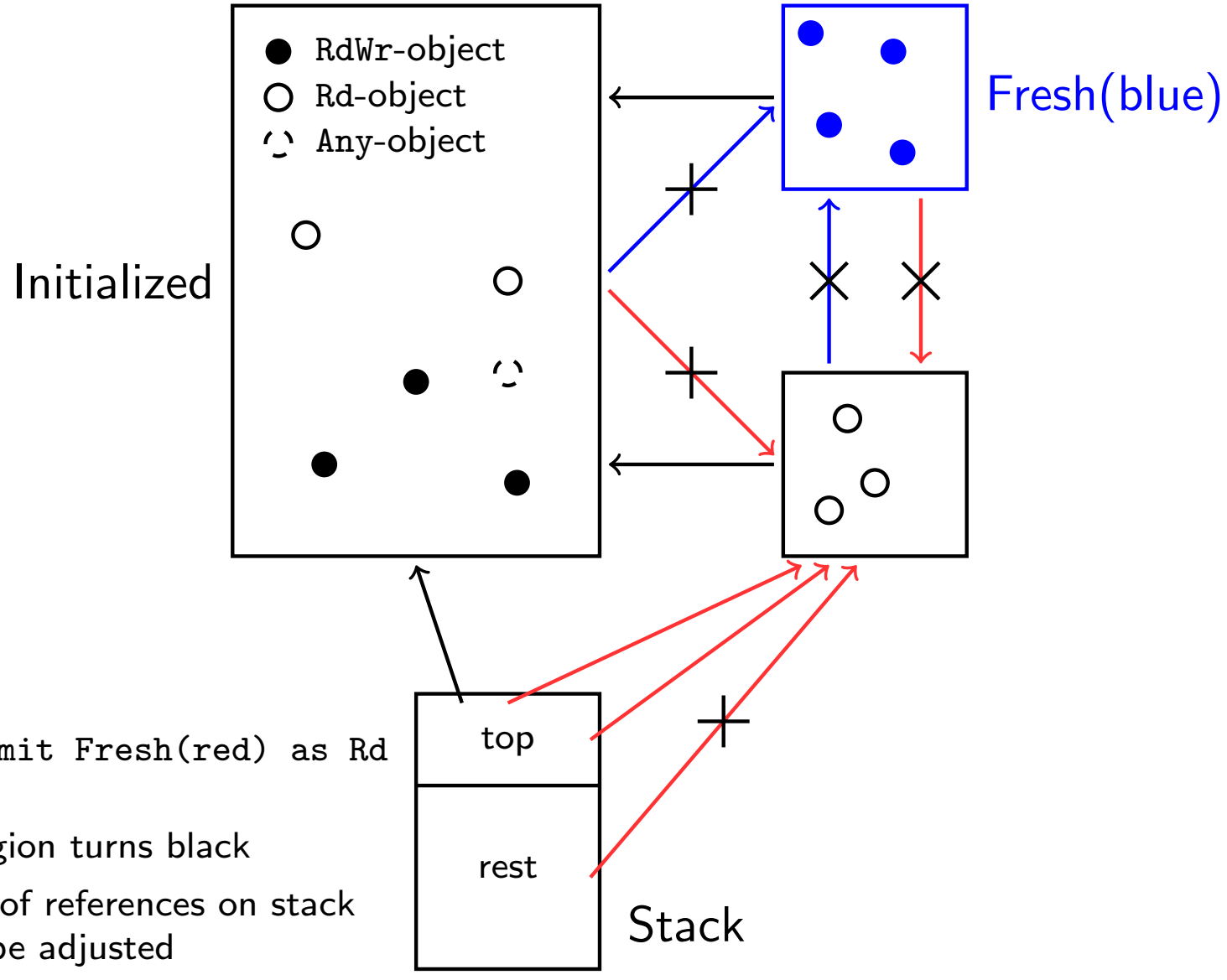
- red region turns black

Soundness of Commit (Finale)



