A Type and Effect System for Deterministic Parallelism in Object-Oriented Languages

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

Department of Computer Science
University of Illinois at Urbana-Champaign
201 N. Goodwin Ave., Urbana, IL 61801
{bocchino,vadve}@cs.uiuc.edu

Abstract

We describe a type and effect system for ensuring deterministic semantics in a concurrent object-oriented language. Our system provides several new capabilities over previous work, including support for linear arrays (important in parallel update traversals), flexible effect specifications and subtyping (important for, e.g., tree-based algorithms), dynamic partitioning into subarrays (important for divide-and-conquer algorithms), and a novel invocation effect for handling higher-level commutative operations such as set insert. We informally describe the key type system features, formally define a core subset of our system, and explain the steps leading to the key soundness result, i.e., that the type and effect annotations allow us to reason soundly about parallel noninterference between sections of code. Finally, we describe our experience with using the system to express realistic parallel algorithms, which validates the importance of the new type system features.

1. Introduction

A parallel programming language has deterministic semantics if it ensures that any legal program produces the same externally visible results in all executions with a particular input. Such a language 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 follow an incremental development strategy, starting with a largely sequential implementation and successively parallelizing it as appropriate, secure in the knowledge that the program behavior will remain unchanged. Program testing, bug reproduction, and debugging can also be significantly simpler than with a non-deterministic parallel language or library.

Deterministic parallel semantics can be enforced via a variety of approaches: language-based solutions with static checking [16, 7, 32], run-time speculation [13, 34, 26, 19, 33], or a combination of static and runtime checks [28, 12, 31, 1]. We believe a language-based solution with static checking, where feasible, is the most attractive because it provides at least three benefits. First, *determinism guarantees are enforced at compile time*, rather than runtime, leading to earlier error detection and correction. Second, static checking eliminates most or all of the overhead and/or complex hardware imposed by the runtime mechanisms. Third, a robust static type system contributes to program understanding by showing *where* in the code parallelism is available – or where code needs to be rewritten to make it available.

Language support for deterministic parallelism is available today for certain styles of parallel programming, such as read-only sharing, functional programming, and certain data-parallel constructs. However, today's programming models generally do not guarantee determinism in the presence of object and array references that may be aliased, passed into method calls, and used to update mutable state. For example, given an array of references in Java, it would be difficult to prove using static analysis that the objects referred to are distinct, so they may be safely updated in parallel. With the emerging trend towards many-core architectures, this is a very important deficiency as many applications that will need to be ported to such architectures are written in imperative, object-oriented languages such as C++, Java and C#.

In this paper, we develop a new *type and effect system* [16, 15, 6, 18] for expressing important patterns of deterministic parallelism in imperative, object-oriented programs. A programmer can use our type system to partition the heap at field-granularity, and to specify effects on the heap. Using simple modular checking, the compiler can ensure that the specifications are correct and that tasks are properly synchronized to ensure determinism.

This work is a first step towards a more complete deterministic programming language. To be useful in real-world settings, the language will also require run-time techniques for more complex sharing patterns the static type system cannot express and support for non-deterministic algorithms like branch-and-bound search. Designing such a language is beyond the scope of this work. Our goal here is to lay the foundations for such a language by developing a type system that is powerful enough to express common patterns of deterministic parallelism, yet simple enough to be checked statically with practical, modular compiler techniques (i.e., without sophisticated interprocedural analysis or a complex theorem prover).

To design such a type system, we can build on previous work. FX [21, 16] shows how to use regions and effects in limited ways for deterministic parallelism in a mostly functional language. Later work on object-oriented effects [15, 6, 18] and object ownership [10, 20, 8] has introduced sophisticated mechanisms for specifying effects. However, studying a wide range of *realistic* programs has shown us that some significantly more powerful capabilities are needed for such programs. In particular, all of the existing work lacks general support for certain fundamental parallel patterns such as parallel array updates, certain important nested effect shapes, "in-place" divide and conquer algorithms, and commutative parallel operations.

The major technical contributions of this paper are as follows:

First: We introduce a *linear array type* that allows references to provably distinct objects to be stored in an array, so that we can express (and check) that operations on the objects can safely be done in parallel, *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.

