Types, Regions, and Effects for Safe Programming with Object-Oriented Parallel Frameworks

Robert L. Bocchino Jr. Vikram S. Adve
Department of Computer Science
University of Illinois at Urbana-Champaign
dpj@cs.uiuc.edu

Abstract

Object-oriented frameworks can make parallel programming easier by providing generic parallel algorithms such as map, reduce, or scan, and letting the user fill in the details with sequential code. However, such frameworks can produce incorrect behavior if they are not carefully used, e.g., if a user-supplied function performs an unsynchronized access to a global variable. We develop novel techniques that a framework designer can use to prevent such errors. Building on a language (Deterministic Parallel Java, or DPJ) with an expressive region-based type and effect system, we show how to write a framework API that enables sound reasoning about the effects of unknown user-supplied methods. We also describe novel extensions to DPJ that enable generic types and effects — essential for flexible frameworks — while retaining soundness. Finally, we show how to make the reasoning modular: using any desired testing or verification technique, the framework author can guarantee noninterference subject to the API constraints; and the compiler can check the constraints to provide a noninterference guarantee for the entire user program. We evaluate our technique by using it to write two parallel frameworks and two realistic parallel algorithms.

1. Introduction

The emergence of multicore desktop architectures is driving parallel programming into the mainstream, posing new productivity, correctness, and performance challenges for programmers who are used to writing sequential code. One way to alleviate these challenges is to use object-oriented frameworks. The framework writer provides most of the code for parallel construction of generic data structures and for generic parallel algorithms such as map, reduce, or scan; and the user fills in the missing pieces with sequential code that is applied in parallel by the framework. Examples include the algorithm templates in Intel's Threading Building Blocks (TBB) [23] and Java's ParallelArray framework [1]. Such a framework is usually easier to reason about than general parallel programming, because the user only has to write sequential code, letting the framework orchestrate the parallelism.

However, current frameworks give no guarantee of noninterference of effect, and this a serious deficiency in terms of correctness and program understanding. For example, ParallelArray's `apply` method applies an arbitrary user-specified function to each element of the array. If that operation performs an unsynchronized update to a global, then an unexpected race will result. One could issue a set of informal guidelines for how to use the API safely, but this is unsatisfactory. It would be much better if (1) the framework developer could write an API expressing a contract (for example, the function provided to `apply` is read-only with respect to global state); and (2) the compiler could check that the contract is met by all code supplied by the user to the framework.

While several tools and techniques exist that support writing and checking assertions at interface boundaries [17, 21, 29], these ideas have not yet been applied to enforce *parallel noninterference*, as discussed in Section 7. Doing so involves several open challenges, which we list below.

Maintaining internal linearity. Useful parallel frameworks need to support parallel updates on contained objects. For example, we would like a `ParallelArray` of distinct objects, where the user can provide an `apply` function that updates an element, and ask the framework to apply it to each distinct object in parallel. To do this safely, the framework must ensure that the objects are really distinct; otherwise the same object could be updated in two parallel iterations, causing a race. We call this property *internal linearity*, by analogy with linear types [31], because each contained object occupies exactly one iteration slot of the container. It is "internal" because we only care about aliasing in the container slots; arbitrary aliasing outside the container (or between containers) is still allowed. For a language like Java with robust reference aliasing, internal linearity is a nontrivial property.

Constraining the effects of user-supplied methods. For a parallel update traversal over the objects in a framework, internal linearity is necessary but not sufficient to ensure noninterference. The framework must also ensure that the effects of the user-supplied methods do not interfere, for example by updating a global variable, or by following a link from one contained object to another.

Making the types and effects generic. Because different uses of the framework need user-supplied methods with different effects, the framework should constrain the effects of user-supplied methods as little as possible while retaining soundness. For example, one use of `apply` may write into each object only; while another may read shared data and write into each object. The framework should also be generic, not specialized to a specific type of contained object. These requirements pose challenges when the framework author needs information about the type of the contained objects and the effect of user-supplied methods in order to provide a noninterference guarantee.

Verifying the framework implementation. The framework author must verify that the internal framework implementation guarantees safe parallelism, given that the API is enforced. For example, even if the framework ensures that the same object is never inserted twice, it must also ensure that any parallel loop inside the framework iterates exactly once over each inserted object.

Notice that the first three challenges are about defining a framework *API* that enables sound reasoning about uses of the framework; while the fourth challenge is about writing a framework *implementation* and proving it correct.

In this work we address the first three challenges, i.e., we show how to write a framework API so that the framework author can reason soundly about interference of effect in arbitrary instantiations of the framework, with unknown user-supplied methods and generic type bindings. We build on Deterministic Parallel Java (DPJ) [5], which expresses effects in terms of *regions* that partition the heap. Regions provide an intuitive way to reason about sharing patterns and a flexible way to express and check effects.

We do not try to solve the fourth challenge; instead we reduce it to a simple logical predicate that can be discharged by other means, such as program logic [14, 22], testing, or model checking. This predicate is completely hidden from the user of the framework, so that the user gets a strong guarantee, assuming a correct framework implementation: if the program type checks, then there is no interference between parallel code sections. A framework such as ParallelArray can further provide *deterministic execution* [3], perhaps subject to some additional requirements (e.g., that reduction operations are associative) that our system does not currently check.

Our approach fosters modular checking in two ways. First, we show how to use a region-based type and effect system for what it does very well — checking the use of a generic framework API — while using any other appropriate form of verification to check the inside of the framework, e.g., to verify set and tree properties. Second, the framework author can verify the generic framework once, and then rely on a type checker to verify each use separately; the user does not have to re-verify the instantiated framework for each use. This is important because the user may have no idea how to verify the framework or how it even works.

Our contributions are the following:

1. We show how to write a framework API using the DPJ type and effect system "off the shelf" so that the framework implementer has all the information necessary to guarantee internal linearity of reference and sound effects for user-supplied methods.
2. We show how to extend the DPJ type and effect system to add generic effects and generic types, making the frameworks more general and useful. For the effects, we add *effect variables*; here the technical challenges are constraining the unknown effects appropriately and providing sound subtyping. For generic types, we introduce *type region parameters* so that the framework author has enough information about the types bound to generic type variables to guarantee internal linearity and soundness of effect, without knowing the exact type.
3. Using a simple logical predicate as the "glue," we show how to make different forms of verification interoperate so that the framework author and user can separately check separate parts of the program with separate verification mechanisms, and guarantee that any composition of the parts results in a correct program.

We formalize a core subset of the system and formally state the soundness results. We also describe the results of an evaluation showing that the system is expressive enough to capture two realistic parallel algorithms, and that the extra annotations required by the system are not unduly burdensome.

2. Background: Deterministic Parallel Java

We begin with a brief introduction to DPJ [5]. DPJ uses *regions* to specify access to the heap: every class field and array cell lies in a single region, and distinct regions represent disjoint collections of memory locations. A region can be a declared name, or a `final` local variable representing a dynamic object reference, as in ownership systems [9,10]. All the regions are arranged in a tree hierarchy, rooted at the special region `Root`. The regions in the subtree at region R can be named as R`:*`. DPJ uses *effect summaries* on method interfaces, expressed in terms of reads and writes to regions, to enable method-local checking of noninterference.

```
1  public class ListNode<region R> {
2      int data in this;
3      ListNode<*> next in this;
4      public ListNode(int data, ListNode<R> next) pure {
5          this.data = data;
6          this.next = next;
7      }
8  }
```

Figure 1. `ListNode` class that will serve as a running example.

```
1  class NodePair {
2      region One, Two;
3      private Node<One> one in Root;
4      private Node<Two> two in Root;
5      NodePair(ListNode<One> one, ListNode<Two> two) pure {
6          this.one = one;
7          this.two = two;
8      }
9      void updateNodes(int oneData, int twoData) {
10         cobegin {
11             /* reads Root writes One : * */
12             one.data = oneData;
13             /* reads Root writes Two : * */
14             two.data = twoData;
15         }
16     }
17 }
```

Figure 2. Using region parameters to distinguish object instances.

As an example, Figure 1 defines a simple list node class that we will also use in subsequent sections. The class has one region parameter `R`. The fields `data` and `next` in lines 2–3 are both located in the object reference region associated with `this`. A reference region is a child of the first region appearing in its type: for example, in Figure 1, region `this` is a child of `R`. In line 4, the effect of the constructor is declared `pure` (no effect) because in DPJ an object is not visible to the rest of the program until the constructor returns, so constructors do not have to report their effects on the constructed object.

Figure 2 presents a simple container class, `ListNodePair`, that stores a pair of list nodes. Lines 3–4 instantiate `ListNode` types using the field region names `One` and `Two`, declared just above. Here `reads Root` comes from the fact that the fields `one` and `two` are located in region `Root`, the top-level region in the hierarchy, as shown in lines 3–4. The effect `writes One:*` comes from the fact that line 12 writes `data`, which is in `this`, which is under the region bound to `R` in the type of `one`, i.e., `One`; and similarly for the effect shown in line 13. Because `One` and `Two` are distinct names, and because the region hierarchy forms a tree, the compiler can conclude that the updates in lines 12 and 14 are disjoint. With these features, together with additional features for arrays, divide and conquer parallelism, and commutative operations such as set inserts, DPJ can express important patterns of parallelism [5].

3. Difficulties with Region-Based Effect Systems

As DPJ illustrates, region-based type and effect systems can be quite expressive, and they are a natural choice for writing safe object-oriented frameworks. However, existing systems impose significant limitations that we must address in our framework design. As we will see, by shifting some of the burden of guaranteeing noninterference from the type system to the framework, we can overcome some of these limitations.