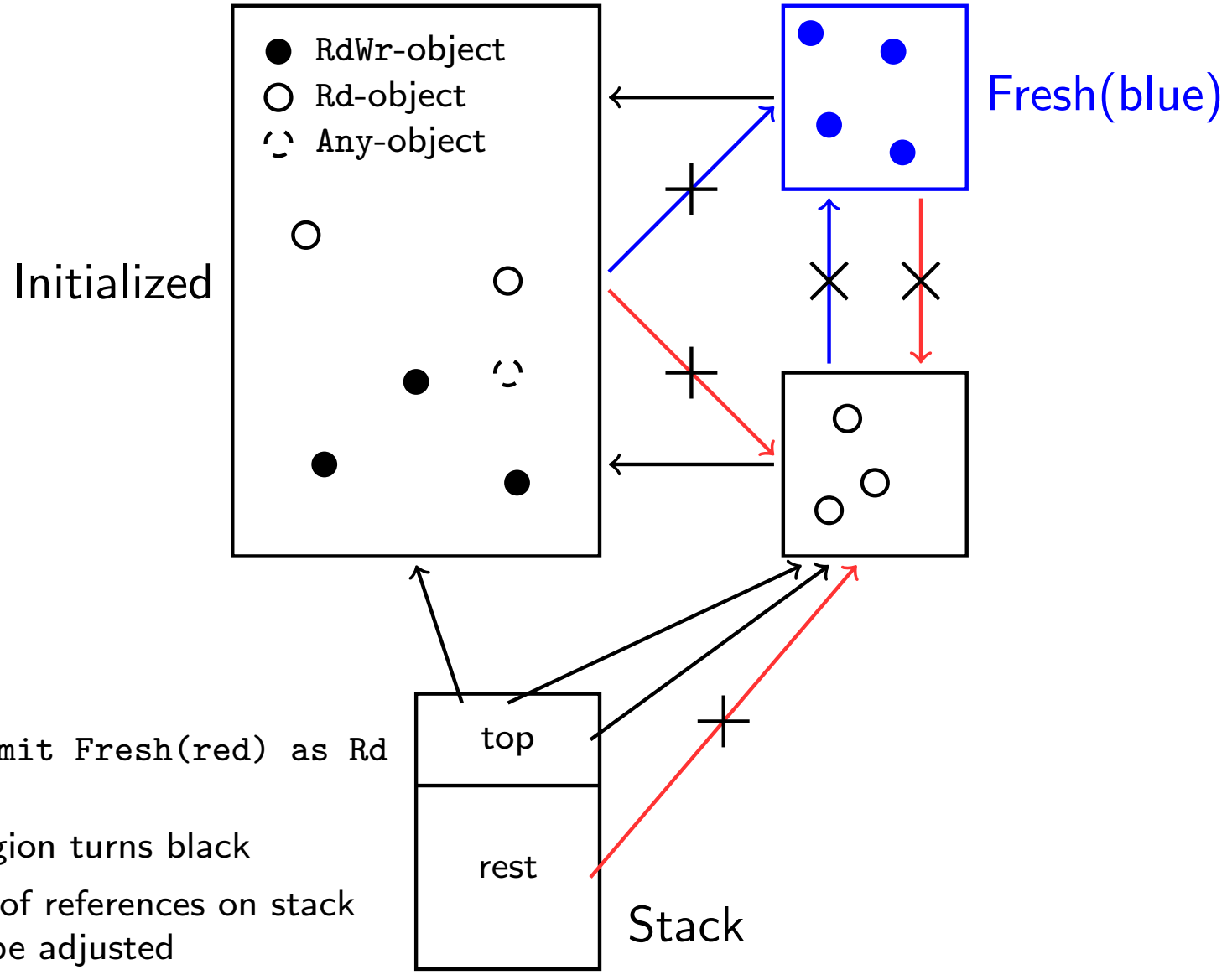
- red region turns black

Soundness of Commit (Finale)



- red region turns black
- colors of references on stack must be adjusted

Soundness of Commit (Finale)



- red region turns black
- colors of references on stack must be adjusted

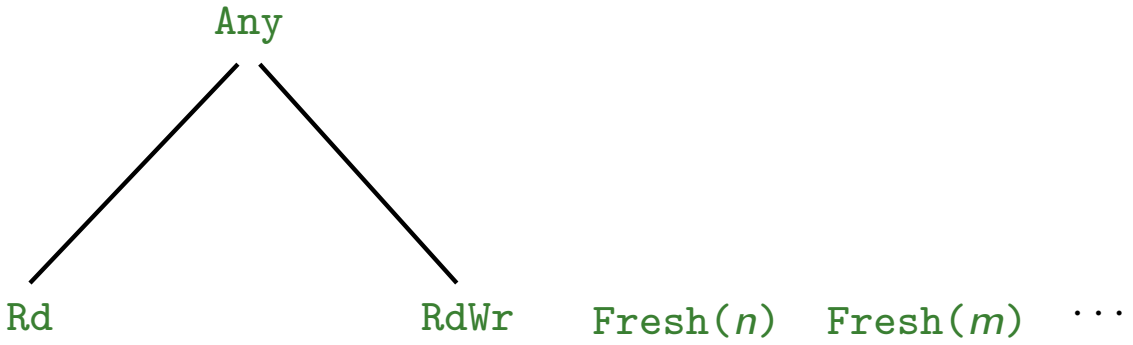
Qualifier Polymorphism for Methods

```
static void copy(Point src, Point dst) {
    dst.x = src.x; dst.y = src.y;
}
```