Second: We introduce a novel way of expressing nested types and effects called *region path lists*, or RPLs. For example, when annotating methods, it is often important to be able to express effect shapes like "writes the left subtree but not the right subtree" or "reads field A of every node but writes field B only under this node." Most ownership-based effect systems cannot express effects on sets of regions like this at all. A few, like JOE [10, 30], can do the first but not the second. We provide a novel way to do both. RPLs also allow more flexible subtyping than previous work.

Third: We introduce the notion of *subarrays* (i.e., one array that shares storage with another) and a *partition operation* that allows 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 [23] that specify array index ranges directly.

Fourth: We 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.

Fifth: For a core subset of the type system, we present a formal definition of the static and dynamic semantics, and we outline a proof that our system allows sound static inference about noninterference of effect. We present the formal elements in more detail in an accompanying technical report [3].

Sixth: The features we describe are part of a language we are developing called *Deterministic Parallel Java* (DPJ). We study five realistic parallel algorithms written in DPJ. Our results show that DPJ can express a range of parallel programming patterns; that all the major type system features except the commutativity annotations are useful in these 5 codes (we expect commutativity to be very useful in other codes); and that the language is effective at achieving significant speedups on these codes on a 24-core SMP.

The rest of this paper proceeds as follows. The next two sections give an overview of some basic features of DPJ, and then an intuitive explanation of the new features in the type system. Section 4 summarizes the formal results for a core subset of the language. Section 5 discusses our evaluation of applications written in DPJ. Section 6 discusses related work, and Section 7 concludes.

2. Basic Capabilities

To provide some context for our work, we briefly illustrate some basic capabilities of DPJ. These capabilities are similar to ones appearing in previous work [21, 18, 15, 8, 9]. We refer to the toy example shown in Fig. 1.

Expressing parallelism. DPJ provides two constructs for expressing parallelism, the `cobegin` block and `foreach` loop. The `cobegin` block executes each statement in its body as a parallel task, as shown in lines 7–10. The `foreach` loop is used in conjunction with arrays and is described in Section 3.

Declaring regions. DPJ allows *field region declarations* (line 2) that declare new region names r. A field region declaration is similar to the declarations described in [15, 18], except that our region declarations are associated with the static class (there are no "instance regions"); this fact allows us to reason soundly about effects without alias restrictions or interprocedural alias analysis. Field regions are inherited like Java fields. *Local regions* (not shown) are similar and declare a region at method local scope.

```
1   class C<R> {
2     region r1, r2;
3     int x in R;
4     int setX(int x) writes R { this.x = x; }
5     static int main() {
6       C<r1> c1 = new C<r1>();
7       C<r2> c2 = new C<r2>();
8       cobegin {
9         c1.setX(0); // effect = writes r1
10        c2.setX(1); // effect = writes r2
11      }
12    }
13  }
```

Figure 1. Basic features for regions, effects, and parallelism. New syntax is highlighted in bold.

Region parameters. DPJ provides class and method region parameters that operate similarly to Java generic parameters. We denote region parameters with $\langle \ldots \rangle$, as shown in lines 1 (class parameters) and 4 (method parameters). In our actual DPJ implementation, we denote region parameters with $\langle\langle \ldots \rangle\rangle$ to distinguish them from Java generic parameters (see [25] for an alternative approach).

Partitioning the heap. Region names R (either declared names or parameters) can be used to partition the heap by placing `in` R after a field declaration, as shown in line 2. An operation on the field is then treated as an operation on the region when specifying and checking effects; this approach is similar to the techniques shown in [18, 15], except that those systems have no region parameters.

Method effect summaries. Every method and constructor must conservatively summarize its heap effects with a declaration of the form `reads [region-list] writes [region-list]`, as shown in line 4. Writes imply reads. When one method overrides another, the effects of the superclass method must contain the effects of the subclass method, so that we can check effects soundly in the presence of polymorphic method invocation [18, 15].

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 memory is not visible until the constructor returns) *unless* a reference to `this` may escape the constructor body. If a method or constructor has no externally visible heap effects, it 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." This scheme allows ordinary sequential Java to work correctly, but it requires the programmer to add the annotations in order to introduce safe parallelism. This approach is similar to how Java handles *raw types*.