One limitation is that, to guarantee soundness, we have to prohibit swapping of `one` and `two` in the example:

```
void swap() {
  ListNode<One> tmp = one;
  /* illegal, can't assign ListNode<Two> to ListNode<One> */
```

```
    one = two;
    /* illegal, can't assign ListNode<One> to ListNode<Two> */
    two = tmp;
  }
```

If we could do such an assignment, then in general we could have multiple references with conflicting types pointing to the same data, and we would no longer be able to draw sound conclusions about effects [4].

For this reason, researchers have introduced wildcard types that allow freer assignments [5, 19]. For example, in lines 3–4 of Figure 2, we could have written both types `ListNode<*>`, where `*` stands in for any region. Now the swapping shown above is fine, because the variable types don't constrain what regions can appear in the dynamic reference types. However, we have lost the ability to distinguish writes to `one.data` and `two.data` using the type system, because now all we know is that the writes in lines 12 and 14 are to `*`. This is true even though by inspecting Figure 2, we (as opposed to the type system) can see that (1) `one` and `two` are distinct coming into the constructor (line 5); and (2) the `swap` operation preserves the distinctness of `one` and `two`. So the state of the art in region-based type systems forces us to choose: either we can prove that two references don't alias, or we can swap the two references, but not both.

In fact, the situation is worse than this. Notice that in Figure 1, we gave the `ListNode` class a field `next` of type `ListNode<*>`, i.e., a `ListNode` with an unspecified region bound to its parameter. Therefore, as shown in Figure 3, a `ListNodePair` holding list nodes can have cross links. The effect system has to make sure that (1) the references stored in the fields `one` and `two` are distinct; and (2) when following the references to access the objects in parallel, the cross links are never followed to update the same object. Further, we probably don't want to "hard code" the operation of writing to `data` into the framework implementation, as shown in lines 12 and 14. Instead, as discussed in the introduction, we would like to express the operation abstractly, and let the user supply the specific operation. We therefore must constrain the effects of the user-supplied method so that we can argue that for any user-supplied method, this kind of interference cannot happen.

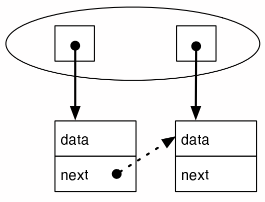

Figure 3. The references stored in the `NodePair` are distinct; but we can still get a race if we follow the cross link represented by the dotted arrow.

4. Writing Safe, Reusable Parallel Frameworks

In this section we show how to address the challenges discussed above to write safe, reusable parallel frameworks. First we define an abstract linear container, which provides a sample framework API to illustrate our ideas. Second, we show how how to write the API so that the framework writer can reason soundly about effects for a container specialized to list nodes. Third, we show how to extend the type system to make the API generic. Finally, we address the problem of verifying the framework implementation.

4.1 Abstract Linear Containers

We define an abstract data type called an *abstract linear container*. This type generalizes the trivial `NodePair` container introduced in the previous section. In Section 6 we discuss how to apply our techniques to more realistic examples.

An abstract linear container is an abstract data type with the following properties:

1. It contains references to other objects. The number of stored references can be fixed up front (as with an array) or changed dynamically (as with a resizable array or set).
2. The elements are conceptually stored in slots. An iteration over the elements in the container iterates over the slots. For example, for an array, the slots are the array cells; for a set the slots are the set elements; and for a tree the slots are the tree nodes.
3. For any two distinct slots, the references stored in the slots are different (point to different objects). So, for example, a set is allowed but a multiset is not (since two slots can have the same element). However, one could emulate a multiset using a set of sets.

Property 1 is standard for a container ADT, e.g., any of the containers in `java.util`. We introduce property 2 just so we have a way to talk about the iteration space of a container that is independent of the internal storage pattern (array, tree, etc.). Property 3 is key to ensuring soundness when the user calls an API method to iterate over the container and update its contents in parallel. We call this property *internal linearity*, by analogy with linear types [31]. The slots are linear, in the sense that at most one slot of any particular linear container points to any object. However, unlike general linear types, multiple containers (or other references outside the container) can point to the same object. Note that both versions of `NodePair` from Section 3 are instances of the abstract linear container type, where the slots are the fields `one` and `two`. For conciseness, we refer to the slots of the container and the container itself as "linear," though we mean that internal linearity holds as to the slots.

4.2 A List Node Container

We now show how to use the DPJ type system [5] to write an abstract linear container API that allows safe parallel updates to its contained objects. There are two problems: maintaining linearity, and reasoning about effects. Our key insight is that through careful API design, together with judicious use of local variables and method region parameters, we can enforce restrictions like "a factory method must return a new object" or "an apply method must write only under the region of the object it is given." Further, we can impose these restrictions without exposing global region names (such as `One` and `Two` in Figure 2), that would otherwise prevent swapping and other linearity-preserving operations inside the framework.

4.2.1 Maintaining Linearity

To maintain linearity, we use the following strategy: (1) every container starts empty and so is trivially linear; and (2) every operation provided by the linear container API is linearity preserving (takes a linear container to another linear container). By a simple induction, we can then conclude that the container is linear throughout its lifetime. The hard part is guaranteeing property (2). There are two types of operations to consider: (a) operations that are totally under the control of the container implementation and (b) operations that must cooperate with (possibly unknown) user code.

An example of (a) is a tree rebalancing or array reshuffling that operates only on internal structure of the container. Here the problem is entirely reduced to writing a correct framework implementation. We discuss this problem in Section 4.4 below.

In the case of (b), however, the framework must restrict what the user can do so that the framework author can reason soundly about uses of the container without knowing exactly what that use will look like. A core example here is putting things into a container. For the container to be useful, the user has to retain control over what is inserted in the container, and how and where those inserted things are created. The trick is to allow some control while still being able to reason about linearity. In our work to date we have explored three strategies: controlled creation of contained objects, building one linear container from another, and backing the container with a set.

Controlled creation of contained objects. Lines 7–11 of Figure 4 illustrate this strategy, for a `NodeContainer` interface that could be implemented in different ways (array, tree, etc). We define the interface with two region parameters, `Node` and `Cont`, because we want to refer separately to the nodes stored in the container, and the container itself. The container implementation does the actual object creation, but the user specifies the number of objects to create and provides a factory method specifying how to create the ith object. For example, a use could look like this, assuming a class `NodeArray` that implements `NodeContainer`:

```
public class MyFactory implements NodeContainer.NodeFactory<N> {
  public <region R>ListNode<R> create(int i) {
    return new ListNode<R>(i, null);
  }
}
NodeContainer<N,C> cont =
  new NodeArray<N,C>(new MyFactory(), 10);
```

This code creates a new `NodeArray` with 10 list nodes, such that the ith one has its `data` field set to i. Here `N` and `C` are region names declared by the user (declarations not shown) and bound to the region arguments in the instantiated types.

The important thing here is that the "factory method" must really be a factory method and not, for example, just fetch some object reference from the heap and store the same one into each slot of the new linear container. The framework author can enforce this requirement by judicious use of a *method region parameter*. Notice that in line 10, the return type of the factory method is written in terms of a parameter `R` that is in scope only in that method. Further, no reference of type `ListNode<R>` enters the method. Therefore, the only way a `ListNode<R>` can escape the method is if it is created inside the method via `new`. To our knowledge, no previous work has shown that region parameters can be used to enforce a restriction that a method must return a fresh object.

Building one linear container from another. If we start with a linear container A, and we create a new linear container B and populate it by copying the reference elements from the slots of A to the slots of B, then B will be linear. An example is creating a tree out of the elements of an array or set.

Lines 4–5 of Figure 4 illustrate how we might implement this strategy in DPJ. They just say that given one object of type `NodeContainer<Node,Cont>` we can create another one. An important special case in DPJ is creating a linear container from an *index-parameterized array*. In DPJ, the index-parameterized array type is an array `A` such that cell `A[`i`]` has a type like `ListNode<[`i`]>` that is parameterized by the integer value i. This guarantees the linearity property for the array, because cell i can never point to type `ListNode<[`j`]>`, for $i \neq j$. However, because the parameterized types are exposed to the rest of the program, it also means that we cannot shuffle the array elements without compromising soundness. (This is exactly the same problem discussed in Section 3, just with array cells rather than fields.) If we construct a linear container by copying in elements from the cells of an index-parameterized array, then we obtain a container that is linear, but on which we can also perform linearity preserving operations, such as reshuffling, that were prohibited for the original array, by doing them *internally* within the framework.

```
1  public interface NodeContainer<region Node,Cont> {
2
3      /* One linear container from another  */
4      public NodeContainer(NodeContainer<Node,Cont> cont)
5          writes Cont;
6
7      /* Controlled creation of contents */
8      public NodeContainer(NodeFactory fact, int size) writes Cont;
9      public interface NodeFactory {
10         public <region R>ListNode<R> create(int i) pure;
11     }
12
13     /* Backed by set */
14     public void add(ListNode<Node> elt) writes Cont;
15
16     /* Data parallel operation on all elements */
17     public void performOnAll(Operation<Node> op)
18         reads Cont writes Node:*;
19     public interface Operation<region Node> {
20         public void operateOn(final ListNode<Node> elt)
21             writes elt;
22     }
23
24 }
```

Figure 4. Framework API for an abstract linear list container.

Backing the container with a set. Line 14 of Figure 4 illustrates the third strategy: we just provide a standard `add` method for the container, but require that any implementation of that method be backed by a set. This is most useful for a linear container that actually is a set, where this backing happens "automatically." For a non-set container such as a tree or array, the implementer must choose: either back insertion with a set (causing extra runtime overhead), or "implement" the `add` method by throwing an `UnsupportedOperationException`, which is effectively a runtime check that this operation never occurs.

