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

Robert L. Bocchino Jr.[1] and Vikram S. Adve[2]

[1] Carnegie Mellon University
[2] University of Illinois at Urbana-Champaign

Abstract. Object-oriented frameworks can make parallel programming easier by providing generic parallel algorithms such as map, reduce, or pipeline 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 can 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 while retaining soundness. We present a formal semantics and soundness properties for the language. Finally, we describe an evaluation showing that our technique can express three parallel frameworks and three realistic parallel algorithms using those frameworks.

1 Introduction

The emergence of commodity multicore systems 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 and manipulation of generic data structures; for generic parallel algorithms such as map, reduce, or scan; or for generic parallel coordination patterns such as pipelines. The user fills in the missing pieces with code that is applied in parallel by the framework. Examples include the algorithm templates in Intel's Threading Building Blocks (TBB) [28] and Java's ParallelArray [1]. Such frameworks are usually easier to reason about than general parallel programming because the user only has to write sequential code, letting the framework orchestrate the parallelism.

* This work was supported by the National Science Foundation under grants CCF 07-02724 and CNS 07-20772, and by Intel, Microsoft, and the University of Illinois through UPCRC Illinois. Robert Bocchino is supported by a Computing Innovation Fellowship.

M. Mezini (Ed.): ECOOP 2011, LNCS 6813, pp. 307–333, 2011.

However, state-of-the-art 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 a race will result. It would be much better if (1) the framework developer could write an API expressing a contract (for example, the function provided to `apply` has no potentially interfering effects on shared 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 [20, 25, 35], these ideas have not yet been applied to enforce *parallel noninterference*. Doing so poses several challenges:

1. *Maintaining disjointness.* 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 define a method 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. For a language like Java with reference aliasing, disjointness of reference is a nontrivial property.
2. *Constraining the effects of user-supplied methods.* For a parallel update traversal over the objects in a framework, disjointness of reference 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.
3. *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 in the type of the contained objects. 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.
4. *Writing the framework implementation.* The framework author must ensure that the internal framework implementation guarantees safe parallelism, given that the API is enforced. For example, the framework implementation must ensure that any parallel loop inside the framework iterates exactly once over each contained 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*.

In this work we primarily 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) [6, 7], which expresses effects in terms of *regions* that partition the heap. Regions provide an intuitive and flexible way to express and check effects.

As to the fourth challenge, we state the properties (type preservation, effect preservation, and noninterference) that a correct framework must satisfy. In many cases DPJ can verify those properties. In some cases, however, the DPJ effect system may be insufficiently expressive to guarantee disjointness of effect. Here the framework author is free to use a different strategy, such as program logic [14, 15], testing, or model checking, to verify disjointness. Such checking is completely hidden from the user of the framework, so that the user gets a strong guarantee: if the program type checks, then there is no interference.

Our contributions are the following:

1. We show how to write a framework API using DPJ as described in [6] so that the framework implementer has all the information necessary to guarantee disjointness of reference and sound effects for user-supplied methods.
2. We show how to extend DPJ to add generic effects and generic types, making the frameworks more general and useful. For the effects, we add *effect variables*, together with *effect constraints* to enforce disjointness of effect. For generic types, we introduce *type region parameters*, a form of type constructor, to guarantee disjointness and soundness of effect, without knowing the exact type bound to type variables.
3. We sketch the formal semantics of a core subset of the extended language and formally state the soundness results. The full semantics and proofs are stated in the first author's Ph.D. thesis [5].
4. We state the requirements for a correct framework implementation, such that if these requirements hold, then noninterference is guaranteed for the entire program. We also show how to use a combination of the DPJ type system and external reasoning to check the requirements informally. We leave as future work the formal verification of the requirements.

To evaluate our techniques, we used them to write three parallel frameworks (Parallel Array, Parallel Tree, and Pipeline) and three applications using those frameworks. We found that the techniques are expressive enough to capture realistic parallel algorithms. We also found that the extra annotations required by the system are fairly simple for framework users and, while more complicated for framework writers, are not unduly burdensome.

2 Background

DPJ. We begin with a brief introduction to DPJ [6]. 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 r, or a colon-separated list of names, such as $r_1 : r_2 : r_3$, called

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

Fig. 1. `Node` class that will serve as a running example

a *region path list* (RPL). RPLs give rise to a natural *nesting* structure: one RPL is "under" another if the second is a prefix of the first. For example, $r_1{:}r_2$ is under r_1. Nesting is useful for expressing *effects* (i.e., what regions are read and written by a particular program statement). The set of all regions under an RPL R is denoted $R{:}*$.

Figure 1 defines a simple list node class that we will also use in subsequent sections. The class has one region parameter `R`. Fields `data` and `next` are both placed in region `R`. When class `Node` is instantiated into a type, both fields will be in the region given as the argument to `R` in the type. 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. In general, a method must summarize its effects; if there is no effect summary, the default is "writes the whole heap."

Figure 2 shows a simple container class, `NodePair`, that stores a pair of list nodes. Line 2 declares region names `First` and `Second`. Lines 3–4 instantiate `Node` types using these names. Line 12 reads field `first`, located in region `First` (line 3). It also writes the `data` field of `first`, which is located in region `R` (line 2 of Figure 1) after substituting `First` for `R`, from the type of `first` (line 3 of Figure 2). Thus the effect of the write is `writes First`. Writes cover reads in DPJ, so the whole effect of line 12 may be summarized as `writes First`, as shown. The same reasoning gives the effect `writes Second` shown in line 13. Because `First` and `Second` are distinct names, the compiler can conclude that the updates in lines 12 and 13 are disjoint. With these features, together with additional features for arrays, divide and conquer parallelism, and commutative operations, DPJ can express important patterns of parallelism [6].

```
1  class NodePair {
2      region First, Second;
3      Node<First> first in First;
4      Node<Second> second in Second;
5      NodePair(Node<First> first,
6               Node<Second> second) pure {
7          this.first = first;
8          this.second = second;
9      }
10     void updateNodes(int fd, int sd) {
11         cobegin {
12             first.data = fd;  // writes First
13             second.data = sd; // writes Second
14         }
15     }
16 }
```

Fig. 2. Using region parameters to distinguish object instances

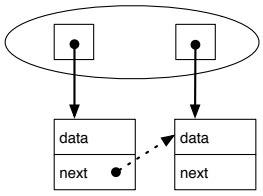

Fig. 3. A potential race caused by cross links. A race can occur if a task operating on the left-hand reference follows the dashed arrow to access the right-hand object.

Difficulties with Region-Based Systems. Region-based effect systems can be quite expressive, and they are a natural choice for writing safe object-oriented frameworks. However, existing systems impose significant limitations. 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 `first` and `second` in the example:

```
void swap() {
    Node<First> tmp = first;
    first = second;          // Illegal: Can't assign Node<Second> to Node<First>
    second = tmp;            // Illegal: Can't assign Node<First> to Node<Second>
}
```

If we could do such an assignment, then 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.

For this reason, DPJ and other region-based systems [23, 9] use wildcard types. For example, in lines 3–4 of Figure 2, we could have written both types `Node<*>`, where `*` stands in for any region. Now the swapping shown above is sound. However, we have lost the ability to distinguish writes to `first.data` and `second.data` using the type system, because now all we know is that the writes are to `*`. So in this case, 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. Notice, however, that (1) regions `First` and `Second` are distinct coming into the constructor (line 5); and (2) the `swap` operation preserves the distinctness of `First` and `Second` in the dynamic types of `first` and `second`. So in fact it is possible both to allow the swap and to prove disjointness, although the type system can't do both.

In fact, the situation is worse than this. As shown in Figure 3, a `NodePair` holding distinct list nodes can have cross links. The effect system must ensure that when following the references to access the objects in parallel, cross links are never followed to update the same object. Further, we probably don't want to encode the write to `data` into the framework implementation, as shown in lines 12–13. Instead, as discussed in the introduction, we would like to express the operation abstractly and let the user supply the specific operation. We must constrain the effects of the user-supplied method so that for any user-supplied method, this kind of interference cannot happen. Finally, we don't really want a `NodePair` class; instead, we want a `Pair<T>`, where `T` is a generic type.

3 Safe, Reusable Parallel Frameworks

We now show how to address the challenges discussed above to write safe, reusable parallel frameworks. First we show how to write a container API that supports sound reasoning about effects for a container specialized to list nodes. Second, we show how to extend the effect system to make the container API

generic. Third, we address the problem of writing a correct framework implementation. Although most of this section focuses on the container example, the work is not specific to containers. In the next section, we formalize the techniques in general terms, without specifically considering disjoint containers. In Section 5 we use the techniques to write a framework for pipeline parallelism.