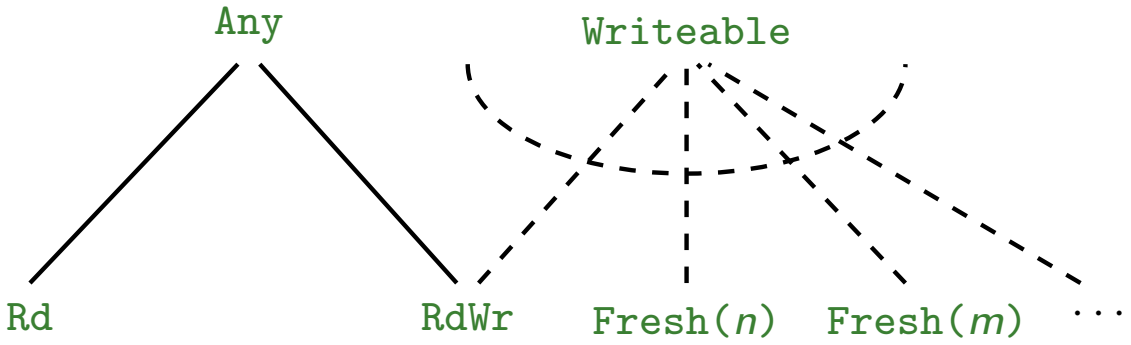
It should be allowed to pass actual `dst`-parameters of types `RdWr Point` and `Fresh(n) Point`.

- This method is similar to `arraycopy()`.
- `arraycopy()` is called in constructors of immutable `String`s.

Qualifier Polymorphism for Methods (cont.)

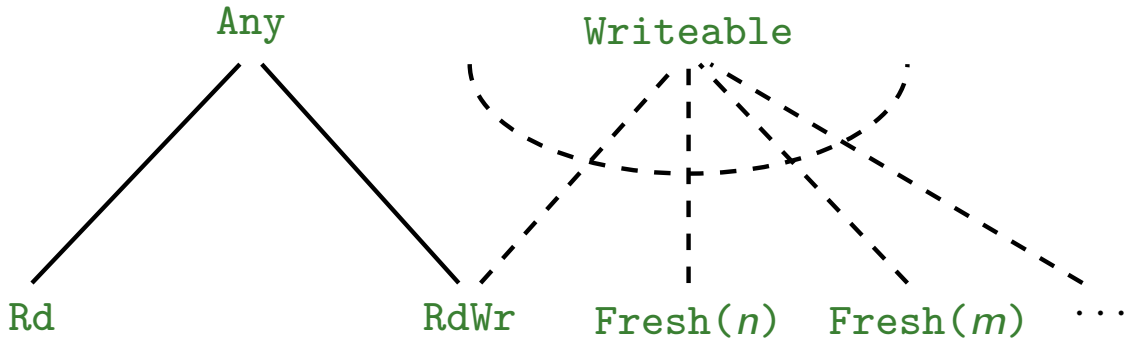


Qualifier Polymorphism for Methods (cont.)



`Writeable` can only be used as a bound, not as a type.

Qualifier Polymorphism for Methods (cont.)



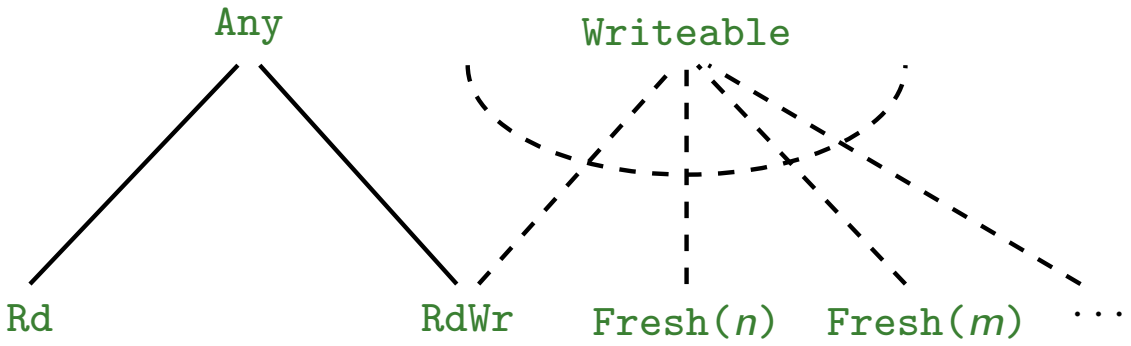
`Writeable` can only be used as a bound, not as a type.