4.2.2 Using Linearity to Reason About Effects

Lines 17–22 of Figure 4 show the part of the API that allows the user to define a method and then pass that method into the container to be applied in parallel to all contained objects. For example, given reference `cont` of type `NodeContainer<N,C>`, the user could do this:

```
public class MyOperation implements NodeContainer.Operation<N> {
  public void operateOn(ListNode<N> elt) writes elt {
    ++elt.data;
  }
}
cont.performOnAll(new MyOperation());
```

This code increments in parallel the `data` field of each of the objects stored in `cont`.

We have carefully written the API so that any user-supplied method updates at most the object it is applied to, and does not (1) follow the cross links to read or write a different object (as illustrated in Figure 3); or (2) update any other shared state (such as `static` variables). In the definition of the abstract `operateOn` method in the `Operation` interface (lines 20–21 of Figure 4), we specify the effect as `writes elt`. The DPJ type system requires that any user-supplied method implementing `operateOn` must have a declared effect that is a *subeffect* of `writes elt`. Updating the `data` field as shown above is legal, because `data` is declared "`in this`" inside `ListNode`, which becomes "`in elt`" (because `elt` is bound to `this`) in the scope of `operateOn`. However, following the `next` field to update `data` of a different object is *not* legal: because the `next` field has type `ListNode<*>`, the effect of that update is `writes *`, which is not a subeffect of `writes elt` and so is not allowed.

4.2.3 Using Hidden Regions for Strong Disjointness

The effect control strategy shown in Figure 4 works well when we want to create an object or graph of objects all at once, then update its fields in parallel with values (including immutable objects). However, using `this` as a parameter prevents us from adding a previously-constructed *mutable object* (i.e., one that supports write effects on its members) as a member of another object, as shown below:

```
class A<region R> {}
class B<region R> { A<this> x = new A<this>(); }
class C<region R> { A<this> x; }
B<Root> b = new B<Root>();
C<Root> c = new C<Root>();
/* Illegal, can't assign A<c> to A<b> */
b.x = c.x;
```

This deficiency can be severe when the user wants to use the mutable result of one computation phase in a subsequent phase, as illustrated in our Monte Carlo example (Section 6).

If the framework API excludes the `add` method (line 14 of Figure 4), and requires that the container contents be created under control of the framework (i.e., the first two linearity strategies), then it can allow more flexible effects by writing the `Operation` interface with a region parameter, and allowing write effects under that parameter:

```
public interface Operation {
  public <region R>void operateOn(final ListNode<R> elt)
    writes R:*, elt;
```

In this way the user can write to objects not parameterized by `this`, while the framework retains control over user-supplied effects.

This strategy is sound because the framework ensures that a fresh object is created for each slot, so the framework can treat each slot i as having a region r_i bound to its parameter, such that for any $i \neq j$, r_i`:*` and r_j`:*` are disjoint (i.e., neither r_j is under r_i nor vice versa in the region tree). In this case we say that the r_i are *strongly disjoint*. Strong disjointness of regions in the slot types implies linearity (because strongly disjoint regions imply different types and therefore different objects), but the converse is not true.

In this approach, the user never sees the strongly disjoint regions r_i, and interacts with them only through the region parameter `R` in the `Operation` API. We call this strategy using "hidden regions." It generalizes the strategy shown in Figure 2, where we argued that `one` and `two` would have (strongly disjoint) regions `One` and `Two` in their dynamic types, even if the static type of both were `Node<*>`. The correctness of this strategy is subtle and, in fact, we initially considered using it for a container supporting `add` backed by a set. That is not sound because in that case we have no control over the actual regions in the objects coming into the set. However, for data structures like arrays that are not backed by a set, it works well. Further, the strategy of creating one container from another (Section 4.2.1) still works, so long as the first container is required to have strongly disjoint regions in its slot types.

4.3 Getting More Flexibility

While the list node container discussed in the previous section is useful, it is too specialized. We now show how to make the example generic. There are two issues to consider: generic effects and generic types.

4.3.1 Making the Effects Generic

The first thing that is too restrictive is the bound on the effects of the user-defined `operateOn`. For instance, what if the user wants to specify an `operateOn` method that reads some other region that is disjoint from `elt` for all `ListNode` objects? That is safe and should be allowed, but it is disallowed by the effect specification `writes elt` in the API.

To solve this problem, we use effect polymorphism [20]. We give the `Operation` interface an effect parameter `E` (similar to a region parameter, but it specifies an effect) that becomes bound to an actual effect when the interface is instantiated into a type. To make this strategy work, we need to solve two problems that have not been solved in previous work: (1) constraining the effect arguments so that the effects of invoking the user-supplied method on different objects are noninterfering; and (2) ensuring soundness of subtyping when we add effect parameters.

```
1  public interface Operation<region Node, effect E> {
2      public void operateOn(final ListNode<Node> elt)
3          writes elt effect E;
4  }
5
6  public <effect E | effect E # writes Node:* effect E>
7    void performOnAll(Operation<Node, effect E> op)
8      reads Cont writes Node:* effect E;
```

Figure 5. Making the effects of the `Operation` interface generic.

Constraining the effect arguments. Obviously the framework cannot let the effect variable `E` become bound to an arbitrary effect in the user's code, because then we would be back to the problem of a user-supplied method with unregulated effects. Instead, we introduce an *effect constraint* that restricts the effect of the user-supplied method.

Figure 5 shows how to write the effect variables and constraints. We define the `Operation` interface (line 1) with one region parameter `Node` and one effect variable `E`. We define the `performOnAll` method (lines 6–8) with a *method effect parameter* `E`. After the parameter declaration is a vertical bar, followed by a constraint specifying that the effect bound to `E` must be noninterfering with `writes Node:* effect E`. This constraint ensures that (1) the supplied effect will not interfere with the effect `writes Node:*` of updating the nodes; and (2) the supplied effect will not interfere with itself. This means that `E` must either be a read-only effect, or it must be an effect such as a set insert that is declared to commute with itself [5].

As an example, here is a user-supplied method that puts all the `ListNode` objects in region `N` and reads region `G` to initialize all the objects with the same global value:

```
public class MyOperation implements
  NodeContainer.Operation<N,reads G> {
    public void operateOn(ListNode<N> elt) reads G writes elt {
      /* Assume global is stored in G */
      elt.data = global;
    }
}
cont.<reads G>performOnAll(new MyOperation());
```

Notice that the constraints are satisfied. First, `G` and `N` are different regions, so `reads G` does not interfere with the effect `writes N:*` of updating the nodes. Second, `reads G` is a read-only effect, so it is noninterfering with itself.

Soundness of subtyping. Once we add class types like `C<`E`>`, where E is an effect argument, we need a rule for when `C<`E_1`>` is a subtype of `C<`E_2`>`. We can then easily extend the rule to handle subclasses, using the same technique as for DPJ with region parameters only [4]. We could require that E_1 and E_2 be identical effects, but this would be unnecessarily restrictive. Instead, we let E_1 be a *subeffect* of E_2. This is similar to the approach we took in DPJ with region parameters only, where we defined subtyping as equivalence up to inclusion of regions [4, 5].

With this approach, soundness of effect falls out naturally if we can show *type preservation*, i.e., that the dynamic types of object references always agree with the static types of variables that hold them. This can be quite subtle, however. For example, if we assign `C<writes r>` to `C<writes *>`, the weaker effect tells us that we don't know the real region, and we have to treat any uses of `C`'s effect parameter as `writes *` when operating through a reference of the weaker type. However, we have to be careful not to allow assignments that would violate type preservation. For example, if class `A<effect E>` has a field `f` of type `B<effect E>`, then we *cannot* simply give `f` the type `B<writes *>` as a member of `A<writes *>`. Instead, we must use the type `B<writes P>`, where `P` is a fresh region parameter (called a *capture parameter*). This is similar to how Java handles generic wildcards and how DPJ already handles partially specified regions [4, 5]. The new part is that we are capturing effects by capturing their component regions. We formalize this notion in the next section.

4.3.2 Making the Type Generic

The second thing that is too restrictive is that we made the class specialized to list nodes. Instead, we would like to write a generic linear container class `LinearContainer<type T, region Cont>`. Notice, however, that there are two places where we used the region parameter of `ListNode` to write the API. First, in writing the `NodeFactory` interface (line 10 of Figure 4), we used a method-local parameter `R` in the return type of `create`. Second, in writing the effect of `performOnAll` (lines 6–8 of Figure 5), we used the region `Node` to write both the effect constraint and the effect of updating the contained objects. If we just replaced these types with an ordinary type variable `T`, then we would not be able to write the node factory pattern at all, we would not be able to constrain the effect `E` properly, and we would be forced to use a more conservative effect (such as `writes *`) for the effect of `operateOn`.

To solve this problem, we introduce the notion of a *type region parameter*, which works as follows:

1. In declaring a type variable `T`, we can write `type T<region R>`, where `R` declares a fresh parameter. This is analogous to declaring a parameter in a class definition. When a type t becomes bound to `T`, t must have at least one region argument, and `R` represents the first region argument.
2. We write uses of the variable `T` as `T<`r`>`, where r is a valid region in scope. `R` itself is valid (because it was declared in the type variable). `T<R>` represents the unmodified type provided as an argument to the variable, while `T<`r`>` represents the same type with the region in its first argument position replaced by r.

For convenience, a bare use of `T` is allowed within the class body, and this is equivalent to `T<R>`. We can also write n parameters (`T<region` $\mathtt{R}_1,\ldots,\mathtt{R}_n$`>`) and arguments ($t$`<`$r_1,\ldots,r_n$`>`), for $n \geq 1$. In this case the argument must have at least n parameters, and the first n region arguments are captured, starting from the left.

Figure 6 shows how to write the final linear container API with generic effects and generic types. Line 1 declares a `LinearContainer` interface with one type parameter `T` and one region parameter `Cont`. The type parameter has one region parameter `Elt` that names the first region argument of the type bound to `T`. In line 10, we write `T<R>` to require that the return type of `create` have the method region parameter `R` as its first region argument. In lines 15 and 17, the region `Elt` is available to constrain the effect variable `E` and to write the effects of `performOnAll`.