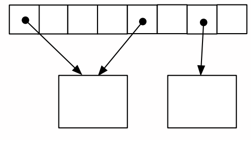
Proving determinism. To type check the program in Fig. 1, the compiler does the following. First, check that `writes P` is a correct summary of the effects of method `setX` (line 4). It is, because field x is declared in region P of class C. Second, check that the the parallelism in lines 9–10 is safe. It is, because the effect of line 9 is `writes r1`, the effect of line 10 is `writes r2`, and `r1` and `r2` are distinct names. Notice that this analysis is entirely intraprocedural.

3. New Capabilities

The capabilities discussed in the previous section allow some sophisticated effect distinctions, but they are not sufficient to express common parallel algorithms. Here we explain four novel elements of DPJ that support realistic parallel algorithms: linear arrays, region path lists, subranges, and commutativity annotations.

3.1 Linear 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



(a)

```
1  class C<P> {
2    int x in P;
3    void update(int y)
4      writes P { x = y; }
5  }
6
7  C<P>[]<[P]> A =
8    new C<P>[N]<[P]>;
9  foreach (int i in 0, N)
10   A[i] = new C<[i]>();
11 foreach (int i in 0, N)
12   A[i].update(i);
```

(b)

Figure 2. (a) A nonlinear array. Updating the objects in parallel through the references causes a data race, even if the traversal is unique. (b) Example using a linear array.

array of values, it is sufficient to prove that each iteration accesses a distinct array element (we call this a *unique traversal*). For an array of references to mutable objects, however, a unique traversal is not enough: we must also prove that the array is *linear*, i.e., no two references in distinct cells point to the same object. See Fig. 2(a). Proving linearity is very hard in general, if arbitrary assignments are allowed into reference cells of arrays. No previous effect system that we are aware of is able to ensure array linearity, and this seriously limits the usefulness of those systems for expressing parallel algorithms.

In DPJ, we make use of the following insight:

Insight 1. *We can define an array type called a* linear array. *We can use types to restrict assignment so that a particular object reference value o is only ever assigned to cell n (where n is a fixed integer constant) of any linear array at runtime. Since no object can be assigned to both cell n and cell n' of the same linear array, for $n \neq n'$, every linear array is guaranteed to be linear.*

To represent linear arrays, we do three things:

1. We introduce a *dynamic array region* written $[n]$, where n is a natural number. At runtime, cell n of every array is located in region $[n]$.
2. We introduce a *static array region* written $[e]$, where e is an integer expression. If e evaluates to n at runtime, then $[e]$ evaluates to $[n]$.
3. We introduce a *linear array type* that allows us to write the type of array cell i using the array region $[i]$. For instance, we can specify that the type of cell i is $C\langle[i]\rangle$.

Fig. 2(b) shows an example. Lines 1–4 declare a class $C\langle P\rangle$ using the basic capabilities from Section 2. Line 6 declares and creates a linear array A. The declaration $C\langle P\rangle[]\langle P\rangle$ says that A is an array of objects, such that the type of cell $A[i]$ is $C\langle[i]\rangle$. Formally, to generate the type of $A[i]$, we replace the parameter name P with $[i]$. On the right-hand side, we are creating an array of the same type with N elements.

To maintain soundness, we just need to enforce the invariant that, at runtime, cell $A[n]$ never points to an object of type $C\langle[n']\rangle$, if $n \neq n'$. One way to enforce this invariant through static checking is by requiring that if $C\langle[e]\rangle$ is assigned to $C\langle[e']\rangle$, then e and e' must always evaluate to the same value at runtime; if we cannot prove this fact, then we conservatively disallow the assignment. We can use standard symbolic analysis for comparing expressions e and e'. In many cases (as in the example above) the check is a straightforward comparison of induction variables.

Note that linear arrays can share references, unlike linear types [15, 6, 7]: when we are traversing the array, we get the benefit of the linearity restriction, 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 shuffle the order of references in the array and maintain the typing discipline. However, note that we can put the same references in another data structure, such as an array or set, and shuffle those references as much as we like; we just cannot use that data structure to do the parallel traversal. Another limitation is that our `foreach` only allows very regular array traversals (including strided traversals), though it could be extended to other unique traversals.

3.2 Region Path Lists (RPLs)

An important concept in effect systems is *region nesting*: without nesting, region names are extremely limited. For example, suppose we have two distinct arrays A and A'. If all we can say is that array cell n is in region $[n]$, then we can distinguish updates to different array elements from each other, but we cannot distinguish $A[i]$ from $A'[i]$. We would like to have a pair of distinct regions r and r' such that each region of $A[i]$ is nested under r, and each region of $A'[i]$ is nested under r', where $r \neq r'$. Then we can distinguish both between individual array cells *and* between whole arrays.

