A Type and Effect System for Deterministic Parallel Java *

Robert L. Bocchino Jr. Vikram S. Adve Danny Dig
Sarita V. Adve Stephen Heumann Rakesh Komuravelli Jeffrey Overbey
Patrick Simmons Hyojin Sung Mohsen Vakilian

Department of Computer Science
University of Illinois at Urbana-Champaign
dpj@cs.uiuc.edu

Abstract

Today's shared-memory parallel programming models are complex and error-prone. While many parallel programs are intended to be deterministic, unanticipated thread interleavings can lead to subtle bugs and nondeterministic semantics. In this paper, we demonstrate that a practical *type and effect system* can simplify parallel programming by *guaranteeing deterministic semantics* with modular, compile-time type checking even in a rich, concurrent object-oriented language such as Java. We describe an object-oriented type and effect system that provides several new capabilities over previous systems for expressing deterministic parallel algorithms. We also describe a language called Deterministic Parallel Java (DPJ) that incorporates the new type system features, and we show that a core subset of DPJ is sound. We describe an experimental validation showing that DPJ can express a wide range of realistic parallel programs; that the new type system features are useful for such programs; and that the parallel programs exhibit good performance gains (coming close to or beating equivalent, nondeterministic multithreaded programs where those are available).

Categories and Subject Descriptors D.1.3 [*Software*]: Concurrent Programming—Parallel Programming; D.3.1 [*Software*]: Formal Definitions and Theory; D.3.2 [*Software*]: Language Classifications—Concurrent, distributed, and parallel languages; D.3.2 [*Software*]: Language Classifications—Object-oriented languages; D.3.3 [*Software*]: Language Constructs and Features—Concurrent Programming Structures

General Terms Languages, Verification, Performance

Keywords Determinism, deterministic parallelism, effects, effect systems, commutativity

1. Introduction

The advent of multicore processors demands parallel programming by mainstream programmers. The dominant model of concurrency today, multithreaded shared memory programming, is inherently complex due to the number of possible thread interleavings that can cause nondeterministic program behaviors. This nondeterminism causes subtle bugs: data races, atomicity violations, and deadlocks. The parallel programmer today prunes away the nondeterminism using constructs such as locks and semaphores, then *debugs* the program to eliminate the symptoms. This task is tedious, error prone, and extremely challenging even with good debugging tools.

The irony is that a vast number of computational algorithms (though not all) are in fact *deterministic*: a given input is always expected to produce the same output. Almost all scientific computing, encryption/decryption, sorting, compiler and program analysis, and processor simulation algorithms exhibit deterministic behavior. Today's parallel programming models force programmers to implement such algorithms in a nondeterministic notation and then convince themselves that the behavior will be deterministic.

By contrast, a *deterministic-by-default programming model* [9] can *guarantee* that any legal program produces the same externally visible results in all executions with a particular input *unless* nondeterministic behavior is explicitly requested by the programmer in disciplined ways. Such a model can make parallel application development and maintenance easier for several reasons. Programmers do not have to reason about notoriously subtle and difficult issues such as data races, deadlocks, and memory models. They can start with a sequential implementation and incrementally add parallelism, secure in the knowledge that the program behavior

* 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.

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

will remain unchanged. They can use familiar sequential tools for debugging and testing. Importantly, they can test an application only once for each input [19].

Unfortunately, while guaranteed determinism is available for some restricted styles of parallel programming (e.g., data parallel, or pure functional), it remains a challenging research problem to guarantee determinism for imperative, object-oriented languages such as Java, C++, and C#. In such languages, object references, aliasing, and updates to mutable state obscure the data dependences between parts of a program, making it hard to prove that those dependences are respected by the program's synchronization. This is a very important problem as many applications that need to be ported to multicore platforms are written in these languages.

We believe that a *type and effect system* [27, 26, 12, 30] is an important part of the solution to providing guaranteed deterministic semantics for imperative, object-oriented languages. A type and effect system (or effect system for short) allows the programmer to give names to distinct parts of the heap (we call them *regions*) and specify the kind of accesses to parts of the heap (e.g., *read or write effects*) in different parts of the program. The compiler can then check, using simple modular analysis, that all pairs of memory accesses either commute with each other (e.g., they are both reads, or they access disjoint parts of the heap) or are properly synchronized to ensure determinism. A robust type and effect system with minimal runtime checks is valuable because it enables checking at compile time rather than runtime, eliminates unnecessary runtime checks (thus leading to less overhead and/or less implementation complexity), and contributes to program understanding by showing *where* in the code parallelism is expressed – and where code must be rewritten to make parallelism available. Effect annotations can also provide an enforceable contract at interface boundaries, leading to greater modularity and composability of program components. An effect system can be supplemented with runtime speculation [23, 51, 38, 31, 50] or other runtime checks [43, 20, 47, 6] to enable greater expressivity.

In this paper, we develop a new type and effect system for expressing important patterns of deterministic parallelism in imperative, object-oriented programs. FX [33, 27] showed how to use regions and effects in limited ways for deterministic parallelism in a mostly functional language. Later work on object-oriented effects [26, 12, 30] and object ownership [16, 32, 14] introduced more sophisticated mechanisms for specifying effects. However, studying a wide range of *realistic* parallel algorithms has shown us that some significantly more powerful capabilities are needed for such algorithms. In particular, all of the existing work lacks general support for fundamental parallel patterns such as parallel updates on distinct fields of nested data structures, parallel array updates, in-place divide and conquer algorithms, and commutative parallel operations.

Our effect system can support all of the above capabilities, using several novel features. We introduce *region path lists*, or RPLs, which enable more flexible effect summaries, including effects on nested structures. RPLs also allow more flexible subtyping than previous work. We introduce an *index-parameterized array type* that allows references to provably distinct objects to be stored in an array *while still permitting arbitrary aliasing of the objects through references outside the array*. We are not aware of any statically checked type system that provides this capability. We define the notions of *subarrays* (i.e., one array that shares storage with another) and *partition operations*, that together enable in-place parallel divide and conquer operations on arrays. Subarrays and partitioning provide a natural object-oriented way to encode disjoint segments of arrays, in contrast to lower-level mechanisms like separation logic [35] that specify array index ranges directly. We also introduce an *invocation effect*, together with simple *commutativity annotations*, to permit the parallel invocation of operations that may actually interfere at the level of reads and writes, but still commute logically, i.e., produce the same final (logical) behavior. This mechanism supports concurrent data structures such as concurrent sets, hash maps, atomic counters, etc.

We have designed a language called *Deterministic Parallel Java* (DPJ) incorporating these features. DPJ is an extension to Java that enforces deterministic semantics via compile-time type checking. Because of the guaranteed deterministic semantics, existing Java code can be ported to DPJ *incrementally*. Furthermore, porting to DPJ will have minimal impact on program testing: developers can use the same tests and testing methodology for the ported parallel code as they had previously used for their sequential code.

The choice of Java for our work is not essential; similar extensions could be applied to other object-oriented languages, and we are currently developing a version of the language and compiler for C++. We are also exploring how to extend our type system and language to provide disciplined support for explicitly nondeterministic computations.

This paper makes the following contributions:

1. **Novel features.** We introduce a new region-based type and effect system with several novel features (RPLs, index-parameterized arrays, subarrays, and invocation effects) for expressing core parallel programming patterns in imperative languages. These features guarantee determinism at compile-time.
2. **Formal definition.** For a core subset of the type system, we have developed a formal definition of the static and dynamic semantics, and a detailed proof that our system allows sound static inference about noninterference of effects. We present an outline of the formal definition and proof in this paper. The full details are in an accompanying technical report [10] available via the Web [1].

3. **Language Definition.** We have designed a language called DPJ that incorporates the type and effect system into a modern O-O language (Java) in such a way that it supports the full flexibility of the sequential subset of Java, enables incremental porting of Java code to DPJ, and guarantees semantic equivalence between a DPJ program and its obvious sequential Java version. We have implemented a prototype compiler for DPJ that performs the necessary type checking and then maps parallelism down to the ForkJoinTask dynamic scheduling framework.
4. **Empirical evaluation.** We study six real-world parallel programs written in DPJ. This experience shows that DPJ can express a range of parallel programming patterns; that all the novel type system features are useful in real programs; and that the language is effective at achieving significant speedups on these codes on a commodity 24-core shared-memory processor. In fact, in 3 out of 6 codes, equivalent, manually parallelized versions written to use Java threads are available for comparison, and the DPJ versions come close to or beat the performance of the Java threads versions.

The rest of this paper proceeds as follows. Section 2 provides an overview of some basic features of DPJ, and Sections 3–5 explain the new features in the type system (RPLs, arrays, and commutativity annotations). Section 6 summarizes the formal results for a core subset of the language. Section 7 discusses our prototype implementation and evaluation of DPJ. Section 8 discusses related work, and Section 9 concludes.

2. Basic Capabilities

We begin by summarizing some basic capabilities of DPJ that are similar to previous work [33, 30, 26, 14, 15]. We refer to the example in Figure 1, which shows a simple binary tree with three nodes and a method `initTree` that writes into the `mass` fields of the left and right child nodes. As we describe more capabilities of DPJ, we will also expand upon this example to make it more realistic, e.g., supporting trees of arbitrary depth.

Region names. In DPJ, the programmer uses named regions to partition the heap, and writes method effect summaries stating what regions are read and written by each method. A *field region declaration* declares a new name r (called a *field region name*) that can be used as a region name. For example, line 2 declares names `Links`, `L`, and `R`, and these names are used as regions in lines 4 and 5.[1] A field region name is associated with the static class in which it is declared; this fact allows us to reason soundly about effects without alias restrictions or interprocedural alias analysis. A field region name functions like an ordinary class member: it is inherited by subclasses, and outside the scope of its defining class, it must be appropriately qualified (e.g., `TreeNode.L`). A *local region declaration* is similar and declares a region name at local scope.

[1] As explained in Section 3, in general a DPJ region is represented as a *region path list* (RPL), which is a colon-separated list of elements such as `Root:L:L:R` that expresses the nested structure of regions. When a simple name r functions as a region, as shown in this section, it is short for `Root:`r.

```
1  class TreeNode<region P> {
2      region Links, L, R;
3      double mass in P;
4      TreeNode<L> left in Links;
5      TreeNode<R> right in Links;
6      void setMass(double mass) writes P { this.mass = mass; }
7      void initTree(double mass) {
8          cobegin {
9              /* reads Links writes L */
10             left.mass = mass;
11             /* reads Links writes R */
12             right.mass = mass;
13         }
14     }
15 }
```

Figure 1. Basic features of DPJ. Type and effect annotations are italicized. Note that method `initTree` (line 7) has no effect annotation, so it gets the default effect summary of "reads and writes the entire heap."

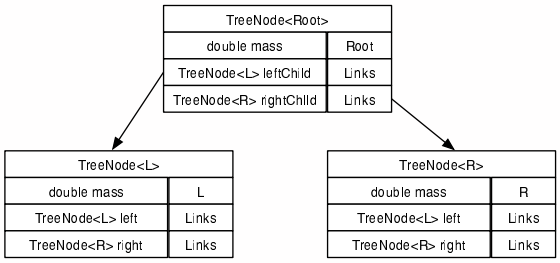

Figure 2. Runtime heap typing from Figure 1

Region parameters. DPJ provides class and method region parameters that operate similarly to Java generic parameters. We declare region parameters with the keyword `region`, as shown in line 1, so that we can distinguish them from Java generic type parameters (which DPJ fully supports). When a region-parameterized class or method is used, region arguments must be provided to the parameters, as shown in lines 4–5. Region parameters enable us to create multiple instances of the same class with their data in different regions.

Disjointness constraints. To control aliasing of region parameters, the programmer may write a disjointness constraint [14] of the form P_1 `#` P_2, where P_1 and P_2 are parameters (or regions written with parameters; see Section 3) that are required to be disjoint. Disjointness of regions is fully explained in Section 3; in the case of simple names, it means the names must be different. The constraints are checked when instantiating the class or calling the method. If the disjointness constraints are violated, the compiler issues a warning.

Partitioning the heap. The programmer may place the keyword `in` after a field declaration, followed by the region,

as shown in lines 3–5. An operation on the field is treated as an operation on the region when specifying and checking effects. This effectively partitions the heap into regions. See Figure 2 for an illustration of the runtime heap typing, assuming the root node has been instantiated with `Root`.

Method effect summaries. Every method (including all constructors) must conservatively summarize its heap effects with an annotation of the form `reads` *region-list* `writes` *region-list*, as shown in line 6. Writes imply reads. When one method overrides another, the effects of the superclass method must contain the effects of the subclass method. For example, if a method specifies a `writes` effect, then all methods it overrides must specify that same `writes` effect. This constraint ensures that we can check effects soundly in the presence of polymorphic method invocation [30, 26]. The full DPJ language also includes *effect variables* [33], to support writing a subclass whose effects are unknown at the time of writing the superclass (e.g., in instantiating a library or framework class); however, we leave the discussion of effect variables to future work.

Effects on local variables need not be declared, because these effects are masked from the calling context. Nor must initialization effects inside a constructor body be declared, because the DPJ type and effect system ensures that no other task can access `this` until after the constructor returns. Read effects on `final` variables are also ignored, because those reads can never cause a conflict. A method or constructor with no externally visible heap effects may be declared `pure`.