We could also have followed the C++ mechanism called *template template parameters* [30], allowing the user to provide a template C and a region R as separate arguments, and having the framework put them together to construct the type C`<`R`>`. We did not adopt this approach because, in addition to the fact that Java does support templates, it obscures the relationship between the type and its region argument in the framework API.

```
1  public interface LinearContainer<type T<region Elt>,region Cont> {
2
3       public LinearContainer(LinearContainer<T,Cont> cont)
4           writes Cont;
5
6       public <effect E | effect E # writes Cont effect E>
7         LinearContainer(Factory<T, effect E> fact, int size)
8           writes Cont effect E;
9       public interface Factory<type T<region Elt>, effect E> {
10          public <region R>T<R> create(int i) effect E;
11      }
12
13      public void add(T elt) writes Cont;
14
15      public <effect E | effect E # reads Cont writes Elt:* effect E>
16        void performOnAll(Operation<T,effect E> op)
17          reads Cont writes Elt:* effect E;
18      public interface Operation<type T, effect E> {
19          public void operateOn(final T elt) writes elt effect E;
20      }
21
22 }
```

Figure 6. API for an abstract linear container with generic types and effects.

4.4 Verifying the Framework Implementation

```
1  public class LinearArray<type T<region Elt>, region Cont>
2    implements LinearContainer<T,Cont> {
3
4       /* Internal array representation */
5       private ArrayList<T,Cont> elts;
6
7       public <effect E | effect E # writes Elt:* effect E>
8         void performOnAll(Operation<T,effect E> op)
9           reads Cont writes Elt:* effect E {
10          foreach (int i in 0, elts.size()) {
11              op.operateOn(elts.get(i));
12          }
13      }
14
15 }
```

Figure 7. Array implementation of a linear container (partial).

Having studied the framework API, we now focus on how to write a correct framework implementation. Figure 7 shows what the inside of `performOnAll` might look like, in the case of an array implementation of `LinearContainer`. We have chosen to represent the array internally as an `ArrayList`, as shown in line 5. The `performOnAll` method uses the DPJ `foreach` construct (line 10) to iterate in parallel over the elements of the `ArrayList` and apply the user-supplied operation to each of its elements.

To verify noninterference (and in this case, deterministic parallelism), it suffices to show that for any two distinct iterations of the `foreach`, the reference values `elts.get(i)` are distinct. We can formalize this statement as a logical predicate:

$$\frac{I \neq J}{\text{disjoint-ref}(\texttt{elts.get(i)}_I, \texttt{elts.get(i)}_J)}$$

Here I and J represent loop iterations. Given just this predicate, and the way we wrote the framework API, we can push through a proof of noninterference using the DPJ type system rules, extended to support effect parameters and type region parameters. We formally state these rules in the following section. We also formally state that the rules are *sound*, in the sense that once we push through the proof we really do get noninterference.

Now, how do we show the predicate? In the case of our array example, we must show two things: (1) for distinct values i, `elts.get(`i`)` is distinct; and (2) `i` attains distinct values i on distinct iterations of the `foreach`. The first statement follows from

$$
\begin{array}{rcl}
\textit{program} & ::= & \textit{class}^{*}\ e \\
\textit{class} & ::= & \texttt{class}\ C\texttt{<}\tau\texttt{<}\rho\texttt{>},\rho,\eta|K\texttt{>}\ \{\ \textit{field}^{*}\ \textit{method}^{*}\ \} \\
K & ::= & \eta \mathbin{\#} E \\
\textit{field} & ::= & T\ f\ \texttt{in}\ R \\
\textit{method} & ::= & T\ m(T\ x)\ E\ \{\ e\ \} \\
R & ::= & z \mid \rho \mid * \\
T & ::= & C\texttt{<}T,R,E\texttt{>} \mid \tau\texttt{<}R\texttt{>} \mid \texttt{void} \\
E & ::= & \emptyset \mid \texttt{reads}\ R \mid \texttt{writes}\ R \mid \eta \mid E \cup E \\
e & ::= & \texttt{let}\ z = e\ \texttt{in}\ e \mid \texttt{this.}f \mid \texttt{this.}f = z \mid \\
 & & z.m(z) \mid z \mid \texttt{new}\ C\texttt{<}T,R,E\texttt{>} \\
z & ::= & \texttt{this} \mid x
\end{array}
$$

Figure 8. Syntax of the core language. C, τ, ρ, η, f, m, and x are identifiers.

the inductive argument we made in Section 4.2.1 about maintaining linearity. This argument can easily be formalized, but we do not do so here. The second statement follows from the semantics of `foreach` in DPJ [5]. More generally, one would follow the same two-pronged strategy to discharge the distinctness predicate for an iterative traversal over an arbitrary linear container: first show linearity of slots, and then show uniqueness of traversal over the slots. To use the hidden regions strategy (Section 4.2.3), we would do the same thing with a predicate disjoint-rgn(z, z') saying that the regions in the types of z and z' are strongly disjoint.

In the case of a recursive traversal (such as over a tree), the problem is more difficult. Here it is not sufficient to prove a predicate like "expression `e` refers to distinct references on distinct iterations"; instead, we need a predicate like "all references in the left subtree are distinct from all references in the right subtree." The DPJ type system supports predicates like this [4, 5], but only by constraining the types such that we cannot rebalance the tree soundly, for the reasons discussed in Section 3. Here we could give up on using the type system to prove disjointness of effect inside the framework and verify it some other way, e.g., through more general program logic [14, 22] or testing. In this case, the "extra" predicate we need is a predicate about disjoint *effects*. Potentially interesting questions here are (1) exactly what such a proof would look like and (2) whether any extensions to our type system could help in constructing such proofs, by providing further "glue" between the different forms of verification. We leave these questions to future work.

In any event, once the framework implementer verifies the inside, the user never has to see or even know about how the verification occurred. From the user's point of view, if the program type checks, then the noninterference property holds. We can thus think of the techniques presented here as making DPJ into an *extensible language*. By writing a suitable API, and doing appropriate proofs, the framework writer can add new capabilities for parallel operations that provide the same guarantees as if those capabilities had been built in as first-class parts of the language. This makes DPJ much more powerful than if the only available verification mechanism were the type system itself.

5. Formal Elements

In this section we formalize the ideas developed in the previous section using a core language that is simple enough to formalize yet illustrates all the essential features.

5.1 Syntax

Figure 8 shows the syntax for the core language. A program consists of zero or more class definitions and an expression to evaluate. A class has one type parameter τ, one region parameter ρ, and one effect parameter η. The type parameter has a region parameter that captures the region argument of the type bound to it. There is one

Judgment	**Meaning**	**Judgment**	**Meaning**
$\rhd$ *program*	Valid program	$\rhd$ *class*	Valid class
$\Gamma \rhd$ *field*	Valid field	$\Gamma \rhd$ *method*	Valid method
$\Gamma \rhd R$	Valid region	$\Gamma \rhd R \subseteq R'$	R included in R'
$\Gamma \rhd R \mathbin{\#} R'$	R disjoint from R'	$\Gamma \rhd T$	Valid type
$\Gamma \rhd T \leq T'$	T a subtype of T'	$\Gamma \rhd E$	Valid effect
$\Gamma \rhd E \subseteq E'$	E a subeffect of E'	$\Gamma \rhd E \mathbin{\#} E'$	Noninterfering effects
$\Gamma \rhd e : T, E$	e has type T and effect E		

Figure 9. Type judgments for the core language. We extend the judgments to groups of things (e.g., $\Gamma \rhd$ *field**) in the obvious way.

$$
\text{(PROGRAM)}\quad \frac{\rhd \textit{class}^{*} \quad \emptyset \rhd e : T, E}{\rhd \textit{class}^{*}\ e}
$$

$$
\text{(CLASS)}\quad \frac{\begin{array}{c}\Gamma = \{(\texttt{this}, C\texttt{<}\tau\texttt{<}\rho\texttt{>},\rho',\eta\texttt{>}), \tau\texttt{<}\rho\texttt{>}, \rho, \rho', \eta, K\} \\ \Gamma \rhd K \quad \Gamma \rhd \textit{field}^{*} \quad \Gamma \rhd \textit{method}^{*}\end{array}}{\rhd \texttt{class}\ C\texttt{<}\tau\texttt{<}\rho\texttt{>},\rho',\eta|K\texttt{>}\ \{\ \textit{field}^{*}\ \textit{method}^{*}\ \}}
$$

$$
\text{(CONSTRAINT)}\quad \frac{\Gamma \rhd \eta \quad \Gamma \rhd E}{\Gamma \rhd \eta \mathbin{\#} E} \qquad \text{(FIELD)}\quad \frac{\Gamma \rhd T \quad \Gamma \rhd R}{\Gamma \rhd T\ f\ \texttt{in}\ R}
$$

$$
\text{(METHOD)}\quad \frac{\begin{array}{c}\Gamma \rhd T_x \quad \Gamma' = \Gamma \cup \{(x, T_x)\} \quad \Gamma' \rhd T_r, E \\ \Gamma' \rhd e : T', E' \quad \Gamma' \rhd T' \leq T_r \quad \Gamma' \rhd E' \subseteq E\end{array}}{\Gamma \rhd T_r\ m(T_x\ x)\ E\ \{\ e\ \}}
$$

Figure 10. Typing of program elements.

$$
\text{(REGION-PARAM)}\quad \frac{\rho \in \Gamma}{\Gamma \rhd \rho} \qquad \text{(REGION-VAR)}\quad \frac{(z, T) \in \Gamma}{\Gamma \rhd z}
$$

$$
\text{(REGION-STAR)}\quad \frac{}{\Gamma \rhd *} \qquad \text{(REGION-CAPTURE)}\quad \frac{(z, C\texttt{<}T, *, E\texttt{>}) \in \Gamma}{\Gamma \rhd \text{rgn}(z)}
$$

$$
\text{(DISJOINT-REF)}\quad \frac{\text{disjoint-ref}(z, z')}{\Gamma \rhd z \mathbin{\#} z'} \qquad \text{(DISJOINT-RGN)}\quad \frac{\text{disjoint-rgn}(z, z')}{\Gamma \rhd \text{rgn}(z) \mathbin{\#} \text{rgn}(z')}
$$

Figure 11. Regions. $\Gamma \rhd R \mathbin{\#} R'$ is symmetric.

effect constraint $K = \eta \mathbin{\#} E$ specifying that the effect argument bound to η must be disjoint from the effect E.

