

Uniqueness and Reference Immutability for Safe Parallelism

Colin Gordon (UW), Matt Parkinson (MSR),
Jared Parsons (MS), Aleks Bromfield (MS),
Joe Duffy (MS)

OOPSLA 2012

A Prototype Extension to C#

- Extend C# for safe parallelism
- Safe task & data parallelism
 - No locks
 - No mutable statics/globals
- Statically enforce data-race freedom
- Real: *millions* of LOC
 - Experience report later

Data-Race-Free, Parallel, Imperative OOP

- Prove soundness for
 - Data-race freedom
 - Reference immutability
 - External uniqueness
 - Borrowing (new technique)
 - Precise generics over permissions

Data-Race-Free, Parallel, Imperative OOP

- Prove soundness for
 - Data-race freedom
 - Reference immutability
 - External uniqueness
 - Borrowing (new technique)
 - Precise generics over permissions

Safe Parallelism Essentials

- Separate read and write effects
 - As in DPJ, Habanero Java, etc.
- Invariant: At all times, for all objects, either



→ Possible write access
→ Read-only access

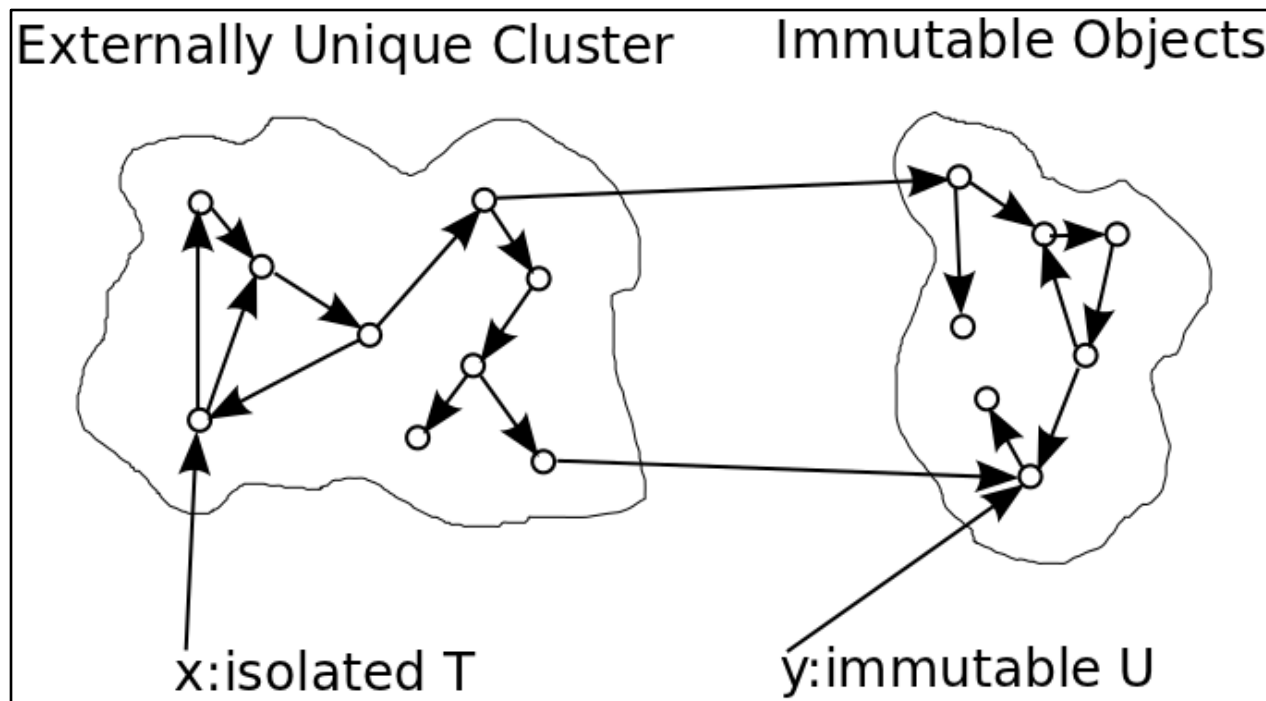
Reference Immutability

- writable: "normal" reference
- readable: deeply read-only
 - Cannot get writable reference through a readable
 - $x:\text{readable} \vdash x.f : t \Rightarrow t$ is never writable
- immutable: permanently immutable
- As in Tschantz et al. '05, Zibin et al. '07, Zibin et al. '10