Effect systems that support nested regions are generally based on object ownership [10, 8] or use explicit declarations that one region is under another [18, 15]. As discussed below, we use a novel approach based on chains of elements called *region path lists*, or RPLs, that provides new capabilities for effects and subtyping.

Fully Specified RPLs: The region path list (RPL) generalizes the notion of a simple region name such as r or [0] discussed above. RPLs are nested, and the nesting forms a tree. There are three forms of RPLs: the *basic form*, the *z form*, and the *P form*. These three forms are called *fully specified RPLs*, because they each fully specify a region of the heap. In what follows, we denote an RPL R, and if we want to say that the RPL is fully specified, we write R_f.

Basic form. The fundamental form of RPL is a colon-separated list of names, called *RPL elements*, beginning with the special name `Root`, which represents the root of the tree. Each element after `Root` is either a declared region name r[1] or an array index element $[e]$, for example, `Root` : r : [0]. We can also use a simple name r or $[e]$ as an RPL, as discussed in the previous sections; in this case, the initial `Root` : is implicit. The syntax of the RPL represents the nested structure of regions: one region is nested under another if the second is a prefix of the first. For example, $A : B$ is nested under A. We write $R \leq R'$ if R is nested under R.

z form. The second form of RPL is to write the name of a `final` local variable z (including `this`) of class type, instead of `Root`, at the head of an RPL, for example $z : r$. In this case, the variable z stands in for the object reference bound to z at runtime. If z's type is $C\langle R, \ldots\rangle$, then $z \leq R$; the first region parameter of a class in this case functions like the *owner parameter* in an object ownership system [11, 10]. This technique allows us to create a tree of *object references*. This form of RPL is particularly useful for divide-and-conquer algorithms, as described in in Section 3.3.

P form. The third form of RPL is to write a region parameter P, instead of `Root`, at the head of an RPL, for example $P : r$. Lines 2–5 of Fig. 3 show how to use parameterized RPLs to build a tree. At runtime, the parameter P of every `Tree` object is bound to a basic or z form RPL such as `Root : L : R`; there are no P form RPLs at runtime, because all parameters are substituted away. However, because P is always bound to the same RPL in a particular scope, we can make sound static inferences like P : `L : R` $\leq P$: `L` and P : `L` is disjoint from P : `R`.

[1] This can be a package- or class-qualified name such as $C.r$, but to keep the presentation simple we use r throughout the discussion.

```
 1  class Tree<P> {
 2    region RD, WR, L, R;
 3    int x in P : RD, y in P : WR;
 4    Tree<P : L> left in P : L;
 5    Tree<P : R> right in P : R;
 6    int recursiveSum() reads Root:*:RD {
 7      int sum = x;
 8      if (left != null) sum += left.recursiveSum();
 9      if (right != null) sum += right.recursiveSum();
10      return sum;
11    }
12    void update(Tree<Root> root)
13     reads Root:*:RD, writes P:WR {
14      y = root.recursiveSum();
15    }
16  }
```

Figure 3. A tree-based computation.

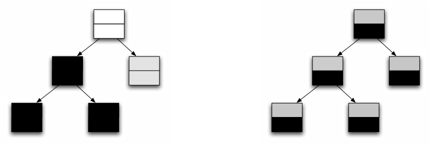

(a) (b)

Figure 4. Distinctions from the left and right. The squares are objects, the smaller rectangles are fields, and the arrows are object reference links. The two shades (black and gray) illustrate the two sets of regions. (a) Distinctions from the left, e.g. `Root` $: r : *$ vs. `Root` $: r' : *$. (b) Distinctions from the right, e.g., `Root` $: * : r$ vs. `Root` $: * : r'$.

RPLs also allow a more general form of linear array declarations: $C\langle P\rangle[]\langle P$ `under` $R\rangle$, where R is an RPL. This notation says that array cell i is located in region $R : [i]$ and has type $C\langle R : [i]\rangle$. If the parameter name P is not needed in the type of the cell, then we can just write R, e.g., $C[]\langle R\rangle$. If the `under` R is omitted (as in Section 3.1) then the default R is `Root`.