A region is a final variable z, a region parameter ρ, or $*$ indicating an unspecified region. A type either instantiates a named class with a type, region, and effect; or it instantiates a type parameter with a region; or it is `void`, indicating an unused type parameter. An effect is a possibly empty union of read effects, write effects, and effect parameters.

5.2 Static Semantics

Judgments. Figure 9 shows the judgments defining the static semantics for the core language. The judgments are defined with respect to an environment Γ containing zero or more of the following elements: (z, T) means that variable z has type T; $\tau\texttt{<}\rho\texttt{>}$ means that type parameter τ is in scope with region parameter ρ; ρ means that region parameter ρ is in scope; η means that effect parameter η is in scope; and $\eta \mathbin{\#} E$ means that effect variable η is constrained to be noninterfering with effect E.

Program elements. Figure 10 shows how to make the judgments for typing of top-level program elements. In rule CLASS, we form the environment Γ containing `this`, the parameters, and the effect constraint, and then we check the effect constraint and the class body in Γ. In rule METHOD, we form the environment Γ' by adding the formal parameter x with its type, then we check the formal parameter type, the method body, and the return type; and we check that the type and effect of the method body are a subtype and subeffect of the return type and declared effect.

Regions. Figure 11 gives the rules for valid regions and disjoint regions. rgn(z) represents the (statically unknown) region in the dynamic type of z, when a $*$ appears in the region of the type. As explained in Section 4.4, the predicate disjoint-ref(z, z') means

$$\text{(TYPE-CLASS)}\quad \frac{\begin{array}{c}\texttt{class}\ C{<}\tau{<}\rho{>}, \rho', \eta | \eta \# E'{>}\{\textit{field}^*\ \textit{method}^*\} \in \textit{program}\\ \Gamma \rhd T, R, E \quad \Gamma \rhd E \# \phi_{C{<}T,R,E{>}}(E')\end{array}}{\Gamma \rhd C{<}T, R, E{>}}$$

$$\text{(TYPE-PARAM)}\quad \frac{\tau{<}\rho{>} \in \Gamma \quad \Gamma \rhd R}{\Gamma \rhd \tau{<}R{>}} \qquad \text{(TYPE-VOID)}\quad \frac{}{\Gamma \rhd \texttt{void}}$$

$$\text{(SUBTYPE-CLASS)}\quad \frac{\Gamma \rhd R \subseteq R' \quad \Gamma \rhd E \subseteq E'}{\Gamma \rhd C{<}T, R, E{>} \le C{<}T, R', E'{>}}$$

$$\text{(SUBTYPE-PARAM)}\quad \frac{\Gamma \rhd R \subseteq R'}{\Gamma \rhd \tau{<}R{>} \le \tau{<}R'{>}}$$

Figure 12. Types. $\Gamma \rhd T \le T'$ is reflexive and transitive.

$$\text{(NI-EMPTY)}\quad \frac{}{\Gamma \rhd \emptyset \# E} \qquad \text{(NI-UNION)}\quad \frac{\Gamma \rhd E \# E'' \quad \Gamma \rhd E' \# E''}{\Gamma \rhd E \cup E' \# E''}$$

$$\text{(NI-RD)}\quad \frac{}{\Gamma \rhd \texttt{reads}\ R \# \texttt{reads}\ R'} \qquad \text{(NI-RD-WR)}\quad \frac{\Gamma \rhd R \# R'}{\Gamma \rhd \texttt{reads}\ R \# \texttt{writes}\ R'}$$

$$\text{(NI-WR)}\quad \frac{\Gamma \rhd R \# R'}{\Gamma \rhd \texttt{writes}\ R \# \texttt{writes}\ R'} \qquad \text{(NI-PARAM)}\quad \frac{\eta \# E \in \Gamma \quad \Gamma \rhd E' \subseteq E}{\Gamma \rhd \eta \# E'}$$

Figure 13. Noninterfering effects. $\Gamma \rhd E \# E'$ is symmetric.

$$\text{(LET)}\quad \frac{\Gamma \rhd e : T, E \quad \Gamma \cup \{(x, T)\} \rhd e' : T', E'}{\Gamma \rhd \texttt{let}\ x = e\ \texttt{in}\ e' : T'[x \leftarrow *], E \cup E'[x \leftarrow *]}$$

$$\text{(ACCESS)}\quad \frac{(\texttt{this}, T) \in \Gamma \quad \text{field}(T, f) = T'\ f\ \texttt{in}\ R}{\Gamma \rhd \texttt{this}.f : T', \texttt{reads}\ R}$$

$$\text{(ASSIGN)}\quad \frac{\begin{array}{c}(\texttt{this}, T) \in \Gamma \quad (z, T') \in \Gamma\\ \text{field}(T, f) = T''\ f\ \texttt{in}\ R \quad \Gamma \rhd T' \le T''\end{array}}{\Gamma \rhd \texttt{this}.f = z : T', \texttt{writes}\ R}$$

$$\text{(INVOKE)}\quad \frac{\begin{array}{c}(z, T) \in \Gamma \quad (z', T') \in \Gamma\\ \text{method}(T, m) = T_r\ m(T_x\ x)\ E\ \{\ e\ \}\\ \text{capture}(z, T) = (T'', \rho) \quad \Gamma \cup \rho \rhd T' \le \phi_{T''}(T_x)\end{array}}{\Gamma \rhd z.m(z') : \phi_T(T_r), \phi_T(E)}$$

$$\text{(VARIABLE)}\quad \frac{(z, T) \in \Gamma}{\Gamma \rhd z : T, \emptyset} \qquad \text{(NEW)}\quad \frac{\Gamma \rhd C{<}T, R, E{>}}{\Gamma \rhd \texttt{new}\ C{<}T, R, E{>} : C{<}T, R, E{>}, \emptyset}$$

Figure 14. Expressions. field(T, f) means the defined field f of the class named in T (which must be a class type). method(T, m) means the defined method m of the class named in T. capture$(z, T) = (T'', \rho)$ means that ρ is a fresh parameter, and T becomes T'' after replacing (a) its type region parameter argument with ρ, if it is $*$; and (b) its region parameter argument with rgn(z), if it is $*$.

that variables z and z' evaluate to distinct object reference values at runtime; while the predicate disjoint-rgn(z, z') says that z and z' have different regions in their types. The warrant for these predicates is provided from outside the type system.

Types. Figure 12 shows the rules for checking types. The interesting rules are TYPE-CLASS and the two subtyping rules. In TYPE-CLASS we check the validity of the type, region, and effect used to instantiate the type. We use the translation mapping ϕ_T (defined below) to instantiate the effect, and then we check the effect constraint. In the subtyping rules, we allow one class type to be a subtype of another if the regions are related by inclusion and the effects related by subeffects.

Effects. The rules for valid effects and subeffects are identical to the rules given in [5], with the addition of effect parameters. For completeness we state the rules in full in an Appendix. Figure 13 gives the rules for noninterfering effects. Rule NI-PARAM says that if the environment guarantees disjointness between a parameter and some effect, then we can infer disjointness of that parameter with any subeffect of the effect.

Expressions. Figure 14 gives the rules for typing expressions. In rule LET, we replace x with $*$ to generate valid types, regions, and effects when the variable x goes out of scope [5, 10]. In rule INVOKE, we use the mapping ϕ_T (defined below) to translate from the callee to the caller context. As discussed in Section 4.3.1, we need to capture any regions or effects containing $*$ to maintain soundness of subtyping. We represent the captured region parameter argument as rgn(z) in order to apply the rule DISJOINT-RGN from Figure 11.

Translation mapping. The mapping ϕ_T translates a type, region, or effect from the context in which it is defined to the context of its use via the type $T = C{<}T', R, E{>}$. It is the same as the context translation described in [4, 5], except that we need to handle effect parameters and type region parameters, as well as plain region parameters:

1. *Types.* To translate a class type, we translate its arguments: $\phi_T(C'{<}T'', R', E'{>}) = C'{<}\phi_T(T''), \phi_T(R'), \phi_T(E'){>}$. To translate a type parameter, we use the instantiating type T, but we replace its region argument with the parameter's region argument, after translating it: $\phi_T(\tau{<}R'{>}) = C{<}T', \phi_T(R'), E{>}$. Finally, $\phi_T(\texttt{void}) = \texttt{void}$.
2. *Regions.* Let the type parameter of C be $\tau{<}\rho'{>}$, and let the region parameter of C be ρ. If $T' = C'{<}T'', R', E'{>}$ or $\tau'{<}R'{>}$, then we replace the region parameter of C with R and the region parameter of the type parameter with R': $\phi_T(R'') = R''[\rho \leftarrow R][\rho' \leftarrow R']$. If $T' = \texttt{void}$, then $\phi_T(R'') = R''[\rho \leftarrow R]$, and it is an error for ρ' to appear in R''.
3. *Effects.* Let the effect parameter of C be η. To form $\phi_T(E')$, first apply ϕ_T to all regions appearing in E', and then replace all occurrences of η with E.