External Uniqueness

Unique reference: **only** reference to an object

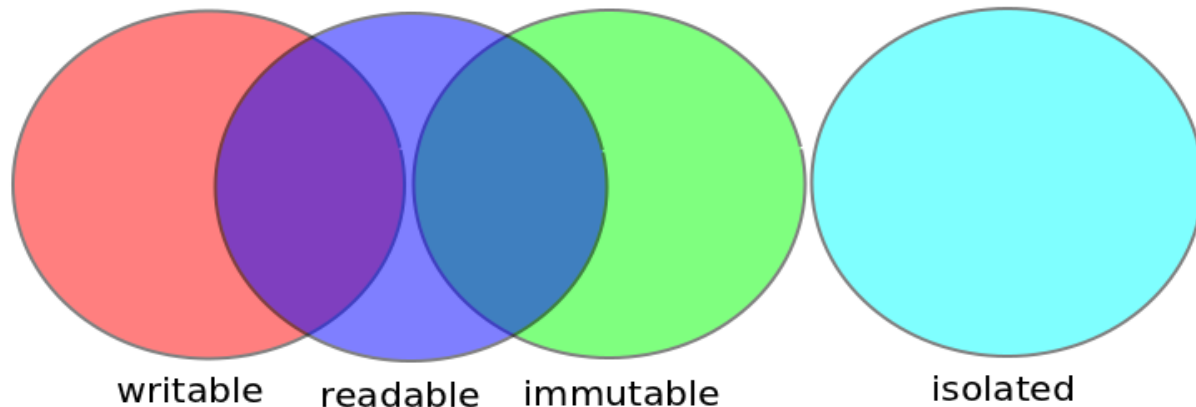
Externally-Unique reference: only **external** reference into a group of objects



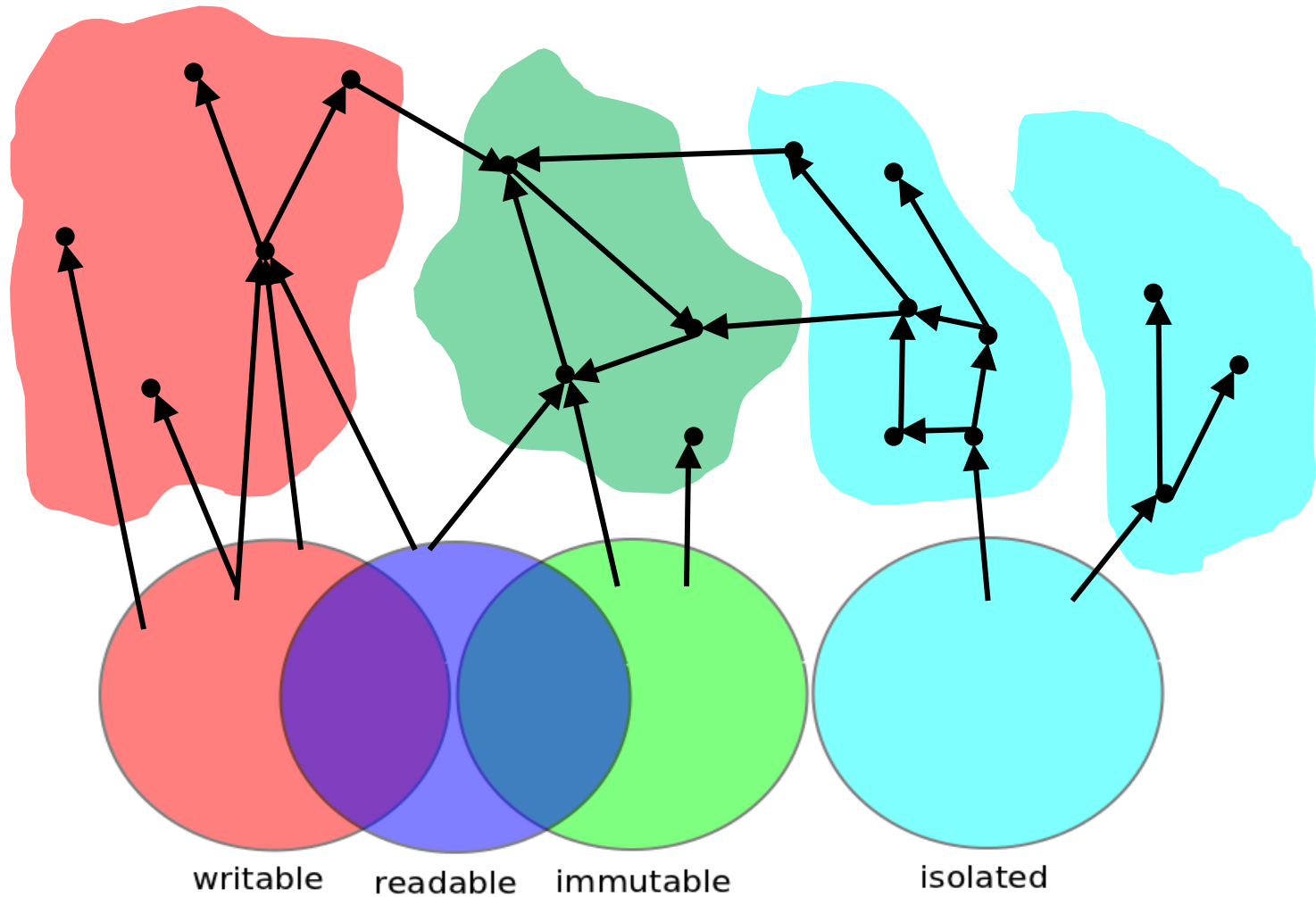
Isolated and Immutable can both reach some objects