Typing rule for field sets (incomplete sketch).

$$\frac{x : q\ C \qquad q \texttt{ extends Writeable}}{x.f = v : \text{ok}}$$

```
static <a, b extends Writeable> void copy(a Point src, b Point dst) {
    dst.x = src.x; dst.y = src.y;
}
```

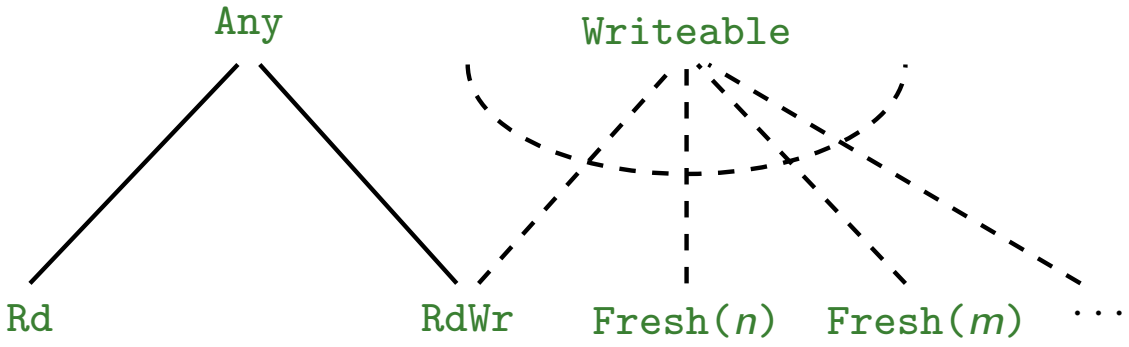
Why can Writeable not be a Type?



This would lead to unsoundness for two reasons:

- fields of type `Writeable` could refer to `Fresh(`n`)`-objects.
  $\Rightarrow$ unsoundness of `commit`
- `Writeable` would be a non-minimal qualifier that allows writes.
  $\Rightarrow$ unsoundness of covariance for `MyAccess` class parameter

More on Qualifier-Polymorphic Methods



- `static <a> void foo(int a [] x)`
  - does not write to object `x` through reference `x`
  - does not write object `x` to the heap
- `static void faa(int Any [] x)`
  - does not write to object `x` through reference `x`
  - may write object `x` to the heap (into `Any`-fields)
- `static <a extends Writeable> void fee(int a [] x)`
  - may write to object `x` through reference `x`
  - does not write object `x` to the heap

Receiver Qualifiers (supported by JSR 308)

- JSR 308 provides a slot for annotations on the receiver type.
- Inspectors can be called on any receivers.
- Mutators can only be called on `Writeable` receivers.

```
class Hashtable<K,V> {

    <a> V get(K key) a { ... }

    <a extends Writeable> V put(K key, V value) a { ... }
}
```

```
newtoken n;
Hashtable<String,String> t = <Fresh(n)>Hashtable<String,String>();
t.put("Tarmo", "Tallinn");
t.put("Vladimir", "Koblenz");
commit Fresh(n) as Rd;
t.get("Tarmo"); // OK
t.put("Reiner", "Gothenburg"); // TYPE ERROR
```

Constructors

Constructors must have one of the following forms:

1. <$\bar{a}$ `extends` $\bar{B}$> q $C(\bar{T}\ \bar{x})$ p { *body* }

   Typically:

   <a `extends` `Writeable`> a $C(\bar{T}\ \bar{x})$ a { *body* }

   - Caller commits.
   - Better choice most of the time.

2. <$\bar{a}$ `extends` $\bar{B}$> q $C(\bar{T}\ \bar{x})$ { `newtoken n;` *body* }

   - Constructor commits.
   - Useful for class immutability in an open world.

It is disallowed to call constructors of the second form using `super()`. The second form is therefore only recommended for `final` classes.

Closed vs. Open World

Class immutability in a closed world.

- Assumes that clients of immutable classes follow the rules of the pluggable type system.

Class immutability in an open world.

- Assumes that clients of immutable classes only follow Java's standard typing rules.

Class immutability in an open world must ensure that representation objects are encapsulated.

Class Immutability (Open World)

A class annotation: `Immutable`

```
Immutable final class String {
    private final char MyAccess [] arr; ...
}
```

Rules.