Effect Specification and Partially Specified RPLs: The point of RPLs is to specify and check effects. Of course we can use a fully specified RPL to specify an effect like `writes` R_f. However, to take any advantage of the nested structure of RPLs, we need to be able to refer to a *set* of regions, such as "all regions nested under R." To do this we need *partially specified RPLs*, which refer to *sets* of regions.

Distinctions from the left. For recursive algorithms, we must specify shapes like "any region under r" and distinguish this shape from "any region under r'." We do this by permitting the symbol $*$ ("star") to appear at the end of an RPL, standing in for an unknown sequence of names. For example, $r : *$ refers to all regions R such that $R \leq r$. This form of RPL is similar to the `under` effect shape of JOE [10]. When we distinguish an effect $R_f : *$ from an effect $R'_f : *$, we call this a "distinction from the left," because we are using the distinctness of the names to the left of the star, and the tree shape of the region hierarchy, to infer that the shapes are distinct. See Fig. 4(a).

Distinctions from the right. Sometimes it is important to specify "all fields x in any node of a tree (or in any object of a linear array)." To support this pattern, we allow names to appear *after* the star; we call this pattern a "distinction from the right," because the names that ensure distinctness appear to the right of any star. See Fig. 4(b) for a picture; Fig. 3 provides a code example. In English, the effect specification in line 13 of Fig. 3 says "reads region `RD` of any node and writes region `WR` of this node." With this specification we can easily prove that invocations of `update` on distinct `Tree` objects (for example, stored in a linear array) have disjoint effects.

More complicated patterns. More complicated RPL patterns like `Root` $: * : r : * : r'$ are supported by the type system (and sometimes arise via parameter substitution when the compiler is checking effects), but the programmer should never have to think about anything so complicated in practice.

In DPJ we write $R \subseteq R'$ to mean that the set of RPLs contained in R is also contained in R'. Note that if R and R' are fully specified, then $R \subseteq R'$ implies $R = R'$. Note that nesting alone does *not* imply inclusion, e.g., $A : B \leq A$ but $A : B \not\subseteq A$.

Subtyping: Partially specified RPLs are also useful for subtyping. For example, suppose we have a linear array A and we wish to write the type of a reference that can alias any cell of A. With fully specified RPLs we cannot do this, because we are not allowed to assign $C\langle[n]\rangle$ to $C\langle[n']\rangle$, if $n \neq n'$. The solution is to use a partially specified RPL, e.g., `Root` $: *$, in the type. 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: $C\langle R\rangle$ is a subtype of $C\langle R'\rangle$ if $R \subseteq R'$. Our rule is more permissive than most ownership systems in which identity of regions is required. The `any` context of [20] is identical to `Root` $: *$ in our system, but we can make more fine-grained distinctions. For example, we can conclude that references stored in fields of $C\langle R : *\rangle$ and $C\langle R' : *\rangle$ can never alias, if R and R' are disjoint.

3.3 Subranges

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 subproblem. DPJ supports this pattern with two built-in classes, together with the z form of RPL discussed in Section 3.2.

The class `DPJArray` wraps an ordinary Java array and provides a view into it. In addition to array access (via get and set methods, similar to Java's `ArrayList`), `DPJArray` provides a method `subarray(intS,int`L`)` that creates a new `DPJArray` of length L pointing into the storage of A starting at position S. Notice that the `subarray` operation does *not* replicate the underlying array; it stores only a reference to the underlying array, and the segment information (start and length). Access to element i of the subarray is translated to access to element $S + i$ by the `DPJArray` object. If $i > L$, then an array bounds exception is thrown, i.e., access through the subarray must stay within the specified segment of the original array.

The class `DPJPartition` is an indexed collection of `DPJArray` objects, all of which point into mutually disjoint segments of the original array. We create a `DPJPartition` by passing a `DPJArray` object into the `DPJPartition` constructor, along with some arguments that say how to do the splitting.

Figure 5 shows how we write quicksort using `DPJArray` and `DPJPartition`. Notice that in lines 14 and 15, we use the local variable `segs` in the RPL used to instantiate the `Quicksort` object, as discussed in Section 3.2. This technique allows us to create a tree of `Quicksort` objects, with each in its own region.

