Uniqueness and Reference Immutability for Safe Parallelism

Colin S. Gordon[†], Matthew J. Parkinson[‡], Jared Parsons[◇], Aleks Bromfield[◇], Joe Duffy[◇]

[†]University of Washington, [‡]Microsoft Research Cambridge, [◇]Microsoft Corporation

csgordon@cs.washington.edu, {mattpark,jaredpar,albromfi,joedu}@microsoft.com

Abstract

A key challenge for concurrent programming is that side-effects (memory operations) in one thread can affect the behavior of another thread. In this paper, we present a type system to restrict the updates to memory to prevent these unintended side-effects. We provide a novel combination of immutable and unique (isolated) types that ensures safe parallelism (race freedom and deterministic execution). The type system includes support for polymorphism over type qualifiers, and can easily create cycles of immutable objects. Key to the system's flexibility is the ability to recover immutable or externally unique references after violating uniqueness without any explicit alias tracking. Our type system models a prototype extension to C# that is in active use by a Microsoft team. We describe their experiences building large systems with this extension. We prove the soundness of the type system by an embedding into a program logic.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Concurrent programming structures; F.3.2 [*Semantics of Programming Languages*]: Program Analysis

General Terms Languages, Verification

Keywords reference immutability, type systems, concurrency, views

1. Introduction

In concurrent programs, side-effects in one thread can affect the behavior of another thread. This makes programs hard to understand as programmers must consider the context in which their thread executes. In a relaxed memory setting even understanding the possible interactions is non-trivial.

We wish to restrict, or tame, side-effects to make programs easier to maintain and understand. To do so, we build on *reference immutability* [37, 39, 40], which uses permission type qualifiers to control object mutation. Typically there are notions of writable references (normal access), read-only references (objects may not be mutated via a read-only reference, and field reads from read-only references produce read-only references), and immutable references (whose referents may never be changed through any alias). There are many applications of this approach to controlling side effects, ranging from improving code understanding to test generation to compiler optimization.

We add to reference immutability a notion of isolation in the form of an extension to external uniqueness [23]. We support the natural use of isolation for object immutability (making objects permanently immutable through all references). But we also show a new use: to *recover isolation* or strengthen immutability assumptions without any alias tracking. To achieve this we give two novel typing rules, which allow recovering isolated or immutable references from arbitrary code checked in environments containing only isolated or immutable inputs.

We provide two forms of parallelism:

Symmetric Assuming that at most one thread may hold writable references to an object at a given point in time, then while all writable references in a context are temporarily forgotten (framed away, in the separation logic sense [29, 33]), it becomes safe to share all read-only or immutable references among multiple threads, in addition to partitioning externally-unique clusters between threads.

Asymmetric If all data accessible to a new thread is immutable or from externally-unique clusters which are made inaccessible to the spawning thread, then the new and old threads may run in parallel without interference.

We provide an extended version of the type system with polymorphism over reference immutability qualifiers. This maintains precision for instantiated uses even through rich patterns like iterators, which was not possible in previous work [39].

There are several aspects of this work which we are the first to do. We are the first to give a denotational meaning to reference immutability qualifiers. We are the first to formalize the use of reference immutability for safe parallelism. We

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1561-6/12/10...$10.00

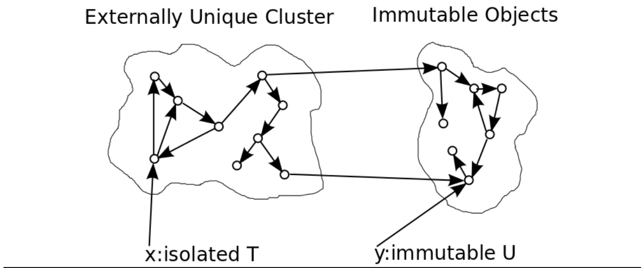


Figure 1. External uniqueness with immutable out-references.

are the first to describe industry experience with a reference immutability type system.

2. Reference Immutability, Uniqueness, and Parallelism

Reference immutability is based on a set of permission-qualified types. Our system has four qualifiers:

writable: An "ordinary" object reference, which allows mutation of its referent.

readable: A read-only reference, which allows no mutation of its referent. Furthermore, no heap traversal through a read-only reference produces a writable reference (writable references to the same objects may exist and be reachable elsewhere, just not through a readable reference). A readable reference may also refer to an immutable object.

immutable: A read-only reference which additionally notes that its referent can never be mutated through any reference. Immutable references may be aliased by read-only or immutable references, but no other kind of reference. All objects reachable from an immutable reference are also immutable.

isolated: An external reference to an externally-unique object cluster. External uniqueness naturally captures thread locality of data. An externally-unique aggregate is a cluster of objects that freely reference each other, but for which only one external reference into the aggregate exists. We define isolation slightly differently from most work on external uniqueness because we also have immutable objects: all paths to *non-immutable* objects reachable from the isolated reference pass through the isolated reference. We allow references out of the externally-unique aggregate to immutable data because it adds flexibility without compromising our uses for isolation: converting clusters to immutable, and supporting non-interference among threads (see Figure 1). This change in definition does limit some traditional uses of externally-unique references that are not our focus, such as resource management tasks.

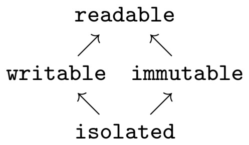


Figure 2. Qualifier conversion/subtyping lattice.

The most obvious use for reference immutability is to control where heap modification may occur in a program, similar to the owner-as-modifier discipline in ownership and universe type systems [14]. For example, a developer can be sure that a library call to a static method with the type signature

```
int countElements(readable ElementList lst);
```

will not modify the list or its elements (through the `lst` reference). Accessing any field of the argument `lst` through the readable reference passed will produce other readable (or immutable) results. For example, a developer could not implement `countElements` like so:

```
int countElements(readable ElementList lst)
{ lst.head = null; return 0; }
```

because the compiler would issue a type error. In fact, any attempt within `countElements()` to modify the list would result in a type error, because `lst` is deeply (transitively) read-only, and writes through read-only references are prohibited.

2.1 Conversion from Isolated

The `isolated` qualifier is atypical in reference immutability work, and is not truly a permission for (im)mutability in the purest sense. In fact, we require that `isolated` references be converted through subtyping to another permission before use, according to the type qualifier hierarchy in Figure 2.

`isolated` references are particularly important in our system for two reasons. First, they naturally support safe parallelism by partitioning mutable data amongst threads. The threads[1] in the following example cannot interfere with each other, because the object graphs they operate on and can mutate are disjoint:

```
isolated IntList l1 = ...;
isolated IntList l2 = ...;
{ l1.map(new Incrementor()); }
   || { l2.map(new Incrementor()); }
```

Second, the control of aliasing allows *conversion* of whole externally-unique object clusters. If there are no external references besides the `isolated` reference, then the whole object graph (up to immutable objects) can be converted at once. An `isolated` reference (and object graph) can trivially be converted to `writable`, by essentially surrendering the aliasing information:

[1] We use || for structured parallelism, and the formal system does not have dynamic thread creation.

```
isolated IntList l = ...;
// implicitly update l's permission to writable
l.head = ...;
```

Or an `isolated` graph can be converted to `immutable`; as with any form of strong update, the decision to treat the whole object graph as immutable is localized:

```
isolated IntList l = ...;
// implicitly update l's permission to immutable
immutable IntList l2 = l;
l.head = ...; // Type Error!
```

The type system is flow sensitive, so although `l` was initially `isolated` after the assignment to `l2` it has been coerced to `immutable` and thus cannot be written to.

2.2 Recovering Isolation

A key insight of our approach is that converting an `isolated` reference to `writable` does not require *permanently* surrendering the aliasing information. In particular, if the input type context for an expression contains only isolated and immutable objects, then if the output context contains a single `writable` reference, we can convert that reference *back to* *`isolated`*. Consider the following method:

```
isolated IntBox increment(isolated IntBox b) {
  // implicitly convert b to writable
  b.value++;
  // convert b *back* to isolated
  return b;
}
```

The first conversion from `isolated` to `writable` occurs naturally by losing aliasing information. The second conversion is safe because if one `writable` reference is left when the initial context contained only `isolated` and `immutable` references, that reference must either refer to an object that was not referenced from elsewhere on entry, or was freshly allocated (our core language and prototype do not allow mutable global variables).

This flexibility is especially useful for algorithms that repeatedly map destructive operations over data in parallel. By keeping data elements as `isolated`, the map operations naturally parallelize, but each task thread can internally violate uniqueness, apply the updates, and recover an `isolated` reference for the spawning context for later parallelization (Section 2.5).

Recovering isolation is reminiscent of borrowing — allowing temporary aliases of a unique reference, often in a scope-delimited region of program text. The main advantage of recovery is that unlike all borrowing designs we are aware of, recovery requires no tracking or invalidation of specific references or capabilities as in other work [10, 23]. Of course this is a result of adding reference immutability, so recovery is not a stand-alone replacement for traditional borrowing; it is an additional benefit of reference immutability.

We also see two slight advantages to our recovery approach. First, a single use of recovery may subsume multiple uses of a scoped approach to borrowing [30], where external uniqueness is preserved by permitting access to only the interior of a particular aggregate within a lexically scoped region of code. Of course, scopeless approaches to borrowing exist with more complex tracking [10, 23]. Second, no special source construct is necessary beyond the reference immutability qualifiers already present for parallelism.

2.3 Recovering Immutability, and Cycles of Immutable Objects

Another advantage of using `isolated` references is that the decision to make data immutable can be deferred (arbitrarily). This makes constructing cycles of immutable objects easy and natural to support. The mechanism for converting an `isolated` reference to `immutable` is similar to recovering isolation, with the natural direct conversion being a special case. If the input context when checking an expression contains only isolated and immutable references, and the output context contains one readable reference (or in general, multiple readable references), then the readable referent must be either an already-immutable object or an object not aliased elsewhere that it is safe to now call immutable. The simplest case of this (equivalent to direct conversion) is to frame away all references but one, convert to `readable`, and then recover immutability:

```
immutable IntBox freeze(isolated IntBox b) {
  // implicitly convert b to readable
  // implicitly recover immutability;
  // the input context was all isolated
  return b;
}
```

Creating cycles of immutable objects is then simply a matter of restricting the input to a conversion to only `isolated` and `immutable` data, then recovering. This can even include recovering immutability from regular code:

```
// The default permission is 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;
}
...
immutable l = make2NodeList();
```

Here the method has no inputs and it returns a writable value, so at the call site anything it returns can be considered `readable`, then recovered to `immutable` (or directly recovered to `isolated`).

Prior reference immutability systems [39] required building immutable cyclic data structures in the constructor of one object, using extensions to pass a partially-initialized object during construction as (effectively) immutable to the constructor of another object. Our use of `isolated` with recovery means we do not need to explicitly model the initialization period of immutable structures.

While we have been using closed static method definitions to illustrate the recovery rules, our system includes a frame rule [29, 33], so these conversions may occur in localized sections of code in a larger context.

2.4 Safe Symmetric Parallelism

Fork-join concurrency is deterministic when neither forked thread interferes with the other by writing to shared memory. Intuitively, proving its safe use requires separating read and write effects, as in Deterministic Parallel Java (DPJ) [4]. With reference immutability, a simpler approach is available that does not require explicit region management, allowing much of the same expressiveness with simpler annotation (see Section 7).

If neither forked thread requires any `writable` reference inputs to type check, then it is safe to parallelize, even if the threads share a `readable` reference to an object that may be mutated later, and even if threads receive `isolated` references.

```
x = new Integer(); x.val = 3; y = x; z = x;
// y and z are readable aliases of x
a = new Integer(); b = new Integer();
// a and b are isolated
// frame away writable references (x)
a.val = y.val; || b.val = z.val;
// get back writable references (x)
x.val = 4;
```

After joining, `x` may be "unframed" and the code regains `writable` access to it. Safety for this style of parallelism is a natural result of reference immutability, but proving it sound (race free) requires careful handling of coexisting `writable` references to the temporarily-shared objects.

We require that each thread in the parallel composition receives disjoint portions of the stack, though richer treatments of variable sharing across threads exist [31, 32].

2.5 Safe Asymmetric Parallelism

C# has an `async` construct that may execute a block of code asynchronously via an interleaving state machine or on a new thread [3], and returns a handle for the block's result in the style of promises or futures. A common use case is asynchronously computing on separated state while the main computation continues. Our formal system models the asymmetric data sharing of this style of use on top of structured parallelism. The formal system (Section 3) does not model the first-class join; in future work we intend to extend this rule to properly isolate `async` expressions.