3.1 A List Node Container

In this section, we show how to use the DPJ effect system as previously described [6] to write a container API that stores `Node` objects and allows safe parallel updates to the stored objects. The API generalizes the trivial `NodePair` class from the previous section into an arbitrary container. The container *implementation* is not specified; it could be any container (set, list, tree, etc.). The point is that we will be able to write an API for a container that (1) holds list nodes, which may have cross links between them, as shown in Figure 3; and (2) allows update operations on the nodes to be done *safely in parallel*, despite the presence of the cross links. We will extend the example further to a more generic (and more useful) container in later sections. Writing the list node container API presents two problems: maintaining disjointness and reasoning about effects.

Maintaining Disjointness. To enable parallel update traversals over the contained nodes, we wish the container to have the following two properties. First, at runtime, every node element e_i stored in the container either is `null` or points to an object with a region R_i in its type. Second, for any $i \neq j$, if e_i and e_j are both non-`null`, then R_i and R_j are *disjoint* (i.e., R_i and R_j refer to nonintersecting sets of regions). In general, we call a container "disjoint" if it satisfies both properties for its elements e_i. Note that the `NodePair` container from Section 2 satisfies this definition. Disjointness ensures that parallel tasks that update the regions of different elements are noninterfering.

To enforce disjointness, we use the following strategy: (1) every container starts empty and so is trivially disjoint; and (2) every operation provided by the disjoint container API is disjointness-preserving (takes a disjoint container to another disjoint container). By a simple induction, we can then conclude that the container is disjoint throughout its lifetime. The hard part is guaranteeing property (2). In some cases, this problem may be reduced entirely to the problem of writing a correct framework implementation (Section 3.3). Examples include

```
1  public interface NodeContainer<region RN,RC | RN:* # RC> {
2       /* One linear container from another  */
3       public NodeContainer(NodeContainer<RN,RC> c) writes RC;
4
5       /* Controlled creation of contents */
6       public NodeContainer(NodeFactory fact, int size) writes RC;
7       public interface NodeFactory { public <region R>Node<R> create(int i) pure; }
8
9       /* Parallel operation on all elements */
10      public void performOnAll(Operation op) reads RC writes RN:*;
11      public interface Operation { public <region R>void operateOn(Node<R> elt) writes R; }
12 }
```

Fig. 4. Framework API for a disjoint list node container

```
1   /* Implement factory interface */
2     public class MyFactory implements NodeContainer.NodeFactory {
3       public <region R>Node<R> create(int i) { return new Node<R>(i, null); }
4   }
5   /* Declare region names A and B */
6   region A, B;
7   /* Use factory to make NodeContainer<A,B> with 10 elements */
8   NodeContainer<A,B> c = new NodeArray<A,B>(new MyFactory(), 10);
```

Fig. 5. Use of the `NodeFactory` API from Figure 4

tree rebalancing and array shuffling operations that modify only the internal structure of the container. In other cases, the framework implementation may need to cooperate with user-provided code. An example is putting things into a container: the user must have some control over objects placed in the container, but the framework must support sound reasoning about disjointness.

We have explored the following two strategies for controlling disjointness: building one disjoint container from another and controlled creation of contained objects. Figure 4 shows an API for a list node container that illustrates these strategies. There are two region parameters, `RN` for the nodes of the container and `RC` for the container itself. In line 1, we use a *region parameter constraint* [6, 9] to require that for any instantiation of `NodeContainer` that binds R_1 to `RN` and R_2 to `RC`, R_1`:*` and R_2 are disjoint. This ensures that reading the container to traverse the elements does not interfere with updating the contained objects.

Building one disjoint container from another. Line 3 of Figure 4 illustrates this strategy: it says that given an object of type `NodeContainer<RN,RC>` we can create another one. An example is creating a tree from an array or set. An important special case in DPJ is creating a disjoint container from an *index-parameterized array* [6], which supports parallel update traversals but does not support reshuffling of elements. (This is exactly the problem discussed in Section 2, just with array cells rather than fields.) A disjoint container created from an index-parameterized array supports disjointness-preserving operations, including shuffling, by doing them *internally* within the framework.

Controlled creation of contained objects. Lines 6–7 of Figure 4 illustrate this strategy, for an interface to `NodeContainer` that could be implemented in different ways (array, tree, etc). The container implementation does the actual object creation, but the user specifies the number of objects to create and provides a factory method that creates the ith object. For example, a use could look as shown in Figure 5, assuming a class `NodeArray` that implements `NodeContainer`.

The important thing here is that the factory method must really create a new object and not, for example, just fetch some object reference from the heap and store it into the container over and over. The framework author can enforce this requirement by judicious use of a *method region parameter*.[1] In line 7 of Figure 4, 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 assignable to type `Node<R>`

[1] Method region parameters are not claimed as new in this work; the contribution here is to use them for safe factory methods.

enters the method. Therefore, the only way a `Node<R>` can escape the method is if it is created inside the method. The effect of this strategy is to hide the actual regions R_i in the types of the created objects from the user code: to create a new object in R_i, the framework binds R_i to `R` and calls the user's factory method. The factory method doesn't know what R_i is, except that it is bound to `R`. On the other hand, the framework doesn't know what the factory method is, except that when called with `R` $= R_i$ it creates a new `Node<`R_i`>`.

Reasoning about Effects. Lines 10–11 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 `c` of type `NodeContainer<A,B>`, the user could do this:

```
public class MyOperation implements NodeContainer.Operation {
    public <region R>void operateOn(Node<R> elt) writes R { ++elt.data; }
}
c.performOnAll(new MyOperation());
```

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

Effect of `operateOn`. In the definition of the abstract `operateOn` method in the `Operation` interface (line 11 of Figure 4), we again use a method region parameter `R`. We write the type of the formal parameter `elt` as `Node<R>`, and we specify the effect as `writes R`. This causes two things to happen. First, the DPJ type system requires that any user-supplied method implementing `operateOn` must have a declared effect that is a *subeffect* of `writes R`. That means all the effects represented by E_2 are also represented by E_1. For example, `reads R` is allowed, but reading or writing some other region is not.[2] In particular, if `MyOperation` had contained the statement `++elt.next.data`, the effect would be `writes *`, which is not a subeffect of `writes R`, and the compiler would catch the error. Thus, the effect annotations prohibit using cross links to cause a race, as shown in Figure 3. Second, the API techniques discussed above ensure that the regions in the contained elements are disjoint. Together, these two facts guarantee that the effects of different parallel tasks operating on different elements are noninterfering.

Effect of `performOnAll`. In Figure 4, we have written the effect of `performOnAll` as `reads RC writes RN:*`. This is correct if, for a particular implementation of the interface, (1) each element i has type `Node<`R_i`>`, where R_i is under `RN`; and (2) the implementation of `performOnAll` reads the container and applies the user's `operateOn` method to the elements. As discussed further in Section 3.3, the framework writer is responsible for ensuring that both facts are true. Further, if the framework internals are written in DPJ, then DPJ can verify these facts.

3.2 Getting More Flexibility

We now show how to generalize the list node container to a generic container. This requires some extensions to the DPJ effect system.

Generic Effects. In the previous API, the effects of `operateOn` are overly restricted. For instance, what if the user wants `operateOn` to read some region

[2] The relevant rules for subeffects are given formally in the next section.

```
1   public interface Operation<effect E> {
2       public <region R>void operateOn(Node<R> elt) writes R effect E;
3   }
4
5   public <effect E | effect E # reads RC writes RN:* effect E>
6     void performOnAll(Operation<effect E> op) reads RC writes RN:* effect E;
```

Fig. 6. Making the effects of the `Operation` interface generic

```
1   public class MyOperation implements NodeContainer.Operation<reads Global> {
2       public <region R>void operateOn(Node<R> elt) reads Global writes R {
3           elt.data = global; // global is in region Global
4       }
5   }
6   c.<reads Global>performOnAll(new MyOperation());
```

Fig. 7. Use of the API from Figure 6

R' disjoint from R`:*`, where R is the region bound to `RN` in the instantiation of the framework interface? That is safe and should be allowed. Yet it is disallowed by the effect specification `writes R` in the API.

To address this problem, we use effect polymorphism [24]. As shown in Figure 6, we give the `Operation` interface an *effect parameter* `E` that becomes bound to an actual effect (for example, `Operation<reads r>`) when the interface is instantiated into a type. To make this strategy work, we need to solve two problems: (1) constraining the effect arguments to ensure noninterference; and (2) ensuring soundness of subtyping.