3.4 Commutativity Annotations

Sometimes to achieve parallelism we need to look at interference in terms of higher-level operations than read and write. For example, updates to 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 (e.g., [17]).

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 Fig. 6(a).

```
 1  class Quicksort<R> {
 2    DPJArrayInt<R> A in R;
 3    Quicksort(DPJArray<R> A) { this.A = A; }
 4    void sort() writes R : * {
 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<R> segs =
12          new DPJPartitionInt<R>(A, p, OPEN);
13        cobegin {
14          new Quicksort<segs:[0]>(segs.get(0)).sort();
15          new Quicksort<segs:[1]>(segs.get(1)).sort();
16        }
17      }
18    }
19  }
```

Figure 5. 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 subranges, i.e., they are open intervals.

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

```
class IntSet<P> {
  void insert(int x) writes P { ... }
  insert commuteswith insert;
}
```

(b) Using `commuteswith` for parallelism

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

(c) Using `invokes` to summarize effects

```
class Inserter<P> {
  void insert(IntSet<P> s, int i)
    invokes IntSet::insert with writes P {
    s.insert(i);
}
IntSet<R> = new IntSet<R>();
Inserter<R> I = new Inserter<R>();
foreach (int i in 0, N)
  // Effect is 'invokes IntSet::insert with writes R'
  I.insert(set, A[i]);
```

Figure 6. Illustration of `commuteswith` and `invokes`.

The commutativity property itself is not checked by the compiler; we must rely on other forms of verification (testing, model checking, etc.) 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.

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. This information is exactly what the type system needs to represent and check effects soundly: for example, in line 4 of Fig. 6(b), the compiler needs to know that `insert` was invoked there (so it can disregard the effects of other `insert` 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).

Meaning	Symbol	Definition
Programs	*program*	$\overline{\textit{region}}\ \overline{\textit{class}}\ e$
Regions	*region*	$\texttt{region}\ r$
Classes	*class*	$\texttt{class}\ C\langle P\rangle\ \{\ \overline{\textit{field}}\ \overline{\textit{method}}\ \overline{\textit{comm}}\}$
RPLs	R_f	$\texttt{Root} \mid P \mid z \mid R_f : r$
	R	$R_f \mid R : * \mid R : r$
Fields	*field*	$T\ f\ \texttt{in}\ R_f$
Types	T	$\mathcal{N} \mid C\langle R\rangle$
Methods	*method*	$T\ m(T\ x)\ E\ \{\ e\ \}$
Effects	E	$\emptyset \mid \texttt{reads}\ R \mid \texttt{writes}\ R \mid$ $\texttt{invokes}\ C.m\ \texttt{with}\ E \mid E \cup E$
Expressions	e	$\texttt{let}\ x = e\ \texttt{in}\ e \mid \texttt{this}.f = z \mid \texttt{this}.f \mid$ $z.m(z) \mid z \mid \texttt{new}\ T \mid \texttt{null}$
Variables	z	$\texttt{this} \mid x$
Commutativity	*comm*	$m\ \texttt{commuteswith}\ m$

Figure 7. Core DPJ syntax. C, P, f, m, x, and r are identifiers. $\mathcal{N}$ represents the type of `null`.

The invocation effect also allows the *programmer* to specify (when writing a method effect summary) that an invocation occurred. For example, in Fig. 6(c), the method `Inserter::insert`, which simply wraps a call to `IntSet::insert`, could simply specify its effects as `writes` P. This would be sound but conservative, and it would hide from the compiler the fact that `Inserter::insert` commutes with itself. Of course we could use `commuteswith` for `Inserter::insert`, but this is 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::insert with writes` P, as shown.

4. The Core DPJ Type System

We formalize 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:

First: We use a sequential language to illustrate the type system. Once we can prove noninterference of effect, then extending the formalism to a language with fork-join parallel constructs is straightforward but unenlightening [3].

Second: We present a simple expression-based language with classes and objects, but no inheritance. Adding more complex syntax, including statements and control flow, adds complexity but raises no significant technical issues. Inheritance raises some subtle issues for formalizing the language but we omit those here for lack of space; again, they are described in [3].

Third: Every class has one region parameter, every method has one formal parameter, and `this` is nested under the class parameter.