Reference Immutability + Uniqueness

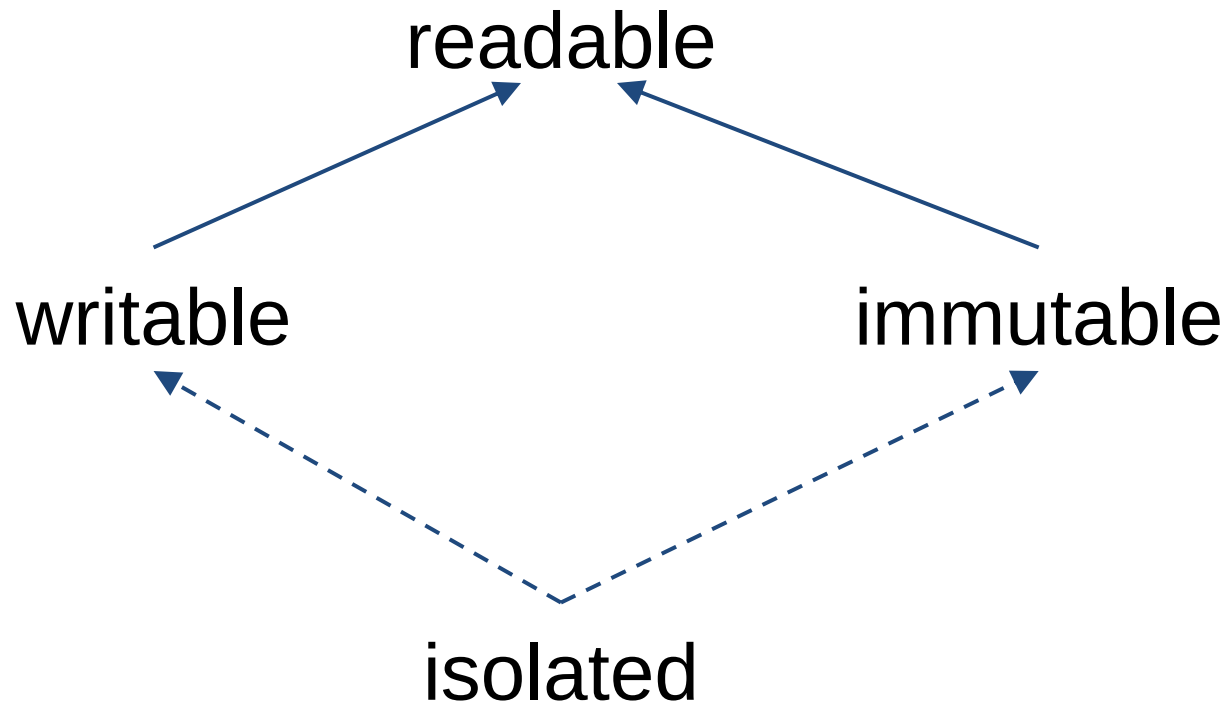
- writable: "normal" reference
- readable: deeply read-only
 - Cannot get writable reference through a readable
 - $x:\text{readable} \vdash x.f : t \Rightarrow t \text{ is never writable}$
- immutable: permanently immutable
- isolated: externally-unique reference