- immutable classes must be final and direct subclasses of `Object`
- methods and constructors may only call static or final methods (transitively)
- all fields must be private
- constructors must have the following form

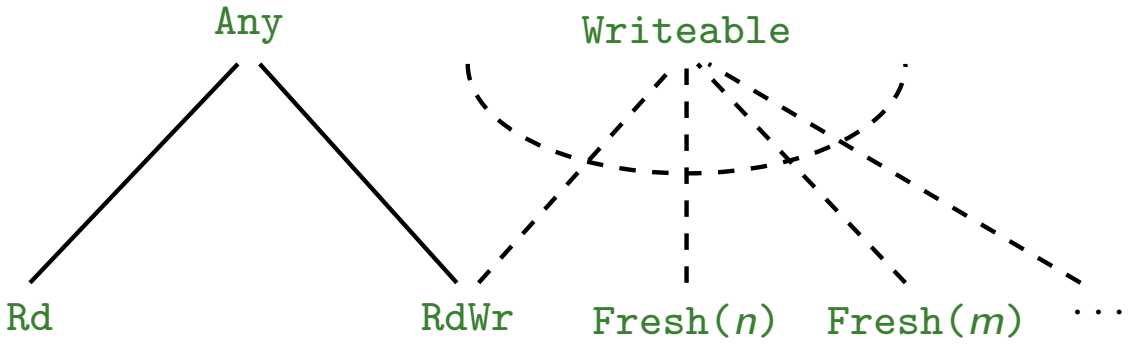
  `Rd C(`$\bar{T}$ $\bar{x}$`) { newtoken n; ...; commit Fresh(n) as Rd; }`

  where `MyAccess` does not occur in $\bar{T}$.
- types of public methods must have the following form:

  `<a> U m(`$\bar{T}$ $\bar{x}$`) a { ... }`

  where `MyAccess` and `a` do not occur in $U, \bar{T}$.

Threads



- We must ensure that thread-shared objects are initialized (i.e., not `Fresh`).
- Why? A thread can commit `Fresh(n)` without other threads knowing about it.

```
class Thread {

    void run() RdWr { }
    void start(); // Treated specially. The type system uses run()'s type.
}
```

- Note that subclasses of `Thread` may override `run()` with receiver type `RdWr` or `Any` (by contravariance).
- Because the thread object is forced to be initialized when the thread is started, so are all objects reachable from it.

Outline

1. A Type System for Immutability

2. Lexically Scoped Regions for Initialization of Object Types
   - Safe Deallocation with Lexically Scoped Regions
   - Lexically Scoped Regions for Immutability
   - A Type System for Non-Nullness

3. References

Lexically Scoped Regions

- Have been used for safe memory deallocation.
  - e.g., in *Cyclone*, a memory-safe dialect of C
- Have been invented by *Tofte and Talpin* for *ML*.
  - emphasized region *inference* to insert deallocation instructions at compile time (*"compile-time garbage collection"*).

A Model Language with Lexically Scoped Regions

- I will present a small model language with lexical scoped regions.
  - Adapted from the Cyclone paper [GMJ+02].
  - See lecture notes for omitted details.
- The language has a primitive for explicit memory deallocation.

Safety Property.

Well-typed programs do not attempt to access deallocated memory locations.

Regions

- *Regions* are named segments of the heap.
- Every object is member of exactly one region.

$n \in \text{RegionName}$	*region names*
$\alpha \in \text{RegionVar}$	*region variables*
$p, q, r \ ::= \ \texttt{Region}(n) \mid \alpha$	*region expressions*

- Actual names of regions are of the form *`Region(n)`*.
- *Region variables* refer to these actual names indirectly.
  - are as class parameters and auxiliary method parameters.

Classes and Types

The model language is based on records (no dynamic dispatch).

Classes are region-parametrized:

$$\texttt{class}\ C\texttt{<}\bar{\alpha}\texttt{>}\ \texttt{\{}\ \bar{T}\ \bar{f}\ \texttt{\}}$$

Reference types:

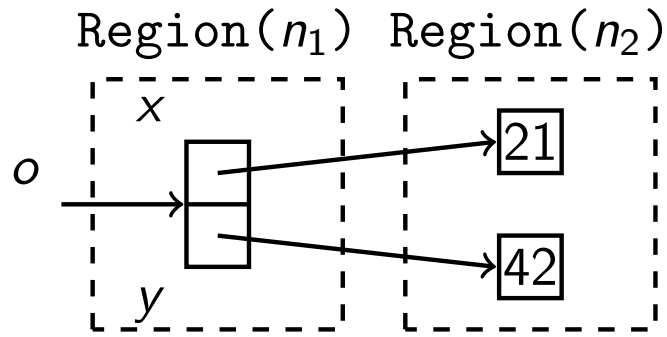
$$q\ C\texttt{<}\bar{r}\texttt{>}$$

- q is the region the object belongs to.
- $\bar{r}$ instantiate the class parameters.