Fourth: We omit the formalization of arrays and array regions. Arrays are formalized similarly to classes, except that we use indexed cells instead of named regions, and we have to be careful about parameter substitution in typing the cells of linear arrays. Array regions $[e]$ are formalized almost identically to named regions r, except that we compare integer expressions e (using symbolic analysis) instead of names r when comparing RPLs.

4.1 Syntax and Static Semantics

Figure 7 defines the syntax of Core DPJ. The syntax consists of the key elements described in the previous section (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.

Figure 8 shows the judgments defining the static semantics of Core DPJ. The judgments are defined with respect to an environment $\mathcal{E}$, which is a set of elements of the form $z \mapsto T$ (variable z has type T) or $P \subseteq R$ (parameter P is in scope and is included in region R). Figure 8(a) defines the validity of various syntactic entities and the environment $\mathcal{E}$. Figure 9 gives the rules for the first five judgments; we omit the other rules, which are similar, in the

(a) Validity Judgments (Figure 9)

$\vdash$ *program*	Valid program	$\mathcal{E} \vdash$ *method*	Valid method
$\vdash$ *class*	Valid class definition	$\mathcal{E} \vdash R$	Valid RPL
$\vdash \mathcal{E}$	Valid environment	$\mathcal{E} \vdash T$	Valid type
$\mathcal{E} \vdash$ *field*	Valid field	$\mathcal{E} \vdash E$	Valid effect

(b) RPLs, Subtyping, and Subeffects (Figure 10)

$\mathcal{E} \vdash R \leq R'$	R under R'	$\mathcal{E} \vdash T \leq T'$	T a subtype of T'
$\mathcal{E} \vdash R \subseteq R'$	R included in R'	$\mathcal{E} \vdash E \subseteq E'$	E a subeffect of E'

(c) Expressions (Figure 11)

$\mathcal{E} \vdash e : T, E$ e has type T and effect E in $\mathcal{E}$

Figure 8. Core DPJ type judgments. We extend the judgments to groups of things (e.g., $\mathcal{E} \vdash \overline{\textit{field}}$) in the obvious way.

$$\frac{\vdash \overline{\textit{class}} \quad \emptyset \vdash e : T, E}{\vdash \overline{\textit{class}}\ e} \qquad \frac{\{\texttt{this} \mapsto C\langle P\rangle\} \vdash \overline{\textit{field}}\ \overline{\textit{method}}\ \overline{\textit{comm}}}{\vdash \texttt{class}\ C\langle P\rangle\ \{\ \overline{\textit{field}}\ \overline{\textit{method}}\ \overline{\textit{comm}}\ \}}$$

$$\frac{\forall z \mapsto T \in \mathcal{E}.\mathcal{E} \vdash T \quad \forall P \subseteq R \in \mathcal{E}.\mathcal{E} \vdash R}{\vdash \mathcal{E}} \qquad \frac{\mathcal{E} \vdash T \quad \mathcal{E} \vdash R}{\mathcal{E} \vdash T\ f\ \texttt{in}\ R}$$

$$\frac{\begin{array}{c}\mathcal{E} \vdash T_r, T_x, E \quad \mathcal{E}' = \mathcal{E} \cup \{x \mapsto T_x\} \quad \mathcal{E}' \vdash e : T', E' \\ \mathcal{E}' \vdash T' \leq T_r \quad \mathcal{E}' \vdash E' \subseteq E\end{array}}{\mathcal{E} \vdash T_r\ m(T_x\ x)\ E\ \{\ e\ \}}$$

$$\frac{\texttt{this} \mapsto C\langle P\rangle \in \mathcal{E} \quad \exists \text{def}(C.m), \text{def}(C.m')}{\mathcal{E} \vdash m\ \texttt{commuteswith}\ m'}$$

Figure 9. Selected rules for validity judgments. def$(C.m)$ means the definition of method m in class C.