Constraining the effect arguments. The framework cannot let the effect variable `E` become bound to an arbitrary effect in the user's code, because that would re-introduce a user-supplied method with unregulated effects. Instead, we use an *effect constraint* that restricts the effect of the user-supplied method, as shown in Figure 6. We give the `Operation` interface (line 1) an effect variable `E`. We also give the `performOnAll` method (lines 5–6) a *constrained method effect parameter* `E`. After the parameter declaration is a constraint specifying that the effect bound to `E` must be noninterfering with `reads RC writes RN:* effect E`. This constraint ensures that the supplied effect will not interfere with any of (1) the effect `reads RC` of reading fields of the container; (2) the effect `writes RN:*` of updating the nodes; or (3) itself. The last constraint implies that either `E` is a read-only effect, or it is an update operation such as an atomic set insert that commutes with itself [6]. As an example, Figure 7 shows a user-supplied method that puts all the `Node` objects in region `A` and reads region `Global` to initialize all the objects with the same global value.

Soundness of subtyping. Once we add class effect parameters, we need a rule for deciding if `C<`E_1`>` is a subtype of `C<`E_2`>`, where E_1 and E_2 are effects. We could require that E_1 and E_2 be identical effects. However, this is more restrictive than necessary. While that alone might be acceptable, it turns out this approach is also not sound if done in the obvious way, as discussed in Section 4.5. Instead, we let E_1 be a subeffect of E_2. For example, E_1 could be `writes` r_1 and E_2 could be `writes` r_1`,` r_2.

This approach introduces a subtle requirement for preserving consistency of types. For example, consider the following snippet:

```
class C<effect E> { C<effect E> f; }
C<writes r> x = new C<pure>();
```

By the subtyping rule stated above, this code is legal. But then what is the static type of `x.f`? The obvious answer is `C<writes r>` (substituting `writes r` from the type of `x` for `E` in the declaration of `f`), but this is incorrect. For in that case, a reference of type `C<writes r>` could be legally assigned to `x.f`. But the dynamic type of `x.f` is `C<pure>`, and `writes r` is not a subeffect of `pure`, so the assignment violates type preservation.

The solution we adopt to make the static type of `x.f` `C<effect` E'`>`, where E' is a fresh effect parameter (called a *capture parameter*). The tricky thing here is that *all nonempty effects must be captured when substituted for an effect parameter in a type.* This is because all nonempty effects are essentially wildcards: the runtime effect could be equal to the static effect, or it could be empty (or possibly something else, e.g., `reads` R instead of `writes` R, or `reads` R_1 instead of `reads` R_1, R_2). Our solution follows in the same vein as generic wildcards [10] (which stand in for several types) and DPJ's partially-specified RPLs [5] (which stand in for several RPLs).

Generic Types. It is also too restrictive to make the class specialized to list nodes. Instead, we want a class `DisjointContainer<type T, region RC>` with a generic type `T`. Notice, however, that the region argument to the `Node` type is essential to writing the API. For example, in writing the `NodeFactory` interface of Figure 4), we used a method-local parameter `R` in the return type of `create`. If we just replaced that type with an ordinary type variable `T`, then we would not be able to write the node factory pattern at all. A similar issue occurs in writing the effect of `performOnAll`.

To solve this problem, we use a type constructor [3, 26] that takes a region argument. A type variable can be declared `T<region R>`, where `R` declares a fresh parameter. We call `R` a *type region parameter*. When a type T becomes bound to a type variable `T`, T must have at least one region argument, and `R` represents the first region argument. We write uses of the variable `T` as `T<`r`>`, where r is a valid region in scope. `T<`r`>` represents the same type with the region in its first argument position replaced by r. Notice that according to this rule the parameter `R` is a valid region, and `T<R>` represents the unmodified type provided as the argument to the variable. For convenience, a bare use of `T` is allowed, and this is equivalent to `T<R>`. Our language also supports multiple type region parameters for a variable `T`; this straightforward extension is discussed in [5].

Final Container API. Figure 8 shows the final disjoint container API. Line 1 declares an interface `DisjointContainer` 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 8, we write `T<R>` to require that the return type of `create` have the method region parameter `R` as its first region argument. In line 12, the region `Elt` is available to write the

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

Fig. 8. API for a disjoint container with generic types and effects

```
1    public class MyOperation implements
2      DisjointContainer.Operation<Node<A>,pure> {
3        public <region R>void operateOn(Node<R> elt) writes R { ++elt.data; }
4    }
5    c.performOnAll(new MyOperation());
```

Fig. 9. Use of the API from Figure 8

effects of `performOnAll`. We do the same thing for the type parameter of the `Operation` interface, in line 13.

Figure 9 shows an example implementation of `operateOn`, assuming `c` has type `DisjointContainer<Node<A>,B>`. The effect argument in line 2 is `pure`, because no effect is needed for this implementation of `operateOn`, except for `writes R`, which is already given by the interface (line 14 of Figure 8). The effect of the call to `performOnAll` in line 5 is `reads B writes A:*`.

3.3 Writing the Framework Implementation

We now address the problem of writing a correct framework implementation. The framework must ensure three properties: type preservation, effect preservation, and noninterference of effect. The key point is that *the API design discussed in the previous sections provides all the information needed to reason soundly about these three properties, even in the presence of unknown user-supplied methods.* Further, the framework author can write the framework in DPJ, thereby using DPJ to check some or all of these properties. However, so long as the properties hold for all user-visible types and effects, the framework author may use *internal* operations, such as swapping references with disjoint regions, that DPJ alone cannot prove correct.

Type Preservation. Type preservation means that the static types of variables agree with the dynamic types of the references they store. If the framework is written in DPJ, then this property will be checked "for free," unless the framework does an assignment (using a cast) that violates the typing rules. Such type casts produce a warning, but the code compiles and runs.

The DPJ subtyping rules are flexible, so we anticipate that unsound assignments will rarely be needed in practice. A more likely case is that casts are used to interface with non-DPJ code. For example, pre-Java 5 code implementing a container might represent the container elements as references to `Object` and require that they be cast back to their actual type when removed from the container. For Java code written with generics, such casts should be rare.

Effect Preservation. Effect preservation means that the static effects of statements correctly summarize their dynamic effects. Again, DPJ guarantees this property, so long as (1) type preservation holds; and (2) every method summary covers the effects of the method body. In DPJ, one can always write a correct method summary (`writes *` is always correct), and in fact an incorrect summary causes a compile-time error. So property (2) will hold if property (1) does. If the framework calls into non-DPJ code, then the framework writer must manually ensure that effect preservation holds for the calling code.

Noninterference of Effect. Noninterference means there are no conflicting memory accesses between parallel tasks. While DPJ can establish noninterference in many cases, in some cases it may not be able to, as in the swap example discussed in Section 2. In such cases, the framework author can write code that causes DPJ to produce an interference warning, and use a different technique to show noninterference. As an example, Figure 10 shows an implementation of `DisjointContainer` as a `DPJArrayList`, which is a Java `ArrayList` annotated with region information. In line 4, the type argument to `DPJArrayList` is `Elt:*`, i.e., the type does not specify which cell of the array is in which region, so reshuffling the array is supported. The `performOnAll` method uses the DPJ `foreach` construct (line 8) to iterate in parallel over the elements. We also add a `swap` method, similar to the method discussed in Section 2, for swapping array elements.

```
1  public class DisjointArray<type T<region Elt>, region RC | Elt:* # RC>
2    implements DisjointContainer<T,RC> {
3       /* Internal array representation */
4       private DPJArrayList<T<Elt:*>,RC> elts in RC;
5       /* Implementation of performOnAll */
6       public <effect E | reads RC writes Elt:* effect E>
7         void performOnAll(Operation<T,effect E> op) reads RC writes Elt:* effect E {
8           foreach (int i in 0, elts.size()) { op.operateOn(elts.get(i)); }
9       }
10      /* Swap elements at idx1 and idx2 */
11      public void swap(int idx1, int idx2) writes RC {
12          T<Elt:*> tmp = elts.get(idx1); elts.add(idx1, elts.get(idx2)); elts.add(idx2, tmp);
13      }
14 }
```

Fig. 10. Array implementation of a disjoint container (partial)

To show noninterference, it suffices to establish two things for the `foreach` construct in line 8: (1) for distinct values i, the region in the dynamic type of `elts.get(`i`)` is distinct; and (2) `i` attains distinct values i on distinct iterations. The first statement follows from the inductive argument we made in Section 2: to change the shape of the array, we either have to use an inherited creation

method, which preserves disjointness as discussed in Section 3.1, or do a swap, which also preserves disjointness, as can be seen from the implementation in line 11. The second statement follows from the semantics of `foreach` in DPJ.

4 Formal Elements

In this section we formalize the ideas developed in the previous section. We use a sequential core language, which suffices to establish type preservation, effect preservation, and noninterference of effect. As discussed in Section 3.3, a framework designer can use these properties to provide deterministic parallelism or other guarantees for correct framework uses. We give a syntax, static semantics, and dynamic semantics for the core language. Then we state the key soundness results, and sketch the proofs. More detail, including proofs, can be found in [5].