To simplify programming and provide interoperability with legacy code, we adopt the rule that no annotation means "reads and writes the entire heap," as shown in Figure 1. This scheme allows ordinary sequential Java to work correctly, but it requires the programmer to add the annotations in order to introduce safe parallelism.

Expressing parallelism. DPJ provides two constructs for expressing parallelism, the `cobegin` block and the `foreach` loop. The `cobegin` block executes each statement in its body as a parallel task, as shown in lines 8–13. The `foreach` loop is used in conjunction with arrays and is described in Section 4.1.

Proving determinism. To type check the program in Figure 1, the compiler does the following. First, check that the summary `writes P` of method `setMass` (line 6) is correct (i.e., it covers all effect of the method). It is, because field `mass` is declared in region `P` (line 3), and there are no other effects. Second, check that the parallelism in lines 8–13 is safe. It is, because the effect of line 10 is `reads Links writes L`; the effect of line 12 is `reads Links writes R`; and `Links`, `L`, and `R` are distinct names. Notice that this analysis is entirely intraprocedural.

3. Region Path Lists (RPLs)

An important concept in effect systems is *region nesting*, which lets us partition the heap hierarchically so we can express that different computations are occurring on different parts of the heap. For example, to extend the code in Figure 1 to a tree of arbitrary depth, we need a tree of nested regions. As discussed in Section 4, we can also use nesting to express that two aggregate data structures (like arrays) are in distinct regions, and the components of those structures (like the cells of the arrays) are in distinct regions, each nested under the region containing the whole structure.

Effect systems that support nested regions are generally based on object ownership [16, 14] or use explicit declarations that one region is under another [30, 26]. As discussed below, we use a novel approach based on chains of elements called *region path lists*, or RPLs, that provides new capabilities for effect specification and subtyping.

3.1 Specifying Single Regions

The region path list (RPL) generalizes the notion of a simple region name r. Each RPL names a single *region*, or set of memory locations, on the heap. The set of all regions partitions the heap, i.e., each memory location lies in exactly one region. The regions are arranged in a tree with a special region `Root` as the root node. We say that one region is *nested under* (or simply under) another if the first is a descendant of the second in the tree. The tree structure guarantees that for any two distinct names r and r', the set of regions under r and the set of regions under r' have empty intersection, and we can use this guarantee to prove disjointness of memory accesses.

Syntactically, an RPL is a colon-separated list of names, called *RPL elements*, beginning with `Root`. Each element after `Root` is a declared region name r,[2] for example, `Root:A:B`. As a shorthand, we can omit the leading `Root`. In particular, a bare name can be used as an RPL, as illustrated in Figure 1. The syntax of the RPL represents the nesting of region names: one RPL is under another if the second is a prefix of the first. For example, `L:R` is under `L`. We write $R_1 \leq R_2$ if R_1 is under R_2.

We may also write a region parameter, instead of `Root`, at the head of an RPL, for example `P:A`, where `P` is a parameter. When a class with a region parameter is instantiated at runtime, the parameter is resolved to an RPL beginning with `Root`. Method region parameters are resolved similarly at method invocation time. Because a parameter `P` is always bound to the same RPL in a particular scope, we can make sound static inferences about parametric RPLs. For example, for all `P`, `P:A` $\leq$ `P`, and `P:A` $\neq$ `P:B` if and only if `A` $\neq$ `B`.

Figure 3 illustrates the use of region nesting and class region parameters to distinguish different fields as well as different objects. It extends the example from Figure 1 by

[2] As noted in Section 2, this can be a package- or class-qualified name such as `C.r`; for simplicity, we use r throughout.

```
1  class TreeNode<region P> {
2      region Links, L, R, M, F;
3      double mass in P:M;
4      double force in P:F;
5      TreeNode<L> left in Links;
6      TreeNode<R> right in Links;
7      void initTree(double mass, double force) {
8          cobegin {
9              /* reads Links writes L:M */
10             left.mass = mass;
11             /* reads Links writes L:F */
12             left.force = force;
13             /* reads Links writes R:M */
14             right.mass = mass;
15             /* reads Links writes R:F */
16             right.force = force;
17         }
18     }
19 }
```

Figure 3. Extension of Figure 1 showing the use of region nesting and region parameters.

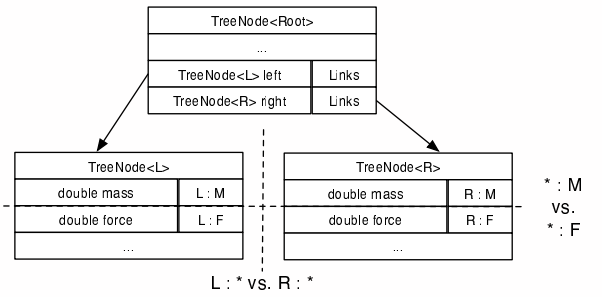

Figure 4. Graphical depiction of the distinctions shown in Figure 3. The * denotes any sequence of RPL elements; this notation is explained further in Section 3.2.

adding a `force` field to the `TreeNode` class, and by making the `initTree` method (line 7) set the `mass` and `force` fields of the left and right child in four parallel statements in a `cobegin` block (lines 9–16).

To establish that the parallelism is safe (i.e., that lines 9–16 access disjoint locations), we place fields `mass` and `force` in distinct regions `P:M` and `P:F`, and the links `left` and `right` in a separate region `Links` (since they are only read). The parameter `P` appears in both regions and `P` is bound to different regions (`L` and `R`) for the left and right subtrees, because of the different instantiations of the parametric type `TreeNode` for the fields `left` and `right`. Because the names `L` and `R` used in the types are distinct, we can distinguish the effects on `left` (lines 10–12) from the effects on `right` (lines 14–16). And because the names `M` and `F` are distinct, we can distinguish the effects on the different fields within an object i.e., lines 10 vs. 14 and lines 12 vs. 16, from each other. Figure 4 shows this situation graphically.

3.2 Specifying Sets of Regions

Partially specified RPLs. To express recursive parallel algorithms, we must specify effects on *sets of regions* (e.g., "all regions under R"). To do this, we introduce *partially specified RPLs*. A partially specified RPL contains the symbol *

```
1  class TreeNode<region P> {
2      region Links, L, R, M, F;
3      double mass in P:M;
4      double force in P:F;
5      TreeNode<P:L> left in Links;
6      TreeNode<P:R> right in Links;
7      TreeNode<*> link in Links;
8      void computeForces() reads Links, *:M writes P:*:F {
9          cobegin {
10             /* reads *:M writes P:F */
11             this.force = (this.mass * link.mass) * R_GRAV;
12             /* reads Links, *:M writes P:L:*:F */
13             if (left != null) left.computeForces();
14             /* reads Links, *:M writes P:R:*:F */
15             if (right != null) right.computeForces();
16         }
17     }
18 }
```

Figure 5. Recursive computation showing the use of partially specified RPLs for effects and subtyping.

("star") as an RPL element, standing in for some unknown sequence of names. An RPL that contains no * is *fully specified*.

For example, consider the code shown in Figure 5. Here we are operating on the same `TreeNode` shown in Figs. 1 and 3, except that we have added (1) a `link` field (line 7) that points to some other node in the tree and (2) a `computeForces` method (line 8) that recursively descends the tree. At each node, `computeForces` follows `link` to another node, reads the `mass` field of that node, computes the force between that node and this one, and stores the result in the `force` field of this node. This computation can safely be done in parallel on the subtrees at each level, because each call writes only the `force` field of `this`, and the operations on other nodes (through `link`) are all reads of the `mass`, which is distinct from `force`. To write this computation, we need to be able to say, for example, that line 13 writes only the left subtree, and does not touch the right subtree.

Distinctions from the left. In lines 11–15 of Figure 5, we need to distinguish the write to `this.force` (line 11) from the writes to the `force` fields in the subtrees (lines 13 and 15). We can use partially specified RPLs to do this. For example, line 8 says that `computeForces` may read all regions under `Links` and write all regions under `P` that end with `F`.

If RPLs R_1 and R_2 are the same in the first n places, they differ in place $n+1$, and neither contains a * in the first $n+1$ places, then (because the regions form a tree) the set of regions under R_1 and the set of regions under R_2 have empty intersection. In this case we say that R_1`:*` and R_2`:*` are *disjoint*, and we know that effects on these two RPLs are noninterfering. We call this a "distinction from the left," because we are using the distinctness of the names to the left of any star to infer that the region sets are non-intersecting. For example, a distinction from the left establishes that the region sets `P:F`, `P:L:*:F`, and `P:R:*:F` (shown in lines 10-15) are disjoint, because the RPLs all start with `P` and differ in the second place.

Distinctions from the right. Sometimes it is important to specify "all fields x in any node of a tree." For example, in lines 10–15, we need to show that the reads of the `mass` fields are distinct from the writes to the `force` fields. We can make this kind of distinction by using different names *after* the star: if R_1 and R_2 differ in the nth place from the right, and neither contains a `*` in the first n places from the right, then a simple syntactic argument shows that their region sets are disjoint. We call this pattern a "distinction from the right," because the names that ensure distinctness appear to the right of any star. For example, in lines 10–15, we can distinguish the reads of `*:M` from the writes to `P:L:*:F` and `P:R:*:F`.

More complicated patterns. More complicated RPL patterns like `Root:*:A:*:B` are supported by the type system. Although we do not expect that programmers will need to write such patterns, they sometimes arise via parameter substitution when the compiler is checking effects.

3.3 Subtyping and Type Casts

Subtyping. Partially specified RPLs are also useful for subtyping. For example, in Figure 5, we needed to write the type of a reference that could point to a `TreeNode<P>`, for any binding to `P`. With fully specified RPLs we cannot do this, because we cannot write a type to which we can assign both `TreeNode<L>` and `TreeNode<R>`. The solution is to use a partially specified RPL in the type, e.g., `TreeNode<*>`, as shown in line 7 of Figure 5. Now we have a type that is flexible enough to allow the assignment, but retains soundness by explicitly saying that we do not know the actual region.

The subtyping rule is simple: $C\texttt{<}R_1\texttt{>}$ is a subtype of $C\texttt{<}R_2\texttt{>}$ if the set of regions denoted by R_1 is included in the set of regions denoted by R_2. We write $R \subseteq R_2$ to denote set inclusion for the corresponding sets of regions. If R_1 and R_2 are fully specified, then $R_1 \subseteq R_2$ implies $R = R_2$. Note that nesting and inclusion are related: $R_1 \leq R_2$ implies $R_1 \subseteq R_2\texttt{:*}$. However, nesting alone does *not* imply inclusion of the corresponding sets. For example, $\texttt{A:B} \leq \texttt{A}$, but $\texttt{A:B} \not\subseteq \texttt{A}$, because `A:B` and `A` denote distinct regions. In Section 6 we discuss the rules for nesting, inclusion, and disjointness of RPLs more formally.

Figure 6 illustrates one possible heap typing resulting from the code in Figure 5. The DPJ typing discipline ensures the object graph restricted to the `left` and `right` references is a tree. However, the full object graph including the `link` references is more general and can even include cycles, as illustrated in Figure 6. Note how our effect system is able to prove that the updates to different subtrees are distinct, even though (1) non-tree edges exist in the graph; and (2) those edges are followed to do possibly overlapping reads.

Type casts. DPJ allows any type cast that would be legal for the types obtained by erasing the region variables. This approach is sound if the region arguments are consistent. For example, given `class B<region R> extends A<R>`, a cast from `A<r>` to `B<r>` is sound, because either the reference is `B<r>`, or it is not any sort of `B`, which will cause a `ClassCastException` at runtime. However, a cast from `Object` to `B<r1>` is unsound and could violate the determinism guarantee, because the `Object` could be a `B<r2>`, which would not cause a runtime exception. The compiler allows this cast, but it issues a warning.

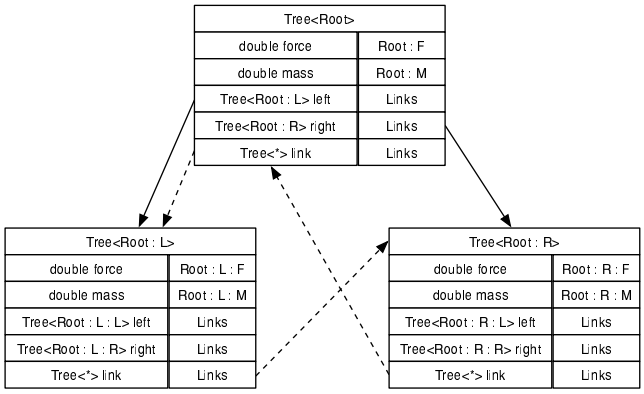

Figure 6. Heap typing from Figure 5. Reference values are shown by arrows; tree arrows are solid, and non-tree arrows are dashed. Notice that all arrows obey the subtyping rules.

4. Arrays

DPJ provides two novel capabilities for computing with arrays: *index-parameterized arrays* and *subarrays*. Index-parameterized arrays allow us to traverse an array of object references and safely update the objects in parallel, while subarrays allow us to dynamically partition an array into disjoint pieces, and give each piece to a parallel subtask.