A natural use for this style of parallelism is to have the asynchronous block process a limited data set in parallel with a "main" thread's execution. One definition of "limited" is to restrict the "worker" thread to isolated and immutable data, allowing the "main" thread to proceed in parallel while retaining `writable` references it may have.

```
writable Integer x = ...;
// construct isolated list of isolated integers
y = new IsolatedIntegerList();
... // populate list
f = new SortFunc();
// Sort in parallel with other work
y.map(f); || x.val = 3;
```

This code also demonstrates the flexibility of combining the rules for recovering isolated or immutable references with parallelism. In the left thread, `f` and `y` are both isolated on entry, and the rule for recovering an isolated reference can be applied to `y` at that thread's finish. Thus, when the threads join, `y` is again isolated, and suitable for further parallelization or full or partial conversion to immutable.

3. Types for Reference Immutability and Parallelism

We describe a simple core imperative, object-oriented language in Figure 3. Commands (statements) include standard field and variable assignments and reads, sequencing, loops, non-deterministic choice (to model conditional statements) and fork-join style parallelism. Our language also includes a destructive read, $x = \texttt{consume}(y.f)$, which reads the field, $y.f$, stores its value in x, and then updates the field to `null`. Our types include primitive types and permission-qualified class types. We include the four permissions from Section 2: `readable`, `writable`, `isolated`, and `immutable`. This section focuses on the language without methods, which are added in Section 3.3. Polymorphism, over both class types and permissions, is described in Section 5.

One of our primary goals for this core system is to understand the design space for source languages with reference immutability and concurrency in terms of an intermediate-level target language. This approach permits understanding source-level proposals for typing higher level language features (such as closures) in terms of translation to a well-typed intermediate form (such as the function objects C# closures compile into), rather than independently reasoning about their source level behavior.

The heart of reference immutability is that a reference's permission applies transitively. Any new references acquired through a reference with a given permission cannot allow modifications that the root reference disallows. We model this through a *permission combining* relation $\rhd$, borrowing intuition and notation from universe types' "viewpoint adaptation" [14]. We define $\rhd$ and lift it to combining with types in Figure 3.

Generally speaking, this relation propagates the weakest, or least permissive, permission. Notice that there are no permission-combining rules for `isolated` receivers and non-`immutable` fields; this reflects the requirement that accessing an `isolated` object graph generally requires upcasting variables first and accessing `isolated` fields requires destructive reads. Also notice that any combination involv-

Metavariables

a	atoms
C	command (statement)
w, x, y, z	variables
t, u	types
T, U	class type
TD	class type declaration
cn	class name
p	permission
fld	field declaration
$meth$	method declaration
f, g	field names
m	method names
n, i, j	nat (indices)

Syntax

$$\begin{aligned} a ::=\; & \\ & \mid x = y \\ & \mid x.f = y \\ & \mid x = y.f \\ & \mid x = \texttt{consume}(y.f) \\ & \mid x = y.m(z_1, ..., z_n) \\ & \mid x = \texttt{new}\, t() \\ & \mid \mathsf{return}\ x \end{aligned}$$

$$\begin{array}{ll} C & ::= a \mid \texttt{skip} \mid C; C \mid C + C \mid C \| C \mid C^* \\ p & ::= \texttt{readable} \mid \texttt{writable} \mid \texttt{immutable} \mid \texttt{isolated} \\ \\ T & ::= cn \\ TD & ::= \texttt{class}\; cn\; [<: T2]\; \{field* \;\; meth* \} \\ fld & ::= t\; fn \\ meth & ::= t\; m(t_1\; x_1, ..., t_n\; x_n) p \{\; C; \mathsf{return}\; x; \;\} \\ t & ::= \texttt{int} \mid \texttt{bool} \mid p\, T \\ \Gamma & ::= \epsilon \mid \Gamma, x : t \end{array}$$

$$\begin{array}{rcl} \multicolumn{3}{l}{\rhd : \mathsf{Permission} \to \mathsf{Permission} \to \mathsf{Permission}} \\ \texttt{immutable} \rhd \_ & = & \texttt{immutable} \\ \_ \rhd \texttt{immutable} & = & \texttt{immutable} \\ \texttt{readable} \rhd \texttt{writable} & = & \texttt{readable} \\ \texttt{readable} \rhd \texttt{readable} & = & \texttt{readable} \\ \texttt{writable} \rhd \texttt{readable} & = & \texttt{readable} \\ \texttt{writable} \rhd \texttt{writable} & = & \texttt{writable} \end{array}$$

$$\begin{aligned} p \rhd \texttt{int} &= \texttt{int} \\ p \rhd \texttt{bool} &= \texttt{bool} \\ p \rhd (p'\, T) &= (p \rhd p')\, T \end{aligned}$$

Figure 3. Core language syntax.

$$\boxed{\vdash p \prec p'} \quad \frac{}{\vdash p \prec p} \quad \frac{}{\vdash p \prec \texttt{readable}} \quad \frac{}{\vdash \texttt{isolated} \prec p}$$

$$\boxed{\vdash T \prec T'} \quad \frac{\texttt{class}\;\; c \;\; <: d \, \{\overline{fld}\; \overline{meth}\; \} \in P}{\vdash c \prec d}\ \textsc{S-Decl}$$

$$\boxed{\vdash t_1 \prec t_2} \quad \frac{\vdash p \prec p'}{\vdash p\, T \prec p'\, T}\ \textsc{S-Perm} \quad \frac{\vdash T \prec T'}{\vdash p\, T \prec p\, T'}\ \textsc{S-Type}$$

$$\frac{}{\vdash t \prec t}\ \textsc{S-Reflexive} \quad \frac{\vdash t_1 \prec t_2 \quad \vdash t_2 \prec t_3}{\vdash t_1 \prec t_3}\ \textsc{S-Trans}$$

$$\boxed{\vdash \Gamma \prec \Gamma'} \quad \frac{}{\epsilon \prec \epsilon}\ \textsc{S-Empty} \quad \frac{\vdash \Gamma \prec \Gamma' \quad \vdash t \prec t'}{\vdash \Gamma, x : t \prec \Gamma', x : t'}\ \textsc{S-Cons}$$

$$\frac{\vdash \Gamma \prec \Gamma'}{\vdash \Gamma, x : t \prec \Gamma'}\ \textsc{S-Drop}$$

Figure 4. Subtyping rules

ing immutable permissions produces an immutable permission; any object reachable from an immutable object is also immutable, regardless of a field's declared permission.

We use type environments Γ, and define subtyping on environments ($\vdash \Gamma \prec \Gamma$) in terms of subtyping for permissions ($\vdash p \prec p$), class types ($\vdash T \prec T$), and permission-qualified types ($\vdash t \prec t$) in Figure 4.

Figure 5 gives the core typing rules. These are mostly standard aside from the treatment of unique references. A destructive field read (T-FieldConsume) is fairly standard, and corresponds dynamically to a basic destructive read: as the command assigns `null` to the field, it is sound to return an `isolated` reference. Writes to `isolated` fields (T-FieldWrite) and method calls with unique arguments (T-Call) treat the `isolated` input references as affine resources, consumed by the operation. We use a metafunction RemIso() to drop "used" `isolated` references:

$$\begin{gathered} \mathsf{RemIso}() : \Gamma \to \Gamma \\ \mathsf{RemIso}(\Gamma) = \mathsf{filter}\ (\lambda x.\, x \neq \texttt{isolated}\ \_)\ \Gamma \end{gathered}$$

This is a slight inconvenience in the core language, but the implementation supports `consume` as a first class effectful expression. The method rule is otherwise straightforward aside from calls on `isolated` receivers (Section 3.3). We also provide structural rules to allow these rules to be used in more general contexts (last two rows of Figure 5). The definition of well-formed programs is mostly standard (shown in our technical report [20]), aside from requiring covariant method permissions for method overrides (Figure 6).

3.1 Recovery Rules

Figure 7 gives the two promotion rules from Sections 2.2 and 2.3 that are key to our system's flexibility: the rules for recovering `isolated` or `immutable` references, used for both precision and conversion. These rules restrict their input contexts to primitives, externally unique references, and immutable references. The T-RecovIso, checks the variable in the premise x must either be null, or point into a freshly-allocated or previously present (in Γ) object aggregate with no other references, and thus it is valid to consider it isolated. Similarly T-RecovImm checks sufficient properties to establish that it is safe to consider it immutable. In practice, using these relies on the frame rule (Figure 5).

Without reference immutability, such simple rules for recovery (sometimes called borrowing) would not be possible. In some sense, the information about permissions in the rules' input contexts gives us "permissions for free." We may essentially ignore particular permissions (isolation) for a block of commands, because knowledge of the input context ensures the `writable` or `readable` output in each premise is sufficiently separated to convert if necessary (taking advantage of our slight weakening of external uniqueness to admit references to shared immutable objects). Section 4.2 elaborates on the details of why we can prove this

$$\boxed{\Gamma_1 \vdash C \dashv \Gamma_2} \qquad \frac{t \neq \texttt{isolated}\ \_}{x : \_, y : t \vdash x = y \dashv y : t, x : t}\ \textsc{T-AssignVar} \qquad \frac{}{\vdash x = \texttt{new}\ T() \dashv x : \texttt{isolated}\ T}\ \textsc{T-New}$$

$$\frac{t'\ f \in T \quad p \neq \texttt{isolated} \vee t' = \texttt{immutable}\ \_ \quad t' \neq \texttt{isolated}\ \_ \vee p = \texttt{immutable}}{x : \_, y : p\ T \vdash x = y.f \dashv y : p\ T, x : p \rhd t'}\ \textsc{T-FieldRead}$$

$$\frac{t\ f \in T}{y : \texttt{writable}\ T, x : t \vdash y.f = x \dashv y : \texttt{writable}\ T, \mathsf{RemIso}(x : t)}\ \textsc{T-FieldWrite}$$

$$\frac{\texttt{isolated}\ T_f\ f \in T}{y : \texttt{writable}\ T \vdash x = \texttt{consume}(y.f) \dashv y : \texttt{writable}\ T, x : \texttt{isolated}\ T_f}\ \textsc{T-FieldConsume}$$

$$\frac{}{x : \_ \vdash x = n \dashv x : \texttt{int}}\ \textsc{T-Int} \quad \frac{}{x : \_ \vdash x = b \dashv x : \texttt{bool}}\ \textsc{T-Bool} \quad \frac{}{x : \_ \vdash x = \mathsf{null} \dashv x : p\ T}\ \textsc{T-Null}$$

$$\frac{\begin{array}{c} t'\ m(\overline{u'\ z'})\ p' \in T \qquad \vdash p \prec p' \qquad \overline{\vdash u \prec u'} \\ p = \texttt{isolated} \implies t \neq \texttt{readable}\ \_ \wedge t \neq \texttt{writable}\ \_ \wedge \mathsf{IsoOrImm}(\overline{z : t}) \wedge p' \neq \texttt{immutable} \end{array}}{y : p\ T, \overline{z : u} \vdash x = y.m(\overline{z}) \dashv y : p\ T, \mathsf{RemIso}(\overline{z : t}), x : t'}\ \textsc{T-Call}$$

$$\frac{\Gamma_1 \prec \Gamma_1' \quad \Gamma_1' \vdash C \dashv \Gamma_2' \quad \Gamma_2' \prec \Gamma_2}{\Gamma_1 \vdash C \dashv \Gamma_2}\ \textsc{T-SubEnv} \quad \frac{\Gamma_1 \vdash C \dashv \Gamma_2}{\Gamma, \Gamma_1 \vdash C \dashv \Gamma, \Gamma_2}\ \textsc{T-Frame} \quad \frac{\Gamma \vdash C \dashv \Gamma}{\Gamma \vdash C* \dashv \Gamma}\ \textsc{T-Loop}$$

$$\frac{\Gamma_1 \vdash C_1 \dashv \Gamma_2 \quad \Gamma_2 \vdash C_2 \dashv \Gamma_3}{\Gamma_1 \vdash C_1; C_2 \dashv \Gamma_3}\ \textsc{T-Seq} \quad \frac{\Gamma_1 \vdash C_1 \dashv \Gamma_2 \quad \Gamma_1 \vdash C_2 \dashv \Gamma_2}{\Gamma_1 \vdash C_1 + C_2 \dashv \Gamma_2}\ \textsc{T-Branch} \quad \frac{\Gamma, y : t', x : t, \Gamma' \vdash C \dashv \Gamma''}{\Gamma, x : t, y : t', \Gamma' \vdash C \dashv \Gamma''}\ \textsc{T-Shuffle}$$

Figure 5. Core typing rules.

$$\frac{\begin{array}{l} TD = \texttt{class}\ cn\ [<: T2]\ \{\overline{fld}\ \overline{meth}\ \} \qquad t'\ m(\overline{t'}\ \overline{x'})\ p' \in T2 \qquad P \vdash t \prec t' \qquad P \vdash p' \prec p \qquad P \vdash \overline{t'} \prec \overline{t} \\ p \neq \texttt{isolated} \qquad \forall i \in [1 \ldots n].\ P \vdash t_i \qquad P \vdash t \qquad \texttt{this} : p\ cn, \overline{t}\ \overline{x} \vdash C; \mathsf{return}\ x \dashv \texttt{result} : t \end{array}}{P; TD \vdash t\ m(\overline{t}\ \overline{x})\ p\ \{\ C; \mathsf{return}\ x;\ \}}\ \textsc{T-Method2}$$