4.1 Syntax

Programs	$\mathcal{P}$	$::= \mathcal{R}^*\ \mathcal{I}^*\ \mathcal{C}^*\ e$
Region Names	$\mathcal{R}$	$::=$ `region` r
Interfaces	$\mathcal{I}$	$::=$ `interface` I`<`τ`<`ρ`>,` $\rho, \eta \# E$`>` `{` S^* `}`
Classes	$\mathcal{C}$	$::=$ `class` C`<`τ`<`ρ`>,` ρ`>` `implements` I`<`T, R, E`>` `{` $F^*\ M^*$ `}`
Method Signatures	S	$::=$ `<`$\rho, \eta \# E$`>`$T\ m$`(`$T\ x$`)` E
Fields	F	$::= T\ f$ `in` R
Methods	M	$::= S$ `{` e `}`
RPLs	R	$::= r \mid \rho \mid R$`:`$r \mid R$`:*`
Types	T	$::= I$`<`T, R, E`>` $\mid C$`<`T, R`>` $\mid \tau$`<`R`>` $\mid$ `Null`
Effects	E	$::= \emptyset \mid$ `reads` $R \mid$ `writes` $R \mid \eta \mid E \cup E$
Expressions	e	$::=$ `this.`$f \mid$ `this.`f`=`$e \mid e$`.<`R`,`E`>`m`(`e`)` $\mid v \mid$ `new` $T \mid$ `null`
Variables	v	$::=$ `this` $\mid x$

Fig. 11. Syntax of the core language. r, I, τ, ρ, η, C, f, m, and x are identifiers

Figure 11 gives the syntax for the core language. A program $\mathcal{P}$ consists of region name declarations, interface definitions, class definitions, and an expression to evaluate. An interface $\mathcal{I}$ consists of an interface name I, the interface parameters, and zero or more method signatures. There is one type parameter τ, one region parameter ρ, and one constrained effect parameter $\eta \# E$. The type parameter τ has a region parameter ρ that captures the region argument of the type bound to it. A method signature S specifies a region parameter, a constrained effect parameter, a return type, a method name m, a typed formal parameter x, and an effect.

A class $\mathcal{C}$ consists of a class name C, the class parameters, the interface type being implemented, and the fields and methods of the class. For simplicity we omit class effect parameters; their treatment is identical to interface effect parameters. A field F specifies a type, a field name f, and an RPL. A method specifies a signature and an expression to evaluate. A region path list (RPL) R is a named region r, a region parameter ρ, or an RPL qualified by appending `:`r or `:*`, where `*` stands in for any chain of names. A type T instantiates a named interface with a type, region, and effect; or it instantiates a named class with a

type and region; or it instantiates a type parameter with a region; or it is `Null`. `Null` is the type of a null reference. It also functions as a base-case type for type parameter arguments (every other type has its own argument). An effect E is a possibly empty union of read effects, write effects, and effect parameters. An expression e is a field access, field assignment, method invocation, variable, object creation, or null reference. A variable v is `this` or a method parameter x.

4.2 Static Semantics

Environment. We define the static semantics with respect to a static environment Γ, defined as follows:

$$\Gamma ::= \emptyset \mid (v, T) \mid \tau \mid \rho \mid \eta \mid \eta \# E \mid \Gamma \cup \Gamma$$

(v, T) means that variable v has type T; τ, ρ, or η means that the parameter is in scope; and $\eta \# E$ means that the effect bound to η constrained not to interfere with effect E.

Translation mapping ϕ_T. We define a mapping ϕ_T for translating a type, region, or effect defined in an interface I or class C to its use as a member of a type T instantiating I or C (the *instantiating type*, which must be a class or interface type). It is the context translation described in [5,6], plus effect parameters and type region parameters. Figure 12 gives the key formal rules for interface types; the rules for class types are similar. Note that when the instantiating type has

$\boxed{\phi_T(T')}$
$\boxed{\phi_T(R)}$
$\boxed{\phi_T(E)}$

$$\begin{array}{ll}
\phi_T(I\texttt{<}T', R, E\texttt{>}) = I\texttt{<}\phi_T(T'), \phi_T(R), \phi_T(E)\texttt{>} & \\
\phi_{I\texttt{<}T,R,E\texttt{>}}(\tau\texttt{<}R'\texttt{>}) = I\texttt{<}T, \phi_{I\texttt{<}T,R,E\texttt{>}}(R'), E\texttt{>} & \\
\phi_{I\texttt{<}T,R,E\texttt{>}}(\rho(I)) = R & \\
\phi_{I\texttt{<}T,R,E\texttt{>}}(\rho_\tau(I)) = \texttt{rgn}(T) \text{ if } T \neq \texttt{Null}, \text{ else } R & \\
\phi_T(\texttt{reads } R) = \texttt{reads } \phi_T(R) & \phi_{I\texttt{<}T,R,E\texttt{>}}(\eta(I)) = E \\
\phi_T(\texttt{writes } R) = \texttt{writes } \phi_T(R) & \phi_T(E \cup E') = \phi_T(E) \cup \phi_T(E')
\end{array}$$

Fig. 12. The translation mapping ϕ_T for interface types (selected rules). $\rho_\tau(T)$ $\rho(T)$, and $\eta(T)$ are the type region parameter, region parameter, and effect parameter of the interface that T instantiates. $\texttt{rgn}(T)$ is the region argument of T.

a type argument of `Null`, we treat ρ_τ as an alias for ρ. That is because in this simple language, `Null` is the only type with no parameters. In the full language, we support classes and interfaces with no type region parameter (or no type parameter at all), and we disallow bindings of types lacking a region argument to a type parameter with a region parameter.

Program elements. Figure 13 gives the judgments and rules for typing top-level program elements. METHOD checks that the method body is well-typed, and that its type and effect agree with the return type and effect specified in the method signature. If a signature with name m also appears in the interface implemented by the enclosing class, then IMPLEMENT checks that the types and effects in the method signature agree with the corresponding types and effects in the signature of the implemented interface.

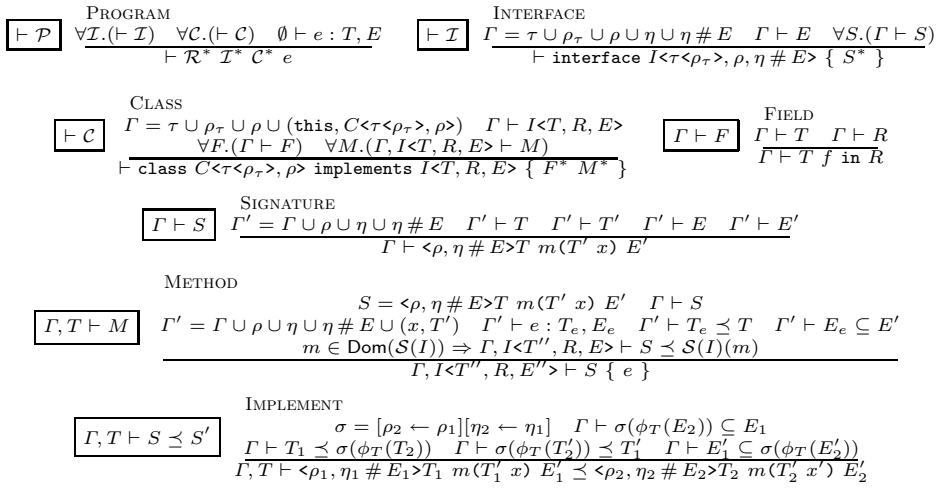

Fig. 13. Typing of program elements. $\mathcal{S}(I)(m)$ is the signature S named m in the definition of I.

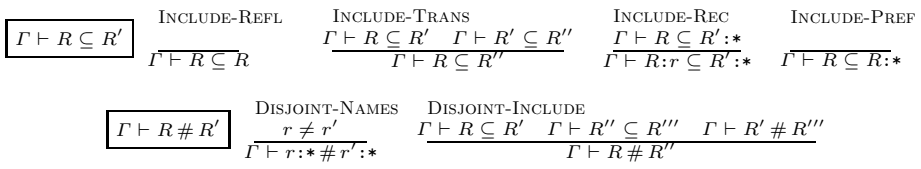

Fig. 14. Inclusion and disjointness of RPLs

RPLs. Figure 14 gives the rules for inclusion and disjointness of RPLs. We use a relevant subset of the rules described in [6]. R is included in R' if R and R' are lexically identical, or if $R' = R'' :*$, and R'' is a prefix of R. For example, $r:*:r'$ is included in $r:*$. Two RPLs are disjoint if they both start with different names r, or if each is included in another RPL, such that the two including RPLs are disjoint. As in [6], inclusion and disjointness of RPLs correspond to inclusion and disjointness of the sets of regions (chains of names r) represented by the RPLs.