Reachability Among Permissions

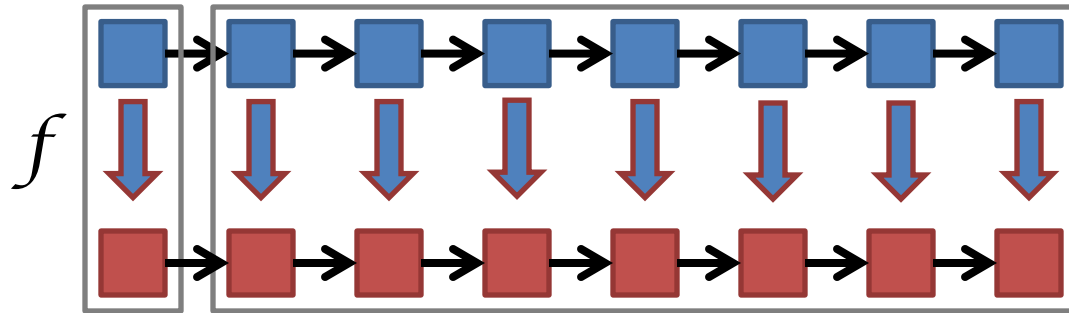


Subtyping Conversions



Converting *from* isolated loses uniqueness

Symmetric Parallelism



Symmetric Parallelism

```
List<X> pmap(List<X> l, Func1<X,X> f) {
```

```
List<X> head = null;  
head = new List<X>;  
head.elem = f(l.elem);
```

```
List<X> rest = null;  
if (l.next != null)  
    rest = pmap(l.next, f);
```

```
head.next = rest;
```

```
return head;
```

```
}
```

$$\frac{\text{NoWrit}(\Gamma_1) \quad \text{NoWrit}(\Gamma_2) \quad \Gamma_1 \vdash C_1 \dashv \Gamma'_1 \quad \Gamma_2 \vdash C_2 \dashv \Gamma'_2}{\Gamma_1, \Gamma_2 \vdash C_1 || C_2 \dashv \Gamma'_1, \Gamma'_2} \text{T-PAR}$$

What About Writable + Parallelism?

$$\frac{\Gamma_1 \vdash C \dashv \Gamma_2}{\Gamma, \Gamma_1 \vdash C \dashv \Gamma, \Gamma_2} \text{ T-FRAME}$$

Asymmetric Parallelism

```
Integer i = ...;  
isolated IntegerList lst=new IntegerList();  
... // populate list  
isolated SortFunc f = new SortFunc();  
// Sort in parallel with other work  
f.apply(lst); || i.val = 3;
```

$$\frac{\text{IsoOrImm}(\Gamma_1) \quad \Gamma_1 \vdash C_1 \dashv \Gamma'_1 \quad \Gamma_2 \vdash C_2 \dashv \Gamma'_2}{\Gamma_1, \Gamma_2 \vdash C_1 || C_2 \dashv \Gamma'_1, \Gamma'_2} \text{T-ASYNC}$$

Extending Isolated References

Want to use isolated refs without losing uniqueness forever:

```
isolated IntBox x = new IntBox();  
x.val = 3; // implicit conversion  
isolated IntBox y = consume(x);
```

Want to use normal code to build isolated object graphs:

```
writable CircularListNode make2NodeList(){...}  
isolated CircularListNode x = make2NodeList();
```

Recovering Uniqueness

$$\frac{\text{IsoOrImm}(\Gamma) \quad \text{IsoOrImm}(\Gamma') \quad \Gamma \vdash C \dashv \Gamma', x : \text{writable } T}{\Gamma \vdash C \dashv \Gamma', x : \text{isolated } T} \text{ T-RECOVISO}$$

- Analagous rule for readable -> immutable

Recovering Isolation in Practice

```
// The default permission is writable
writable CircularListNode make2NodeList() {
    CircularListNode n1 = new CircularListNode();
    CircularListNode n2 = new CircularListNode();
    n1.next = n2; n1.prev = n2;
    n2.next = n1; n2.prev = n1;
    return n1;
}
...
isolated l_iso = make2NodeList();
immutable l_imm = l_iso;
```

make2NodeList doesn't care about uniqueness

Recovery vs. Source Language

- Recovery subsumes multiple source features
 - Borrowing w/o special treatment
 - Creating immutable objects
 - Easy construction of complex isolated structures
 - Method dispatch on isolated receivers
- Wouldn't have found this by formalizing the source language!