Figure 6. Method override definition typing

$$\frac{\begin{array}{c} \mathsf{IsoOrImm}(\Gamma) \quad \mathsf{IsoOrImm}(\Gamma') \\ \Gamma \vdash C \dashv \Gamma', x : \texttt{writable}\ T \end{array}}{\Gamma \vdash C \dashv \Gamma', x : \texttt{isolated}\ T}\ \textsc{T-RecovIso}$$

$$\frac{\begin{array}{c} \mathsf{IsoOrImm}(\Gamma) \quad \mathsf{IsoOrImm}(\Gamma') \\ \Gamma \vdash C \dashv \Gamma', x : \texttt{readable}\ T \end{array}}{\Gamma \vdash C \dashv \Gamma', x : \texttt{immutable}\ T}\ \textsc{T-RecovImm}$$

where $\mathsf{IsoOrImm}(\Gamma) \overset{\text{def}}{=} \forall (x : p\,T) \in \Gamma.\ \vdash p \prec \texttt{immutable}$

Figure 7. Recovery rules

$$\frac{\mathsf{NoWrit}(\Gamma_1) \quad \mathsf{NoWrit}(\Gamma_2) \quad \Gamma_1 \vdash C_1 \dashv \Gamma_1' \quad \Gamma_2 \vdash C_2 \dashv \Gamma_2'}{\Gamma_1, \Gamma_2 \vdash C_1 || C_2 \dashv \Gamma_1', \Gamma_2'}\ \textsc{T-Par}$$

$$\frac{\mathsf{IsoOrImm}(\Gamma_1) \quad \Gamma_1 \vdash C_1 \dashv \Gamma_1' \quad \Gamma_2 \vdash C_2 \dashv \Gamma_2'}{\Gamma_1, \Gamma_2 \vdash C_1 || C_2 \dashv \Gamma_1', \Gamma_2'}\ \textsc{T-Async}$$

where $\mathsf{NoWrit}(\Gamma) \overset{\text{def}}{=} \forall (x : p\,T) \in \Gamma.\, p \neq \texttt{writable}$

Figure 8. Type rules for safe parallelism. IsoOrImm is defined in Figure 7

is sound. Additionally, the permission qualifications specify which references may safely interact with an externally-unique aggregate, and which must be prevented from interacting via the frame rule (`readable` and `writable` references). This distinction normally requires precise reasoning about aliases.

3.2 Safe Parallelism

Figure 8 gives the rules for safe parallelism. They ensure data race freedom, and therefore (for the concurrency primitives we provide) deterministic execution. T-PAR corresponds to safe symmetric parallelism, when all `writable` references are framed out. The second rule T-ASYNC corresponds to the safety criteria for asymmetric parallelism (named for C#'s async block). This rule obviously produces structured parallelism, not the unstructured task-based concurrency present in C#. But it models the state separation required for safe task parallelism: all input to a task must be isolated or immutable. The implementation provides safe task parallelism of this form, as described in Section 6.1, as well as structured parallelism.

3.3 Methods

The type rule for a method call (T-CALL) is shown in Figure 5. It is mostly standard (the method exists in the receiver type, actual arguments are subtypes of formal arguments), with a couple of complications. First, `isolated` actual arguments are forgotten by the typing context, in lieu of extending the method syntax for destructive reads.

Second, methods have required calling permissions, which restrict the side effects a method may have on the receiver. The permission on the receiver at the call site must be at least as permissive as the required permission (e.g., a program cannot call a `writable` method on a `readable` receiver). This is standard for reference immutability [37, 39, 40].

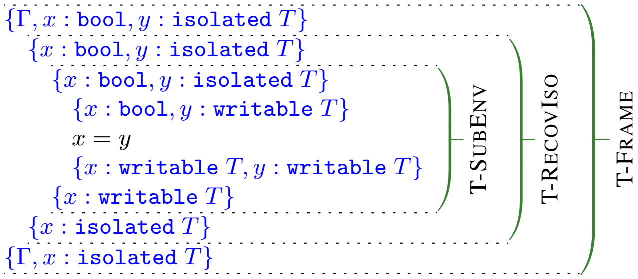
Finally, additional restrictions apply when the receiver is isolated. Intuitively, no isolated method may return an alias to an object inside its isolation bubble; alternatively, the restrictions ensure that an inlined method body is suitable for upcasting the `isolated` receiver's permission, executing, and finally applying `T-RecovIso`. Because no method is type checked with `this : isolated` T (by T-METHOD* in Figure 6 and our technical report [20], no method may require an `isolated` receiver permission), no method may leverage recovery rules (Figure 7) to recover an `isolated` or `immutable` reference to its receiver. Thus any methods returning primitives (`int` and `bool`) or `isolated` or `immutable` references is returning a value that does not violate external uniqueness for the receiver's bubble.

3.4 Examples

For brevity, because the type environment is flow sensitive, we write typing derivations in the same style as a proof in a program logic, with pre- and post-conditions of a statement in braces before and afterwards. Unmarked adjacent assertions represent use of the rule of implication (subtyping). Uses of other transformation rules are labeled. In Section 4, it will become clear how this style directly models the corresponding proof of type checking in the program logic.

3.4.1 Assigning an Isolated Variable

Assigning an `isolated` variable consists of framing away outer context, upcasting the `isolated` reference to `writable`, assigning normally, weakening to drop the source variable, and an application of T-RECOVISO to recover the isolation property on the destination variable. It is possible to add an admissible rule for the direct consumption. It is also possible to preserve access to the source variable by also overwriting it with a primitive value such as null, which is equivalent to an encoding of a traditional destructive read on a variable.



$\{\Gamma, x : \texttt{bool}, y : \texttt{isolated}\ T\}$
$\{x : \texttt{bool}, y : \texttt{isolated}\ T\}$
$\{x : \texttt{bool}, y : \texttt{isolated}\ T\}$
$\{x : \texttt{bool}, y : \texttt{writable}\ T\}$
$x = y$
$\{x : \texttt{writable}\ T, y : \texttt{writable}\ T\}$
$\{x : \texttt{writable}\ T\}$
$\{x : \texttt{isolated}\ T\}$
$\{\Gamma, x : \texttt{isolated}\ T\}$
T-SUBENV
T-RECOVISO
T-FRAME

3.4.2 Temporarily Violating Isolation

Figure 9 shows the type derivation for a simple use of the T-RECOVISO rule, adding a node to an isolated list. The inner portion of the derivation is not notable, simply natural use of sequencing, allocation, and field write rules. But that inner portion is wrapped by a use of subtyping, followed by recovering an isolated reference. Using T-RECOVIMM to recover an immutable reference would be similar, using a `readable` reference to the list after the updates.

4. Type Soundness

To prove soundness, we must define the dynamic language semantics and relate the typing rules. The dynamic semantics for commands C are standard small step operational semantics over the states we define below, so we omit them here. The operational rule for reducing an atom a appeals to a denotational semantics of atoms, which is defined in an entirely standard way and therefore also omitted (method calls have some subtlety, but conform to standard intuition of evaluating method calls by inlining method bodies, fully detailed in our technical report [20]). We relate the type rules to the semantics by defining a denotation of type environments in terms of an extended machine state.

We define abstract machine states as:

$$\mathcal{S} \overset{\text{def}}{=} \mathsf{Stack} \times \mathsf{Heap} \times \mathsf{TypeMap}$$

where $\mathsf{Stack} \overset{\text{def}}{=} \mathsf{Var} \rightharpoonup \mathsf{Val}$ and is ranged over by s, $\mathsf{Heap} \overset{\text{def}}{=} \mathsf{OID} \times \mathsf{Field} \rightharpoonup \mathsf{Val}$ and is ranged over by h, and $\mathsf{TypeMap} \overset{\text{def}}{=} \mathsf{OID} \rightharpoonup \mathsf{Class}$ and is ranged over by t.

We only consider well-typed states. To define well-typed states, we assume a function that gives the type and permission for each field of each class, reflects inheritance of fields, and in Section 5 handles instantiating field types of polymorphic types:

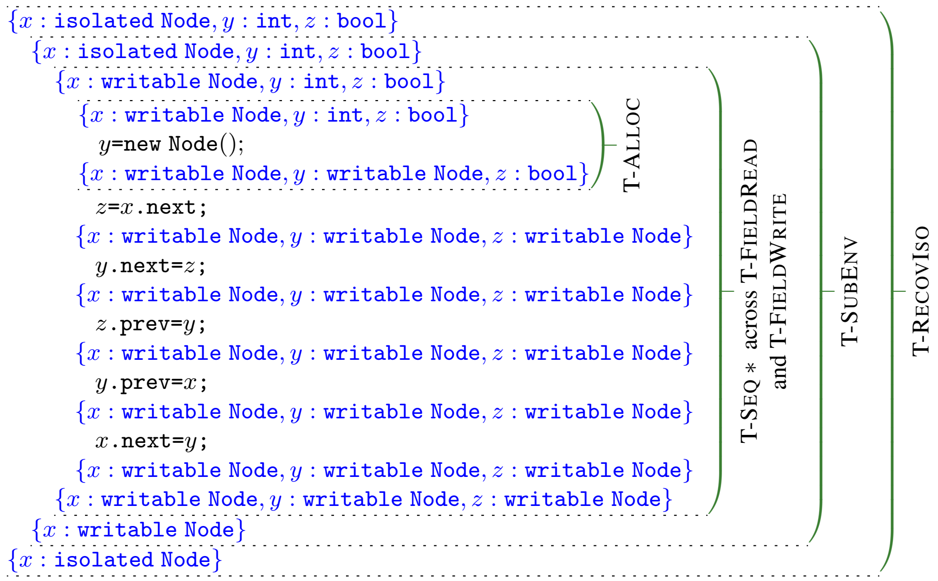
$$\mathsf{FType} : \mathsf{Class} \times \mathsf{Field} \to \mathsf{Type}$$

We can describe a state (s, h, t) as well-typed iff

$$\begin{aligned} &\mathsf{WellTyped}(s,h,t) = \forall o, f.\\ &\quad (\exists v.\, h(o,f) = v)\\ &\qquad\quad \iff (\exists ft.\, \mathsf{FType}(t(o), f) = ft)\\ &\wedge \forall ft.\, \mathsf{FType}(t(o), f) = ft \implies\\ &\quad \begin{pmatrix} ft = p\ c' \wedge h(o,f) \neq \mathsf{null} \implies \vdash t(h(o,f)) \prec c'\\ \wedge\ ft = \mathtt{bool} \implies \exists b.\, h(o,f) = b\\ \wedge\ ft = \mathtt{int} \implies \exists n.\, h(o,f) = n \end{pmatrix} \end{aligned}$$

The first conjunct requires that the type map contains a type for every object in the heap, and vice-versa; it limits the type map to the heap contents. The second conjunct simply enforces that each field holds well-typed contents.

Reasoning about which objects are immutable and the permissions of various references is somewhat difficult for such a basic state space, so we define an *instrumented state* with additional metadata: a partitioning of objects among



```
{x : isolated Node, y : int, z : bool}
  {x : isolated Node, y : int, z : bool}
    {x : writable Node, y : int, z : bool}
      {x : writable Node, y : int, z : bool}
        y=new Node();
      {x : writable Node, y : writable Node, z : bool}
        z=x.next;
      {x : writable Node, y : writable Node, z : writable Node}
        y.next=z;
      {x : writable Node, y : writable Node, z : writable Node}
        z.prev=y;
      {x : writable Node, y : writable Node, z : writable Node}
        y.prev=x;
      {x : writable Node, y : writable Node, z : writable Node}
        x.next=y;
      {x : writable Node, y : writable Node, z : writable Node}
    {x : writable Node, y : writable Node, z : writable Node}
  {x : writable Node}
{x : isolated Node}
```

Figure 9. Typing derivation for adding a node to an isolated doubly-linked list.

regions, and permission to each region (important for safe parallelism).

We map each object to a region

$$r : \mathsf{RegionMap} = \mathsf{OID} \rightharpoonup \mathsf{Region}$$

We have three forms of region:

- $\mathsf{Root}(\rho)$ is a root region with abstract root ρ
- $\mathsf{Field}(o, f)$ means the region is only accessible through the isolated field f of object o.
- $\mathsf{Immutable}$ means immutable

We associate two permissions with each root region:

$$\pi : \mathsf{RegionPerm} = \mathsf{Root} \rightharpoonup \mathsf{Update}[0,1] \times \mathsf{Reference}(0,1]$$

where

- Update: Is used to indicate if objects in the region can be modified. Full (1) means this is the case. An update permission will be split for the period of a parallel composition, as a fractional permission [9].
- Reference: Is used to indicate whether there is a framed-out reference to this region (< 1). This prevents the conversion of a region to isolated or immutable when there are framed-out `readable` or `writable` references to it. Note that 0 reference permission is not allowed; states with no permission at all to a region do not have that permission in their permission map.