Types. Figure 15 gives the rules for checking types. TYPE-INTERFACE checks the disjointness requirement for the effect argument to an interface type. $\Gamma \vdash T \preceq T'$ means that T is a subtype of T', and $\Gamma \vdash T \subseteq T'$ means that T and T' are the same type, except that the region and effect arguments are related by inclusion. The inclusion relation $\subseteq$ implies subtyping (rule SUBTYPE-INCLUDE), but not vice versa. Note that it would not be sound to put $\Gamma \vdash T \preceq T'$ in the condition of INCLUDE-INTERFACE or INCLUDE-CLASS, for the same reason that it is not sound to treat C<C'> as a subtype of C<Object> in ordinary Java [17]. It is

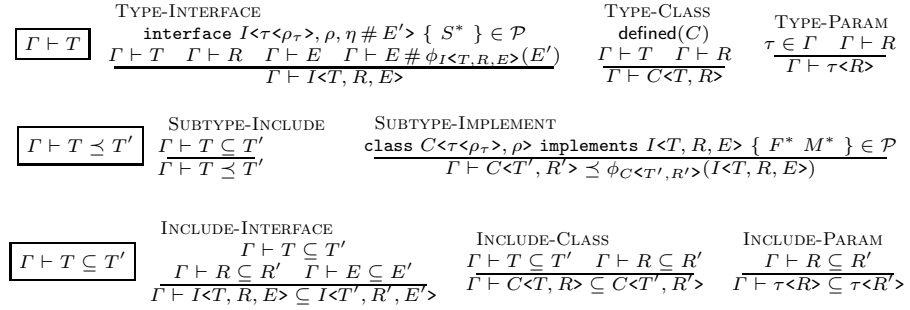

Fig. 15. Types (selected rules). defined(C) means that class C is defined in the program.

sound, however, to make inclusion a condition of subtyping, because we capture regions and effects as discussed below.

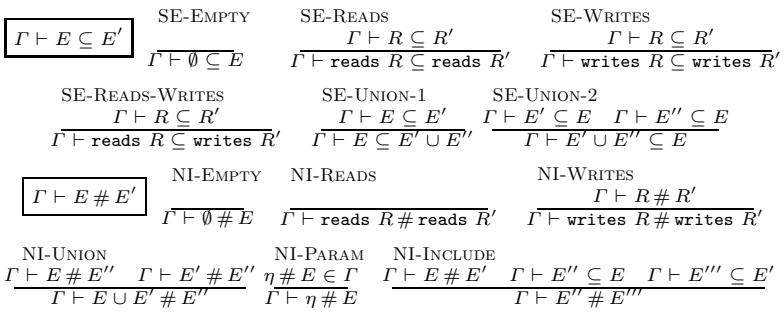

Fig. 16. Subeffects and disjoint effects

Effects. Figure 16 gives the relevant rules for subeffects and noninterfering effects. For subeffects, reads effects on R cover reads of R' if R includes R, and write effects on R cover both reads and writes of R'. For noninterfering effects, read effects are always noninterfering, writes effects are noninterfering if the regions are disjoint, and parametric effects are disjoint if disjointness is specified in a constraint.

Expressions. Figure 17 gives the rules for typing expressions. Invoke translates types and effects in the method signature by (1) substituting for the method region and effect arguments; and (2) applying the translation mapping ϕ_T defined above. Here T is the type of the dispatch expression, after capturing its region and effect arguments. As discussed in Section 3.2, we must capture all partially specified RPLs and all effects. We capture an RPL or effect by replacing it with a fresh parameter, and adding the parameter to the environment. We do this for the RPL and effect arguments of T, and recursively for the type argument of T. The formal rules for this procedure are straightforward and are stated in full in [5].

$$\boxed{\Gamma \vdash e : T, E}$$

$$\textsc{Access}\ \frac{(\texttt{this}, C\texttt{<}\tau\texttt{<}\rho_\tau\texttt{>}, \rho\texttt{>}) \in \Gamma \quad \mathcal{F}(C)(f) = T\ f\ \texttt{in}\ R}{\Gamma \vdash \texttt{this}.f : T, \texttt{reads}\ R} \qquad \textsc{Assign}\ \frac{(\texttt{this}, C\texttt{<}\tau\texttt{<}\rho_\tau\texttt{>}, \rho\texttt{>}) \in \Gamma \quad \Gamma \vdash e : T, E \quad \mathcal{F}(C)(f) = T'\ f\ \texttt{in}\ R \quad \Gamma \vdash T \preceq T'}{\Gamma \vdash \texttt{this}.f\texttt{=}e : T, E \cup \texttt{writes}\ R}$$

$$\textsc{Variable}\ \frac{(v, T) \in \Gamma}{\Gamma \vdash v : T, \emptyset}$$

$$\textsc{Invoke}\ \frac{\begin{array}{c}\Gamma \vdash e_1 : T_1, E_1 \quad \Gamma \vdash e_2 : T_2, E_2 \\ \mathcal{S}(T_1)(m) = \texttt{<}\rho, \eta\ \#\ E_3\texttt{>}T_3\ m(T_4\ x)\ E_4 \quad \sigma = [\rho \leftarrow R][\eta \leftarrow E_5] \\ \Gamma \vdash E_5\ \#\ \sigma(\phi_{T_1}(E_3)) \quad \Gamma \vdash \mathsf{capt}(T_1) = (T_c, \Gamma_c) \quad \Gamma_c \vdash T_2 \preceq \sigma(\phi_{T_c}(T_4))\end{array}}{\Gamma \vdash e_1.\texttt{<}R, E_5\texttt{>}m(e_2) : \sigma(\phi_{T_1}(T_3)), E_1 \cup E_2 \cup \sigma(\phi_{T_1}(E_4))}$$

$$\textsc{New}\ \frac{\Gamma \vdash C\texttt{<}T, R\texttt{>}}{\Gamma \vdash \texttt{new}\ C\texttt{<}T, R\texttt{>} : C\texttt{<}T, R\texttt{>}, \emptyset} \qquad \textsc{Null}\ \frac{}{\Gamma \vdash \texttt{null} : \texttt{Null}, \emptyset}$$

Fig. 17. Expressions. $\mathcal{F}(C)(f)$ means field f declared in class C. $\mathcal{S}(T)$ means $\mathcal{S}(I)$ or $\mathcal{S}(C)$, corresponding to the interface or class named in T. $\Gamma \vdash \mathsf{capt}(T) = (T', \Gamma')$ means that capturing type T in environment Γ yields type T' and environment Γ'.

4.3 Dynamic Semantics

$$\boxed{(e, \Sigma, H) \rightarrow (o, H', E)}$$

$$\textsc{Dyn-Access}\ \frac{(\texttt{this}, o) \in \Sigma \quad H(o) = (O, C\texttt{<}T, R\texttt{>}) \quad \mathcal{F}(C)(f) = T'\ f\ \texttt{in}\ R'}{(\texttt{this}.f, \Sigma, H) \rightarrow (O(f), H, \texttt{reads}\ \phi_{\Sigma,H}(R'))}$$

$$\textsc{Dyn-Assign}\ \frac{(e, \Sigma, H) \rightarrow (o, H', E) \quad (\texttt{this}, o') \in \Sigma \quad H'(o') = (O, C\texttt{<}T, R\texttt{>}) \quad \mathcal{F}(C)(m) = T'\ f\ \texttt{in}\ R'}{(\texttt{this}.f\texttt{=}e, \Sigma, H) \rightarrow (o, H'[o' \mapsto (O[f \mapsto o], C\texttt{<}T, R\texttt{>})], E \cup \texttt{writes}\ \phi_{\Sigma,H}(R'))}$$

$$\textsc{Dyn-Invoke}\ \frac{\begin{array}{c}(e_1, \Sigma, H_1) \rightarrow (o_1, H_2, E_2) \quad (e_2, \Sigma, H_2) \rightarrow (o_2, H_3, E_3) \quad H_3(o_1) = (O, C\texttt{<}T_1, R'\texttt{>}) \\ \mathcal{M}(C)(m) = \texttt{<}\rho, \eta\ \#\ E_4\texttt{>}T_2\ m(T_3\ x)\ E_5\ \{\ e_3\ \} \\ \Sigma' = (\texttt{this}, o_1) \cup (x, o_2) \cup (\rho, \phi_{\Sigma,H}(R)) \cup (\eta, \phi_{\Sigma,H}(E_1)) \quad (e_3, \Sigma', H_3) \rightarrow (o_3, H_4, E_6)\end{array}}{(e_1.\texttt{<}R, E_1\texttt{>}m(e_2), \Sigma, H_1) \rightarrow (o_3, H_4, E_2 \cup E_3 \cup E_6)}$$

$$\textsc{Dyn-Variable}\ \frac{(z, o) \in \Sigma}{(z, \Sigma, H) \rightarrow (o, H, \emptyset)} \qquad \textsc{Dyn-New}\ \frac{o \notin \mathsf{Dom}(H) \quad H' = H \cup o \mapsto (\texttt{new}(C), \phi_{\Sigma,H}(C\texttt{<}T, R\texttt{>}))}{(\texttt{new}\ C\texttt{<}T, R\texttt{>}, \Sigma, H) \rightarrow (o, H', \emptyset)}$$

Fig. 18. Program evaluation. $f[a \mapsto b]$ denotes the function identical to f everywhere on its domain, except that it maps a to b. $\texttt{new}(C)$ is the function taking each field of class C to `null`. The translation function $\phi_{\Sigma,H}$ does the following: (1) it substitutes actual regions and effects for parameters as specified by the bindings in Σ; (2) if $(\texttt{this}, o) \in \Sigma$ and $(O, T) \in H$, it applies ϕ_T. $\mathcal{M}(C)(m)$ denotes the method named m in the definition of class C.