4.1 Index-Parameterized Arrays

A basic capability of any language for deterministic parallelism is to operate on elements of an array in parallel. For a loop over an array of values, it is sufficient to prove that each iteration accesses a distinct array element (we call this a *unique traversal*). For a loop over an array of references to mutable objects, however, a unique traversal is not enough: we must also prove that any memory locations updated by following references in distinct array cells (possibly through a chain of references) are distinct. Proving this property is very hard in general, if assignments are allowed into reference cells of arrays. No previous effect system that we are aware of is able to ensure disjointness of updates by following references stored in arrays, and this seriously limits the ability of those systems to express parallel algorithms.

In DPJ, we make use of the following insight:

Insight 1. *We can define a special array type with the restriction that an object reference value o assigned to cell n (where n is a natural number constant) of such an array has a runtime type that is parameterized by n. If accesses through cell n touch only region n (even by following a chain*

```
1  class Body<region P> {
2      region Link, M, F;
3      double mass in P:M;
4      double force in P:F;
5      Body<*> link in Link;
6      void computeForce() reads Link, *:M writes P:F {
7          force = (mass * link.mass) * R_GRAV;
8      }
9  }
10
11 final Body<[_]>[]<[_]> bodies = new Body<[_]>[N]<[_]>;
12 foreach (int i in 0, N) {
13     /* writes [i] */
14     bodies[i] = new Body<[i]>();
15 }
16 foreach (int i in 0, N) {
17     /* reads [i], Link, *:M writes [i]:F */
18     bodies[i].computeForce();
19 }
```

Figure 7. Example using an index-parameterized array.

of references), then the accesses through different cells are guaranteed to be disjoint.

We call such an array type an *index-parameterized array*. To represent such arrays, we introduce two language constructs:

1. An *array RPL element* written `[`e`]`, where e is an integer expression.
2. An *index-parameterized array type* that allows us to write the region and type of array cell e using the array RPL element `[`e`]`. For example, we can specify that cell e resides in region `Root:[`e`]` and has type `C<Root:[`e`]>`.

At runtime, if e evaluates to a natural number n, then the static array RPL element `[`e`]` evaluates to the *dynamic array RPL element* `[`n`]`.

The key point here is that we can distinguish `C<[`e_1`]>` from `C<[`e_2`]>` if e_1 and e_2 always evaluate to unequal values at runtime, just as we can distinguish `C<`r_1`>` from `C<`r_2`>`, where r_1 and r_2 are declared names, as discussed in Section 3.1. Obviously, the compiler's capability to distinguish such types will be determined by its ability to prove the inequality of the symbolic expressions e_1 and e_2. This is possible in many common cases, for the same reason that array dependence analysis is effective in many, though not all, cases [24]. The key benefit is that *the type checker has then proved the uniqueness of the target objects, which would not follow from dependence analysis alone*.

In DPJ, the notation we use for index-parameterized arrays is T`[]<`R`>#`i, where T is a type, R is an RPL, `#`i declares a fresh integer variable i in scope over the type, and `[`i`]` may appear as an array RPL element in T or R (or both). This notation says that array cell e (where e is an integer expression) has type $T[i \leftarrow e]$ and is located in region $R[i \leftarrow e]$. For example, `C<r1:[i]>[]<r2:[i]>#i` specifies an array such that cell e has type `C<r1:[`e`]>` and resides in region `r2:[`e`]`. If T itself is an array type, then nested index variable declarations can appear in the type. However, the most common case is a single-dimensional array, which needs only one declaration. For that case, we provide a simplified notation: the user may omit the `#`i and use an underscore (_) as an implicitly declared variable. For example, `C<[_]>[]<[_]>` is equivalent to `C<[i]>[]<[i]>#i`.

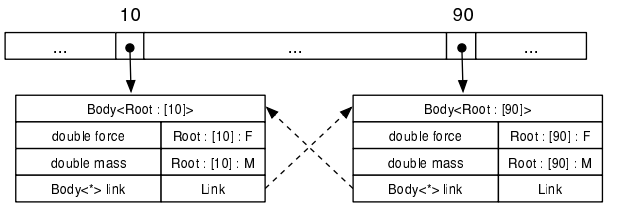

Figure 8. Heap typing from Figure 7. The type of array cell i is parameterized by i. Cross-links are possible, but if any links are followed to access other array cells, the effects are visible.

Figure 7 shows an example, which is similar in spirit to the Barnes-Hut force computation discussed in Section 7. Lines 1–9 declare a class `Body`. Line 11 declares and creates an index-parameterized array `bodies` with `N` cells, such that cell i resides in region `[`i`]` and points to an object of type `Body<[`i`]>`. Figure 8 shows a sample heap typing, for some particular value n of `N`.

Lines 12–15 show a `foreach` loop that traverses the indices $i \in [0, n-1]$ in parallel and initializes cell i with a new object of type `Body<[`i`]>`. The loop is noninterfering because the type of `bodies` says that cell `bodies[`i`]` resides in region `[`i`]`, so distinct iterations i and j write disjoint regions `[`i`]` and `[`j`]`. Lines 16–19 are similar, except that the loop calls `computeForce` on each of the objects. In iteration i of this loop, the effect of line 16 is `reads [`i`]`, because it reads `bodies[`i`]`, together with `reads Link, *:M writes [`i`]:F`, which is the declared effect of method `computeForce` (line 6), after substituting `[`i`]` for `P`. Again, the effects are noninterfering for $i \neq j$.

To maintain soundness, we just need to enforce the invariant that, at runtime, cell `A[`i`]` never points to an object of type `C<[`j`]>`, if $i \neq j$. The compiler can enforce this invariant through symbolic analysis, by requiring that if type `C<[`e_1`]>` is assigned to type `C<[`e_2`]>`, then e_1 and e_2 must always evaluate to the same value at runtime; if it cannot prove this fact, then it must conservatively disallow the assignment. In many cases (as in the example above) the check is straightforward.

Note that because of the typing rules, no two distinct cells of an index-parameterized array can point to the same object. However, it is perfectly legal to reach the same object by following chains of references from distinct array cells, as shown in Figure 8. In that case, in a parallel traversal over the array, either the common object is not updated, in which case the parallelism is safe; or a write effect on the same region appears in two distinct iterations of a parallel loop, in which case the compiler can catch the error.

Note also that while no two cells in an index-parameterized array can alias, references may be freely shared with other

```
1  class QSort<region P> {
2      DPJArrayInt<P> A in P;
3      QSort(DPJArray<P> A) pure { this.A = A; }
4      void sort() writes P:* {
5          if (A.length <= SEQ_LENGTH) {
6              seqSort();
7          } else {
8              /* Shuffle A and return pivot index */
9              int p = partition(A);
10             /* Divide A into two disjoint subarrays at p */
11             final DPJPartitionInt<P> segs =
12                 new DPJPartitionInt<P>(A, p, OPEN);
13             cobegin {
14                 /* writes segs:[0]:* */
15                 new QSort<segs:[0]:*>(segs.get(0)).sort();
16                 /* writes segs:[1]:* */
17                 new QSort<segs:[1]:*>(segs.get(1)).sort();
18             }
19         }
20     }
21 }
```

Figure 9. Writing quicksort with the partition operation. `DPJArrayInt` and `DPJPartitionInt` are specializations to `int` values. In line 12, the argument `OPEN` indicates that we are omitting the partition index from the subarrays, i.e., they are open intervals.

variables (including cells in other index-parameterized arrays), unlike linear types [26, 12, 13]. For example, if cell i of a particular array has type `C<[`i`]>`, the object it points to could be referred to by cell i of any number of other arrays (with the same type), or by any reference of type `C<*>`. Thus, when we are traversing the array, we get the benefit of the alias restriction imposed by the typing, but we can still have as many other outstanding references to the objects as we like.

The pattern does have some limitations: for example, we cannot move an element from position i to position j in the array C`<[i]>[]#i`. However, we can copy the references into a different array C`<*>[]` and shuffle those references as much as we like, though we cannot use those references to update the objects in parallel. We can also make a new copy of element i with type C`<[`j`]>` and store the new copy into position j. This effectively gives a kind of reshuffling, although the copying adds performance overhead. Another limitation is that our `foreach` currently only allows regular array traversals (including strided traversals), though it could be extended to other unique traversals.

4.2 Subarrays

A familiar pattern for writing divide and conquer recursion is to partition an array into two or more disjoint pieces and give each array to a subtask. For example, Figure 9 shows a standard implementation of quicksort, which divides the array in two at each recursive step, then works in parallel on the halves. DPJ supports this pattern with three novel features, which we illustrate with the quicksort example.

First, DPJ provides a class `DPJArray` that wraps an ordinary Java array and provides a view into a contiguous segment of it, parameterized by start position S and length L. In Figure 9, the `QSort` constructor (line 3) takes a `DPJArray` object that represents a contiguous subrange of the caller's array. We call this subrange a *subarray*. Notice that the `DPJArray` object does *not* replicate the underlying array; it stores only a reference to the underlying array, and the values of S and L. The `DPJArray` object translates access to element i into access to element $S + i$ of the underlying array. If $i < 0$ or $i \geq L$, then an array bounds exception is thrown, i.e., access through the subarray must stay within the specified segment of the original array.

Second, DPJ provides a class `DPJPartition`, representing an indexed collection of `DPJArray` objects, all of which point into mutually disjoint segments of the original array. To create a `DPJPartition`, the programmer passes a `DPJArray` object into the `DPJPartition` constructor, along with some arguments that say how to do the splitting. Lines 11–12 of Figure 9 show how to split the `DPJArray` `A` at index `p`, and indicate that position `p` is to be left out of the resulting disjoint segments. The programmer can access segment i of the partition `segs` by saying `segs.get(`i`)`, as shown in lines 15 and 17.

Third, to support recursive computations, we need a slight extension to the syntax of RPLs (Section 3). Notice that we cannot use a simple region name, like r, for the type of a partition segment, because different partitions can divide the same array in different ways. Instead, we allow a `final` local variable z (including `this`) of class type to appear at the head of an RPL, for example z`:r`. The variable z stands in for the object reference o stored into the variable at runtime, which is the actual region. Using the object reference as a region insures that different partitions get different regions, and making the variable `final` ensures that it always refers to the same region.

We make these "z regions" into a tree as follows. If z's type is `C<`R`,...>`, then z is nested under R; the first region parameter of a class functions like the *owner parameter* in an object ownership system [18, 16]. In the particular case of `DPJPartition`, if the type of z is `DPJPartition<`R`>`, then the type of z`.get(`i`)` is z`:[`i`]:*`, where $z \leq R$. Internally, the `get` method uses a type cast to generate a `DPJArray` of type `this:[`i`]:*` that points into the underlying array. The type cast is not sound within the type system, but it is hidden from the user code in such a way that all well-typed uses of `DPJPartition` are noninterfering.

In Figure 9, the sequence of recursive `sort` calls creates a tree of `QSort` objects, each in its own region. The `cobegin` in lines 13–17 is safe because `DPJPartition` guarantees that the segments `segs.get(0)` and `segs.get(1)` passed into the recursive parallel `sort` calls are disjoint. In the user code, the compiler uses the type and effect annotations to prove noninterference as follows. First, from the type of `QSort` and the declared effect of `sort` (line 4), the compiler determines that the effects of lines 15 and 17 are `writes segs:[0]:*` and `writes segs:[1]:*`, as shown. Second, the regions `segs:[0]:*` and `segs:[1]:*` are dis-

joint, by a distinction from the left (Section 3.2). Finally, the effect `writes P:*` in line 4 correctly summarizes the effects of `sort`, because lines 6 and 9 write `P`, lines 15 and 17 write under `segs`, and `segs` is under `P`, as explained above.

Notice that `DPJPartition` can create multiple references to overlapping data with different regions in the types. Thus, there is potential for unsoundness here if we are not careful. To make this work, we must do two things. First, if z_1 and z_2 represent different partitions of the same array, then z_1`.get(0)` and z_2`.get(1)` could overlap. Therefore, we must not treat them as disjoint. This is why we put $*$ at the end of the type z`:[`i`]:*` of z`.get(`i`)`; otherwise we could incorrectly distinguish z_1`:[0]` from z_2`:[1]`, using a distinction from the right. Second, if z has type `DPJPartition<`R`>`, then z`.get(`i`)` has type `DPJArray<`z`:[`i`]:*>` and points into a `DPJArray<`R`>`. Therefore, we must not treat z`:[`i`]:*` as disjoint from R. Here, we simply do not include this distinction in our type system. All we say is that z`:[`i`]:*` $\leq R$. See Section 6.3 and Appendix C.2 for further discussion of the disjointness rules in our type system.

```
1  class IntSet<region P> {
2      void add(int x) writes P { ... }
3      add commuteswith add;
4  }
```

(a) Declaration of `IntSet` class with commutative method `add`

```
1  IntSet<R> set = new IntSet<R>();
2  foreach (int i in 0, N)
3      /* invokes IntSet.add with writes R */
4      set.add(A[i]);
```

(b) Using `commuteswith` for parallelism

```
1  class Adder<region P> {
2      void add(IntSet<P> set, int i)
3        invokes IntSet.add with writes P {
4          set.add(i);
5      }
6  }
7  IntSet<R> set = new IntSet<R>();
8  Adder<R> adder = new Adder<R>();
9  foreach (int i in 0, N)
10     /* invokes IntSet.add with writes R */
11     adder.add(set, A[i]);
```

(c) Using `invokes` to summarize effects

Figure 10. Illustration of `commuteswith` and `invokes`.