For both permissions, the total permission available to the program for any given region is 1. These two permission types capture two interference concepts. You can interfere with yourself; and you cannot interfere with other threads. Interference with other threads is prevented by the update permission, only one thread can ever have an update permission to a region.

These states also satisfy a well-formedness predicates. We require instrumented states to be *well-regioned*: e.g. an `immutable` reference points to an object in region $\mathsf{Immutable}$, no `readable` or `writable` reference outside a given externally unique aggregate points to an object in an isolated region, etc. We define well-regioned given in Figure 10. The first conjunct ensures full region information for the heap's objects. The second, largest conjunct enforces restrictions on references between regions. Intra-region pointers must be either within the $\mathsf{Immutable}$ region, or following `readable` or `writable` fields. Cross-region pointers must not be pointing out of $\mathsf{Immutable}$ (which is closed under field dereference), and must either pointing into $\mathsf{Immutable}$ from fields with appropriate permissions, an `isolated` field pointing into an appropriate Field region, or a `readable` reference between root regions. The next conjunct requires permissions on any root region, and the final conjunct limits the region map's Fields to those whose entry points are present in the heap.

We can thus define an instrumented state as:

$$\mathcal{M} = \left\{ \begin{array}{l} m \in \mathcal{S} \times \mathsf{RegionMap} \times \mathsf{RegionPerm} \\ \qquad \mid \mathsf{WellRegioned}(m) \wedge \mathsf{WellTyped}(\lfloor m \rfloor) \end{array} \right\}$$

where we define an erasure $\lfloor \cdot \rfloor : \mathcal{M} \rightarrow \mathcal{S}$ that projects instrumented states to the common components with $\mathcal{S}$. We use $m.s$, $m.h$, $m.t$, $m.r$, and $m.\pi$ to access the stack, heap, type map, region map and region permission map, respectively.

We view type environments denotationally, in terms of the set of instrumented states permitted by a given envi-

$$
\begin{array}{l}
\mathsf{WellRegioned}(s,h,t,r,\pi) = \\
\quad \mathsf{CompleteRegionInfo}(s,h,t,r,\pi) \wedge \\
\quad \left(\begin{array}{l}
\forall o,f,v,p.\, h(o,f) = v \wedge v \in \mathit{OID} \wedge \mathsf{FType}(t(o),f) = p\ \_ \\
\implies \\
\left(\begin{array}{l}
(r(o) = r(v) \implies \\
\qquad r(o) = \mathsf{Immutable} \vee p \in \{\texttt{readable}, \texttt{writable}\}) \\
\wedge\, (r(o) \neq r(v) \implies \mathsf{ValidXRegionRef}(r,o,f,p,v))
\end{array} \right)
\end{array} \right) \\
\wedge \quad (\forall \rho.\, \mathsf{Root}\ \rho \in \mathit{Img}(r) \implies \pi(\rho)(\geq,>)(0,0)) \\
\wedge \quad (\forall o,f.\, \mathsf{Field}(o,f) \in \mathit{Img}(r) \implies (o,f) \in \mathit{dom}(h))
\end{array}
$$

where

$$
\begin{array}{l}
\mathsf{ValidXRegionRef}(r,o,f,p,v) = \\
\quad \left(\begin{array}{l}
(r(o) \neq \mathsf{Immutable}) \\
\wedge\, (r(v) = \mathsf{Immutable} \implies p \in \{\texttt{immutable}, \texttt{readable}\}) \\
\wedge\, (r(v) = \mathsf{Field}(o,f) \implies p = \texttt{isolated}) \\
\wedge\, (r(v) = \mathsf{Root}(\_) \implies p = \texttt{readable} \wedge r(o) = \mathsf{Root}(\_))
\end{array} \right)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{CompleteRegionInfo}(s,h,t,r,\pi) = \\
\quad \left(\begin{array}{l}
\forall o,f.\, h(o,f) \text{ defined} \implies \\
\qquad t(o) \text{ defined} \wedge r(o) \text{ defined} \wedge \\
\qquad\quad (\forall \rho.\, r(o) = \mathsf{Root}(\rho) \implies \pi(\rho) \text{ defined})
\end{array} \right) \\
\quad \wedge (\forall o, r(o) \text{ defined} \implies \exists f, h(o,f) \text{ defined})
\end{array}
$$

Figure 10. Definition of Well-Regioned

$$
\begin{array}{l}
[\![x : \texttt{isolated}\, T]\!] = \\
\quad \left\{ m \in \mathcal{M} \;\middle|\; \begin{array}{l}
m.\pi(m.r(m.s(x))) = (1,1) \\
\wedge m.t(m.s(x)) = T' \wedge \vdash T' \prec T \\
\wedge \mathsf{RootClosed}(m.r(m.s(x)), m) \\
\vee m.s(x) = \mathsf{null}
\end{array} \right\} \\
[\![x : \texttt{immutable}\, T]\!] = \\
\quad \left\{ m \in \mathcal{M} \;\middle|\; \begin{array}{l}
m.r(m.s(x)) = \mathsf{Immutable} \\
\wedge m.t(m.s(x)) = T' \wedge \vdash T' \prec T \\
\vee m.s(x) = \mathsf{null}
\end{array} \right\} \\
[\![x : \texttt{readable}\, T]\!]_\pi = \\
\quad \left\{ m \in \mathcal{M} \;\middle|\; \begin{array}{l}
m.s(x) = \mathsf{null} \vee \\
((\exists \rho.\, \mathsf{Up}(\pi(\rho)) > 0 \wedge \mathsf{Up}(m.\pi(\rho)) > 0 \\
\qquad \wedge\, m.r(m.s(x)) = \mathsf{Root}\ \rho) \\
\vee m.r(m.s(x)) = \mathsf{Immutable}) \\
\wedge m.t(m.s(x)) = T' \wedge \vdash T' \prec T
\end{array} \right\} \\
[\![x : \texttt{writable}\, T]\!]_\pi = \\
\quad \left\{ m \in \mathcal{M} \;\middle|\; \begin{array}{l}
m.s(x) = \mathsf{null} \vee \\
(\exists \rho.\, \pi(\rho) = (1,\_) \wedge m.\pi(\rho) = (1,\_) \\
\wedge m.r(m.s(x)) = \mathsf{Root}\ \rho) \\
\wedge m.t(m.s(x)) = T' \wedge \vdash T' \prec T
\end{array} \right\} \\
[\![\Gamma, x : \texttt{isolated}\, T]\!]_\pi = [\![x : \texttt{isolated}\, T]\!] * [\![\Gamma]\!]_\pi \\
[\![\Gamma, x : \texttt{immutable}\, T]\!]_\pi = [\![x : \texttt{immutable}\, T]\!] * [\![\Gamma]\!]_\pi \\
[\![\Gamma, x : \texttt{readable}\, T]\!]_\pi = [\![x : \texttt{readable}\, T]\!]_\pi \cap [\![\Gamma]\!]_\pi \\
[\![\Gamma, x : \texttt{writable}\, T]\!]_\pi = [\![x : \texttt{writable}\, T]\!]_\pi \cap [\![\Gamma]\!]_\pi \\
[\![\epsilon]\!]_\pi = \\
\quad \left\{ m \in \mathcal{M} \;\middle|\; \begin{array}{l}
m.\pi \geq \pi \\
\wedge\, (\forall \rho \in \mathit{dom}(\pi).\, \mathsf{Up}(\pi(\rho)) > 0) \\
\wedge\, \forall o,f,o'.\, m.r(o) \in \mathit{dom}(\pi) \wedge m.h(o,f) = o' \\
\qquad \implies \left(\begin{array}{l}
m.r(o') \in \mathit{dom}(\pi) \vee \\
m.r(o') = \mathsf{Immutable} \vee \\
m.r(o') = \mathsf{Field}(o,f)
\end{array} \right)
\end{array} \right\}
\end{array}
$$

where $\mathsf{RootClosed}(\rho, m) \overset{\text{def}}{=}$

$$
\left(\begin{array}{l}
\forall o,f,o'.\, m.r(o) = \mathsf{Root}(\rho) \wedge m.h(o,f) = o' \implies \\
\qquad (m.r(o') = m.r(o) \vee m.r(o') = \mathsf{Immutable} \vee \\
\qquad\quad m.r(o') = \mathsf{Field}(o,f))
\end{array} \right).
$$

Figure 11. Denoting types and type environments.

ronment. Figure 11 defines the type environment denotation $[\![\Gamma]\!]_\pi$. Isolated and immutable denotations are mostly straightforward, though they rely on a notion of partial separating conjunction $*$ of instrumented states. To define this separation, we must first define composition of instrumented states $\bullet$:

$$\bullet = (\bullet_{\rightharpoonup}, \bullet_{\cup}, \bullet_{\cup}, \bullet_{\cup}, \bullet_{\pi})$$

where

$$
\begin{array}{l}
s \in (s_1 \bullet_{\rightharpoonup} s_2) \overset{\text{def}}{=} \mathit{dom}(s_1) \cap \mathit{dom}(s_2) = \emptyset \wedge s = s_1 \cup s_2 \\
x \in (x_1 \bullet_{\cup} x_2) \overset{\text{def}}{=} x = x_1 \cup x_2 \\
\pi \in (\pi_1 \bullet_{\pi} \pi_2) \overset{\text{def}}{=} \forall \rho.\ \pi(\rho) = \pi_1(\rho)(+,+)\pi_2(\rho)
\end{array}
$$

Partial separating conjunction then simply requires the existence of two states that compose:

$$m \in P * Q \overset{\text{def}}{=} \exists m'.\, \exists m''.\, m' \in P \wedge m'' \in Q \wedge m \in m' \bullet m''$$

This partial separation makes denotation of immutable or isolated references mostly independent of other state. For example, an `isolated` reference in the environment must be the only reference to some root region, and it must be possible to split that full permission away from the state described by the rest of the environment without invaliding other parts of the context. We cannot define the meaning of `readable` and `writable` individually, because we need an externally visible bound on the regions involved in denoting a `readable` or `writable` reference when proving conversions (T-RECOVISO and T-RECOVIMM) sound. We give the meaning of a typing context with respect to some local permission map π, which the denotations of `readable` and `writable` references refer to, in addition to checking permissions in the concrete state. Because this π bounds the set of regions supporting an environment, when π contains only full permissions we can prove that certain region-changing operations will not interfere with other threads. It also enables proving parallel composition is race free, as our proof of safe composition gives full update permission on a shared region to neither thread, meaning neither thread may denote a writable reference to a shared object (as in T-PAR).

Section 4.1 briefly describes specifics of how we interact with an existing program logic [15] to prove soundness. Even without reading Section 4.1, the actual soundness proof in Section 4.2 should be understandable enough to build an intuition for soundness with only intuition for $*$. The proofs are based around a relation $\sqsubseteq$, which can be viewed as saying what changes to the π and r components of instrumented states are allowed, such that other threads can preserve their view of the typing of the state.

4.1 Views Framework

Our soundness proof builds upon a version of the Views Framework [15], which is in some sense a generalization of the ideas behind separation logic and rely-guarantee reasoning. The definitions we gave in the previous section, $\mathcal{S}$, $\mathcal{M}$ and $\bullet$, happen to coincide with definitions required by this framework. Given a few operations and relations over $\mathcal{M}$, the framework gives a natural structure to the soundness proof as an embedding of type derivations into the view's program logic. To do this we must define:

- An operation $\bullet : \mathcal{M} \to \mathcal{M} \to \mathcal{M}$ that is commutative and associative.
- A preorder interference relation $\mathcal{R} \subseteq \mathcal{M} \times \mathcal{M}$ that defines permissible interference on an instrumented state. The relation must distribute over composition:

$$\begin{aligned}&\forall m_1, m_2, m.\, (m_1 \bullet m_2)\mathcal{R}m \implies\\ &\qquad \exists m_1', m_2'.\, m_1\mathcal{R}m_1' \wedge m_2\mathcal{R}m_2' \wedge m \in m_1' \bullet m_2'\end{aligned}$$

- A (left and right) unit to $\bullet$ that is closed wrt to $\mathcal{R}$ (in our case, an instrumented state where all the components are empty maps).
- A denotation of static assertions (in our case, types) in terms of instrumented states: $[\![\Gamma]\!]_\pi$ as in Figure 11.