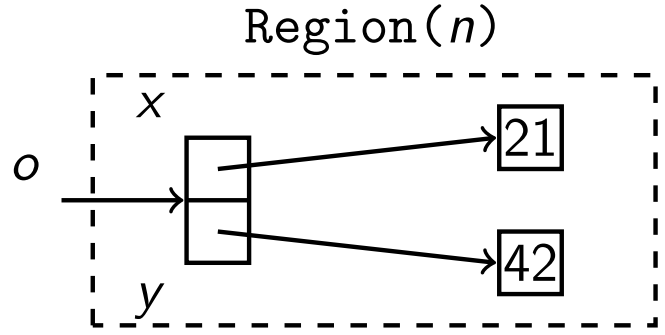
Example

```
class Integer { int val; }
class Point<r> { r Integer x; r Integer x; }
```

- o of type $Region(n_1)$ Point<$Region(n_2)$>:



- o of type $Region(n)$ Point<$Region(n)$>:



Methods

Methods are region-polymorphic:

$$<\bar{\alpha}>\ U\ m(\bar{T}\ \bar{x})\{e\}$$

- The scope of $\bar{\alpha}$ is $U, \bar{T}, e$.

Example

```
<r,q> q Point<q> copy(r Point<r> arg) {
    q Point<q> result = new q Point<q>;
    // result.x = arg.x;      THIS WOULD BE A TYPE ERROR
    result.x = new q Integer;
    result.y = new q Integer;
    result.x.val = arg.x.val;
    result.y.val = arg.y.val;
    return result;
}
```

- The method type gives information about method behaviour:
  - *copy() cannot return a shallow copy of its argument.*

Expressions

e ::=

...	
`new` q $C\langle\bar{r}\rangle$	*object creation*
`newName` n `in` e `end`	*region generation (scope of n is e)*

`new` q $C\langle\bar{r}\rangle$

- Creates a new object of class $C\langle\bar{r}\rangle$ and places it in region q.

`newName` n `in` e `end`

- Generates a fresh region name for an initially empty $\mathit{Region}(n)$, then executes e, and finally deallocates all objects in $\mathit{Region}(n)$.

Crucial property enforced by type system:

- *Objects in $\mathit{Region}(n)$ are not reachable outside the lexical scope of n.*

Example

```
newName n in
    Region(n) Point<Region(n)> p;
    p = new Region(n) Point<Region(n)>;
    p.x = new Region(n) Integer;
    p.y = new Region(n) Integer;
    p.x.val = 21;
    p.y.val = 42;
    ... // Do something with point p.
end // All objects in Region(n), including p, p.x and p.y, get deallocated.
```

Typing Rules

$\Delta \subseteq \mathsf{RegionName} \cup \mathsf{RegionVar}$ *region environments*

$\Gamma \in \mathsf{Var} \rightarrow \mathsf{Ty}$ *type environments*

$$\frac{n \notin \Delta \quad \Delta, n; \Gamma \vdash e : T \quad \Delta \vdash \Gamma, T : \mathsf{ok}}{\Delta; \Gamma \vdash \texttt{newName}\ n\ \texttt{in}\ e\ \texttt{end} : T}$$

where: $\Delta \vdash \mathcal{J} : \mathsf{ok} \triangleq \{n, \alpha \mid n, \alpha \text{ occurs in } \mathcal{J}\} \subseteq \Delta$

Examples

The following unsafe programs do not typecheck:

```
Region(n) Point<Region(n)> pt0;
pt0 = newName n in
          Region(n) Point<Region(n)> pt1;
          ...
          pt1 // TYPE ERROR: n in pt0's and pt1's type aren't the same
      end
pt0.x; // UNSAFE MEMORY ACCESS
```

Examples

The following unsafe programs do not typecheck:

```
Region(n) Point<Region(n)> pt0;
pt0 = newName n in
          Region(n) Point<Region(n)> pt1;
          ...
          pt1 // TYPE ERROR: n in pt0's and pt1's type aren't the same
      end
pt0.x; // UNSAFE MEMORY ACCESS
```

```
Region(n) Point<Region(n)> pt0;
newName n in
    Region(n) Point<Region(n)> pt1;
    ...
    pt0 = pt1; // TYPE ERROR: n in pt0's and pt1's type aren't the same
    ...
end
pt0.x; // UNSAFE MEMORY ACCESS
```

Memory Safety

For objects o, define:

$$\text{regionNames}(o) \triangleq \text{set of region names that occur in } o\text{'s type}$$

Heap invariant.