5. Commutativity Annotations

Sometimes to express parallelism we need to look at interference in terms of higher-level operations than read and write [29]. For example, insertions into a concurrent `Set` can go in parallel and preserve determinism even though the order of interfering reads and writes inside the `Set` implementation is nondeterministic. Another such example is computing connected components of a graph in parallel.

In DPJ, we address this problem by adding two features. First, classes may contain declarations of the form m `commuteswith` m', where m and m' are method names, indicating that any pair of invocations of the named methods may be safely done in parallel, *regardless of the read and write effects of the methods*. See Figure 10(a). In effect, the `commuteswith` annotation says that (1) the two invocations are *atomic* with respect to each other, i.e., the result will be as if one occurred and then the other; and (2) either order of invocation produces the same result.

The commutativity property itself is not checked by the compiler; we must rely on other forms of checking (e.g., more complex program logic [52] or static analysis [42, 4]) to ensure that methods declared to be commutative are really commutative. In practice, we anticipate that `commuteswith` will be used mostly by library and framework code that is written by experienced programmers and extensively tested. Our effect system does guarantee deterministic results for an application using a commutative operation, assuming that the operation declared commutative is indeed commutative.

Second, our effect system provides a novel *invocation effect* of the form `invokes` m `with` E. This effect records that an invocation of method m occurred with underlying effects E. The type system needs this information to represent and check effects soundly in the presence of commutativity annotations: for example, in line 4 of Fig. 10(b), the compiler needs to record that `add` was invoked there (so it can disregard the effects of other `add` invocations) *and* that the underlying effect of the method was `writes R` (so it can verify that there are no other interfering effects, e.g., reads or writes of `R`, in the invoking code).

When there are one or more intervening method calls between a `foreach` loop and a commutative operation, it may also be necessary for a method effect summary in the *program text* to specify that an invocation occurred inside the method. For example, in Figure 10(c), the `add` method is called through a wrapper object. We could have correctly specified the effect of `Adder.add` as `writes P`, but this would hide from the compiler the fact that `Adder.add` commutes with itself. Of course we could use `commuteswith` for `Adder.add`, but this is highly unsatisfactory: it just propagates the unchecked commutativity annotation out through the call chain in the application code. The solution is to specify the invocation effect `invokes IntSet.add with writes P`, as shown.

Notice that the programmer-specified invocation effect exposes an internal implementation detail (i.e., that a particular method was invoked) at a method interface. However, we believe that such exposure will be rare. In most cases, the effect `invokes` $C.m$ `with` E will be conservatively summarized as E (Section 6.1 gives the formal rules for covering effects). The invocation effect will *only* be used for cases where a commutative method is invoked, and the commutativity information needs to be exposed to the caller. We believe these cases will generally be confined to high-level public API methods, such as `Set.add` in the example given in Figure 10.

Meaning	Symbol	Definition
Programs	*program*	*region** *class** e
Regions	*region*	`region` r
Classes	*class*	`class` C<P> { *field** *method** *comm** }
RPLs	R	`Root` $\mid P \mid z \mid R : r \mid R : [i] \mid R : *$
Fields	*field*	$T\ f$ `in` R_f
Types	T	C<R> $\mid T[]$<R>$\#i$
Methods	*method*	$T\ m(T\ x)\ E\ \{\ e\ \}$
Effects	E	$\emptyset \mid$ `reads` $R \mid$ `writes` $R \mid$ `invokes` $C.m$ `with` $E \mid E \cup E$
Expressions	e	`let` $x = e$ `in` $e \mid$ `this`$.f = z \mid$ `this`$.f \mid z[n] = z \mid z[n] \mid z.m(z) \mid z \mid$ `new` C<R> $\mid$ `new` $T[n]$<R>$\#i$
Variables	z	`this` $\mid x$
Commutativity	*comm*	m `commuteswith` m

Figure 11. Core DPJ syntax. C, P, f, m, x, r, and i are identifiers, and n is a natural number. R_f denotes a fully specified RPL (i.e., containing no $*$).

6. The Core DPJ Type System

We have formalized a subset of DPJ, called *Core DPJ*. To make the presentation more tractable and to focus attention on the important aspects of the language, we make the following simplifications:

1. We present a simple expression-based language, omitting more complicated aspects of the real language such as statements and control flow.
2. Our language has classes and objects, but no inheritance.
3. Region names r are declared at global scope, instead of at class scope. Every class has one region parameter, and every method has one formal parameter.
4. To avoid dealing with integer variables and expressions, we require that array indices are natural number literals.

Removing the first simplification adds complexity but raises no significant technical issues. Adding inheritance raises standard issues for formalizing an object-oriented language. We omit those here in order to focus on the novel aspects of our system, but we describe them in [10]. Removing simplifications 3 and 4 is mostly a matter of bookkeeping. To handle arrays in the full language, we need to prove equivalence and non-equivalence of array index expressions, but this is a standard compiler capability.

We have chosen to make Core DPJ a sequential language, in order to focus on our mechanisms for expressing effects and noninterference. In Section 6.4, we discuss how to extend the formalism to model the `cobegin` and `foreach` constructs of DPJ.

6.1 Syntax and Static Semantics

Figure 11 defines the syntax of Core DPJ. The syntax consists of the key elements described in the previous sections (RPLs, effects, and commutativity annotations) hung upon a toy language that is sufficient to illustrate the features yet reasonable to formalize. A program consists of a number of region declarations, a number of class declarations, and an expression to evaluate. Class definitions are similar to Java's, with the restrictions noted above.

(a) Programs

$\triangleright$ *program*	Valid program	$\triangleright$ *class*	Valid class definition
$\triangleright \Gamma$	Valid environment	$\Gamma \triangleright$ *field*	Valid field
$\Gamma \triangleright$ *method*	Valid method	$\Gamma \triangleright$ *comm*	Valid commutativity annotation

(b) RPLs

$\Gamma \triangleright R$ Valid RPL $\quad \Gamma \triangleright R \leq R'$ R under R'
$\Gamma \triangleright R \subseteq R'$ R included in R'

(c) Types

$\Gamma \triangleright T$ Valid type $\quad \Gamma \triangleright T \leq T'$ T a subtype of T'

(d) Effects

$\Gamma \triangleright E$ Valid effect $\quad \Gamma \triangleright E \subseteq E'$ E a subeffect of E'

(e) Expressions

$\Gamma \triangleright e : T, E$ e has type T and effect E in Γ

Figure 12. Core DPJ type judgments. We extend the judgments to groups of things (e.g., $\Gamma \triangleright$ *field**) in the obvious way.

We define the static semantics of Core DPJ with the judgments stated in Figure 12. The judgments are defined with respect to an environment Γ, where each element of Γ is one of the following:

- A binding $z \mapsto T$ stating that variable z has type T. These elements come into scope when a new variable (`let` variable or formal parameter) is introduced.
- A constraint $P \subseteq R$ stating that region parameter P is in scope and included in region R. These elements come into scope when we capture the type of a variable used for an invocation (see the discussion of expression typing judgments below).
- An integer variable i. These elements come into scope when we are evaluating an array type or new array expression.

The formal rules for making the judgments are stated in full in Appendix A. Below we briefly discuss each of the five groups of judgments.

Programs. These judgments state that a program and its top-level components (classes, methods, etc.) are valid. Most rules just require that the component's components are valid in the surrounding environment. The rule for valid method definitions (METHOD) requires that the method body's type and effect are a subtype and subeffect of the return type and declared effect. These constraints ensure that we can use the method declaration to reason soundly about a method's return type and effect when we are typing method invocation expressions.

RPLs. These judgments define validity, nesting, and inclusion of RPLs. Most rules are a straightforward formal translation of the relations that we described informally in Section 3.2. The key rule states that if R is under R' in some environment, then R is included in $R' : *$ in that environment:

$$\text{(INCLUDE-STAR)}\quad \frac{\Gamma \triangleright R \leq R'}{\Gamma \triangleright R \subseteq R' : *}$$

Types. These define when one type is a subtype of another. The class subtyping rule is just the formal statement of the rule we described informally in Section 3.3:

$$\text{(SUBTYPE-CLASS)}\quad \frac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright C\texttt{<}R\texttt{>} \leq C\texttt{<}R'\texttt{>}}$$

The array subtyping rule is similar:

$$\text{(SUBTYPE-ARRAY)}\quad \frac{\Gamma \cup \{i\} \triangleright R \subseteq R'[i' \leftarrow i] \quad T \equiv T'}{\Gamma \triangleright T[]\texttt{<}R\texttt{>}\#i \leq T'[]\texttt{<}R'\texttt{>}\#i'}$$

Here $\equiv$ means identity of element types up to the names of integer variables i. More flexible element subtyping is not possible without sacrificing soundness. We could allow unsound assignments and check for them at runtime (as Java does for class subtyping of array elements), but this would require that we retain the class region binding information at runtime.

Effects. These judgments define when an effect is valid, and when one effect is a subeffect of another. Intuitively, "E is a subeffect of E'" means that E' contains all the effects of E, i.e., we can use E' as a (possibly conservative) summary of E. The rules for reads, writes, and effect unions are standard [16, 33], but there are two new rules for invocation effects. First, if E' covers E, then an invocation of some method with E' covers an invocation of the same method with E:

$$\text{(SE-INVOKES-1)}\quad \frac{\Gamma \triangleright E \subseteq E'}{\Gamma \triangleright \texttt{invokes}\ C.m\ \texttt{with}\ E \subseteq \texttt{invokes}\ C.m\ \texttt{with}\ E'}$$

Second, we can conservatively summarize the effect `invokes` $C.m$ `with` E as just E:

$$\text{(SE-INVOKES-2)}\quad \frac{}{\Gamma \triangleright \texttt{invokes}\ C.m\ \texttt{with}\ E \subseteq E}$$

Expressions. These judgments tell us how to compute the type and effect of an expression. They also ensure that the types of component expressions (for example at assignments and method parameter bindings) match in a way that guarantees soundness. The rules for field and array access and assignment, variable lookup, and new classes and arrays are straightforward. In the rule for `let` $x = e$ `in` e', we type e, bind x to the type of e, and type e'. If x appears in the type or effect of e', we replace it with $R\texttt{:*}$ to generate a type and effect for the whole expression that is valid in the outer scope.

In the rule for method invocation (INVOKE), we translate the type T_x of the method formal parameter to the current context by creating a fresh region parameter P included in the region R of z's type. This technique is similar to how Java handles the capture of a generic wildcard. Note that simply substituting R for param(C) in translating T_x would not be sound; see [10] for an explanation and an example.

Meaning	Symbol	Definition
RPLs	dR	`Root` $\mid o \mid dR : r \mid dR : [i] \mid dR : [n] \mid dR : *$
Types	dT	C<dR>
Effects	dE	$\emptyset \mid$ `reads` $dR \mid$ `writes` $dR \mid$ `invokes` $C.m$ `with` $dE \mid dE \cup dE$

Figure 13. Dynamic syntax of Core DPJ. dR_f denotes a fully-specified dynamic RPL (i.e., containing no $*$).

We also check that the actual argument type is a subtype of the declared formal parameter type, and we report the invocation of the method with its declared effect.

6.2 Dynamic Semantics

The syntax for entities appearing in the dynamic semantics is shown in Figure 13. At runtime, we have dynamic regions (dR), dynamic types (dT) and dynamic effects (dE), corresponding to static regions (R), types (T) and effects (E) respectively. Dynamic regions and effects are not recorded in a real execution, but here we thread them through the execution state so we can formulate and prove soundness results [16]. We also have object references o, which are the actual values computed during the execution.

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 elements of the form $z \mapsto o$ (variable z is bound to value o) or $P \mapsto dR$ (region parameter P is bound to region dR). $d\Gamma$ defines a natural substitution on RPLs, where we replace the variables with values and the region parameters with regions as specified in the environment. We denote this substitution on RPL R as $d\Gamma(R)$, and we extend this notation to types and effects in the obvious way. Notice that we get the syntax of Figure 13 by applying the substitution $d\Gamma$ to the syntax of Figure 11.

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 because we need to track the actual types of null references to establish soundness. Since the actual implementation does not need to do this tracking, it can just use the single value `null`. Every object reference $o \in \text{Dom}(H)$ has a type, determined when the object is created, and we write $H \triangleright o : dT$ to mean that the reference o has type dT with respect to heap H.

We write the evaluation rules in large-step semantics notation, using the following evaluation function:

$$(e, d\Gamma, H) \rightarrow (o, H', dE),$$

where e is an expression to evaluate, $d\Gamma$ and H give the dynamic context for evaluation, o is the result of the evaluation, H' is the updated heap, and dE represents the effects of the evaluation. A program evaluates to reference o with heap H and effect dE if its main expression is e and $(e, \emptyset, \emptyset) \rightarrow (o, H, dE)$.