Soundness follows from proving that the denotation of a typing derivation ($[\![\Gamma \vdash C \dashv \Gamma]\!]$) respects some lifting of the operational semantics to instrumented states, by embedding the typing derivations into a program logic. The advantage of choosing this approach over a more traditional technique like syntactic type soundness is that after proving a few lemmas about how type environment denotations behave with respect to composition and interference, a number of typically distinct concepts (including forking and joining threads, frame rules, and safety of environment reordering) become straightforward applications of simpler lemmas.

$$\begin{aligned}
&(s,h,t,r,\pi)\mathcal{R}_0(s',h',t',r',\pi') \overset{\text{def}}{=}\\
&\quad s = s' \wedge \pi = \pi'\\
&\wedge(\forall o, f.\, h(o,f) \neq h'(o,f) \implies \pi(r(o)) = (0, \_))\\
&\wedge \left(\begin{array}{l}
\text{let } O = \mathit{dom}(t) \text{ in}\\
\text{let } O' = \mathit{dom}(t') \text{ in}\\
\quad (\forall o.\, o \in O \cap O' \implies\\
\qquad (r(o) \neq r'(o) \implies\\
\qquad\qquad \pi(r(o)) = (0, \_) \wedge \pi(r'(o)) = (0, \_)))\\
\qquad \wedge\, t(o) = t'(o)\\
\ \wedge\, (\forall o.\, o \in O \setminus O' \implies r(o) = \mathsf{Field}(\_, \_))\\
\ \wedge\, (\forall o.\, o \in O' \setminus O \wedge r'(o) \in \mathit{dom}(\pi')\\
\qquad \implies \pi'(r'(o)) = (0, \_))
\end{array}\right)
\end{aligned}$$

Figure 12. The thread interference relation $\mathcal{R}_0$.

We define the interference permitted by a single action of another thread (heap modifications) $\mathcal{R}_0$ in Figure 4.1. The interference relation for individual actions allows relatively little to change. The stack and region permissions must remain constant. The types and region map must remain constant, aside from objects initially in Field regions disappearing (as in another view performing a destructive read), new objects appearing in any region the current view has no update permission for, and moving objects between root regions with no update permission (intuitively, another view merging two root region contents). The heap has similar restrictions, though it additionally permits field changes in root regions with 0 update permissions. Note that WellTyped and WellRegioned constrain the domains of the region and type maps, and the object-only projection of the heap domain, to all be equal. So if an object appears in the region map, it appears in the type map and heap as well, and so on. We define the final interference relation $\mathcal{R}$ as the reflexive transitive closure of $\mathcal{R}_0$.

Given the specific definitions of composition and interference, the Views Framework defines a number of useful concepts to help structure and simplify the soundness proof. First, it defines a *view* as the subset of instrumented states that are *stable* under interference:

$$\mathsf{View} \overset{\text{def}}{=} \{M \in \mathcal{P}(\mathcal{M}) \mid \mathcal{R}(M) \subseteq M\}$$

The program logic is proven sound with respect to views [15], and our denotation of type environments is a valid view (stable with respect to $\mathcal{R}$). The framework also describes a useful concept called the *view shift* operator $\sqsubseteq$, that describes a way to reinterpret a set of instrumented states as a new set of instrumented states with the same erasures to $\mathcal{S}$, accounting for any requirement of other views. It requires that:

$$p \sqsubseteq q \overset{def}{\iff} \forall m \in \mathcal{M}.\, \lfloor p * \{m\} \rfloor \subseteq \lfloor q * \mathcal{R}(\{m\}) \rfloor$$

This specifies how the region information can be changed soundly. That is, we can only change the region information such that all other possible threads can maintain compatible views. This corresponds to precisely what subtyping must satisfy in a concurrent setting and underlies the majority of encoding the complex typing rules into the Views Framework.

4.2 Soundness Proof

As mentioned earlier, soundness of a type system in the Views Framework proceeds by embedding the types' denotation into a sound program logic [15]. The logic itself contains judgments of the form $\{p\}C\{q\}$ for views p and q and commands C, and the logic's soundness criteria, subject to our definitions of composition, interference, etc. satisfying the required properties, is

Theorem 1 (Views Logic Soundness [15])**.** *If* $\{p\}C\{q\}$ *is derived in the logic, then for all* $s \in \lfloor p \rfloor$*, and* $s \in \mathcal{S}$*, if* $(C, s) \longrightarrow^* (\mathit{skip}, s')$ *then* $s' \in \lfloor q \rfloor$*.*

Thus because our definitions of $\mathcal{S}$, $\mathcal{M}$, $\bullet$ and $\mathcal{R}$ satisfy the required properties, if every type derivation denotes a valid derivation in the Views Framework's logic, then the type system is sound. We can define the denotation of a typing judgment as:

$$\begin{aligned}&\llbracket \Gamma_1 \vdash C \dashv \Gamma_2 \rrbracket \stackrel{\text{def}}{=} \\ &\quad \forall \pi.\, (\exists \rho.\, \pi(\rho) = (1, \_)) \Rightarrow \{\llbracket \Gamma_1 \rrbracket_\pi\} C \{\llbracket \Gamma_2 \rrbracket_\pi\}\end{aligned}$$

We map each judgement onto a collection of judgements in the Views Framework. This allows us to encode the rules for recovery, as the logic does not directly support them. Specifically, closing over π allows us to prove that permissions are preserved. Thus, if a block of code is encoded with a set of initially full permissions, it will finish with full permissions, allowing conversion back to `isolated` if necessary.

We always require there to be at least one local region that is writable: $(\exists \rho.\, \pi(\rho) = (1, \_))$. This is required to prove soundness for the subtyping rule, to allow us to cast `isolated` to `writable`.

Here we describe the major lemmas supporting the soundness proof, and omit natural but uninteresting lemmas, such as proving that the denotation of a type environment is stable under interference. We also omit methods here. When not otherwise discussed, lemmas are typically proven by induction on a type environment Γ. More details are available in our extended technical report [20]. To prove soundness for method calls, we extended the Views Framework with support for method calls. The semantics are mostly intuitive (reducing a call statement to an inlined method body with freshly bound locals), and both the semantics and proof extensions are described in detail in our technical report [20].

The most important lemmas are those for recovering isolated or immutable references, which prove the soundness of the type rules T-RecovIso and T-RecovImm:

Lemma 1 (Recovering Isolation)**.**

$$\begin{aligned}&\mathit{IsoOrImm}(\Gamma) \implies \mathit{FullPermsOnly}(\pi) \implies \\ &\quad \llbracket \Gamma, x : \mathtt{writable} \rrbracket_\pi \sqsubseteq \llbracket \Gamma, x : \mathtt{isolated} \rrbracket_\emptyset\end{aligned}$$

Lemma 2 (Recovering Immutability)**.**

$$\begin{aligned}&\mathit{IsoOrImm}(\Gamma) \implies \mathit{FullPermsOnly}(\pi) \implies \\ &\quad \llbracket \Gamma, x : \mathtt{readable} \rrbracket_\pi \sqsubseteq \llbracket \Gamma, x : \mathtt{immutable} \rrbracket_\emptyset\end{aligned}$$

Both Lemmas 1 and 2 rely on the fact that `readable` and `writable` references into root regions refer only to regions in π. Without that restriction, and the fact that the denotation of type environments separates `isolated` and `immutable` references from regions in π, recovering isolation or immutability would not be possible. Another important factor for these lemmas is our slight weakening of external uniqueness, to allow references out of an aggregate into immutable data; without this, recovering isolation would not be possible with immutable references in Γ.

Environment subtyping is given by the following lemma.

Lemma 3 (Subtyping Denotation)**.**

$$(\exists \rho.\, \pi(\rho) = (1, \_)) \ \wedge\ \vdash \Gamma_1 \prec \Gamma_2 \implies \llbracket \Gamma_1 \rrbracket_\pi \sqsubseteq \llbracket \Gamma_2 \rrbracket_\pi$$

The lemmas for framing (and unframing) parts of the type environment require defining a weakened type denotation $\llbracket \Gamma \rrbracket^\omega_\pi$, described in detail in our expanded technical report [20]. This denotation is mostly the same as the regular denotation but requires only a non-zero update permission in π, with 0 update permission in the state, but checking reference permission against a π matching the "unframed" state. This makes the environment unusable for executing commands but retaining enough information to restore the environment later. We also use a transformation function on π to frame out a reference permission, preventing the recovery rules from being applied in cases where a `readable` or `writable` reference to some region is framed out: $\mathsf{frame\_out\_perm}\ (u, r) := (u, r/2)$.

Lemma 4 (Type Framing)**.**

$$\llbracket \Gamma, \Gamma' \rrbracket_\pi \sqsubseteq \llbracket \Gamma \rrbracket^\omega_{(\mathsf{map\ frame\_out\_perm}\ \pi)} * \llbracket \Gamma' \rrbracket_{(\mathsf{map\ frame\_out\_perm}\ \pi)}$$

Lemma 5 (Type Unframing)**.**

$$\llbracket \Gamma \rrbracket^\omega_{(\mathsf{map\ frame\_out\_perm}\ \pi)} * \llbracket \Gamma' \rrbracket_{(\mathsf{map\ frame\_out\_perm}\ \pi)} \sqsubseteq \llbracket \Gamma, \Gamma' \rrbracket_\pi$$

Lemma 6 (Symmetric Decomposition)**.**

$$\begin{aligned}&\mathit{NoWrit}(\Gamma) \implies \\ &\quad \llbracket \Gamma, \Gamma' \rrbracket_\pi \sqsubseteq \llbracket \Gamma \rrbracket_{(\mathsf{map\ halve\_perm}\ \pi)} * \llbracket \Gamma' \rrbracket_{(\mathsf{map\ halve\_perm}\ \pi)}\end{aligned}$$

Lemma 7 (Join)**.** $\llbracket \Gamma \rrbracket_\pi * \llbracket \Gamma' \rrbracket_{\pi'} \sqsubseteq \llbracket \Gamma, \Gamma' \rrbracket_{\pi \bullet \pi'}$

Lemma 8 (Asymmetric Decomposition)**.**

$$\mathit{IsoOrImm}(\Gamma) \implies \llbracket \Gamma, \Gamma' \rrbracket_\pi \sqsubseteq \llbracket \Gamma \rrbracket_\emptyset * \llbracket \Gamma' \rrbracket_\pi$$

Theorem 2 (Type Soundness)**.**

$$\Gamma_1 \vdash C \dashv \Gamma_2 \implies \llbracket \Gamma_1 \vdash C \dashv \Gamma_2 \rrbracket$$

Proof. By induction on the derivation $\Gamma_1 \vdash C \dashv \Gamma_2$. □

5. Polymorphism

Any practical application of this sort of system naturally requires support for polymorphism over type qualifiers. Otherwise code must be duplicated, for example, for each possible permission of a collection and each possible permission for the objects contained within a collection. Of course, polymorphism over unique and non-unique references with mutable state still lacks a clean solution due to the presence of destructive reads (using a destructively-reading collection for non-unique elements would significantly alter semantics, though in the pure setting some clever techniques exist [26]).

To that end we also develop a variant of the system with both type and method polymorphism, over class types and permissions. As in C#, we allow a sort of dependent kinding for type parameters, allowing type and permission parameters to bound each other (without creating a cycle of bounding). We separate permission parameters from class parameters for simplicity and expressivity. A source level variant may wish to take a single sort of parameter that quantifies over a permission-qualified type as in IGJ [39]. There is a straightforward embedding from those constraints to the more primitive constraints we use, and our separation makes our language suitable as an intermediate language that can be targeted by variants of a source language that may change over time.

Figure 13 gives the grammar extensions for the polymorphic system. Our language permits bounding a polymorphic permission by any other permission, including other permission parameters, and by concrete permissions that produce parameters with only a single valid instantiation (such as `immutable`). This allows for future extensions, for example distinguishing multiple `immutable` data sources. We add a context Δ containing type bounds to the typing and subtyping judgements. We extend the previous typing and subtyping rules in the obvious way. Δ is invariant for the duration of checking a piece of code, and the main soundness theorem, restated below, relies on an executing program having an empty Δ (a program instantiates all type and permission parameters to concrete classes and permissions). Concretely, type judgements and subtyping judgements now take the forms:

$$\Delta \mid \Gamma \vdash C \dashv \Gamma \qquad \Delta \vdash t \prec t$$

Figure 14 gives auxiliary judgements and metafunctions used in the polymorphic system. Most of the extensions are straightforward, leveraging bounds in Δ to type check interactions with generic permissions and class types. We discuss a couple of the most interesting rules here, presenting the full type system in our technical report [20].