If object o is reachable from object p,
then $\text{regionNames}(o) \subseteq \text{regionNames}(p)$.

- Consequence: Types of local variables must refer to all regions reachable from them.
- But types of local variables can only refer to regions that are in lexical scope.

$\Rightarrow$ All reachable regions are in lexical scope.

Lexically Scoped Regions and Typestate Transitions

How do lexically scoped regions help with typestate transitions?

- Instead of deallocating a region at the end of its scope, the type system *recycles the region with a different type*.

In region-based memory management:

- `newName` has an effect at runtime.

For typestate transitions:

- `newName` has *no* effect at runtime.

Lexically Scoped Regions for Immutability

Adding access qualifiers for persistent typestates:

$$p, q, r \ ::= \ \texttt{RdWr} \ \mid \ \texttt{Rd} \ \mid \ \texttt{Any} \ \mid \ \texttt{Fresh}(n) \ \mid \ \alpha$$

`newName` n `for` q `in` e `end`

- This has no effect at runtime. To the typechecker, it introduces a new name n with scope e and tells the typechecker that `Fresh`(n) becomes q after e terminates.

$$\frac{n \notin \Delta \quad \Delta, n; \Gamma \vdash e : T \quad \Delta \vdash \Gamma, q : \text{ok}}{\Delta; \Gamma \vdash \texttt{newName}\ n\ \texttt{for}\ q\ \texttt{in}\ e\ \texttt{end} : T[q/\texttt{Fresh}(n)]}$$

A Type System for Non-Nullness

The type system distinguishes between non-null and maybe-null.

$C!$	C-object and not `null`
C	C-object or `null`

Safety Property.

Well-typed programs do not attempt to dereference `null`.

I will present

- Fähndrich and Xia's system of delayed types (essentially),
- which is under the hood of $\mathrm{Spec}^\sharp$'s non-nullness checker.

Problem Comparison: Non-nullness vs. Immutability

Non-nullness system	Immutability system
After object initialization:	
guarantees an object property (namely that certain fields are non-null).	restricts an object permission (namely the permission to write).
Initialization phase is associated with:	
an obligation (to establish a property)	a temporary permission (to write)

Additional complication in non-nullness system:

- the type system has to ensure that the initialization phase establishes non-nullness of all non-null object fields

Types

Types

$$T, U, V \ ::= \ q\,\tau \ \mid \ \texttt{int} \ \mid \ \texttt{void}$$

Unqualified Types

$$\tau \ ::= \ C \ \mid \ C!$$

Type Qualifiers

$p, q, r \ ::=$

`Initialized`	initialized object
`Fresh(`n`)`	fresh object under initialization
α	qualifier variable

Subtyping

$$q\ C! <: q\ C$$

Write Effects

Write Effects

$$E \subseteq \mathsf{Var} \times \mathsf{FieldId}$$

Methods

$$<\bar{\alpha}>\ U\ m(\bar{T}\ \bar{x})\ E\{e\} \qquad (\text{where } E \subseteq \{x.f \mid x \in \bar{x}, f \in \mathsf{FieldId}\})$$

Judgment Format for Expressions

$$\Delta; \Gamma \vdash e : T; E$$

Enforcing Initialization of Non-null Fields

- Tie object initialization to a `let`-expression `let` x `=new Fresh(`n`)` C `in` e `end`.
- Check that all non-null fields of x have been written at the end of e

Enforcing Initialization of Non-null Fields

- Tie object initialization to a `let`-expression `let` x `= new Fresh(`n`)` C `in` e `end`.
- Check that all non-null fields of x have been written at the end of e

$$\frac{C \text{ declared} \quad x \notin \mathrm{dom}(\Gamma) \qquad \Delta; \Gamma, x : \texttt{Fresh}(n)\ C \vdash e : T; E \quad \mathrm{nnfields}(C) \subseteq \pi_x(E)}{\Delta; \Gamma \vdash \texttt{let}\ x = \texttt{new Fresh}(n)\ C\ \texttt{in}\ e\ \texttt{end} : T; (E \setminus \pi_x(E))}$$

The Trouble with Writing Fields

The obvious rule.

$$\frac{\Delta;\Gamma \vdash x : q\ C!;\emptyset \quad \texttt{class}\ C\ \{..\ T\ f\ ..\} \quad \Delta;\Gamma \vdash e : T[q/\texttt{mystate}];E}{\Delta;\Gamma \vdash x.f = e : T[q/\texttt{mystate}];E \cup \{x.f\}}$$

This rule is too restrictive!