We give a large-step semantics for program execution, using the transition relation $(e, \Sigma, H) \rightarrow (o, H', E)$. e is a program expression. The dynamic environment Σ maps variables v to object references o, region parameters ρ to regions R, and effect parameters η to effects E:

$$\Sigma ::= (v, o) \mid (\rho, R) \mid (\eta, E)$$

The heap H is a partial function from object references o to pairs $(O, C\texttt{<}T, R\texttt{>})$, where O is an object, and $C\texttt{<}T, R\texttt{>}$ is the type of O:

$$H ::= \texttt{null} \mid o \mapsto (O, C\texttt{<}T, R\texttt{>}) \mid H \cup H$$

`null` is a special reference that is in $\mathsf{Dom}(H)$ but does not map to an object. Attempting to access a field of `null` causes execution to fail. An object O is a mapping from field names f to object references o:

$$O ::= \emptyset \mid f \mapsto o \mid O \cup O$$

The effect E collects the effect of the evaluation. A program evaluates to reference o with heap H and effect E if its main expression is e and $(e, \texttt{null}, \emptyset) \rightarrow (o, H, E)$, according to the rules shown in Figure 18. The rules describe a standard semantics for an object-oriented language, except that we bind actual regions and effects to method parameters in rule Invoke, and we accumulate the effects of every expression evaluation.

4.4 Valid Execution State

To state the soundness results, we need to define valid heaps, environments, and execution states.

$$\boxed{\vdash H} \quad \textsc{Heap}\ \frac{H \vdash H}{\vdash H} \qquad \boxed{H \vdash H'} \quad \textsc{Heap-Null}\ \frac{}{H \vdash \texttt{null}} \qquad \textsc{Heap-Object}\ \frac{H \vdash (O,T)}{H \vdash o \mapsto (O,T)} \qquad \textsc{Heap-Union}\ \frac{H \vdash H' \quad H \vdash H''}{H \vdash H' \cup H''}$$

$$\boxed{H \vdash o : T} \quad \textsc{Type-Object}\ \frac{o \mapsto (O,T) \in H}{H \vdash o : T} \qquad \textsc{Type-Null}\ \frac{}{H \vdash \texttt{null} : \texttt{Null}}$$

$$\boxed{H \vdash (O,T)} \quad \textsc{Object}\ \frac{\emptyset \vdash C\texttt{<}T,R\texttt{>} \quad \forall (f \in \mathsf{Dom}(\mathcal{F}(C))).(\mathcal{F}(C)(f) = T'\ f\ \texttt{in}\ R' \wedge H \vdash O(f) : T'' \wedge \emptyset \vdash T'' \preceq \phi_{C\texttt{<}T,R\texttt{>}}(T'))}{H \vdash (O, C\texttt{<}T,R\texttt{>})}$$

Fig. 19. Well-typed heaps

Heaps. Figure 19 gives the rules for typing heaps. A heap is valid if its elements are valid. An object-type pair (O,T) is valid if (1) T is a valid type; and (2) for every field f in $\mathcal{F}(C)$, $O(f)$ is defined, and its type is a subtype of the static type of f, after translation via ϕ_T. At runtime we check types in the empty environment, because all parameters have been substituted away.

Environments. A static environment is valid ($\vdash \Gamma$) if it binds variables to valid types, and if the effects named in the constraints are valid. A dynamic environment is valid ($H \vdash \Sigma$) if it binds variables to valid object references and parameters to valid regions and effects. We omit the formal rules for valid environments, but they are straightforward and are stated in full in [5].

Execution state. Figure 20 gives the rules for a valid execution state (e, Σ, H), with respect to the environment Γ that typed e in the static semantics. The rules state that H, Σ, and Γ are valid; e is well typed in Γ; and Σ instantiates Γ ($H \vdash \Sigma \preceq \Gamma$). That means the types of the variable bindings in Σ and Γ match, and the bindings in Σ obey the disjointness constraints specified by Γ.

$$\boxed{\Gamma \vdash (e, \Sigma, H) : T, E} \quad \textsc{State}\ \frac{\vdash \Gamma \quad \vdash H \quad H \vdash \Sigma \quad H \vdash \Sigma \preceq \Gamma \quad \Gamma \vdash e : T, E}{\Gamma \vdash (e, \Sigma, H) : T, E}$$

$$\boxed{H \vdash \Sigma \preceq \Gamma} \quad \textsc{Instantiate}\ \frac{\Sigma, H \vdash \Sigma \preceq \Gamma}{H \vdash \Sigma \preceq \Gamma}$$

$$\boxed{\Sigma, H \vdash \Sigma' \preceq \Gamma} \quad \textsc{Inst-Var}\ \frac{H \vdash o : T \quad \emptyset \vdash T \preceq \phi_{\Sigma,H}(T')}{\Sigma, H \vdash (z, o) \preceq (z, T')} \quad \textsc{Inst-Constraint}\ \frac{\emptyset \vdash \phi_{\Sigma,H}(\eta) \# \phi_{\Sigma,H}(E)}{\Sigma, H \vdash \emptyset \preceq \eta \# E}$$

Fig. 20. Valid execution state (selected rules)

4.5 Soundness Results

Preservation of type and effect. The first soundness result states that the static types and effects computed according to Figure 17 approximate the dynamic types and effects produced by execution according to Figure 18. More precisely, if we evaluate e to o starting in a valid execution state, then the resulting heap is valid; o is well typed, and its type is a subtype of the static type of e; and the resulting effect is valid and a subeffect of the static effect of e.

Theorem 1 (Preservation of type and effect). *If* $\vdash \mathcal{P}$ *and* $\Gamma \vdash (e, \Sigma, H) : T_s, E_s$ *and* $(e, \Sigma, H) \rightarrow (o, H', E)$*, then (a)* $\vdash H'$*; (b)* $H' \vdash o : T$*; (c)* $\emptyset \vdash T \preceq \phi_{\Sigma,H'}(T_s)$*; (d)* $\emptyset \vdash E$*; and (e)* $\emptyset \vdash E \subseteq \phi_{\Sigma,H'}(E_s)$*.*

The proof, stated in full in [5], is by induction on the structure of e, showing the result for each of the rules given in Figure 18. For all rules but INVOKE, the result follows straightforwardly from the assumptions and the induction hypothesis. For INVOKE, we must show two facts: first, that the dynamic environment in which the method body is executed instantiates the static environment in which we typed the method; and second, that the preservation properties are preserved when we translate back to the environment in which we typed the method invocation. Both facts are proved by keeping careful track of the substitutions that occur in translating from one static environment to another, and in translating from static to dynamic environments.

Here it helps the proof that $C\texttt{<}E\texttt{>}$ is a subtype of $C\texttt{<}E'\texttt{>}$ if E is a subeffect of E', as discussed in Section 3.2. If we required $E = E'$ in the subtype judgment, then we would not be able to conclude that T a subtype of T' implies $\phi_T(T'')$ a subtype of $\phi_{T'}(T'')$. In that case, to ensure sound subtyping for method invocations, we would need to introduce some ad-hoc restrictions on the use of type region parameters in effects.

Noninterference. The second soundness result states that the static noninterference judgment for expressions is sound: if two expressions have statically noninterfering effects, then the execution of the two expressions is noninterfering at runtime.

Theorem 2 (Noninterference). *If* $\vdash \mathcal{P}$ *and* $\Gamma \vdash (e, \Sigma, H) : T_s, E_s$ *and* $\Gamma \vdash (e', \Sigma, H') : T'_s, E'_s$ *and* $\Gamma \vdash E_s \# E'_s$ *and* $(e, \Sigma, H) \rightarrow (o, H', E)$ *and* $(e', \Sigma, H') \rightarrow (o', H'', E')$*, then there are no conflicting memory operations in the evaluations of* e *and* e'*.*

"Conflicting accesses" means a pair of operations on the same memory location, one or both of which is a write. Again the proof is stated in full in [5]. Theorem 1 says that the static effects contain the dynamic effects, so it suffices to show that conflicting accesses produce interfering effects. But this is straightforward from (1) the way that the rules in Figure 18 record effects; and (2) the definition of noninterfering effects in Figure 16.