One of the most interesting rules is the field read:

$$\frac{t'\ f \in T \qquad p \neq \texttt{isolated} \vee t' = \texttt{immutable}\ \_ \qquad t' \neq \texttt{isolated}\ \_ \vee p = \texttt{immutable}}{\Delta \mid x : \_, y : p\,T \vdash x = y.f \dashv y : p\,T, x : p \rhd_\Delta t'}$$

It uses a variant of permission combining parameterized by Δ, given in Figure 14, and lifted to types as before. When reading the definition of $\rhd_\Delta$, bear in mind that no permission variable may ever be instantiated to `isolated`. The final case of $\rhd_\Delta$ produces a *deferred permission combination* ($p \rightsquigarrow p$). The two cases previous to it that combine uninstantiated permission parameters leverage the bounding relation in Δ to give a sound answer that might produce `writable` or `immutable` results that can be used locally (though in the case that P is instantiated to `immutable`, this can lose some precision compared to instantiated uses). In the unrelated case, there is always an answer to give: `readable`. But this is too imprecise for uses such as container classes. There is always a more precise answer to give, but it cannot be known until all parameters are instantiated. To this end, we also change the type and permission substitution to reduce $p \rightsquigarrow q$ to $p \rhd_\Delta q$ if p and q are both concrete permissions after substitution. Note that these deferred permissions are effectively equivalent to `readable` in terms of what actions generic code using them may perform. This deferred combination plays a pivotal role in supporting highly polymorphic collection classes, as Section 6.3 describes.

$$\frac{\textsf{class}\ cn\langle\overline{X}\rangle\langle\overline{P}\rangle\ \textsf{where}\ \ldots\{\overline{field}\,t\ f\ \overline{field}\ \overline{method}\} \in P}{t[\overline{P/p}, \overline{X/T}]\ f \in cn\langle\overline{T}\rangle\langle\overline{p}\rangle}$$

$$\frac{\textsf{class}\ cn\langle\overline{X}\rangle\langle\overline{P}\rangle\ \textsf{where}\ \ldots\ \{\overline{field}\ \overline{method}\} \in P \quad m \in \overline{method} \quad m = t'\ m\langle\overline{Y}\rangle\langle\overline{Q}\rangle(\overline{u'\ z'})\ p'\ \texttt{where}\ \overline{Y <: V}, \overline{Q <: q}\ldots}{m[\overline{P/p}, \overline{X/T}] \in cn\langle\overline{T}\rangle\langle\overline{p}\rangle}$$

$$\textsf{IsoOrImm}_\Delta(\Gamma) \overset{\text{def}}{=} \forall(x : p\,T) \in \Gamma.\ \Delta \vdash p \prec \texttt{immutable}$$

$$\begin{array}{rcl}
\multicolumn{3}{l}{\rhd_\Delta : \textsf{Permission} \to \textsf{Permission} \to \textsf{Permission}} \\
\texttt{immutable} \rhd_\Delta \_ & = & \texttt{immutable} \\
\_ \rhd_\Delta \texttt{immutable} & = & \texttt{immutable} \\
\texttt{readable} \rhd_\Delta \texttt{writable} & = & \texttt{readable} \\
\texttt{readable} \rhd_\Delta \texttt{readable} & = & \texttt{readable} \\
\texttt{writable} \rhd_\Delta \texttt{readable} & = & \texttt{readable} \\
\texttt{writable} \rhd_\Delta \texttt{writable} & = & \texttt{writable} \\
\texttt{readable} \rhd_\Delta Q & = & \texttt{readable} \\
P \rhd_\Delta \texttt{readable} & = & \texttt{readable} \\
\texttt{writable} \rhd_\Delta Q & = & Q \\
P \rhd_\Delta \texttt{writable} & = & P \\
P \rhd_\Delta Q & = & \begin{cases} Q & \Delta \vdash P \prec Q \\ P & \Delta \vdash Q \prec P \\ P \rightsquigarrow Q & \text{otherwise} \end{cases}
\end{array}$$

Figure 14. Auxiliary judgements and metafunctions for the polymorphic system.

We also support covariant subtyping on `readable` and `immutable` references, as in IGJ [39].

$$\frac{\Delta \vdash p\,c\langle\overline{T}^{i-1}, T_i, \overline{T}^{m-i}\rangle\langle\overline{p}\rangle \quad \Delta \vdash p\,c\langle\overline{T}^{i-1}, T'_i, \overline{T}^{m-i}\rangle\langle\overline{p}\rangle \quad p = \texttt{readable} \vee p = \texttt{immutable} \quad \Delta \vdash T_i \prec T'_i}{\Delta \vdash p\,c\langle\overline{T}^{i-1}, T_i, \overline{T}^{m-i}\rangle\langle\overline{p}\rangle \prec p\,c\langle\overline{T}^{i-1}, T'_i, \overline{T}^{m-i}\rangle\langle\overline{p}\rangle}$$

There is another rule for safe covariant permission subtyping as well.

Soundness for Generics At a high level, the soundness proof for the polymorphic system is similar to the monomorphic system, because we only need to embed fully-instantiated programs (the top level program expression is type checked with an empty type bound context). The definition for type maps in the concrete machine states and views are redefined to have a range of only fully-instantiated types, making type environment denotations defined only over fully-instantiated types.

$$
\begin{array}{ll}
W, X, Y, Z & \text{type variables} \\
P, Q, R & \text{permission variables} \\
T, U, V & ::= cn\langle\overline{T}\rangle\langle\overline{p}\rangle \mid X
\end{array}
\quad\Bigg|\quad
\begin{array}{ll}
TD & ::= \texttt{class}\ cn\langle\overline{X}\rangle\langle\overline{P}\rangle\ [<: T2]\ \texttt{where}\ \overline{X <: T}, \overline{P <: p}\ \{\overline{field}\ \overline{meth}\ \} \\
meth & ::= t\ m\langle\overline{X}\rangle\langle\overline{P}\rangle(t_1\ x_1, ..., t_n\ x_n)\ p\ \texttt{where}\ \overline{X <: T}, \overline{P <: p}\ \{\ C; \mathsf{return}\ x;\ \} \\
p, q & ::= \ldots \mid P \mid p \rightsquigarrow p \\
\Delta & ::= \epsilon \mid \Delta, X <: T \mid \Delta, P <: p
\end{array}
$$

Figure 13. Grammar extensions for the polymorphic system.

Several auxiliary lemmas are required such as that substituting any valid permission or type instantiations into a generic derivation yields a consistent derivation. Additionally, the denotation of `writable` and `isolated` references must use a strengthened subtyping bound on their referents, to ensure they are viewed at a type that does not change any field types (thus preventing the classic reference subtyping problem while allowing covariant subtyping of read-only references). More details are given in our technical report [20].

6. Evaluation

A source-level variant of this system, as an extension to C#, is in use by a large project at Microsoft, as their primary programming language. The group has written several million lines of code, including: core libraries (including collections with polymorphism over element permissions and data-parallel operations when safe), a webserver, a high level optimizing compiler, and an MPEG decoder. These and other applications written in the source language are performance-competitive with established implementations on standard benchmarks; we mention this not because our language design is focused on performance, but merely to point out that heavy use of reference immutability, including removing mutable static/global state, has not come at the cost of performance in the experience of the Microsoft team. In fact, the prototype compiler exploits reference immutability information for a number of otherwise-unavailable compiler optimizations.

6.1 Differences from Formal System

The implementation differs from the formal system described earlier in a number of ways, mostly small. The most important difference is that the implementation supports proper task parallelism, with a first-class (unstructured) join. Task parallelism is supported via library calls that accept `isolated` delegates (closures, which must therefore capture only isolated or immutable state, in correspondence with T-ASYNC) and return `isolated` promises, thus interacting nicely with recovery and framing, since the asynchronous task's mutable memory is disjoint from the main computation's. `async` blocks are not currently checked according to T-ASYNC, mostly because we restrict `async` block task execution to single-threaded cooperative behavior, multiplexing `async` block tasks on a single CLR thread[2], which already reduces concurrency errors from its use, so the team has not yet decided to undertake the maintenance task of turning on such checking. This permits some unchecked concurrency, but single-threaded (avoiding at least memory model issues) and with only explicit points of interference (an `await` expression basically acts as a yield statement; essentially cooperative concurrency). The team plans to eventually enable checking of `async` blocks as well. T-PAR is not used for asynchronous tasks because it is unsound: recovery (T-RECOVISO and T-RECOVIMM) is not valid if a shared `readable` reference to mutable data can live arbitrarily long after the "recovery block" in an asynchronous task. Thus T-PAR is used only for applicable static constructs such as parallel for loops.

There are also a few source-level conveniences added as compared to the system here. The most notable is immutable classes. Immutable classes are simply classes whose constructors are required to have a final type environment with `this : immutable` rather than `isolated`. This allows the constructor to internally treat the self pointer as `writable` or `isolated`, before the type system conceptually uses T-RECOVIMM. Thus, `writable` and `readable` constructor arguments are permitted, they simply cannot be stored directly into the object. The disadvantage of this is that it is not possible, without unsafe casts, to create a cycle of objects of immutable *classes* (cycles of immutable objects in general remain possible as in Section 2.3).

The source variation also includes an *unstrict* block, where permission checking is disabled. The eventual goal is for this to be used only in trusted code (whose .NET assemblies are marked as such), for optimizations like lazy initialization of immutable data when an accessor is called; the core libraries offer a set of standard abstractions to encapsulate these unsafe actions (Section 6.6). Finally, the source language uses only a single type parameter list, where each argument may be instantiated with a single permission or full permission-qualified type.

C# also permits compound expressions, and expressions with side-effects, which our core language disallows. `consume` is an expression in the source language, which performs a destructive read on its l-value argument and returns the result. This makes using isolated method arguments

[2] In C#'s implementation, `async` blocks may run on other threads [3], but the team decided prior to adding reference immutability that such behavior was too error prone.

more convenient than in our core language, which allows statement consumption of fields, but treats isolated variables as affine resources when passed to methods.

A common focus for safe data-parallelism systems is handling of arrays. The implementation currently does not support arrays directly, but via trusted library abstractions for safe data parallelism. We are currently designing a notion of a sub-array, using a combination of isolation and immutability to allow safe array partitioning for in-place updates, as well as functional data-parallelism. This part of the design is still evolving.

6.2 Differences from C#

Beyond adding the obvious immutability-related extensions and restricting `async` block task execution to a single-threaded model (augmented with a true task parallelism library) as already discussed, the only additional difference from C# is that all static (global) state must be immutable. This is necessary for safe parallelism and for the recovery rules to avoid capturing shared mutable state. This restriction does lead to some different coding patterns, and required introducing several internally-unsafe but externally-safe library abstractions for things like global caches, which we will discuss shortly.

6.3 Source Level Examples

Generic Collections Collection libraries are a standard benchmark for any form of generics. The source variant of our system includes a full collections library, including support for polymorphism over permissions of the collection itself and elements of the collection. An illustrative example is the following simplified collections interface (using a lifting of our notation to a source language with interfaces, retaining our separation of permission and class parameters):

```
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;
}
```

This collection interface is parameterized over a class type for elements and a permission for the elements (which may never be instantiated to `isolated`). The `add()` method is natural, but the interesting case is `getEnumerator()`. This method returns a `writable` enumerator, but the enumerator manages two permissions: the permission with which `getEnumerator()` is called (which governs the permission the enumerator will hold on the collection) and the permission the collection has for the elements.

These separate permissions come into play in the type of the enumerator's `getNext()` method, which uses deferred permission composition ($p \rightsquigarrow p$, Section 5) to return elements with as precise a permission as possible. Simply specifying a single permission for the elements returned requires either specifying a different enumerator variant for every possible permission on the collection, or losing precision. For example, given a `writable` collection of `immutable` elements, it is reasonable to expect an iterator to return elements with an `immutable` permission. This is straightforward with a `getEnumerator()` variant specific to `writable` collections, but difficult using polymorphism for code reuse. Returning (using the enumerator definition's parameters) `PElem` elements is in general not possible with a generic `PColl` permission on the collection because we cannot predict at the method definition site the result of combining the two permissions when the enumerator accesses the collection; it would be sound for a `writable` collection of `immutable` objects, but not for an `immutable` collection of `writable` objects since `immutable` $\rhd_\triangle$ `writable` $=$ `immutable`, not `writable`. It also preserves precision, as any element from enumerating an `immutable` collection of `readable` references should ideally return `immutable` elements rather than the sound but less precise `readable`.

```
public class LinkedList<Elem><PElem>
  implements ICollection<Elem><PElem> {
    protected writable Node<Elem><PElem> head;
    protected class Node<Elem><PElem> {
        public PElem Elem item;
        public writable Node<Elem><PElem> next;
    }
    protected class LLEnum<E><PColl,PE>
      implements IEnumerator<E><PColl,PE> {
        private PColl Node<Elem><PE> next;
        public LLEnum(PColl LinkedList<E><PE> coll) {
            next = coll.head;
        }
        public bool hasNext() readable { return next == null; }
        public PColl⇝PElem E getNext() writable {
            if (next != null) {
                PColl⇝PElem E nextElem = next.item;
                next = next.next;
                return nextElem;
            }
            return null;
        }
    }
    public LinkedList() { head = null; }
    public void add(PElem Elem e) writable {
        writable Node<Elem><PElem> n = new Node<Elem><PElem>();
        n.item = e;
        n.next = head;
        head = n;
    }
    public writable Enumerator<Elem><P,PElem> getEnumerator() P {
        return new LLEnum<Elem><P,PElem>(this);
    }
}
```

Figure 15. A simplified collection with a polymorphic enumerator.