Section B of the Appendix states the rules for program evaluation. The rules are standard for an imperative language, except that we record read effects in DYN-FIELD-

ACCESS and DYN-ARRAY-ACCESS and write effects in DYN-FIELD-ASSIGN and DYN-ARRAY-ASSIGN. Rules DYN-LET and DYN-INVOKE accumulate the effects of the component expressions. Note that when we evaluate `new` T we eliminate any $*$ from T in the dynamic type of the new reference, e.g., `new` C`<Root:*>` is the same as `new` C`<Root>`; this rule ensures that all object fields are allocated in fully specified regions. This rule is sound for the same reason that assigning C`<Root>` to a variable of type C`<Root:*>` is sound.

6.3 Soundness

Our key soundness result is that we can define and check a static property of noninterference of effect between expressions in the language, such that static noninterference implies dynamic noninterference. Appendix C states the major steps of the proof in formal terms. We divide the steps into three groups: type and effect preservation (Section C.1), disjointness (Section C.2), and noninterference of effect (Section C.3).We provide further explanation and a full proof in our technical report [10].

Type and effect preservation. In Section C.1, we assert some preliminary definitions and the preservation result. A dynamic environment $d\Gamma$ is valid (**Definition 1**) if the types and RPLs appearing on the right of its bindings are valid, and it is internally consistent. A heap H is valid (**Definition 2**) if the reference stored in every object field or array cell of H is consistent with the declared type of the field or cell, translated to $d\Gamma$. A dynamic environment $d\Gamma$ *instantiates* a static environment Γ (**Definition 3**) if the bindings to variables in $d\Gamma$ are consistent with the bindings to the corresponding variables in Γ, after translation to $d\Gamma$.

Theorem 1 establishes that we can use the static types and effects (Section 6.1) to reason soundly about dynamic types and effects (Section 6.2). It states that if we type an expression e in environment Γ, and we evaluate e in dynamic environment $d\Gamma$, where $d\Gamma$ instantiates Γ, then (a) the evaluation takes a valid heap to a valid heap; (b) the static type of e bounds the dynamic type of the value o that results from the evaluation; and (c) the static effect of e bounds the dynamic effect that results from the evaluation.

Disjoint RPLs. In Section C.2, we formally define a disjointness relation on pairs of RPLs ($\Gamma \triangleright R \# R'$). The relation formalizes distinctions from the left and right, as discussed informally in Section 3.2. **Definition 4** formally expresses how to interpret a dynamic RPL as a set of fully-specified RPLs (i.e., regions). **Definition 5** shows how to associate every object field and array cell with a region of the heap. **Proposition 1** states that disjoint RPLs imply disjoint sets of fully specified regions, i.e., disjoint sets of locations. **Proposition 2** states that at runtime, disjoint fully-specified regions imply disjoint locations.

Noninterference. In Section C.3, we formally define a noninterference relation on pairs of static effects ($\Gamma \triangleright E \# E'$). The rules express four basic facts: (1) reads commute with reads; (2) writes commute with reads or writes if the regions are disjoint; (3) invocations commute with other effects if the underlying effects are disjoint; and (4) two invocations commute if the methods are declared to commute, regardless of interference between the underlying effects.

Theorem 2 expresses the main soundness property of Core DPJ, which is that the execution order of noninterfering expressions does not matter. It states that in a well-typed program, if e and e' are expressions with types T and T' and effects E and E', and E and E' are noninterfering, then either order of evaluating e and e' produces the same values o and o', the same effects dE and dE', and the same final heap H.

The claim is true for dynamic effects from the commutativity of reads, the disjointness results of Section C.2, and the assumed correctness of the commutativity specifications for methods. The claim is true for static effects by the type and effect preservation property above. See [10] for the formal proof.

6.4 Deterministic Parallelism

As discussed in Sections 2 and 4, the actual DPJ language includes `foreach` for parallel loops and `cobegin` for a block of parallel statements. We briefly discuss how to extend the formalism to model these constructs.

We can easily simulate `cobegin` by adding a parallel composition operator $e|e'$, which says to execute e and e' in the same environment, in an unspecified order, with an implicit join at the end of the execution. We can simulate `foreach` by allowing an induction variable i to appear in expressions inside the scope of a `foreach`, mapping i to n over the index range of the `foreach`, and evaluating all e_n in unspecified order. In both cases we can extend the static typing rules to say that for any pair of expressions e and e' as to which the order of execution is unspecified, then the effects of e and e' must be noninterfering.

It follows directly from Theorem 2 that parallel composition of noninterfering expressions produces the same result as sequential composition of those expressions. This guarantees determinism of execution regardless of the order of parallel execution. The formalization of this property is straightforward, and we omit it from our technical report.

7. Evaluation

We have carried out a preliminary evaluation of the language and type system features presented in this paper. Our evaluation addressed the following questions:

- **Expressiveness.** Can the type system express important parallel algorithms on object-oriented data structures? When does it fail to capture parallelism and why?
- **Coverage.** Are each of the *new* features in the DPJ type system important to express one or more of these algorithms?

- **Performance.** For each of the algorithms, what increase in performance is realized in practice? This is a quantitative measure of how much parallelism the type system can express for each algorithm (note that the runtime overheads introduced by DPJ are negligible).

To do the evaluation, we extended Sun's *javac* compiler so that it compiles DPJ into ordinary Java source. We built a runtime system for DPJ using the *ForkJoinTask* framework that will be added to the `java.util.concurrent` standard library in Java 1.7 [2]. *ForkJoinTask* supports dynamic scheduling of lightweight parallel tasks, using a work-stealing scheduler similar to that in Cilk [8]. The DPJ compiler automatically translates `foreach` to a recursive computation that successively divides the iteration space, to a depth that is tunable by the programmer, and it translates a `cobegin` block into one task for every statement. Code using *ForkJoinTask* is compatible with Java threads so an existing multithreaded Java program can be incrementally ported to DPJ. Such code may still have some guarantees, e.g., the DPJ portions will be guaranteed deterministic if the explicitly threaded and DPJ portions are separate phases that do not run concurrently.

Using the DPJ compiler, we studied the following programs: Parallel merge sort, two codes from the Java Grande parallel benchmark suite (a Monte Carlo financial simulation and IDEA encryption), the force computation from the Barnes-Hut n-body simulation [45], k-means clustering from the STAMP benchmarks [34], and a tree-based collision detection algorithm from a large, real-world open source game engine called JMonkey (we refer to this algorithm as Collision Tree). For all the codes, we began with a sequential version and modified it to add the DPJ type annotations. The Java Grande benchmarks are explicitly parallel versions using Java threads (along with equivalent sequential versions), and we compared DPJ's performance against those. We also wrote and carefully tuned the Barnes-Hut force computation using Java threads as part of understanding performance issues in the code, so we could compare Java and DPJ for that one as well.

7.1 A Realistic Example

We use the Barnes-Hut force computation to show how to write a realistic parallel program in DPJ. Figure 14 shows a simplified version of this code. The main simplification is that the `Vector` objects are immutable, with `final` fields (so there are no effects on these objects), whereas our actual implementation uses mutable objects. The class `Node` represents an abstract tree node containing a mass and position. The mass and position represent the actual mass and position of a body (at a leaf) or the center of mass of a subtree (at an inner node). The `Node` class has two subclasses: `InnerNode`, representing an inner node of the tree, and storing an array of children; and `Body`, representing the body data stored at the leaves, and storing a force. The `Tree` class stores the tree,

```
1  /* Abstract class for tree nodes */
2  abstract class Node<region R> {
3      region MP;           /* Region for mass and position */
4      double mass in R:MP; /* Mass */
5      Vector pos in R:MP;  /* Position */
6  }
7
8  /* Inner node of the tree */
9  class InnerNode<region R> extends Node<R> {
10     region Children;
11     Node<R:*>[]<R:Children> children in R:Children;
12 }
13
14 /* Leaf node of the tree */
15 class Body<region R> extends Node<R> {
16     region Force;            /* Region for force */
17     Vector force in R:Force; /* Force on this body */
18
19     /* Compute force of entire subtree on this body */
20     Vector computeForce(Node<R:*> subtree)
21         reads R:*:Children, R:*:MP { ... }
22 }
23
24 /* Barnes-Hut tree */
25 class Tree<region R> {
26     region Tree;                      /* Region for tree */
27     Node<R> root in R:Tree;                   /* Root */
28     Body<R:[i]>[]<R:[i]>#i bodies in R:Tree;  /* Leaves */
29
30     /* Compute forces on all bodies */
31     void computeForces() writes R:* {
32         foreach (int i in 0, bodies.length) {
33             /* reads R:Tree, R:*:Node.Children, R:[i],
34                R:*:Node.MP writes R:[i]:Node.Force */
35             bodies[i].force = bodies[i].computeForce(root);
36         }
37     }
38 }
```

Figure 14. Using DPJ to write the Barnes-Hut force computation.

together with an array of `Body` objects pointing to the leaves of the tree.

The method `Tree.computeForces` does the force computation by traversing the array of bodies and calling the method `Body.computeForce` on each one, to compute the force between the body `this` and `subtree`. If `subtree` is a body, or is sufficiently far away that it can be approximated as a point mass, then `Body.computeForce` computes and returns the pairwise interaction between the nodes. Otherwise, it recursively calls `computeForce` on the children of `subtree`, and accumulates the result.

We use a region parameter on the node classes to distinguish instances of these nodes. Class `Tree` uses the parameters to create an index-parameterized array of references to distinct body objects; the parallel loop in `computeForces` iterates over this array. This allows distinctions from the left for operations on `bodies[i]` (Section 3). We also use distinct region names within each class (in particular, for the force, masses and positions, and the children array) to enable distinctions from the right.

The key fact is that, from the effect summary in line 21 and the code in line 35, the compiler infers the effects shown in lines 33–34. Using distinctions from the left and right, the compiler can now prove that (1) the updates are distinct for distinct iterations of the `foreach`; and (2) all the updates are distinct from the reads. Notice also how the nested RPLs

allow us to describe the entire effect of `computeForces` as `writes R:*`. That is, to the outside world, `computeForces` just writes under the region parameter of `Tree`. Thus with careful use of RPLs, we can enforce a kind of encapsulation of effects, which is important for modular software design.

7.2 Expressiveness and Coverage

We used DPJ to express *all* available parallelism (except for vector parallelism, which we do not consider here) for Merge Sort, Monte Carlo, IDEA, K-Means, and Collision Tree. For Barnes-Hut, the overall program includes four major phases in each time step: tree building; center-of-mass computation; force calculations; and position calculations. Expressing the force, center of mass, and position calculations is straightforward, but we studied only the force computation (the dominant part of the overall computation) for this work. DPJ can also express the tree-building phase, but we would have to use a divide-and-conquer approach, instead of inserting bodies from the root via "hand-over-hand locking," as in in [45].

Briefly, we parallelized each of the codes as follows. MergeSort uses subarrays (Section 4.2) to perform in-place parallel divide and conquer operations for both merge and sort, switching to sequential merge and sort for subproblems below a certain size. Monte Carlo uses index-parameterized arrays (Section 4.1) to generate an array of tasks and compute an array of results, followed by commutativity annotations (Section 5) to update to globally shared data inside a reduction loop. IDEA uses subarrays to divide the input array into disjoint pieces, then uses `foreach` to operate on each of the pieces. Section 7.1 describes our parallel Barnes-Hut force computation. Collision Tree recursively walks two trees, reading the trees and collecting a list of intersecting triangles. At each node, a separate triangle list is computed in parallel for each subtree, and then the lists are merged. Our implementation uses method-local regions to distinguish the writes to the left and right subtree lists. K-Means uses commutativity annotations to perform simultaneous reductions, one for each cluster. Table 1 summarizes the novel DPJ capabilities used for each code.

Table 1. Capabilities Used In The Benchmarks
1. Index-parameterized array; 2. Distinctions from the left; 3. Distinctions from the right; 4. Recursive subranges; 5. Commutativity annotations.

Benchmark	1	2	3	4	5
Merge Sort	-	Y	-	Y	-
Monte Carlo	Y	Y	-	-	Y
IDEA	-	Y	-	Y	-
Barnes-Hut	Y	Y	Y	-	-
Collision Tree	-	Y	-	-	-
K Means	-	-	-	-	Y

Our evaluation and experience showed some interesting limitations of the current language design. To achieve good cache performance in Barnes-Hut, the bodies must be reordered according to their proximity in space on each time step [45]. As discussed in Section 7.1, we use an index-parameterized array to update the bodies in parallel. As discussed in Section 4.1, this requires that we copy each body with the new destination regions at the point of re-insertion. As future work, we believe we can ease this restriction by adding a mechanism for disjointness checking at runtime.

Num Cores	Monte Carlo DPJ	Monte Carlo Java	IDEA DPJ	IDEA Java	Barnes Hut DPJ	Barnes Hut Java
2	2.00	1.80	1.95	1.99	1.98	1.99
3	2.82	2.50	2.88	2.97	2.96	2.94
4	3.56	3.09	3.80	3.91	4.94	3.88
7	5.53	4.65	6.40	6.70	6.79	7.56
12	8.01	6.46	9.99	11.04	11.4	13.65
17	10.02	7.18	12.70	14.90	15.3	19.04
22	11.50	7.98	18.70	17.79	23.9	23.33

Table 2. Comparison of DPJ vs. Java threads performance for Monte Carlo, IDEA encryption, and Barnes Hut.