5.3 Dynamic Semantics

Execution state. The runtime values are object references o. These are the only entities we would need in an actual implementation; but to formulate and prove soundness results we also need to keep track of dynamic types *dT*, dynamic regions *dR*, and dynamic effects *dE* corresponding to static types, regions, and effects. These entities are defined by the following syntax:

$$\begin{array}{lcl} dR & ::= & o \mid * \\ dT & ::= & C{<}dT, dR, dE{>} \mid \texttt{void} \\ dE & ::= & \emptyset \mid \texttt{reads}\ dR \mid \texttt{writes}\ dR \mid dE \cup dE \end{array}$$

Notice that there are no type, region, or effect parameters in the runtime syntax, because all such parameters are eliminated at runtime via substitution.

The dynamic execution state consists of (1) a heap H, which is a function taking values to objects; and (2) a dynamic environment $d\Gamma$, which is a set of bindings (z, o) meaning that variable z is bound to object reference o. An object is a partial function taking field names to object references. If the function is undefined on all field names, then we say it is a *null object*. We use null objects to avoid having to represent the special type of `null`. In an actual implementation, we can just use the single value `null` for uninitialized reference variables. Every object reference $o \in$ Dom(H) has a type, and we write $H \rhd o$: *dT* to mean that the reference o has type *dT* in the domain of heap H.

Evaluating programs. Figure 15 gives the rules for program evaluation. A program evaluates to value o with heap H and effect dE if its main expression is e, and $(e, \emptyset, \emptyset) \to (o, H, dE)$. Notice that in rules DYN-ACCESS, DYN-ASSIGN, and DYN-NEW, we use the translation mapping ϕ defined in the previous section to

$$\text{(DYN-LET)}\quad \frac{(e, d\Gamma, H) \rightarrow (o, H', dE) \quad (e', d\Gamma \cup \{(x, o)\}, H') \rightarrow (o', H'', dE')}{(\texttt{let}\ x = e\ \texttt{in}\ e', d\Gamma, H) \rightarrow (o', H'', dE \cup dE')} \qquad \text{(DYN-VARIABLE)}\quad \frac{(z, o) \in d\Gamma}{(z, d\Gamma, H) \rightarrow (o, H, \emptyset)}$$

$$\text{(DYN-ACCESS)}\quad \frac{(\texttt{this}, o) \in d\Gamma \quad H \rhd o : C\texttt{<}dT, dR, dE\texttt{>} \quad \text{field}(C, f) = T\ f\ \texttt{in}\ R}{(\texttt{this}.f, d\Gamma, H) \rightarrow (H(o)(f), H, \texttt{reads}\ \phi_{C\texttt{<}dT, dR, dE\texttt{>}}(R))}$$

$$\text{(DYN-ASSIGN)}\quad \frac{(\texttt{this}, o) \in d\Gamma \quad (z', o') \in d\Gamma \quad H \rhd o : C\texttt{<}dT, dR, dE\texttt{>} \quad \text{field}(C, f) = T\ f\ \texttt{in}\ R}{(\texttt{this}.f = z, d\Gamma, H) \rightarrow (o', H \cup \{o \mapsto (H(o) \cup \{f \mapsto o'\})\}, \texttt{writes}\ \phi_{C\texttt{<}dT, dR, dE\texttt{>}}(R))}$$

$$\text{(DYN-INVOKE)}\quad \frac{\begin{array}{c}(z, o) \in d\Gamma \quad (z', o') \in d\Gamma \quad H \rhd o : C\texttt{<}dT, dR, dE\texttt{>} \\ \text{method}(C, m) = T_r\ m(T_x\ x)\ E_m\ \{\ e\ \} \quad (e, \{(\texttt{this}, o), (x, o')\}, H) \rightarrow (o'', H', dE')\end{array}}{(z.m(z'), d\Gamma, H) \rightarrow (o'', H', dE')}$$

$$\text{(DYN-NEW)}\quad \frac{(\texttt{this}, o) \in d\Gamma \quad H \rhd o : dT \quad o' \notin \text{Dom}(H) \quad H' = H \cup \{o' \mapsto \text{new}(C, dT)\} \quad H' \rhd o' : \phi_{dT}(C\texttt{<}T, R, E\texttt{>})}{(\texttt{new}\ C\texttt{<}T, R, E\texttt{>}, d\Gamma, H) \rightarrow (o', H', \emptyset)}$$

Figure 15. Program evaluation. If $f : A \rightarrow B$ is a function, then $f \cup \{x \mapsto y\}$ is the function $f' : A \cup \{x\} \rightarrow B \cup \{y\}$ defined by $f'(a) = f(a)$ if $a \neq x$ and $f'(x) = y$. new(C, dT) is the function taking each field of class C with type T to a null reference of type $\phi_{dT}(T)$.

translate static regions and types to their corresponding runtime representations. Here the use context is given by the runtime type of the object bound to `this` in $d\Gamma$.

5.4 Soundness Results

Dynamic judgments for regions, types, and effects. To state and prove the preservation result, we need to establish runtime judgments for regions, types, and effects corresponding to the static judgments defined in Section 5.2. The rules are nearly identical to their static counterparts. We describe how to generate the rules via simple substitution.

First, replace the rules REGION-CAPTURE, DISJOINT-REF, DISJOINT-RGN, and TYPE-CLASS with the following rules:

$$\text{(NI-WR)}\quad \frac{o \neq o'}{H \rhd \texttt{writes}\ o \mathbin{\#} \texttt{writes}\ o'} \qquad \text{(NI-RD-WR)}\quad \frac{o \neq o'}{H \rhd \texttt{reads}\ o \mathbin{\#} \texttt{writes}\ o'}$$

$$\text{(DYN-TYPE-CLASS)}\quad \frac{\begin{array}{c}\texttt{class}\ C\texttt{<}\tau\texttt{<}\rho\texttt{>}, \rho', \eta | \eta \mathbin{\#} E'\texttt{>}\{\mathit{field}^*\ \mathit{method}^*\} \in \mathit{program} \\ H \rhd dE \mathbin{\#} \phi_{C\texttt{<}dT, dR, dE\texttt{>}}(E')\end{array}}{H \rhd C\texttt{<}dT, dR, dE\texttt{>}}$$

Second, delete the rules REGION-PARAM, TYPE-PARAM, and NI-PARAM (because there are no type, region, or effect parameters at runtime). Third, for every other rule, do the following: (1) append DYN- to the front of the name; (2) replace Γ with H; and (3) replace T with dT, R with dR, and E with dE.

Preservation of type and effect. We first define a valid heap:

Definition 1 (Valid heaps)**.** *A heap H is valid ($\rhd H$) if (1) for each $o \in Dom(H)$, $H \rhd o : dT$, $H \rhd dT$, and $dT = C\texttt{<}dT', dR, dE\texttt{>}$; and (2) for each field $T\ f\ \texttt{in}\ R \in def(C)$, if $H(o)(f)$ is defined, then $H \rhd H(o)(f) : dT''$ and $H \rhd dT''$ and $H \rhd dT'' \leq \phi_{dT}(T)$.*

This definition says that every object reference is well typed with a valid type, and every field of every object is either undefined (causing execution to fail if it is accessed) or contains a reference with a valid type that is bounded by its static type, translated to the dynamic environment.

Next we define $H \rhd d\Gamma \leq \Gamma$ ("$d\Gamma$ instantiates Γ in H"). We write $\phi_{d\Gamma,H}$ as a shorthand for "ϕ_{dT}, where $(\texttt{this}, o) \in d\Gamma$ and $H \rhd o : dT$."

Definition 2 (Instantiation of static environments)**.** *A dynamic environment $d\Gamma$ instantiates a static environment Γ ($H \rhd d\Gamma \leq \Gamma$) if the same variables appear in $d\Gamma$ as in Γ; for each pair $(z, dT) \in \Gamma$ and $(z, o) \in d\Gamma$, $H \rhd o : dT$ and $H \rhd dT \leq \phi_{d\Gamma,H}(T)$; and for each constraint $\eta \mathbin{\#} dE \in \Gamma$, $H \rhd \phi_{d\Gamma,H}(\eta) \mathbin{\#} \phi_{d\Gamma,H}(E)$.*

This definition specifies a correspondence between static typing environments and dynamic execution environments, such that we can use the typing in the static environment to draw sound inferences about execution in the dynamic environment.