Consider a linked list as in Figure 15. The heart of the iterator's flexibility is in the type checking of the first assignment in `LLEnum.getNext()`. There the code has a `PColl` permissioned reference `next` to a linked list node that contains a `PElem` permissioned reference field `item` to an element. Thus the result type of `next.item` is `PColl⇝PElem PE` by T-FIELDREAD and $\rhd_\triangle$. When the linked list type is instantiated, and `getEnumerator()` is called with a certain permission, the enumerator type becomes fully instantiated and the deferred combination is reduced to a concrete permission. For example:

```
writable LinkedList<IntBox><writable> ll =
    new LinkedList<IntBox><writable>();
writable IEnumerator<IntBox><writable,writable> e =
    ll.getEnumerator(); // P instantiated as writable
writable IntBox b = e.getNext();

writable LinkedList<IntBox><readable> llr =
    new LinkedList<IntBox><readable>();
writable IEnumerator<IntBox><writable,readable> e =
    llr.getEnumerator(); // P instantiated as readable
writable IntBox b = e.getNext(); // Type Error!
// e.getNext() returns readable, since w⇝r=r
readable IntBox b = e.getNext(); // OK
```

A slightly richer variant of this enumerator design underlies the prototype's `foreach` construct, and is used widely in the Microsoft team's code.

Data Parallelism Reference immutability gives our language the ability to offer unified specializations of data structures for safe concurrency patterns. Other systems, such as the collections libraries for C# or Scala separate concurrency-safe (immutable) collections from mutable collections by separate (but related) trees in the class hierarchy.[3]

A fully polymorphic version of a `map()` method for a collection can coexist with a parallelized version `pmap()` specialized for `immutable` or `readable` collections. Consider the types and extension methods [34] (intuitively similar to mixins on .NET/CLR, though the differences are non-trivial) in Figure 16, adding parallel map to a `LinkedList` class for a singly-linked list (assuming the list object itself acts as a list node for this example). Each maps a function[4] across the list, but if the function requires only `readable` permission to its arguments, `pmap` may be used to do so in parallel. Note that the parallelized version can still be used with `writable` collections through subtyping and framing as long as the mapped operation is pure; no duplication or creation of an additional collection just for concurrency is needed. With the eventual addition of static method overloading by permissions (as in Javari [37]), these methods could share the same name, and the compiler could automatically select the parallelized version whenever possible.

6.4 Optimizations

Reference immutability enables some new optimizations in the compiler and runtime system. For example, the concurrent GC can use weaker read barriers for immutable data. The compiler can perform more code motion and caching, and an MSIL-to-native pass can freeze immutable data into the binary.

```
public abstract class Func1<In,Out><Pin,Pout,Prun> {
  public abstract Pout Out invoke(Pin In in) Prun;
}
public static class LinkedListExtensions {
  // A parallel map
  public static readable LinkedList<readable><X>
    pmap<X>(
      this readable LinkedList<readable><X>,
      immutable Func1<X,X><readable,readable,readable> fun)
    readable {
      readable LinkedList<readable><X> rest = null;
      isolated LinkedList<readable><X> head = null;
      head =                          || if (list.next != null)
        new LinkedList<readable><X>;  ||   rest =
      head.elem = fun(list.elem);     ||     list.next.map<X>(fun);
      head.next = rest;
      return head;
  }
  // A polymorphic map
  public static writable LinkedList<PL⇝PE><X>
    map<X><PE>(
      this PL LinkedList<PE><X> list,
      immutable Func1<X,X><PL⇝PE,PL⇝PE,readable> fun) PL {
      writable LinkedList<PL⇝PE><X> result =
          new LinkedList<PL⇝PE><X>;
      result.elem = fun(list.elem);
      writable LinkedList<PL⇝PE><X> newCurr = result;
      PL LinkedList<PE><X> oldCurr = list;
      while (oldCurr.next != null) {
        newCurr.next = new LinkedList<PL⇝PE><X>;
        newCurr = newCurr.next;
        oldCurr = oldCurr.next;
        newCurr.elem = fun(oldCurr.elem);
      }
      return result;
  }
}
```

Figure 16. Extension methods to add regular and parallel map to a linked list.

A common concern with destructive reads is the additional memory writes a naïve implementation (such as ours) might incur. These have not been an issue for us: many null writes are overwritten before flushing from the cache; the compiler's MSIL is later processed by one of two optimizing compilers (.NET JIT or an ahead-of-time MSIL-to-native compiler) that often optimize away shadowed null writes; and in many cases the manual treatment of uniqueness would still require storing null.

6.5 Evolving a Type System

This type system grew naturally from a series of efforts at safe parallelism. The initial plans included no new language features, only compiler plugins, and language extensions were added over time for better support. The earliest version was simply copying Spec#'s `[Pure]` method attribute [1], along with a set of carefully designed task- and data-parallelism libraries. To handle rough edges with this approach and ease checking, `readable` and `immutable` were added, followed by library abstractions for `isolated` and `immutable`. After some time using unstrict blocks to implement those abstractions, we gradually saw a way to integrate them into the type system. With all four permissions, the team was much more eager to use reference im-

[3] C# and Scala have practical reasons for this beyond simply being unable to check safety of parallelism: they lack the temporary immutability of our system due to the presence of unstructured parallelism.

[4] `Func1` is the intermediate-language encoding of a higher-order procedure. C# has proper types for these, called *delegate types* [34], each compiled to an abstract class with an `invoke` method with the appropriate arity and types. We restrict our examples to the underlying object representation for clarity.

mutability. After seeing some benefit, users eagerly added `readable` and `immutable` permissions.

Generics were the most difficult part of the design, but many iterations on the design of generic collections produced the design shown here. The one aspect we still struggle with is the occasional need for *shallow* permissions, such as for a collection with immutable membership, but mutable elements. This is the source of some unstrict blocks.

The entire design process was guided by user feedback about what was difficult. Picking the right defaults had a large impact on the users' happiness and willingness to use the language: `writable` is the default annotation, so any single-threaded C# that does not access global state also compiles with the prototype. This also made converting existing code much easier.

The system remains a work in progress. We initially feared the loss of traditional threads could be a great weakness, but the team placed slightly higher value on correct concurrency than easy concurrency, leading to the design point illustrated here. To recover some flexibility, we are currently adding actor concurrency, using `isolated` and `immutable` permissions for safe message passing [22, 24].

We continue to work on driving the number of unstrict blocks as low as possible without over-complicating the type system's use or implementation. Part of this includes codifying additional patterns that currently require unstrict use. We understand how to implement some of these (such as permission conversion for stateless objects like empty arrays) safely within the type system, but the engineering trade-offs have not yet been judged worth the effort to implement.

6.6 User Experience

Overall, the Microsoft team has been satisfied with the additional safety they gain from not only the general software engineering advantages of reference immutability [37, 39, 40] but particularly the safe parallelism.

Anecdotally, they claim that the further they push reference immutability through their code base, the more bugs they find from spurious mutations. The main classes of bugs found are cases where a developer provided an object intended for read-only access, but a callee incorrectly mutated it; accidental mutations of structures that should be immutable; and data races where data should have been immutable or thread local (i.e. `isolated`, and one thread kept and used a stale reference).

Annotation burden has been low. There is roughly 1 annotation (permission or consume) per 63 lines of code. These are roughly 55% `readable`, 16.8% `consume`, 16.5% `immutable`, 4.7% `writable`, 4.1% `isolated`, and 2.8% generic permissions. This is partly due to the `writable` default, as well as C#'s local type inference (e.g. `var x = ...;`). Thus most annotations appear in method signatures. Note that because users added additional qualifiers for stricter behavior checking, this is not the minimal annotation burden to type check, but reflects heavy use of the system.

The type system does make some common tasks difficult. We were initially concerned that immutable-only global state would be too restrictive, but has been mitigated by features of the platform the Microsoft team develops on top of. The platform includes pervasive support for capability-based resource access for resources such as files. Global caches are treated as capabilities, which must be passed explicity through the source (essentially writable references). This requires some careful design, but has not been onerous. Making the caches global per process adds some plumbing effort, but allows better unified resource management.

Another point of initial concern was whether isolation would be too restrictive. In practice it also adds some design work, but our borrowing / recovery features avoid viral linearity annotations, so it has not been troublesome. It has also revealed subtle aliasing and concurrency bugs, and it enables many affine reference design patterns, such as checking linear hand-off in pipeline designs.

The standard library also provides trusted internally-unstrict abstractions for common idioms that would otherwise require wider use of unstrict blocks. Examples include lazy initialization and general memoization for otherwise immutable data structures, caches, and diagnostic logging. There are relatively few unstrict blocks, of varying sizes (a count does not give an accurate estimate of unchecked code). Most of these are in safe (trusted) standard library abstractions and interactions with the runtime system (GC, allocator, etc., which are already not memory-safe). Over the course of development, unstrict blocks have also been useful for the Microsoft team to make forward progress even while relying on effectively nightly builds of the compiler. They have been used to temporarily work around unimplemented features or compiler bugs, with such blocks being marked, and removed once the compiler is updated.

The Microsoft team was surprisingly receptive to using explicit destructive reads, as opposed to richer flow-sensitive analyses [8, 28] (which also have non-trivial interaction with exceptions). They value the simplicity and predictability of destructive reads, and like that it makes the transfer of unique references explicit and easy to find. In general, the team preferred explicit source representation for type system interactions (e.g. `consume`, permission conversion).

The team has also naturally developed their own design patterns for working in this environment. One of the most popular is informally called the "builder pattern" (as in building a collection) to create frozen collections:

```
isolated List<Foo> list = new List<Foo>();
foreach (var cur in someOtherCollection) {
  isolated Foo f = new Foo();
  f.Name = cur.Name;
  // etc ...
  list.Add(consume f);
}
immutable List<Foo> immList = consume list;
```

This pattern can be further abstracted for elements with a deep clone method returning an `isolated` reference.

Nearly all (non-`async`) concurrency in the system is checked. The unchecked concurrency is mostly due to development priorities (e.g. feature development has been prioritized over converting the remaining code); removing all unchecked concurrency from the system remains an explicit goal for the team. Nonetheless, the safe concurrency features described here have handled most of the team's needs. Natural partitioning of tasks, such as in the H.264 and JPEG decoders (both verified for safe concurrency) is "surprisingly common," and well-supported by these abstractions. Sometimes breaking an algorithm into Map-Reduce-style phases helps fit problems into these abstractions. The main difficulties using the model come in two forms. The first form is where partitioning is dynamic rather than structural. This is difficult to express efficiently; we are working on a framework to compute a partitioning blueprint dynamically. Second, sometimes communication among tasks is not required for correctness, but offers substantial performance benefits: for example, in a parallelized search algorithm, broadcasting the best-known result thus far can help all threads prune the search space. Currently unstrict code is used for a few instances of this, which motivates current work to add actors to the language [22, 24].

7. Related Work

Reference immutability [37, 39, 40] is a family of work characterized by the ability to make an object graph effectively immutable to a region of code by passing read-only references to objects that may be mutated later, where the read-only effect of a reference applies transitively to all references obtained through a read-only reference. Common extensions include support for class immutability (classes where all instances are permanently immutable after allocation) and object immutability (making an individual instance of a class permanently immutable). Surprisingly, despite safe parallelism being cited as a natural application of reference immutability, we are the first to formalize such a use.

Immutability Generic Java (IGJ) [39] is the most similar reference immutability work to ours, though it does not address concurrency. IGJ uses Java generics support to embed reference immutability into Java syntax (it still requires a custom compiler to handle permission subtyping). Thus reference permissions are specified by special classes as the first type parameter of a generic type. IGJ's support of object immutability is also based on the permission passed to a constructor, rather than conversion, so object immutability is enforced at allocation, and may not be deferred as our T-RECOVIMM rule allows. This means that creating a cycle of immutable objects requires a self-passing constructor idiom, where a constructor for cyclic immutable data structures must pass its `this` pointer into another constructor call as Immutable. Haack and Poll relax this restriction by lexically scoping the modification lifetime of an immutable instance [21].

Ownership [6, 7, 11, 13] and Universe [14] types describe a notion of some objects "owning" others as a method of structuring heap references and mutation. The "owner-as-modifier" interpretation of ownership resembles reference immutability: any modification to an object o must be done through a reference to o's owner. These systems still permit references across ownership domain, but such references (`any` references in Universe types) are deeply read-only. Universe types specify a "viewpoint adaptation" relation used for adapting type declarations to their use in a given context, which directly inspired our permission adaptation relation. Leino, Müller, and Wallenburg [25] boost the owner-as-modifier restriction to object immutability by adding a `freeze` operation that transfers ownership of an object to a hidden owner unavailable to the program source. Since the owner cannot be mentioned, no modifications may be made to frozen objects. In general, ownership transfer, (as in Leino et al.'s system or UTT [27]) relies on uniqueness and treats ownership domains as regions to merge.