7.3 Performance

We measured the performance of each of the benchmarks on a Dell R900 multiprocessor running Red Hat Linux with 24 cores, comprising four six-core Xeon processors, and a total of 48GB of main memory. For each data point, we took the minimum of five runs on an idle machine.

We studied multiple inputs for each of the benchmarks and also experimented with different limits for recursive codes. We present results for the inputs and parameter values that show the best performance, since our main aim is to evaluate how well DPJ can express the parallelism in these codes. The sensitivity of the parallelism to input size and/or recursive limit parameters is a property of the algorithm and not a consequence of using DPJ.

Figure 15 presents the speedups of the six programs for $p \in \{1, 2, 3, 4, 7, 12, 17, 22\}$ processors. All speedups are relative to an equivalent sequential version of the program, *with no DPJ or other multithreaded runtime overheads*. All six codes showed moderate to good scalability for all values of p. Barnes-Hut and Merge Sort showed near-ideal performance scalability, with Barnes-Hut showing a superlinear increase for $p = 22$ due to cache effects.

Notably, as shown in Table 2, for the three codes where we have manually parallelized Java threads versions available, the DPJ versions achieved speedups close to (IDEA and Barnes Hut), or better than (Monte Carlo), the Java versions, for the same inputs on the same machines. We believe the Java threads codes are all reasonably well tuned; the two Java Grande benchmarks were tuned by the original authors and the Barnes Hut code was tuned by us. The manually parallelized Monte Carlo code exhibited a similar leveling off in speedup as the DPJ version did beyond about 7 cores because both have a significant sequential component that makes copies of a large array for each parallel task. Overall, in all three programs, DPJ is able to express the available parallelism as efficiently as a lower-level hand coded parallel programming model that provides no guarantees of determinism or even race-freedom.

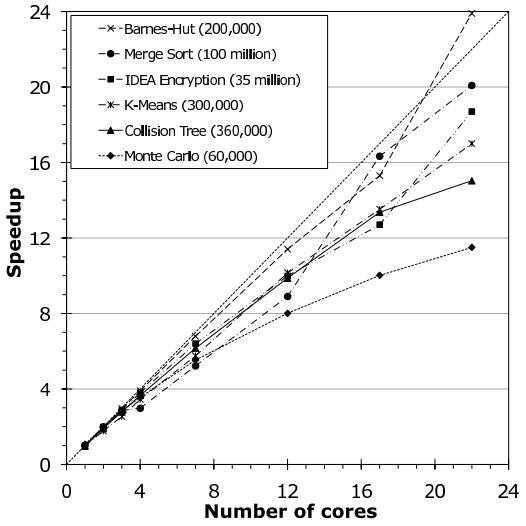

Figure 15. Speedups. Numbers in legend are input sizes.

Our experience so far has shown us that DPJ itself can be very efficient, even though both the compiler and runtime are preliminary. In particular (except for very small runtime costs for the dynamic partitioning mechanism for subarrays), our type system requires no runtime checks or speculation and therefore *adds negligible runtime overhead for achieving determinism*. On the other hand, it is possible that the type system may constrain algorithmic design choices. The limitation on reordering the array of bodies in Barnes-Hut, explained in Section 7.2, is one such example.

7.4 Porting Effort

Table 3 shows the number of source lines changed and the number of annotations, relative to the program size. Program size is given in non-blank, non-comment lines of source code, counted by *sloccount*. The next column shows how many LOC were changed when annotating. The last four columns show (1) the number of declarations using the `region` keyword (i.e., field regions, local regions, and region parameters); (2) the number of RPLs appearing as arguments to `in`, types, methods, and effect summaries; (3) the number of method effect summaries, counting `reads` and `writes` separately; and (4) the number of commutativity annotations. As the table shows, the fraction of lines of code changed was not large, averaging 10.7% of the original lines. Most of the changed lines were due to writing RPL arguments when instantiating types (represented in column four), followed by writing method effect summaries (column five).

More importantly, we believe that the overall effort of writing, testing, and debugging a program with *any* parallel programming model is dominated by the time required to understand the parallelism and sharing patterns (including aliases), and to debug the parallel code. The regions and effects in DPJ provide *concrete guidance to the programmer on how to reason about parallelism and sharing*. Once the programmer understands the sharing patterns, he or she explicitly documents them in the code through region and effect annotations, so other programmers can gain the benefit of his or her understanding.

Program	Total SLOC	Annotated SLOC	Region Decls	RPLs	Effect Summ.	Comm.
MergeSort	295	38 (12.9%)	15	41	7	0
Monte Carlo	2877	220 (7.6%)	13	301	161	1
IDEA	228	24 (10.5%)	8	22	2	0
Barnes-Hut	682	80 (11.7%)	25	123	38	0
CollisionTree	1032	233 (22.6%)	82	408	58	0
K-means	501	5 (1.0%)	0	3	3	1
Total	**5615**	**600 (10.7%)**	143	**898**	**269**	**2**

Table 3. Annotation counts for the case studies.

Further, programming tools can alleviate the burden of writing annotations. We have developed an interactive porting tool, DPJIZER [49], that infers many of these annotations, using iterative constraint solving over the whole program. DPJIZER is implemented as an Eclipse plugin and correctly infers method effect summaries for a program that is already annotated with region information. We are currently extending DPJIZER to infer RPLs, assuming that the programmer declares the regions.

In addition, a good set of defaults can further reduce the amount of manually written annotations. For example, if the programmer does not annotate a class field, its default region could be the RPL *default-parameter*:*field-name*. This default distinguishes both instances of the same class and fields within a class. The programmer can override the defaults if she needs further refinements.

8. Related Work

We group the related work into five broad categories: effect systems (not including ownership-based systems); ownership types (including ownership with effects); unique references; separation logic; and runtime checks.

Effect Systems: The seminal work on types and effects for concurrency is FX [33, 27], which adds a region-based type and effect system to a Scheme-like, implicitly parallel language. Leino et al. [30] and Greenhouse and Boyland [26] first added effects to an object-oriented language. None of these systems can represent arbitrarily nested structures or array partitioning, and they cannot specify arbitrarily large sets of regions. Also, the latter two systems rely on alias restrictions and/or supplementary alias analysis for soundness of effect, whereas DPJ does not.

Ownership Types: Some ownership-based type systems have been combined with effects to enable reasoning about noninterference. Both JOE [16, 46] and MOJO [14] have sophisticated effect systems that allow nested regions and effects. However, neither has the capabilities of DPJ's array partitioning and partially specified RPLs, which are crucial

to expressing the patterns addressed in this paper. JOE's `under` effect shape is similar to DPJ's $*$, but it cannot do the equivalent of our distinctions from the right. JOE allows slightly more precision than our rule LET when a type or effect uses a local variable that goes out of scope, but we have found that this precision is not necessary for expressing deterministic parallelism. MOJO has a wildcard region specifier ?, but it pertains to the orthogonal capability of *multiple ownership*, which allows objects to be placed in multiple regions. Leino's system also has this capability, but without arbitrary nesting.

Lu and Potter [32] show how to use effect constraints to break the owner dominates rule in limited ways while still retaining meaningful guarantees. The `any` context of [32] is identical to `Root:*` in our system, but we can make more fine-grained distinctions. For example, we can conclude that a pair of references stored in variables of type `C<`R_1`:*>` and `C<`R_2`:*>` can never alias, if R_1`:*` and R_2`:*` are disjoint.

Several researchers [11, 3, 28] have described effect systems for enforcing a locking discipline in nondeterministic programs, to prevent data races and deadlocks. Because they have different goals, these effect systems are very different from ours, e.g., they cannot express arrays or nested effects.

Finally, an important difference between DPJ and most ownership systems is that we allow *explicit region declarations*, like [33, 30, 26], whereas ownership systems generally couple region creation with object creation. We have found many cases where a new region is needed but a new object is not, so the ownership paradigm becomes awkward. Supporting field granularity effects also is difficult with ownership.

Unique References: Boyland [13] shows how to use alias restrictions to guarantee determinism for a simple language with pointers. Terauchi and Aiken [48] have extended this work with a type inference algorithm that simplifies the type annotations and elegantly expresses some simple patterns of determinism. Alias restrictions are a well-known alternative to effect annotations for reasoning about heap access, and in some cases they can complement effect annotations [26, 12]. However, alias restrictions severely limit the expressivity of an object-oriented language. It is not clear whether the techniques in [13, 48] could be applied to a robust object-oriented language. Clarke and Wrigstad's external uniqueness [17] is better suited to an object-oriented style, but it is not clear whether external uniqueness is useful for deterministic parallelism.

Separation Logic: Separation logic [40] (SL) is a potential alternative to effect systems for reasoning about shared resources. O'Hearn [35] and Gotsman et al. [25] use SL to check race freedom, though O'Hearn includes some simple proofs of noninterference. Parkinson [37] has extended C# with SL predicates to allow sound inference in the presence of inheritance. Raza et al. [39] show how to use separation logic together with shape analysis for automatic parallelization of a sequential program.

While SL is a promising approach, applying it to realistic programs poses two key issues. First, SL is a *low-level* specification language: it generally treats memory as a single array of words, on which notions of objects and linked data structures must be defined using SL predicates [40, 35]. Second, SL approaches generally *either* require heavyweight theorem proving and/or a relatively heavy programmer annotation burden [37] *or* are fully automated, and thereby limited by what the compiler can infer [25, 39].

In contrast, we chose to start from the extensive prior work on regions and effects, which is more mature than SL for OO languages. As noted in [40], type systems and SL systems have many common goals but have developed largely in parallel; as future research it would be useful to understand better the relationship between the two.

Runtime Checks: A number of systems enforce some form of disciplined parallelism via runtime checks. Jade [43] and Prometheus [5] use runtime checks to guarantee deterministic parallelism for programs that do not fail their checks. Jade also supports a simple form of commutativity annotation [41]. Multiphase Shared Arrays [20] and PPL1 [47] are similar in that they rely on runtime checks that may fail if determinism is violated. None of these systems checks nontrivial sharing patterns at compile time.

Speculative parallelism [7, 23, 51] can achieve determinism with minimal programmer annotations, compared to DPJ. However, speculation generally either incurs significant software overheads or requires special hardware [38, 31, 50]. Grace [7] reduces the overhead of software-only speculation by running threads as separate processes and using commodity memory protection hardware to detect conflicts at page granularity. However, Grace does not efficiently support essential sharing patterns such as (1) fine-grain access distinctions (e.g., distinguishing different fields of an object, as in Barnes-Hut); (2) dynamically scheduled fine-grain tasks (e.g., *ForkJoinTask*); or (3) concurrent data structures, which are usually finely interleaved in memory. Further, unlike DPJ, a speculative solution does not document the parallelization strategy or show how the code must be rewritten to expose parallelism.

Kendo [36] and DMP [21] use runtime mechanisms to guarantee equivalence to some (arbitrary) serial interleaving of tasks; however, that interleaving is not necessarily obvious from the program text, as it is in DPJ. Further, Kendo's guarantee fails if the program contains data races, and DMP requires special hardware support. SharC [6] uses a combination of static and dynamic checks to enforce race freedom, but not necessarily deterministic semantics, in C programs.

Finally, a determinism checker [44, 22] instruments code to detect determinism violations at runtime. This approach is not viable for production runs because of the slowdowns caused by the instrumentation, and it is limited by the cover-

age of the inputs used for the dynamic analysis. However, it is sound for the observed traces.

9. Conclusion

We have described a novel type and effect system, together with a language called DPJ that uses the system to enforce deterministic semantics. Our experience shows that the new type system features are useful for writing a range of programs, achieving moderate to excellent speedups on a 24-processor system with guaranteed determinism.

Our future goals are to exploit region and effect annotations for optimizing memory hierarchy performance; to add runtime support for more flexible operationson index-parameterized arrays; to add support for object-oriented parallel frameworks; and to add support for explicitly nondeterministic algorithms.

Acknowledgments

The following persons provided invaluable insight at various stages of this work: Ralph Johnson, Christopher Rodrigues, Marc Snir, Dan Grossman, Brad Chamberlain, John Brant, Rajesh Karmani, and Maurice Rabb.

References

[1] `http://dpj.cs.uiuc.edu`.

[2] `http://gee.cs.oswego.edu/dl/concurrency-interest`.

[3] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *TOPLAS*, 2006.

[4] F. Aleen and N. Clark. Commutativity analysis for software parallelization: letting program transformations see the big picture. *ASPLOS*, 2009.

[5] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization sets: A dynamic dependence-based parallel execution model. *PPOPP*, 2009.

[6] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking data sharing strategies for multithreaded C. *PLDI*, 2008.

[7] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. *OOPSLA*, 2009.

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *PPOPP*, 1995.

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

[10] 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.

[11] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. *OOPSLA*, 2002.

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

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

[14] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple ownership. *OOPSLA*, 2007.

[15] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *OOPSLA*, 2005.

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

[17] D. Clarke and T. Wrigstad. External uniqueness is unique enough. *ECOOP*, 2003.

[18] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. *OOPSLA*, 1998.

[19] J. Dennis. Keynote address. *PPOPP*, 2009.