It Lives! A Prototype in Use

Formal language models a prototype C# extension
in *active use at Microsoft*.

- Some differences from formal system
 - e.g. first-class tasks, unstrict blocks
- Millions of lines of code
- Most parallelism checked
 - 100% within a couple weeks
- Anecdotally: More RI finds more bugs
 - Races found before safe parallelism was turned on

Rough Spots in Practice

Today's code works around a few limitations:

- Optimizations
 - Idioms for performance (e.g. caches)
 - Lazy initialization for immutable structures
 - Safe library abstractions for these
- Dynamic partitioning
 - Sub-Arrays
- Communication for performance

More in the Paper

- Design Evolution
- Generics
 - Retain precision for combining generic permissions
 - E.g. one iterator impl. for all collection and element permission pairs
 - Safe covariant subtyping
- Proof by embedding into a program logic
 - The Views Framework
(Dinsdale-Young et al., accepted to POPL'13)

Contributions

- Combine reference immutability & external uniqueness
- Reference immutability for data race freedom
- New borrowing technique
- Generics over RI permissions
- Largest industrial use of RI

Choose Your Own Backup Slides:

- [Design History & Evolution](#)
- [Generics](#)
- [Soundness Proof](#)

History

1. Spec#'s [Pure] method attribute
 2. readable and writable
 3. Library abstractions for isolated and immutable
 4. All four permissions
 5. Generics
 6. Proof
- 1-5 guided by user feedback

User-Friendly Design

- Default permission: writable
 - Single-threaded C# w/o globals compiles unchanged
- Local Type Inference
 - Inherited from C#
- Approx. 1 annotation / 63 LOC
 - Mostly on method declarations
- People* like it, eagerly add readable refs
 - *mostly ex-systems-C++ people fed up with locks
 - Success related to team priorities for correctness vs. ease

Generics Goal: Polymorphic Iterator

```
w LinkedList<IntBox><w> ll = new LinkedList<IntBox><w>();
```

```
w IEnumerator<IntBox><w,w> e = ll.GetEnumerator();
```

```
w IntBox b = e.getNext();
```

```
w LinkedList<IntBox><r> llr = new LinkedList<IntBox><r>();
```

```
w IEnumerator<IntBox><w,r> e2 = llr.GetEnumerator();
```

```
w IntBox b = e2.getNext(); // Type Error!
```

```
r IntBox b = e2.getNext(); // OK
```

Polymorphic Iterator

```
public interface ICollection<Elem><PElem> {  
    public void add(PElem Elem e) writable;  
    public writable Enumerator<Elem><P,PElem>  
        GetEnumerator() P;  
}  
  
public interface IEnumerator<Elem><PColl,PElem> {  
    public bool hasNext() readable;  
    public PColl  PElem Elem getNext() writable;  
}
```

Formal Permission Polymorphism

$$\frac{t' \ f \in T \quad p \neq \text{isolated} \vee t' = \text{immutable} _}{t' \neq \text{isolated} _ \vee p = \text{immutable}}$$

$$\Delta \mid x : _, y : p \ T \vdash x = y.f \dashv y : p \ T, x : p \triangleright_{\Delta} t'$$

$$P \triangleright_{\Delta} Q = \begin{cases} Q & \Delta \vdash P \prec Q \\ P & \Delta \vdash Q \prec P \\ P \rightsquigarrow Q & \text{otherwise} \end{cases}$$

- Key insight: deferred combination $P \rightsquigarrow Q$
 - Combination deferred until full instantiation
- Formalization inspired by Dietl & Müller's GUT

Proving Safe Parallelism

- Proof by embedding into Views Framework
 - Program logic generalizing Separation Logic and Rely-Guarantee reasoning
 - States form partial cancellative monoid
 - Separation corresponds to threads OR env. framing
- Each type environment denotes some set of program states
- Those denotations are *stable with respect to R* (possible interference)
- Details in the paper; more details in the TR