The most similar treatment of polymorphism over mutability-related qualifiers to our work is Generic Universe Types (GUT). GUT provides polymorphism over permission-qualified types as in our prototype source language, rather than separating qualifiers and class types as our core language does. GUT's viewpoint adaptation (roughly equivalent to our permission combining relation $\rhd_\Delta$) deals immediately with concrete qualifier combinations, and preserves some precision when combining a concrete ownership modifier with a generic one. But when combining two generic ownership modifiers, the result is always `any`, roughly equivalent to `readable` in our system. In practice, this is sufficient precision for GUT, because passing a generic type across ownership domains typically converts to `any` anyways. Our use cases require additional precision, which we retain by using deferred permission combination ($p \rightsquigarrow p$) to postpone the combination until all type parameters are instantiated. Without this, the generic enumerator in Section 6.3 could only return `readable` elements, even for a `writable` collection of `immutable` elements. IGJ [39] does not discuss this type of permission combining, but appears to have a similar loss of precision: i.e. accessing a field with generic permission through a reference with generic permission always yields a `readable` reference. OIGJ [40] can express generic iterators, but mostly because reference immutability in OIGJ is not transitive: a read-only iterator over a mutable list is permitted to mutate the list, and an iterator over an immutable list of mutable elements can return mutable references to those elements.

Ownership type systems have been used to achieve data race freedom [6, 7, 13], essentially by using the ownership hierarchy to associate data with locks and beyond that enforcing data race freedom in the standard way [17–19].

Clarke et al. [12] use ownership to preserve thread locality, but allow immutable instances and "safe" references (which permit access only to immutable, e.g. Java-style `final`, portions of the referent) to be shared freely across threads, and add transfer of externally unique references.

Östlund et al. [30] present an ownership type and effect system for a language Joe_3 with the explicit aim of supporting reference immutability idioms by embedding into an ownership type system. Owner polymorphic methods declare the effects they may have on each ownership domain, treating ownership domains as regions [35, 36]. Joe_3 uses a similar technique to ours for delayed initialization of immutable instances, as it has (externally) unique references, and writing a unique reference to an immutable variable or field converts the externally unique cluster into a cluster of immutable objects. While Joe_3 has blocks for borrowing unique references, our T-RecovImm rule is more general, combining borrowing and conversion. Their borrowing mechanism also creates local owners to preserve encapsulation, requiring explicit ownership transfer to merge aggregates. Our type system also permits invoking some methods directly on unique references (as opposed to the required source-level borrowing in Joe_3) because our frame rule makes it easy to prove the invocation preserves uniqueness with the additional argument restrictions (Figure 5).

Our T-RecovIso rule is in some ways a simplification of existing techniques for borrowing unique references, given the presence of reference immutability qualifiers. The closest work on borrowing to ours in terms of simplicity and expressiveness is Haller and Odersky's work [23] using capabilities that guard access to regions containing externally-unique clusters. Regions of code that return an input capability have only borrowed from a region, and have not violated its external uniqueness. Local variable types $\rho \triangleright \tau$ are references to an object of type τ in region ρ. When a reference to some object in an externally unique aggregate is written to a heap location, that aggregate's capability is consumed in a flow-sensitive fashion, and all local variables guarded by that capability become inaccessible. Our recovery rule requires no invalidation, though its use may require environment weakening. We believe the expressiveness of the two approaches to be equal for code without method calls. For method calls, Haller and Odersky track borrowing of individual arguments by what permissions are returned. Our system would require returning multiple `isolated` references through the heap, though our recovery rules would allow inferring some method returns as `isolated` in the proper contexts, without cooperation of the method called. We also add some flexibility by allowing arbitrary paths to immutable state reachable from an externally-unique aggregate.

Another interesting variant of borrowing is *adoption* [10, 16], where one piece of state is logically embedded into another piece, which provides a way to manage unique references without destructive reads.

Boyland proposed fractional permissions [9] to reason statically about interference among threads in a language with fork-join concurrency. We use a twist on fractional permissions in the denotation of our types, including to denote uniqueness, though the fractions do not appear in the source program or type system. The Plaid language uses fractional permissions to manage aliasing and updates to objects when checking typestate [2]. Recent work by Naden et al. to simplify Plaid [28], like our system, does not require any fractions or explicit permission terms to appear in source code, though unlike us an implementation of their type system must reason about fractional permissions (our fractional permissions appear only in our meta-proofs). Like us they use type qualifiers to specify access permissions to each object, though their permissions do not apply transitively (a distinction driven largely by our differing motivations). Naden's language supports externally-unique and immutable references, and more fine-grained control than our system over how permissions for each argument to a method are changed (e.g. preserving uniqueness as an indication of borrowing), though his language does not address concurrency.

Deterministic Parallel Java (DPJ) [4] uses effect typing and nested regions [35, 36] to enable data parallelism and deterministic execution of parallel programs. An expression may have a read or write effect on each of some number of regions, and expressions with non-conflicting effects (one thread with write effect and none with read effects, or multiple threads with read effects and no write effects, on each region) may safely be parallelized. This ensures not only the absence of data races, but determinism as well (later revisions add controlled nondeterminism [5]). Our system is similar in spirit, but requires no mention of regions in the source, only mention of the permissions required for each object a method accesses, not where they reside. This means, for example, that region polymorphism is implicit in our system; methods need not bind to a specific number of regions, while DPJ requires methods and classes to declare fixed numbers of regions over which they operate (it is possible to instantiate multiple region arguments with the same region). The upside to the explicit treatment of regions in DPJ is that non-interfering writes may be parallelized without requiring any sort of reference uniqueness (the type system must still be able to prove two regions are distinct). DPJ also treats data parallelism over arrays, whereas we do not.

Westbrook et al. [38] describe Habanero Java with Permissions (HJp), a language with parallelism structure between our formal and source languages: $\mathsf{async}(e)$ begins an asynchronous task and returns unit; $\mathsf{finish}(e)$ waits for all tasks spawned within e, rather than allowing joins with individual tasks. They use qualifiers to distinguish thread-local read, thread-local write, and thread-shared read access to objects (the latter is mutually exclusive with the first two). They must distinguish between thread local and shared read-only access because they cannot guarantee the inaccessibility of writable references to objects for the duration of an `async`; doing so would require a flow-sensitive set of variables inaccessible until the enclosing `finish()` because the end of an `async` is not statically scoped, and `async` blocks may capture any shareable state, not only unique or immutable state. Their treatment of storing (totally) unique references in unique fields and embedding the referent's permissions is more flexible for concurrency than our `isolated` fields. Their embedding allows natural read-only access to unique field referents if the containing object is shared read-only, while `isolated` fields of shared `readable` objects are inaccessible until recovery or conversion. Reads from non-unique fields in HJp have no static permissions; dereferencing such fields requires dynamically acquiring permissions. We treat permission polymorphism, while they do not.

8. Conclusions

We have used reference immutability to ensure safe (interference-free) parallelism, in part by combining reference immutability with external uniqueness. Applying our approach to an intermediate-level language led us to derive recovery rules for recovering isolation or immutability in certain contexts, which offers a natural approach to borrowing for languages with reference immutability. Our type system models a reference immutability system in active use in industry, and we have described their experiences with it.

Acknowledgments

Thanks to Dan Grossman, Brian Burg, and the anonymous reviewers for helpful comments, and to Michael Ernst for helpful conversations about related work.

The first author's design work occurred during a Microsoft internship; proof work was supported by NSF grants CCF-1016701 and CNS-0855252.

References

[1] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and Verification: The Spec# Experience. *Commun. ACM*, 54(6):81–91, June 2011.

[2] K. Bierhoff and J. Aldrich. Modular Typestate Checking of Aliased Objects. In *OOPSLA*, 2007.

[3] G. Bierman, C. Russo, G. Mainland, E. Meijer, and M. Torgersen. Pause n Play: Formalizing Asynchronous C$^\sharp$. In *ECOOP*, 2012.

[4] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.

[5] R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe Nondeterminism in a Deterministic-by-default Parallel Language. In *POPL*, 2011.

[6] C. Boyapati and M. Rinard. A Parameterized Type System for Race-Free Java Programs. In *OOPSLA*, 2001.

[7] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA*, 2002.

[8] J. Boyland. Alias Burying: Unique Variables without Destructive Reads. *Software Practice & Experience*, 31(6), 2001.

[9] J. Boyland. Checking Interference with Fractional Permissions. In *SAS*, 2003.

[10] J. T. Boyland and W. Retert. Connecting Effects and Uniqueness with Adoption. In *POPL*, 2005.

[11] D. Clarke, S. Drossopoulou, and J. Noble. Aliasing, Confinement, and Ownership in Object-Oriented Programming. In *ECOOP 2003 Workshop Reader*, 2004.

[12] D. Clarke, T. Wrigstad, J. Östlund, and E. Johnsen. Minimal Ownership for Active Objects. In *APLAS*, 2008.

[13] D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universe Types for Race Safety. In *VAMP*, 2007.

[14] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *ECOOP*, 2007.

[15] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional Reasoning for Concurrent Programs. Technical report, 2012. URL `https://sites.google.com/site/viewsmodel/`.

[16] M. Fahndrich and R. DeLine. Adoption and Focus: Practical Linear Types for Imperative Programming. In *PLDI*, 2002.

[17] C. Flanagan and M. Abadi. Object Types against Races. In *CONCUR*, 1999.

[18] C. Flanagan and M. Abadi. Types for Safe Locking. In *ESOP*, 1999.

[19] C. Flanagan and S. N. Freund. Type-Based Race Detection for Java. In *PLDI*, 2000.

[20] C. S. Gordon, M. J. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and Reference Immutability for Safe Parallelism (Extended Version). Technical Report MSR-TR-2012-79, 2012.

[21] C. Haack and E. Poll. Type-Based Object Immutability with Flexible Initialization. In *ECOOP*, 2009.

[22] P. Haller. *Isolated Actors for Race-Free Concurrent Programming*. PhD thesis, EPFL, Lausanne, Switzerland, 2010.

[23] P. Haller and M. Odersky. Capabilities for Uniqueness and Borrowing. In *ECOOP*, 2010.

[24] C. Hewitt, P. Bishop, I. Greif, B. Smith, T. Matson, and R. Steiger. Actor Induction and Meta-Evaluation. In *POPL*, 1973.

[25] K. Leino, P. Müller, and A. Wallenburg. Flexible Immutability with Frozen Objects. In *VSTTE*, 2008.

[26] K. Mazurak, J. Zhao, and S. Zdancewic. Lightweight Linear Types in System F$^\circ$. In *TLDI*, 2010.

[27] P. Müller and A. Rudich. Ownership Transfer in Universe Types. In *OOPSLA*, 2007.

[28] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A Type System for Borrowing Permissions. In *POPL*, 2012.

[29] P. O'Hearn, J. Reynolds, and H. Yang. Local Reasoning about Programs that Alter Data Structures. In *Computer Science Logic*, 2001.

[30] J. Östlund, T. Wrigstad, D. Clarke, and B. Åkerblom. Ownership, Uniqueness, and Immutability. In *Objects, Components, Models and Patterns*, 2008.

[31] M. Parkinson, R. Bornat, and C. Calcagno. Variables as Resource in Hoare Logics. In *LICS*, 2006.

[32] U. S. Reddy and J. C. Reynolds. Syntactic Control of Interference for Separation Logic. In *POPL*, 2012.

[33] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, 2002.

[34] J. Richter. *CLR Via C♯, Second Edition*. Microsoft Press, 2006. ISBN 0735621632.

[35] J.-P. Talpin and P. Jouvelot. Polymorphic Type, Region, and Effect Inference. *JFP*, 2(2), 1992.

[36] M. Tofte and J.-P. Talpin. Implementation of the Typed Call-by-Value λ-calculus Using a Stack of Regions. In *POPL*, 1994.

[37] M. S. Tschantz and M. D. Ernst. Javari: Adding Reference Immutability to Java. In *OOPSLA*, 2005.

[38] E. Westbrook, J. Zhao, Z. Budimli, and V. Sarkar. Practical Permissions for Race-Free Parallelism. In *ECOOP*, 2012.

[39] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kiezun, and M. D. Ernst. Object and Reference Immutability Using Java Generics. In *ESEC-FSE*, 2007.

[40] Y. Zibin, A. Potanin, P. Li, M. Ali, and M. D. Ernst. Ownership and Immutability in Generic Java. In *OOPSLA*, 2010.