5 Evaluation

We have evaluated the techniques discussed above with two goals in mind. First, can we use the techniques to write realistic frameworks and user programs? Do any additional issues arise in real frameworks or user code? Second, what is the complexity and annotation overhead of using the techniques to write framework APIs and client code?

We extended the DPJ compiler [6, 7] to support the new language features discussed in Sections 3 and 4. Then we studied how to (1) use our techniques to write generic array, tree, and pipeline frameworks; and (2) use the frameworks to write three parallel codes: a Monte Carlo simulation algorithm, a Barnes-Hut n-body computation using a tree to partition physical space, and radix sort expressed as a pipeline. We chose these three algorithms because they exemplify different styles of parallelism.

5.1 DPJ Frameworks

Array. We wrote a framework **DPJDisjointArray** with an API similar to a subset of Java's ParallelArray [1]. Our API supports creating an array, mapping one array to another with a user-supplied element mapping function, and reducing the array to a single element with a user-supplied binary reduction method (i.e., that reduces two elements into one). For the array creation and mapping interfaces, we used exactly the techniques discussed in Section 3. For the reduction operation, we had to solve the following problem: the user-supplied binary reduction method might violate disjointness by, e.g., storing one of its argument objects into a field of the other. To prevent that, we parameterized each of the arguments with a separate method region parameter, as follows:

```
public interface Reducer<type T<region R>, effect E> {
    public <region R1,R2>T<R1> op(T<R1> a, T<R2> b) writes R1,R2 effect E;
}
```

Tree. We wrote a framework **DPJDisjointTree** that provides a tree with a user-specified branching factor. The tree stores a data object of generic type `T` in each node. The API supports building a tree by inserting bodies from the root and doing a recursive parallel postorder traversal over the tree. The build method takes a user-supplied `index` function that computes which of the children of a particular node to follow next when inserting an object in the subtree rooted at that node. The postorder visitor takes a user-supplied `visit` method. The

input to the method (furnished by the framework implementation) consists of the data object at the current node and an ArrayList of result objects produced from visiting the children (or `null` if the current node is a leaf). The output is a result object for the current node. Again we use two region parameters to ensure that the `visit` method preserves disjointness for the data objects.

Pipeline. We implemented a framework **DPJPipeline** that represents data flowing through a series of pipeline stages, each of which applies some operation to the data. Following the Threading Building Blocks (TBB) library [28] and the StreamIt language [34], we call the operation applied by each stage a *filter*. Each data element flows sequentially through the stages, but different stages can apply their filters to different elements at the same time, creating pipeline parallelism.

The API provides two interfaces for the user to implement: a filter and a factory method for creating a filter. Method region parameters on the factory methods ensure that each filter and each element is a freshly-created object. The filter interface provides an operation method for the user to override. Using method region parameters and constrained effect variables, as in the other examples, the API ensures that the user-defined filter operation is limited to updating the regions of the data object and the filter state, and doing any noninterfering effects on other state. In particular, the filter operation may not update data operated on by a concurrent filter, or a different filter.

5.2 Client Code

Monte Carlo Simulation. We studied the Monte Carlo simulation benchmark from the Java Grande suite [30]. 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 DPJDisjointArray. The first two loops were straightforward to parallelize with the mapping operation. For the third loop, we wrote a binary reduction method that takes two objects produced by the second stage, reads the accumulated sum from both, adds them, stores the result in the first one, and returns it. We could also have created a new object and returned it, but that would be less efficient.

Barnes-Hut Center of Mass. Next we studied the Barnes-Hut n-body simulation [29], which uses an octree (eight-ary tree) to represent three-dimensional space hierarchically, storing the bodies in the leaves. We focused on the center-of-mass computation, which traverses the tree recursively in parallel and, for each node, computes and stores the center of mass of the subtree rooted at that node. It would be straightforward to parallelize the force computation using the same array-based techniques that we used for Monte Carlo.

We wrote a program that builds a tree and performs a center of mass computation for a binary tree computation in one-dimensional space. That simplified the computation, while retaining the algorithm structure. To do this, we instantiated `DPJDisjointTree` with a `Node` class that has subclasses `Cell` for the inner

node data and `Body` for the leaf data, similarly to the original and SPLASH versions [29]. Then, studying the original algorithm, we put the logic for creating the tree into the user-supplied `index` function and the logic for computing and storing the center of mass into the user-supplied `visit` function.

Radix Sort. We wrote a pipelined version of radix sort that is directly modeled after the StreamIt RadixSort benchmark [34]. The first stage produces a stream of arrays to sort, and the successive stages each sort the arrays on a different radix, with the radix recorded in the `Filter` object as `final` variable (so reading it produces no effect). Each sort stage also stores two temporary arrays as persistent mutable data in the filter region (such that accessing the arrays produces an effect on the filter region). When an array enters a sort stage, the filter for that stage adds each array element to one of the temporary arrays, depending on whether the element has a 0 or 1 at the bit position corresponding to the radix for that filter. The filter then copies all the 0 elements followed by all the 1 elements back into the original array, and passes it to the next stage.

5.3 Discussion of Evaluation Results

Expressivity. We were able to use the techniques discussed in Section 3 to write realistic parallel frameworks, with no significant additional challenges. Getting the region and effect annotations correct for the APIs did require some careful thought. However, all the APIs have a similar pattern; once we mastered that pattern, writing the APIs was straightforward.

Table 1 summarizes the effect annotation counts for the framework code. The leftmost data column shows the annotated over the total source lines of code (SLOC), counted with `sloccount`. From the left, the other columns show the number of class (including interface) definitions, class region and effect parameters, class region and effect constraints, region and effect arguments to types, method definitions, method effect summaries, method region and effect parameters, method region and effect constraints, and region and effect arguments to methods. For arguments to class types, the denominator is the total number of types appearing in the program; and for arguments to methods, the denominator is the total number of method invocations. As expected, the annotations are nontrivial; this is simply a cost of the safety guarantee we provide. We believe that production frameworks would have a higher ratio of internal to API code than our simple frameworks do, so the relative annotation overhead would be lower in practice. Further, some type and effect annotations could be inferred. In particular [36] shows how to infer method effect summaries for DPJ as described in [6]; this approach could be extended to inferring arguments to region and effect parameters in types and method invocations.

Table 1. Annotation counts for the framework code

		Classes				Methods				
	SLOC	**Defs**	**Params**	**Constr.**	**Args**	**Defs**	**Summ.**	**Params**	**Constr.**	**Args**
Array	41/97	12	21	0	10/88	20	11	7	4	1/21
Tree	61/169	11	19	0	32/100	18	16	6	2	4/42
Pipeline	35/112	8	9	1	14/44	19	18	2	0	2/28

Table 2. Annotation counts for the client code

		Classes				Methods				
	SLOC	Defs	Params	Constr.	Args	Defs	Summ.	Params	Constr.	Args
Monte Carlo	236/1389	21	10	0	90/492	195	136	8	0	3/350
Spatial Tree	55/172	6	5	0	42/90	10	7	4	0	3/45
Radix Sort	31/102	6	3	0	36/46	11	6	4	0	0/13

Framework Client Experience. Table 2 shows the annotation counts for the client code, with the same layout as Table 1. Overall, the annotation burden is less than for the framework code. As in [6], most of the annotations are method effect summaries and region arguments to types. In the client codes, the arguments to effect variables were simple: either `pure` or one or two read effects. As expected, there were no effect constraints in the client code. Again, type and effect inference [36] could reduce the annotation burden.

It is also instructive to compare the client experience to DPJ as presented in [6,7]. In [6], we wrote Monte Carlo using an index-parameterized array for the first two loops; for the third loop, we encapsulated the reduction sum in a method implemented with locks and declared that method `commutative`. This is not attractive because it puts the burden of writing low-level synchronization code on the application developer. To write the Barnes-Hut center of mass computation using the techniques shown in [6], each tree node would need a distinct type. Because the destination node of a body is not known at the time the body is originally created, we would have to recopy the bodies on insertion into the tree; this is similar to the swapping example discussed in Section 2. This approach works, but it adds overhead. For pipelined radix sort, we could write this program using the features presented in [7] for safe nondeterminism, but we would need to use low-level synchronization techniques in the client program, and we would not get the framework encapsulation or any determinism guarantee.

Overall, the advantages of the framework approach are (1) simplifying the DPJ types exposed to the client, by avoiding index parameterized arrays or recursive types; (2) eliminating low-level code for common patterns such as reductions; (3) supporting operations such as reshuffling that the type system prohibits; and (4) extending the language with more flexible parallel control idioms. On the other hand, the non-framework DPJ code is closer in structure to the original sequential program. This last point is not specific to our work, but is a general issue with frameworks.