- It disallows writing `Initialized` objects to `mystate`-fields of `Fresh` objects.
- This is needed when inserting `Fresh` objects into `Initialized` cyclic data structures.

A More Liberal Rule for Writing Fields

Fähndrich and Xia use the following more liberal rule:

$$\frac{\Delta;\Gamma \vdash x : q\ C!;\emptyset \quad \texttt{class}\ C\ \{..\ T\ f\ ..\} \quad T[q/\texttt{mystate}] = p\ \tau \qquad \Delta;\Gamma \vdash e : p'\ \tau;E \quad p' = \texttt{Initialized or } p' = p}{\Delta;\Gamma \vdash x.f = e : \texttt{void};E \cup \{x.f\}}$$

Now we can write `Initialized` objects to `mystate`-fields of `Fresh` objects.

The Trouble with Reading Fields

- But now the system permits writing values whose type does not match the field type.
- This creates trouble for the rule for reading fields!

A Possible Rule for Reading

The following rule entirely disallows reading `Fresh`-typed fields:

$$\frac{\Delta; \Gamma \vdash e : q\ C!; E \quad \texttt{class}\ C\ \{..\ T\ f\ ..\} \quad T[q/\texttt{mystate}] = \texttt{Initialized}\ \tau}{\Delta; \Gamma \vdash e.f : T[q/\texttt{mystate}]; E}$$

This rule is sound.

A More Liberal Rule for Reading

Fähndrich and Xia use a more liberal that:

- Allows reading `Fresh`-typed fields.
- Associates the retrieved value with a *qualifier wildcard*.

$$\frac{\Delta;\Gamma \vdash e : q\ C!;E \quad \texttt{class}\ C\ \{..\ T\ f\ ..\} \quad T[q/\texttt{mystate}] = p\ \tau <: p\ D \qquad U = \begin{cases} p\ \tau & \text{if } p = \texttt{Initialized} \\ ?\ D & \text{otherwise} \end{cases}}{\Delta;\Gamma \vdash e.f : U;E}$$

Effectively, the qualifier wildcard makes it impossible to write to the retrieved object.

Wildcard Capture

- In order to be able to use wildcard-qualified objects at all, we need a rule for wildcard capture.
- This rule allows replacing ? by a fresh qualifier variable.

$$\frac{\Delta;\Gamma \vdash e : ?\,\tau; E \quad \alpha \notin \Delta \quad x \notin \mathsf{dom}(\Gamma) \quad \Delta,\alpha;\Gamma,x:\alpha\,\tau \vdash e' : T; E'}{\Delta;\Gamma \vdash \texttt{unpack}\,\alpha, x = e\ \texttt{in}\ e'\texttt{end} : T; E \cup (E' \setminus \pi_x(E'))}$$

Effectively:

Wildcard-qualified objects can be read but can't be written.

Delayed Types in Spec♯

- Spec♯'s surface syntax uses a single type qualifier `Delayed`.
- The surface syntax can be desugared into the Fähndrich and Xia's core language.
- For instance, `new C(...)` is translated to:

```
newName n in
  let tmp = new Fresh(n) C in
    tmp.<Fresh(n)>ctor(...);
  end;
  tmp
end
```

- See my lecture notes or Fähndrich and Xia's paper for some additional details on the desugaring.

Summary

Lexically scoped regions are a useful technique for enabling flexible initialization of pluggable object types.

That's it for today!

Outline

1. A Type System for Immutability
2. Lexically Scoped Regions for Initialization of Object Types
   - Safe Deallocation with Lexically Scoped Regions
   - Lexically Scoped Regions for Immutability
   - A Type System for Non-Nullness
3. References

References

M. Fähndrich and S. Xia.

Establishing object invariants with delayed types.

In *Object-Oriented Programming Systems, Languages, and Applications*, pages 337–350. ACM, 2007.

D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney.

Region-based memory management in Cyclone.

In *Programming Languages Design and Implementation*, pages 282–293, 2002.

C. Haack.

Supplementary notes for this talk.

http://viinistu.cost-ic0701.org/Christian-Haack, 2009.

C. Haack and E. Poll.

Type-based object immutability with flexible initialization.

Technical Report ICIS-R09001, Radboud University, Nijmegen, January 2009.