Finally, we state the type and effect preservation result:

Theorem 1 (Preservation)**.** *For a well-typed program, if $\Gamma \rhd e : T, E$ and $H \rhd d\Gamma \leq \Gamma$ and $(e, d\Gamma, H) \rightarrow (o, H', dE')$, then $\rhd H'$; $H' \rhd o : dT'$; $H \rhd dT' \leq \phi_{d\Gamma,H}(T)$; $H \rhd dE'$; and $H \rhd dE' \subseteq \phi_{d\Gamma,H}(E)$.*

Noninterference of effect. Now we can prove that expressions with noninterfering static effects are noninterfering at runtime. First we define $\mathcal{R}_f(o, H)$, the region of field f of object $o \in$ Dom(H). This definition formalizes the idea that regions R in the field declarations $T\ f\ \texttt{in}\ R$ partition the heap:

Definition 3 (Region of a field)**.** *If $H \rhd o : C\texttt{<}dT, dR, dE\texttt{>}$ and $T\ f\ \texttt{in}\ R \in def(C)$, then $\mathcal{R}_f(o, H) = \phi_{C\texttt{<}dT, dR, dE\texttt{>}}(R)$.*

Proposition 1. *At runtime, disjoint regions imply disjoint locations. That is, if $H \rhd \mathcal{R}_f(o, H) \mathbin{\#} \mathcal{R}_{f'}(o', H)$, then either $o \neq o'$ or $f \neq f'$.*

Next we state a proposition about the dynamic effects produced by program execution: if we evaluate e and e' with the same dynamic environment $d\Gamma$, and if the two evaluations have noninterfering effects, then the individual read and write effects of e and e' can be arbitrarily interleaved, with identical results:

Proposition 2. *If $(e, d\Gamma, H_0) \rightarrow (o, H_1, dE)$ and $(e', d\Gamma, H_0') \rightarrow (o', H_1', dE')$ and $H_1 \cup H_1' \rhd dE \mathbin{\#} dE'$, then the read and write effects of the two evaluations are pairwise commutative.*

By extending this result to static effects, we obtain the main soundness property of the core language:

Theorem 2. *If $\Gamma \rhd e : T, E$ and $\Gamma \rhd e' : T', E'$ and $\Gamma \rhd E \mathbin{\#} E'$ and $H \rhd d\Gamma \leq \Gamma$ and $(e, d\Gamma, H_0) \rightarrow (o, H_1, dE)$ and $(e', d\Gamma, H_0') \rightarrow (o', H_1', dE')$, then the read and write effects of the two evaluations are pairwise commutative.*

Theorem 2 says that if two expressions have noninterfering *static* effects, then their actual runtime effects are noninterfering as well. Therefore, we can use the static effect information to reason soundly about noninterference at runtime.

6. Evaluation

We have evaluated the techniques discussed above with two goals in mind:

1. Can we use the techniques to write realistic frameworks and user programs? Do any additional issues come up in real frame-

works or user code that present difficulties for the abstract linear container model?

2. What is the user experience of using such an API? How burdensome is it to write the type and effect annotations, and how difficult is it to get the annotations correct?

To perform our evaluation, we first extended the DPJ compiler [5] to support effect variables, effect constraints, and type region parameters as discussed in Sections 4.3 and 5. Then we studied how to (1) use our techniques to write generic array and tree frameworks; and (2) use the frameworks to write two parallel codes: a Monte Carlo simulation algorithm and a Barnes-Hut n-body computation using a spatial octtree. We chose these two algorithms because they exemplify different styles of parallelism: Monte Carlo uses direct loop-style parallelism over arrays, while Barnes-Hut uses recursive, divide-and-conquer parallelism over trees.

6.1 Array and Tree Frameworks

We focused on the framework operations needed for the two benchmarks but ensured that the operations themselves were *general*, i.e., were not specifically tied to the needs of the benchmarks, as discussed below. Adding more operations is not difficult.

Parallel array framework. We implemented a framework called `DPJLinearArray` with an interface similar to a subset of the ParallelArray API for Java [1]. The API supports the following operations:

1. A `create` method that creates an array with a user-supplied factory method, as discussed in Section 4.2.1 and shown in its final generic form in Figure 6.
2. A `withMapping` method that maps one array to another, element by element, with a user-supplied mapping function. We provide an indexed (taking an index variable) and an unindexed form of the mapping, as ParallelArray does. As in the factory method pattern, we use a method region parameter `R` to ensure that the mapping function creates a new output object for each element, and the mapping function is allowed to write under `R`.
3. A `reduce` method that reduces the array to an object, given a starting element and a user-specified `Reducer` that combines two elements into one. Following the hidden regions pattern discussed in Section 4.2.3, the two elements coming into the `Reducer` method are parameterized by method region parameters `R1` and `R2`, and the user-supplied method is allowed to write under these parameters. Using distinct parameters ensures that the `Reducer` cannot violate strong disjointness, e.g., by storing one object into a field of the other.

The framework implementation is a thin wrapper that uses a `ParallelArray` instance internally to provide all the operations.

Parallel tree framework. We also wrote a framework that provides a tree of user-specified arity (i.e., each inner node has at most `arity` children) with data of generic type `T` stored in every node. The API supports the following operations:

1. A `buildTree` method that takes a `DPJLinearContainer elts` of objects of type `T` and a positive `arity` and inserts the bodies into the leaves of the tree. The user provides an `index` function that, given a `T` to insert, a `T` at the current (inner or leaf) node, and a `T` at the parent node (if any) of the current node, says which of the children of the current node to follow next when inserting the object in the subtree rooted at the current node. The framework creates the inner nodes as necessary and populates each one with a fresh object of type `T`, using a user-specified factory method.
2. A `visitPO` method that recursively does a parallel postorder tree traversal. As shown in Figure 16, this method takes a user-supplied `visit` method that, given a `T` object at the current node and an ArrayList of `V` (result) objects produced from visiting the children (or `null` if the current node is a leaf), produces a `V` object for this node. Again we use two region parameters, `R1` and `R2`, to ensure that strong disjointness of the `T` objects is preserved by the traversal.

```
public class DPJLinearTree<type T<region Elt>, region Cont> {

    public <effect E | effect E # reads Cont writes Elt:* effect E>
      double visitPO(POVisitor<T, effect E> visitor)
        reads Cont writes Elt:* effect E { ... }

    public interface POVisitor<type T<region Elt>,
                               type V<region VR>, effect E> {
        public <region R1, R2> V<R2>
          visit(final T<R1> data,
                final ArrayList<V<R2>, Cont> childResults)
            reads Cont writes data:*, R1:*, R2:* effect E;
    }
}
```

Figure 16. Postorder visitor from region-based spatial tree.

6.2 Application Code

Monte Carlo simulation. We studied the Monte Carlo simulation benchmark from the Java Grande suite [26]. The computation contains three parallelizable loops: the first one creates `Task` objects; the second one iterates over the objects to compute a return rate for each one; and the third one reduces the return rates into a cumulative average.

We parallelized all three loops using `DPJLinearArray`. For the first loop, we used the indexed form of `withMapping`. Apart from writing to the `Task` object itself (which does not have to be reported), the effect of the `Task` constructor is read-only, so it can be validly used for aggregate array creation, as shown in line 6 of Figure 6.

For the second loop, we used the unindexed `withMapping`. We wrote a mapping function that takes a `Task<Tasks>` object to a `Result<R>` object, where `Tasks` is a declared region name, and `R` is the method parameter provided by the framework. The computation in the mapping function writes to `R`.

For the third loop, we wrote a `Reducer` that takes two objects of type `Result<R>`, reads the accumulated sum out of both, adds them, stores the result in the first one, and returns it. The write effect is bounded by `writes R`, as required in the API. We could also have avoided the write effect entirely by creating a new object and returning it, but that would be less efficient.

Barnes-Hut center of mass computation. Next we studied the Barnes-Hut n-body simulation [25], which uses an octtree (eight-ary tree) to represent three-dimensional space hierarchically, storing the bodies in the leaves. We focused on the center-of-mass computation, which recurses down the tree in parallel and computes, for each node, the center of mass of the subtree rooted at that node. Proving noninterference is nontrivial because the computation writes into each node as it traverses it, so the computation requires that the traversal is over a tree. Because of this fact, the center of mass computation is hard to do efficiently in baseline in DPJ; we discuss this point further in Section 6.3 below. It would be straightforward to parallelize the force computation using the same array-based techniques as we used for Monte Carlo, but for lack of time we have not done that.

We wrote a program that builds a tree and performs a center of mass computation for a binary tree computation in one-dimensional (1-D) space. 1-D space simplifies the computation, without changing the essential patterns of parallelism. We instantiated `DPJLinearTree` with a `Node` class that has subclasses `Cell` for the inner node data and `Body` for the leaf data, similarly to both the original and Splash-2 versions of Barnes-Hut [25]. To build the tree, we wrote an `index` method that puts each inserted node in the left or right subtree based on its position, and a factory method that constructs fresh `Cell` objects for each inner node in the tree. To compute the center of mass, we wrote a `postOrderVisitor` that computes the average position and total mass for the bodies in the subtree rooted at each inner node stores them at the node. This visitor returns a pair of `double` values (for type `V` in the API) for the average position and total mass at the current node.

6.3 Discussion of Evaluation Results

Difficulties for realistic frameworks. Our experience shows that the framework techniques in this work can be used to write realistic parallel algorithms. For example, Barnes Hut uses a complex traversal pattern with in-place updates for both tree construction and center-of-mass computation. For these codes, we did not find any significant challenges over and above the framework API we discussed in Section 4. In the future, we could also easily support other operations, such as ParallelArray's filter and apply.

One question left open by our evaluation is whether we could support some of ParallelArray's operations for combining arrays, for instance concatenating two arrays. It is hard to do this and maintain linearity. The best we could do is probably to back the array with a set and provide set union; but in this case we would (1) pay the overhead of maintaining a set and (2) lose the benefit of strong disjointness.

Framework user experience. We found that the annotations required to support the frameworks shifts much of the burden of reasoning about safety from framework *users* to framework *designers*. First we found that getting the region and effect annotations correct for the framework was sometimes tricky. However, the framework user just has to use the API correctly, and the use is automatically checked by the DPJ compiler. Second, most of the complexity introduced by the effect variables (including the effect constraints) was in the framework API itself. The user arguments to effect variables were simple: either `pure` or one or two read effects. We expect that many of the method effect arguments could be inferred (similarly to how Java infers generic method arguments and DPJ infers method region arguments), but we leave this to future work.

It is also instructive to compare the user experience for these algorithms written using frameworks to the corresponding ones using baseline DPJ, as presented in [5]. For Monte Carlo, we had used an index-parameterized array to guarantee strong disjointness in the first two loops, by making the `Task` and `Result` types parameterized by the index `i`. For the third loop, we encapsulated the reduction sum in a method implemented with locks and declared that method `commutative`.