6 Related Work

Effects. The seminal work on types and effects for concurrency is FX [19, 24], which adds a region-based type and effect system to a Scheme-like, implicitly parallel language. Later work added effects to object-oriented languages [18,21]. DPJ [6,7] builds upon this work to provide an expressive type and effect system for deterministic-by-default parallelism. None of this work teaches how to write a framework API for safe parallelism using disjoint data structures. Nor does it support mechanisms such as effect constraints and type region parameters that are necessary for generic frameworks.

Several sophisticated effect systems are based on *object ownership* [9,12,22,23]. There are many variants, all unified by the idea that objects define groupings of data on the heap (which we call regions). DPJ is similar in that it uses regions to group data on the heap, but it is different in that a region is specified by an RPL, which is primarily a sequence of declared names like `A:B:C`. DPJ as described in [6] incorporates a notion of ownership by allowing an object to appear first in an RPL (as in *o*`:A:B:C`). Though useful for some parallel patterns, this form is not used in the present work. Ownership domains [2, 31] provide an alternative way to combine declared names with owner objects.

Linear Types. Linear types [37] allow in-place updates while preserving the semantic guarantees of pure functional programming. However, linear types prohibit reference aliasing, making many common patterns of imperative programming awkward or impossible.

Several researchers have worked to make linear types less restrictive while maintaining meaningful guarantees. Fähndrich and DeLine [16] introduced *adoption and focus* to create aliases of a linear reference with a limited lifetime. Clarke and Wrigstad [13] 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. Boyland and others [8, 33] have used *fractional permissions* to enforce linearity of write references, while allowing sharing of read-only references.

Our idea of *disjoint data structures* is related to these mechanisms, but also different from all of them. Our insight is that for parallel traversals over the elements of a data structure, all we care about is whether the elements have different regions in their types. This implies that the elements are distinct objects, but it does not preclude aliasing with other references in the program. DPJ's indexed parameterized arrays [6] provide disjoint regions, but they do so by making the regions explicit in user code, thereby preventing reference swapping as discussed in Section 2.

Enforcing API Contracts. The Eiffel language [35] introduced *design by contract*, which uses preconditions and postconditions to specify interaction between classes. Spec# [4] and the Java Modeling Language (JML) [20] provide ways to write design-by-contract specifications for C# and Java; the specifications can be checked with a combination of static verification and online checking.

Design-by-contract ideas have been applied to concurrent programming languages. Meyer's Systematic Concurrent Object-Oriented Programming (SCOOP) concurrent programming model [25] is based on Eiffel. The Fortress programming language [32] provides a way to write assertions at interface boundaries that can be checked at runtime. X10 [11] 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.

Separation Logic. Recent work on separation logic (SL) [14, 15] shows how to specify abstractions such as barriers, locks, and sets and verify separately that (1) programs using the abstractions are correct and (2) the abstractions are correctly implemented. While similar in spirit to ours, this work does not consider the abstractions we have studied, including data structures containing references to disjoint mutable objects, frameworks with internal parallelism, and frameworks applying methods with unknown effects. The technical mechanisms are also very different (SL-based program verification vs. types and effects). Compared to SL, our effect system is better integrated with the language, easier to use, and amenable to lightweight checking; however, it is probably also less powerful. More complex logics such as SL could be used to prove that a framework implementation satisfies the properties stated in Section 3.3.

Type Constructors. Type constructors are well known in functional languages like Haskell. Recently type constructors have been applied to object-oriented languages [3, 26]. Standard type constructors have no notion of region parameters or effects. Further, we exploit the fact that a parametric type implicitly provides a type constructor: in our language one can define a class `List<type T<region R>>` and use `T` either as the type T bound to the type argument of `List`, or as the type constructor that results from ignoring the binding to the first region parameter in T. This can be viewed as syntactic sugar for a more standard approach, where one would specify the region `R` as a separate class parameter. Ownership Generic Java [27] uses a similar approach in specifying type bounds, but it does not have any notion of effects or support type constructors.

7 Conclusion

We have shown how to use an effect system with polymorphic effects and type constructors to write a generic framework API that enables sound reasoning about its uses. The framework internals can be checked once, and then the compiler can guarantee noninterference for any user program written using the framework. As future work, we would like to explore ways to formally verify properties of the framework implementation that DPJ cannot prove.

References

1. `http://gee.cs.oswego.edu/dl/jsr166/dist/extra166ydocs/index.html?extra166y/package-tree.html`
2. Aldrich, J., Chambers, C.: Ownership domains: Separating aliasing policy from mechanism. In: Vetta, A. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 1–25. Springer, Heidelberg (2004)
3. Altherr, P., Cremet, V.: Adding type constructor parameterization to Java. In: Formal Techniques for Java-like Programs, FTFJP (2007)

4. Barnett, M., et al.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
5. Bocchino, R.: An Effect System and Language for Deterministic-by-Default Parallel Programming. PhD thesis, Univ. of Illinois, Urbana-Champaign, IL (2010)
6. Bocchino, R., et al.: A type and effect system for deterministic parallel Java. In: OOPSLA (2009)
7. Bocchino, R., et al.: Safe nondeterminism in a deterministic-by-default parallel language. In: POPL (2011)
8. Boyland, J.: In: Cousot, R. (ed.) SAS 2003. LNCS, vol. 2694, Springer, Heidelberg (2003)
9. Cameron, N., et al.: Multiple ownership. In: OOPSLA (2007)
10. Cameron, N., Gairing, M., Bateni, M.: A model for Java with wildcards. In: Ryan, M. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 2–26. Springer, Heidelberg (2008)
11. Charles, P., et al.: X10: An object-oriented approach to non-uniform cluster computing. In: OOPSLA (2005)
12. Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA (2002)
13. Clarke, D., Wrigstad, T.: External uniqueness is unique enough. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, Springer, Heidelberg (2003)
14. Dinsdale-Young, T., Dodds, M., Gardner, P., Parkinson, M.J., Vafeiadis, V.: Concurrent abstract predicates. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 504–528. Springer, Heidelberg (2010)
15. Dodds, M., et al.: Modular reasoning for deterministic parallelism. In: POPL (2011)
16. Fähndrich, M., DeLine, R.: Adoption and focus: Practical linear types for imperative programming. In: PLDI (2002)
17. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. Addison-Wesley Longman, Amsterdam (2005)
18. Greenhouse, A., Boyland, J.: An object-oriented effects system. In: Liu, H. (ed.) ECOOP 1999. LNCS, vol. 1628, Springer, Heidelberg (1999)
19. Hammel, R., Gifford, D.: FX-87 performance measurements: Dataflow implementation. Technical Report MIT/LCS/TR-421 (1988)
20. Leavens, G., et al.: Preliminary design of JML: A behavioral interface specification language for Java. SIGSOFT Softw. Eng. Notes (2006)
21. Leino, K., et al.: Using data groups to specify and check side effects. In: PLDI (2002)
22. Li, P., et al.: Mojojojo — More ownership for multiple owners. In: FOOL (2010)
23. Lu, Y., Potter, J.: Protecting representation with effect encapsulation. In: POPL (2006)
24. Lucassen, J., et al.: Polymorphic effect systems. In: POPL (1988)
25. Meyer, B.: Systematic concurrent object-oriented programming. In: CACM (1993)
26. Moors, A., et al.: Generics of a higher kind. In: OOPSLA (2008)
27. Potanin, A., et al.: Generic ownership for generic Java. In: OOPSLA (2006)
28. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly Media, Sebastopol (2007)
29. Singh, J., et al.: SPLASH: Stanford parallel applications for shared-memory. Technical report, Stanford Univ. (1992)
30. Smith, L., Bull, J.: A multithreaded Java grande benchmark suite. In: Third Workshop on Java for High Performance Computing (2001)
31. Smith, M.: Towards an effects system for ownership domains. In: Gao, X.-X. (ed.) ECOOP 2005. LNCS, vol. 3586, Springer, Heidelberg (2005)

32. Sun Microsystems, Inc. The Fortress language specification, version 1.0. Technical report, Sun Microsystems, Inc. (March 2008)
33. Terauchi, T., Aiken, A.: A capability calculus for concurrency and determinism. In: TOPLAS (2008)
34. Thies, W., et al.: StreamIt: A language for streaming applications. In: CC (2002)
35. Thomas, P., Weedon, R.: Object-Oriented Programming in Eiffel, 2nd edn. Addison-Wesley Longman, Amsterdam (1998)
36. Vakilian, M., et al.: Inferring method effect summaries for nested heap regions. In: ASE (2009)
37. Wadler, P.: Linear types can change the world! In: Working Conf. on Programming Concepts and Methods (1990)