[20] J. DeSouza and L. V. Kalé. MSA: Multiphase specifically shared arrays. *LCPC*, 2004.

[21] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. *ASPLOS*, 2009.

[22] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. *SPAA*, 1997.

[23] C. Flanagan and M. Felleisen. The semantics of future and its use in program optimization. *POPL*, 1995.

[24] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. *PLDI*, 2001.

[25] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. *PLDI*, 2007.

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

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

[28] B. Jacobs, F. Piessens, J. Smans, K. R. M. Leino, and W. Schulte. A programming model for concurrent object-oriented programs. *TOPLAS*, 2008.

[29] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. *PLDI*, 2007.

[30] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. *PLDI*, 2002.

[31] W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: a TLS compiler that exploits program structure. *PPOPP*, 2006.

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

[33] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. *POPL*, 1988.

[34] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. *IISWC*, 2008.

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

[36] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. *ASPLOS*, 2009.

[37] M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. *POPL*, 2008.

[38] M. K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. *PPOPP*, 2003.

[39] M. Raza, C. Calcagno, and P. Gardner. Automatic parallelization with separation logic. *ESOP*, 2009.

[40] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. *Symp. on Logic in Comp. Sci.*, 2002.

[41] M. C. Rinard. *The design, implementation and evaluation of Jade: A portable, implicitly parallel programming language*. PhD thesis, Stanford University, 1994.

[42] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *TOPLAS*, 1997.

[43] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of Jade. *TOPLAS*, 1998.

[44] C. Sadowski, S. N. Freund, and C. Flanagan. SingleTrack: A dynamic determinism checker for multithreaded programs. *ESOP*, 2009.

[45] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical report, Stanford University, 1992.

[46] M. Smith. Towards an effects system for ownership domains. *ECOOP*, 2005.

[47] M. Snir. Parallel Programming Language 1 (PPL1), V0.9 — Draft. Technical Report UIUCDCS-R-2006-2969, U. Illinois, 2006.

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

[49] M. Vakilian, D. Dig, R. Bocchino, J. Overbey, V. Adve, and R. Johnson. Inferring Method Effect Summaries for Nested Heap Regions. *ASE*, 2009. To appear.

[50] C. von Praun, L. Ceze, and C. Caşcaval. Implicit parallelism with ordered transactions. *PPOPP*, 2007.

[51] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. *OOPSLA*, 2005.

[52] K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. *PLDI*, 2008.

A. Static Semantics Rules

We divide the static semantics in to five parts: rules for valid program elements (**Figure 16**), rules for validity, nesting, and inclusion of RPLs (**Figure 17**), rules for valid types and subtypes (**Figure 18**), rules for valid effects and subeffects (**Figure 19**), and rules for typing expressions (**Figure 20**).

B. Dynamic Semantics Rules

Figure 21 gives the rules for evaluating programs. If $f : A \to B$ is a function, then $f \cup \{x \mapsto y\}$ is the function $f' : A \cup \{x\} \to B \cup \{y\}$ defined by $f'(a) = f(a)$ if $a \neq x$ and $f'(x) = y$. new(C) is the function taking each field of class C with type T to a null reference of type $d\Gamma(T)$, and new$(T[n])$ is the function taking each index $n' \in [0, n-1]$ to a null reference of type $d\Gamma(T)[i \leftarrow n']$.

The rules for dynamic RPLs, types, and effects are nearly identical to their static counterparts. Instead of writing out all the rules, we describe how to generate them via simple substitution from the rules given in Section A. For every rule given there except RPL-VAR, RPL-PARAM, UNDER-VAR, INCLUDE-PARAM, and INCLUDE-FULL, do the following: (1) append DYN- to the front of the name; (2) replace Γ with H and $[i]$ with $[n]$; and (3) replace R with dR, T with dT, and E with dE. For example, here are the rules for dynamic class subtyping, generated by the substitution above from the rule SUBTYPE-CLASS:

$$\text{(DYN-SUBTYPE-CLASS)} \quad \frac{H \triangleright dR \subseteq dR'}{H \triangleright C\texttt{<}dR\texttt{>} \leq C\texttt{<}dR'\texttt{>}}$$

Then add the following rules:

$$\text{(DYN-RPL-REF)} \quad \frac{H \triangleright o : dT}{H \triangleright o} \qquad \text{(DYN-UNDER-REF)} \quad \frac{H \triangleright o : C\texttt{<}dR\texttt{>}}{H \triangleright o \leq dR}$$

$$\text{(DYN-TYPE-ARRAY)} \quad \frac{H \triangleright dT[i \leftarrow n] \quad H \triangleright dR[i \leftarrow n]}{H \triangleright dT[]\texttt{<}dR\texttt{>}\#i}$$

C. Soundness

C.1 Type and Effect Preservation

Definition 1 (Valid dynamic environments)**.** *A dynamic environment $d\Gamma$ is valid with respect to heap H ($H \triangleright d\Gamma$) if the following hold: (1) for every binding $z \mapsto o \in d\Gamma$, $H \triangleright o : dT$; (2) for every binding $P \mapsto dR \in d\Gamma$, $H \triangleright dR$; and (3) if $\texttt{this} \mapsto o \in d\Gamma$, then $H \triangleright o : C\texttt{<}dR\texttt{>}$, and param$(C) \mapsto dR \in d\Gamma$.*

Definition 2 (Valid heaps)**.** *A heap H is valid ($\triangleright H$) if for each $o \in$ Dom(H), one of the following holds:*

1. *(a) $H \vdash o : C\texttt{<}dR\texttt{>}$ and (b) $H \triangleright C\texttt{<}dR\texttt{>}$ and (c) for each field $T\ f\ \texttt{in}\ R_f \in$ def(C), if $H(o)(f)$ is defined, then $H \triangleright H(o)(f) : dT$ and $H \triangleright dT$ and $H \triangleright dT \leq T[o \leftarrow \texttt{this}][dR \leftarrow$ param$(C)]$; or*
2. *(a) $H \triangleright o : dT[]\texttt{<}dR\texttt{>}\#i$ and (b) $H \triangleright dT[]\texttt{<}dR\texttt{>}\#i$ and (c) if $H(o)(n)$ is defined, then $H \triangleright H(o)(n) : dT'$ and $H \triangleright dT$ and $H \triangleright dT' \leq dT[i \leftarrow n]$.*

Definition 3 (Instantiation of static environments)**.** *A dynamic environment $d\Gamma$ instantiates a static environment Γ ($H \triangleright d\Gamma \leq \Gamma$) if $\triangleright\Gamma$, $\triangleright H$, and $H \triangleright d\Gamma$; the same variables appear in Dom(Γ) as in Dom$(d\Gamma)$; and for each pair $z \mapsto T \in \Gamma$ and $z \mapsto o \in d\Gamma$, $H \triangleright v : dT$ and $H \triangleright dT \leq d\Gamma(T)$.*

Theorem 1 (Preservation)**.** *For a well-typed program, if $\Gamma \triangleright e : T, E$ and $H \triangleright d\Gamma \leq \Gamma$ and $(e, d\Gamma, H) \to (o, H', dE)$, then (a) $\triangleright H'$; and (b) $H' \triangleright dT \leq d\Gamma(T)$, where $H' \triangleright o : dT$; and (c) $H' \triangleright dE$; and (d) $H' \triangleright dE \subseteq d\Gamma(E)$.*

(PROGRAM) $\dfrac{\triangleright \mathit{class}^* \quad \emptyset \triangleright e : T, E}{\triangleright \mathit{class}^*\ e}$ (CLASS) $\dfrac{\{\texttt{this} \mapsto C\texttt{<}P\texttt{>}\} \triangleright \mathit{field}^*\ \mathit{method}^*\ \mathit{comm}^*}{\triangleright \texttt{class}\ C\texttt{<}P\texttt{>}\ \{\ \mathit{field}^*\ \mathit{method}^*\ \mathit{comm}^*\ \}}$ (ENV) $\dfrac{\forall z \mapsto T \in \Gamma.\Gamma \triangleright T \quad \forall P \subseteq R \in \Gamma.\Gamma \triangleright R}{\triangleright \Gamma}$

(FIELD) $\dfrac{\Gamma \triangleright T \quad \Gamma \triangleright R}{\Gamma \triangleright T\ f\ \texttt{in}\ R}$ (METHOD) $\dfrac{\Gamma \triangleright T_r, T_x, E \quad \Gamma' = \Gamma \cup \{x \mapsto T_x\} \quad \Gamma' \triangleright e : T', E' \quad \Gamma' \triangleright T' \leq T_r \quad \Gamma' \triangleright E' \subseteq E}{\Gamma \triangleright T_r\ m(T_x\ x)\ E\ \{\ e\ \}}$

(COMM) $\dfrac{\texttt{this} \mapsto C\texttt{<}P\texttt{>} \in \Gamma \quad \exists \text{def}(C.m), \text{def}(C.m')}{\Gamma \triangleright m\ \texttt{commuteswith}\ m'}$

Figure 16. Rules for valid program elements. $\text{def}(C.m)$ means the definition of method m in class C.

(RPL-ROOT) $\dfrac{}{\Gamma \triangleright \texttt{Root}}$ (RPL-VAR) $\dfrac{z \mapsto C\texttt{<}R\texttt{>} \in \Gamma}{\Gamma \triangleright z}$ (RPL-PARAM) $\dfrac{\texttt{this} \mapsto C\texttt{<}P\texttt{>} \in \Gamma \vee P \subseteq R \in \Gamma}{\Gamma \triangleright P}$ (RPL-NAME) $\dfrac{\Gamma \triangleright R \quad \texttt{region}\ r \in \mathit{program}}{\Gamma \triangleright R : r}$

(RPL-INDEX) $\dfrac{\Gamma \triangleright R \quad i \in \Gamma}{\Gamma \triangleright R : [i]}$ (RPL-STAR) $\dfrac{\Gamma \triangleright R}{\Gamma \triangleright R : *}$ (UNDER-ROOT) $\dfrac{}{\Gamma \triangleright R \leq \texttt{Root}}$ (UNDER-VAR) $\dfrac{z \mapsto C\texttt{<}R\texttt{>} \in \Gamma}{\Gamma \triangleright z \leq R}$

(UNDER-NAME) $\dfrac{\Gamma \triangleright R \leq R'}{\Gamma \triangleright R : r \leq R'}$ (UNDER-INDEX) $\dfrac{\Gamma \triangleright R \leq R'}{\Gamma \triangleright R : [i] \leq R'}$ (UNDER-STAR) $\dfrac{\Gamma \triangleright R \leq R'}{\Gamma \triangleright R : * \leq R'}$ (UNDER-INCLUDE) $\dfrac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright R \leq R'}$

(INCLUDE-NAME) $\dfrac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright R : r \subseteq R' : r}$ (INCLUDE-INDEX) $\dfrac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright R : [i] \subseteq R' : [i]}$ (INCLUDE-STAR) $\dfrac{\Gamma \triangleright R \leq R'}{\Gamma \triangleright R \subseteq R' : *}$

(INCLUDE-PARAM) $\dfrac{P \subseteq R \in \Gamma}{\Gamma \triangleright P \subseteq R}$ (INCLUDE-FULL) $\dfrac{\Gamma \triangleright R \subseteq R_f}{\Gamma \triangleright R_f \subseteq R}$

Figure 17. Rules for valid RPLs, nesting of RPLs, and inclusion of RPLs. The nesting and inclusion relations are reflexive and transitive (obvious rules omitted).

(TYPE-CLASS) $\dfrac{\exists \text{def}(C) \quad \Gamma \triangleright R}{\Gamma \triangleright C\texttt{<}R\texttt{>}}$ (TYPE-ARRAY) $\dfrac{\Gamma \cup \{i\} \triangleright T, R}{\Gamma \triangleright T[]\texttt{<}R\texttt{>}\#i}$

(SUBTYPE-CLASS) $\dfrac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright C\texttt{<}R\texttt{>} \leq C\texttt{<}R'\texttt{>}}$ (SUBTYPE-ARRAY) $\dfrac{\Gamma \cup \{i\} \triangleright R \subseteq R'[i' \leftarrow i] \quad T \equiv T'}{\Gamma \triangleright T[]\texttt{<}R\texttt{>}\#i \leq T'[]\texttt{<}R'\texttt{>}\#i'}$

Figure 18. Rules for valid types and subtypes. $\text{def}(C)$ means the definition of class C. $T \equiv T'$ means that T and T' are identical up to the names of variables i.