Similarly, we could use baseline DPJ to parallelize the center of mass computation in Barnes-Hut. However, we would have to give each tree node a distinct type and *recopy the bodies on insertion into the tree*, because we cannot soundly change the type of a reference in DPJ, as discussed in Section 3. We could support such "ownership transfer" with runtime reference counting [2], but this would add its own overhead.

Overall, the advantages of the framework approach are (1) simplifying the DPJ types exposed to the user, by avoiding index parameterized arrays or recursive types; (2) eliminating low-level code for common patterns such as reductions; and (3) avoiding copies where the baseline type system might require them, as in Barnes-Hut. On the other hand, the baseline DPJ code is closer to the original sequential code, because it uses parallel control constructs directly, rather than factoring the code into helper functions and framework API calls. This last point is not specific to our work, but is a general issue with using frameworks.

7. Related Work

Effect systems. The seminal work on types and effects for concurrency is FX [16, 20], which adds a region-based type and effect system to a Scheme-like, implicitly parallel language. FX supports effect polymorphic closures, which are similar to our Java interfaces annotated with effect variables.

Leino et al. [18] and Greenhouse and Boyland [15] first added effects to an object-oriented language. Later work on ownership types [6,9,10,19] introduced sophisticated ways of reasoning about nested effects, which are necessary to support recursive structures. DPJ [4, 5] builds upon this work to provide an expressive type and effect system for deterministic parallelism.

None of this work teaches how to write a framework API for safe parallelism using linear data structures. Nor does it support mechanisms such as effect constraints and type region parameters that are necessary for generic frameworks.

Linear type systems. Wadler [31] introduced linear types as a way to allow in-place updates while preserving the semantic guarantees of pure functional programming. A linear type system can enforce strong guarantees of program correctness [12]. However, linear types prohibit reference aliasing, which makes many common patterns of imperative programming awkward or impossible.

Several researchers have looked at ways to make linear types less restrictive while maintaining meaningful guarantees. Fähndrich and DeLine [13] introduced *adoption and focus* to create aliases of a linear reference with a limited lifetime. Clarke and Wrigstad [11] have observed that *external uniqueness* — the property that every object has at most one reference to it located outside its containing data structure — can express important patterns, such as a unique reference to a doubly-linked list. Boyland and others [8, 28] have used *fractional permissions* to enforce linearity of write references, while allowing sharing of read-only references. Finally, several researchers have shown how to combine unique references with effect systems in interesting ways [7, 15].

Our idea of *internal linearity* of data structures is related to these mechanisms, but also different from all of them. Our insight is that for parallel traversals over the slots of a data structure, all we care about is whether the slots point to different objects. Thus we don't need the full power of a linear type system, or even one of the weaker systems mentioned above.

Together, our ideas of *hidden regions* and *strong disjointness* provide more power than linear data structures, while still providing more aliasing than linear types. However, this approach does not support sets. DPJ's indexed parameterized arrays [5] provide both strong disjointness and linearity, but they do so by making the type region arguments explicit in user code, thereby preventing reference swapping as discussed in Section 3.

Enforcing API contracts. The Eiffel language [29] introduced the idea of *design by contract*, which uses preconditions and postconditions to specify interaction between classes. The Java Modeling Language (JML) [17] provides a powerful way to write design-by-contract specifications for Java, which can be checked with a combination of static verification and online checking.

Design by contract ideas have been applied to concurrent programming. Meyer's Systematic Concurrent Object-Oriented Programming (SCOOP) concurrent programming model [21] is based on Eiffel. The Fortress programming language [27] provides a way to write assertions at interface boundaries that can be checked at runtime. X10 [24] has a sophisticated dependent type system that can specify and check interface assertions, also supported with runtime checking. None of this work addresses parallel noninterference or safe frameworks for shared memory parallelism.

Our annotated generic framework APIs also provide a kind of design by contract, because the framework writer bounds the effects of user-supplied methods. As far as we know, we are the first to study the problem of guaranteeing parallel noninterference for a framework operating on linear data structures in a shared memory context. We are also the first to show how to use a *type and effect system* for design by contract in a parallel framework API. Compared to more general specification methods (such as JML), an effect system has the advantage that the annotations are easier for the programmer to write and the compiler to check without runtime checks or heavyweight constraint solving or theorem proving.

8. Conclusion

We have shown how to use a type and effect system with effect variables and type region parameters to write a generic framework API that enables sound reasoning about its uses. The framework internals can be verified once, by the framework writer, and then the compiler can provide a guarantee of noninterference (and consequently, deterministic parallel semantics) for any user program written using the framework. As future work, we would like to explore ways to verify properties of the framework implementation, such as linearity, using static and dynamic techniques. We would also like to explore what if any additional "glue" is necessary to connect this sort of checking with the guarantees provided by our type system to do proofs of correctness for the composite program.

References

[1] http://gee.cs.oswego.edu/dl/jsr166/dist/extra166ydocs/index.html?extra166y/package-tree.html.

[2] Z. Anderson et al. SharC: Checking data sharing strategies for multithreaded C. *PLDI*, 2008.

[3] R. Bocchino, V. Adve, S. Adve, and M. Snir. Parallel programming must be deterministic by default. In *First USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2009.

[4] R. L. Bocchino and V. S. Adve. Formal definition and proof of soundness for Core DPJ. Technical Report UIUCDCS-R-2008-2980, U. Illinois, 2008.

[5] R. L. Bocchino, V. S. Adve, et al. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.

[6] C. Boyapati et al. Ownership types for safe programming: Preventing data races and deadlocks. *OOPSLA*, 2002.

[7] J. Boyland. The interdependence of effects and uniqueness. *Workshop on Formal Techs. for Java Programs*, 2001.

[8] J. Boyland. Checking interference with fractional permissions. *SAS*, 2003.

[9] N. R. Cameron et al. Multiple ownership. *OOPSLA*, 2007.

[10] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. *OOPSLA*, 2002.

[11] D. Clarke and T. Wrigstad. External uniqueness. In *FOOL*, 2003.

[12] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI*, 2001.

[13] M. Fähndrich and R. DeLine. Adoption and Focus: Practical linear types for imperative programming. *PLDI*, 2002.

[14] A. Gotsman et al. Thread-modular shape analysis. *PLDI*, 2007.

[15] A. Greenhouse and J. Boyland. An object-oriented effects system. *ECOOP*, 1999.

[16] R. T. Hammel and D. K. Gifford. FX-87 performance measurements: Dataflow implementation. Technical Report MIT/LCS/TR-421, 1988.

[17] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 2006.

[18] K. R. M. Leino et al. Using data groups to specify and check side effects. *PLDI*, 2002.

[19] Y. Lu and J. Potter. Protecting representation with effect encapsulation. *POPL*, 2006.

[20] J. M. Lucassen et al. Polymorphic effect systems. In *POPL*, 1988.

[21] B. Meyer. Systematic concurrent object-oriented programming. *CACM*, 1993.

[22] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comp. Sci.*, 2007.

[23] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2007.

[24] V. A. Saraswat, V. Sarkar, and C. von Praun. X10: Concurrent programming for modern architectures. In *PPoPP*, 2007.

[25] J. P. Singh et al. SPLASH: Stanford parallel applications for shared-memory. Technical report, 1992.

[26] L. A. Smith and J. M. Bull. A multithreaded Java grande benchmark suite. In *Third Workshop on Java for High Performance Computing*, 2001.

[27] Sun Microsystems, Inc. The Fortress language specification, version 1.0. Technical report, Sun Microsystems, Inc., March 2008.

[28] T. Terauchi and A. Aiken. A capability calculus for concurrency and determinism. *TOPLAS*, 2008.

[29] P. Thomas and R. Weedon. *Object-Oriented Programming in Eiffel: 2nd Ed.* Addison-Wesley Longman, 1998.

[30] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, November 2002.

[31] P. Wadler. Linear types can change the world! *IFIP*, 1990.

A. Appendix

Here are the rules for valid effects:

$$\text{(EFFECT-EMPTY)}\ \frac{}{\Gamma \triangleright \emptyset} \qquad \text{(EFFECT-RD)}\ \frac{\Gamma \triangleright R}{\Gamma \triangleright \texttt{reads}\ R}$$

$$\text{(EFFECT-WR)}\ \frac{\Gamma \triangleright R}{\Gamma \triangleright \texttt{writes}\ R} \qquad \text{(EFFECT-PARAM)}\ \frac{\eta \in \Gamma}{\Gamma \triangleright \eta}$$

$$\text{(EFFECT-UNION)}\ \frac{\Gamma \triangleright E \quad \Gamma \triangleright E'}{\Gamma \triangleright E \cup E'}$$

Here are the rules for subeffects, repeated from [4]:

$$\text{(SE-EMPTY)}\ \frac{}{\Gamma \triangleright \emptyset \subseteq E} \qquad \text{(SE-RD)}\ \frac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright \texttt{reads}\ R \subseteq \texttt{reads}\ R'}$$

$$\text{(SE-WR)}\ \frac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright \texttt{writes}\ R \subseteq \texttt{writes}\ R'} \qquad \text{(SE-RD-WR)}\ \frac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright \texttt{reads}\ R \subseteq \texttt{writes}\ R'}$$

$$\text{(SE-UNION-1)}\ \frac{\Gamma \triangleright E \subseteq E'}{\Gamma \triangleright E \subseteq E' \cup E''} \qquad \text{(SE-UNION-2)}\ \frac{\Gamma \triangleright E' \subseteq E \quad \Gamma \triangleright E'' \subseteq E}{\Gamma \triangleright E' \cup E'' \subseteq E}$$

$\Gamma \triangleright E \subseteq E'$ is reflexive and transitive. Inclusion of regions is given by reflexivity and transitivity, together with the single rule

$$\text{(REGION-INCLUDE)}\ \frac{}{\Gamma \triangleright R \subseteq *}$$

which says that any region is included in $*$.