(EFFECT-EMPTY) $\dfrac{}{\Gamma \triangleright \emptyset}$ (EFFECT-READS) $\dfrac{\Gamma \triangleright R}{\Gamma \triangleright \texttt{reads}\ R}$ (EFFECT-WRITES) $\dfrac{\Gamma \triangleright R}{\Gamma \triangleright \texttt{writes}\ R}$ (EFFECT-INVOKES) $\dfrac{\exists \text{def}(C.m) \quad \Gamma \triangleright E}{\Gamma \triangleright \texttt{invokes}\ C.m\ \texttt{with}\ E}$

(EFFECT-UNION) $\dfrac{\Gamma \triangleright E \quad \Gamma \triangleright E'}{\Gamma \triangleright E \cup E'}$ (SE-EMPTY) $\dfrac{}{\Gamma \triangleright \emptyset \subseteq E}$ (SE-READS) $\dfrac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright \texttt{reads}\ R \subseteq \texttt{reads}\ R'}$ (SE-WRITES) $\dfrac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright \texttt{writes}\ R \subseteq \texttt{writes}\ R'}$

(SE-READS-WRITES) $\dfrac{\Gamma \triangleright R \subseteq R'}{\Gamma \triangleright \texttt{reads}\ R \subseteq \texttt{writes}\ R'}$ (SE-INVOKES-1) $\dfrac{\Gamma \triangleright E \subseteq E'}{\Gamma \triangleright \texttt{invokes}\ C.m\ \texttt{with}\ E \subseteq \texttt{invokes}\ C.m\ \texttt{with}\ E'}$

(SE-INVOKES-2) $\dfrac{}{\Gamma \triangleright \texttt{invokes}\ C.m\ \texttt{with}\ E \subseteq E}$ (SE-UNION-1) $\dfrac{\Gamma \triangleright E \subseteq E' \vee \Gamma \triangleright E \subseteq E''}{\Gamma \triangleright E \subseteq E' \cup E''}$ (SE-UNION-2) $\dfrac{\Gamma \triangleright E' \subseteq E \quad \Gamma \triangleright E'' \subseteq E}{\Gamma \triangleright E' \cup E'' \subseteq E}$

Figure 19. Rules for valid effects and subeffects.

(LET) $\dfrac{\Gamma \triangleright e : C\texttt{<}R\texttt{>}, E \quad \Gamma \cup \{x \mapsto C\texttt{<}R\texttt{>}\} \triangleright e' : T', E'}{\Gamma \triangleright \texttt{let}\ x = e\ \texttt{in}\ e' : T'[x \leftarrow R : *], E \cup E'[x \leftarrow R : *]}$ (FIELD-ACCESS) $\dfrac{T\ f\ \texttt{in}\ R_f \in \text{def}(C) \quad \texttt{this} \mapsto C\texttt{<}\text{param}(C)\texttt{>} \in \Gamma}{\Gamma \triangleright \texttt{this}.f : T, \texttt{reads}\ R_f}$

(FIELD-ASSIGN) $\dfrac{\texttt{this} \mapsto C\texttt{<}\text{param}(C)\texttt{>} \in \Gamma \quad z \mapsto T \in \Gamma \quad T'\ f\ \texttt{in}\ R_f \in \text{def}(C) \quad \Gamma \triangleright T \leq T'}{\Gamma \triangleright \texttt{this}.f = z : T, \texttt{writes}\ R_f}$

(ARRAY-ACCESS) $\dfrac{z \mapsto T[]\texttt{<}R\texttt{>}\#i \in \Gamma}{\Gamma \triangleright z[n] : T[i \leftarrow n], \texttt{reads}\ R[i \leftarrow n]}$ (ARRAY-ASSIGN) $\dfrac{\{z \mapsto T[]\texttt{<}R\texttt{>}\#i, z' \mapsto T'\} \subseteq \Gamma \quad \Gamma \triangleright T' \leq T[i \leftarrow n]}{\Gamma \triangleright z[n] = z' : T', \texttt{writes}\ R[i \leftarrow n]}$

(INVOKE) $\dfrac{\{z \mapsto C\texttt{<}R\texttt{>}, z' \mapsto T\} \subseteq \Gamma \quad T_r\ m(T_x\ x)\ E\ \{\ e\ \} \in \text{def}(C) \quad \Gamma \cup \{P \subseteq R\} \triangleright T \leq T_x[\texttt{this} \leftarrow z][\text{param}(C) \leftarrow P]}{\Gamma \triangleright z.m(z') : T_r[\texttt{this} \leftarrow z][\text{param}(C) \leftarrow R], \texttt{invokes}\ C.m\ \texttt{with}\ E[\texttt{this} \leftarrow z][\text{param}(C) \leftarrow R]}$

(VAR) $\dfrac{z \mapsto T \in \Gamma}{\Gamma \triangleright z : T, \emptyset}$ (NEW-CLASS) $\dfrac{\Gamma \triangleright C\texttt{<}R\texttt{>}}{\Gamma \triangleright \texttt{new}\ C\texttt{<}R\texttt{>} : C\texttt{<}R\texttt{>}, \emptyset}$ (NEW-ARRAY) $\dfrac{\Gamma \triangleright T[]\texttt{<}R\texttt{>}\#i}{\Gamma \triangleright \texttt{new}\ T[n]\texttt{<}R\texttt{>}\#i\ : T[]\texttt{<}R\texttt{>}\#i, \emptyset}$

Figure 20. Rules for typing expressions. $\text{param}(C)$ means the parameter of class C.

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

$$\text{(DYN-FIELD-ACCESS)}\quad \frac{\texttt{this} \mapsto o \in d\Gamma \quad H \triangleright o : C\texttt{<}dR\texttt{>} \quad T\ f\ \texttt{in}\ R_f \in \text{def}(C)}{(\texttt{this}.f, d\Gamma, H) \to (H(o)(f), H, \texttt{reads}\ d\Gamma(R_f))}$$

$$\text{(DYN-FIELD-ASSIGN)}\quad \frac{\{\texttt{this} \mapsto o, z \mapsto o'\} \subseteq d\Gamma \quad H \triangleright o : C\texttt{<}dR\texttt{>} \quad T\ f\ \texttt{in}\ R_f \in \text{def}(C)}{(\texttt{this}.f = z, d\Gamma, H) \to (o', H \cup \{o \mapsto (H(o) \cup \{f \mapsto o'\})\}, \texttt{writes}\ d\Gamma(R_f))}$$

$$\text{(DYN-ARRAY-ACCESS)}\quad \frac{z \mapsto o \in d\Gamma \quad H \triangleright o : dT[]\texttt{<}dR\texttt{>}\#i}{(z[n], d\Gamma, H) \to (H(o)(n), H, \texttt{reads}\ dR[i \leftarrow n])}$$

$$\text{(DYN-ARRAY-ASSIGN)}\quad \frac{\{z \mapsto o, z' \mapsto o'\} \subseteq d\Gamma \quad H \triangleright o : dT[]\texttt{<}dR\texttt{>}\#i}{(z[n] = z', d\Gamma, H) \to (o', H \cup \{o \mapsto (H(o) \cup \{n \mapsto o'\})\}, \texttt{writes}\ dR[i \leftarrow n])}$$

$$\text{(DYN-INVOKE)}\quad \frac{H \vdash o : C\texttt{<}dR\texttt{>} \quad T_r\ m(T_x\ x)\ E\ \{\ e\ \} \in \text{def}(C) \quad (e, \{\texttt{this} \mapsto o, \text{param}(C) \mapsto dR, x \mapsto o'\}, H) \to (o'', H', dE)}{(z.m(z'), \{z \mapsto o, z' \mapsto o'\} \cup d\Gamma, H) \to (o'', H', \texttt{invokes}\ C.m\ \texttt{with}\ dE)}$$

$$\text{(DYN-NEW-CLASS)}\quad \frac{o \notin \text{Dom}(H) \quad H' = H \cup \{o \mapsto \text{new}(C)\} \quad H' \triangleright o : C\texttt{<}d\Gamma(R[: * \leftarrow \epsilon])\texttt{>}}{(\texttt{new}\ C\texttt{<}R\texttt{>}, d\Gamma, H) \to (o, H', \emptyset)}$$

$$\text{(DYN-NEW-ARRAY)}\quad \frac{o \notin \text{Dom}(H) \quad H' = H \cup \{o \mapsto \texttt{new}(T[n])\} \quad H' \triangleright o : d\Gamma(T)[]\texttt{<}d\Gamma(R[: * \leftarrow \epsilon])\texttt{>}}{(\texttt{new}\ T[n]\texttt{<}R\texttt{>}\#i, d\Gamma, H) \to (o, H', \emptyset)}$$

Figure 21. Rules for program evaluation.

$$\text{(DISJOINT-LEFT-NAME)}\quad \frac{r \neq r' \quad \Gamma \triangleright R \leq R_f : r \quad \Gamma \triangleright R' \leq R_f : r'}{\Gamma \triangleright R \# R'}$$

$$\text{(DISJOINT-LEFT-INDEX)}\quad \frac{i \neq i' \quad \Gamma \triangleright R \leq R_f : [i] \quad \Gamma \triangleright R' \leq R_f : [i']}{\Gamma \triangleright R \# R'}$$

$$\text{(DISJOINT-LEFT-NAME-INDEX)}\quad \frac{\Gamma \triangleright R \leq R_f : r \quad \Gamma \triangleright R' \leq R_f : [i]}{\Gamma \triangleright R \# R'}$$

$$\text{(DISJOINT-RIGHT-NAME)}\quad \frac{r \neq r'}{\Gamma \triangleright R : r \# R' : r'}$$

$$\text{(DISJOINT-RIGHT-INDEX)}\quad \frac{i \neq i'}{\Gamma \triangleright R : [i] \# R' : [i']}$$

$$\text{(DISJOINT-RIGHT-NAME-INDEX)}\quad \frac{}{\Gamma \triangleright R : r \# R' : [i]}$$

$$\text{(DISJOINT-NAME)}\quad \frac{\Gamma \triangleright R \# R'}{\Gamma \triangleright R : r \# R' : r}$$

$$\text{(DISJOINT-INDEX)}\quad \frac{\Gamma \triangleright R \# R'}{\Gamma \triangleright R : [i] \# R' : [i]}$$

Figure 22. Rules for disjointness of RPLs. The disjointness relation is symmetric (obvious rule omitted).

C.2 Disjointness

Figure 22 gives the rules for concluding that two static RPLs are disjoint; we extend them to dynamic RPLs as in Section B.

Definition 4 (Set interpretation of dynamic RPLs)**.** *Let $\triangleright H$ and $H \triangleright dR$. Then $S(dR, H)$ is defined as follows: (1) $S(dR_f, H) = \{dR_f\}$; (2) $S(dR : r, H) = \{dR_f : r | dR_f \in S(dR, H)\}$; (3) $S(dR : [n], H) = \{dR_f : [n] | dR_f \in S(dR, H)\}$; and (4) $S(dR : *, H) = \{dR_f | H \triangleright dR_f \leq dR\}$.*

Definition 5 (Region of a field or array cell)**.** *If $H \triangleright o : C\texttt{<}dR\texttt{>}$ and $T\ f\ \texttt{in}\ R_f \in \text{def}(C)$, then region$(o, f, H) = R_f[\texttt{this} \leftarrow o][\text{param}(C) \leftarrow dR]$. If $H \triangleright o : dT[]\texttt{<}dR\texttt{>}\#i$, then region$(o, n, H) = dR[i \leftarrow n]$.*

$$\text{(NI-READ)}\quad \frac{}{\Gamma \triangleright \texttt{reads}\ R \# \texttt{reads}\ R'}$$

$$\text{(NI-READ-WRITE)}\quad \frac{\Gamma \triangleright R \# R'}{\Gamma \triangleright \texttt{reads}\ R \# \texttt{writes}\ R'}$$

$$\text{(NI-WRITE)}\quad \frac{\Gamma \triangleright R \# R'}{\Gamma \triangleright \texttt{writes}\ R \# \texttt{writes}\ R'}$$

$$\text{(NI-INVOKES-1)}\quad \frac{\Gamma \triangleright E \# E'}{\Gamma \triangleright \texttt{invokes}\ C.m\ \texttt{with}\ E \# E'}$$

$$\text{(NI-INVOKES-2)}\quad \frac{m\ \texttt{commuteswith}\ m' \in \text{def}(C)}{\Gamma \triangleright \texttt{invokes}\ C.m\ \texttt{with}\ E \# \texttt{invokes}\ C.m'\ \texttt{with}\ E'}$$

$$\text{(NI-EMPTY)}\quad \frac{}{\Gamma \triangleright \emptyset \# E}$$

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

Figure 23. The noninterference relation on effects. Noninterference is symmetric (obvious rule omitted).

Proposition 1 (Disjointness of region sets)**.** *If $H \triangleright dR \# dR'$, then $S(dR, H) \cap S(dR', H) = \emptyset$.*

Proposition 2 (Distinctness of disjoint regions)**.** *If $H \triangleright$ region$(o, f, H)\#$region(o', f', H), then either $o \neq o'$ or $f \neq f'$; and if $H \triangleright$ region$(o, n, H)\#$region(o', n', H), then either $o \neq o'$ or $n \neq n'$.*

C.3 Noninterference of Effect

Figure 23 gives the noninterference relation on static effects. We extend this relation to dynamic effects as in Section B.

Theorem 2 (Soundness of noninterference)**.** *If $\Gamma \triangleright e : T, E$ and $\Gamma \triangleright e' : T', E'$ and $\Gamma \triangleright E \# E'$ and $H \triangleright d\Gamma \leq \Gamma$ and $(e, d\Gamma, H) \to (o, H', dE)$ and $(e', d\Gamma, H') \to (o', H'', dE')$, then there exists H''' such that $(e', d\Gamma, H) \to (o', H''', dE')$ and $(e, d\Gamma, H''') \to (o, H'', dE)$.*