(a) Nested RPLs

$$\frac{}{\mathcal{E} \vdash R \leq \texttt{Root}} \qquad \frac{}{\mathcal{E} \vdash R : r \leq R} \qquad \frac{\mathcal{E} \vdash R \leq R'}{\mathcal{E} \vdash R : r \leq R' : r}$$

$$\frac{z \mapsto C\langle R\rangle \in \mathcal{E}}{\mathcal{E} \vdash z \leq R} \qquad \frac{}{\mathcal{E} \vdash R \leq R : *}$$

(b) Inclusion of RPLs

$$\frac{\mathcal{E} \vdash R \leq R'}{\mathcal{E} \vdash R \subseteq R' : *} \qquad \frac{\mathcal{E} \vdash R \subseteq R'}{\mathcal{E} \vdash R : r \subseteq R' : r} \qquad \frac{P \subseteq R \in \mathcal{E}}{\mathcal{E} \vdash P \subseteq R}$$

(c) Subtyping

$$\frac{}{\mathcal{E} \vdash \mathcal{N} \leq T} \qquad \frac{\mathcal{E} \vdash R \subseteq R'}{\mathcal{E} \vdash C\langle R\rangle \leq C\langle R'\rangle}$$

(d) Subeffects

$$\frac{}{\mathcal{E} \vdash \emptyset \subseteq E} \qquad \frac{\mathcal{E} \vdash R \subseteq R'}{\mathcal{E} \vdash \texttt{reads}\ R \subseteq \texttt{reads}\ R'} \qquad \frac{\mathcal{E} \vdash R \subseteq R'}{\mathcal{E} \vdash \texttt{reads}\ R \subseteq \texttt{writes}\ R'}$$

$$\frac{\mathcal{E} \vdash R \subseteq R'}{\mathcal{E} \vdash \texttt{writes}\ R \subseteq \texttt{writes}\ R'} \qquad \frac{}{\mathcal{E} \vdash \texttt{invokes}\ C.m\ \texttt{with}\ E \subseteq E}$$

$$\frac{\mathcal{E} \vdash E \subseteq E'}{\mathcal{E} \vdash \texttt{invokes}\ C.m\ \texttt{with}\ E \subseteq \texttt{invokes}\ C.m\ \texttt{with}\ E'}$$

$$\frac{\mathcal{E} \vdash E \subseteq E' \vee \mathcal{E} \vdash E \subseteq E''}{\mathcal{E} \vdash E \subseteq E' \cup E''} \qquad \frac{\mathcal{E} \vdash E' \subseteq E \quad \mathcal{E} \vdash E'' \subseteq E}{\mathcal{E} \vdash E' \cup E'' \subseteq E}$$

Figure 10. Nesting and inclusion of RPLs, subtyping, and subeffects. Nesting and inclusion are reflexive and transitive, and subtyping is reflexive (obvious rules omitted). The transitivity of subtyping follows from the other rules.

interest of brevity. All the rules except the method typing rule simply say, in formal terms, that an entity is valid if its components are valid in their surrounding environment. The method typing rule says that a method definition is valid if its return type, parameter type, and effect are valid; its body type-checks in the environment plus $x \mapsto T_x$; and the body's type and effect are, respectively, a subtype of the return type and a subeffect of the declared effect.

$$\frac{\begin{array}{c}\mathcal{E} \vdash e : C\langle R\rangle, E \quad \mathcal{E} \cup \{x \mapsto C\langle R\rangle\} \vdash e' : T', E' \\ \sigma = \{x \mapsto R : *\}\end{array}}{\mathcal{E} \vdash \texttt{let}\ x = e\ \texttt{in}\ e' : \sigma(T'), E \cup \sigma(E')}$$

$$\frac{T_f\ f\ \texttt{in}\ R_f \in \text{def}(C) \quad \texttt{this} \mapsto C\langle P\rangle \in \mathcal{E}}{\mathcal{E} \vdash \texttt{this}.f : T_f, \texttt{reads}\ R_f}$$

$$\frac{\texttt{this} \mapsto C\langle P\rangle \in \mathcal{E} \quad z \mapsto T \in \mathcal{E} \quad T_f\ f\ \texttt{in}\ R_f \in \text{def}(C) \quad \mathcal{E} \vdash T \leq T_f}{\mathcal{E} \vdash \texttt{this}.f = z : T, \texttt{writes}\ R_f}$$

$$\frac{\begin{array}{c}\{z \mapsto C\langle R\rangle, z' \mapsto T\} \subseteq \mathcal{E} \quad T_r\ m(T_x\ x)\ E_m\ \{\ e\ \} \in \text{def}(C) \\ \sigma = \{\texttt{this} \mapsto z, \text{param}(C) \mapsto R\} \\ \sigma' = \{\texttt{this} \mapsto z, \text{param}(C) \mapsto P\} \quad \mathcal{E} \cup \{P \subseteq R\} \vdash T \leq \sigma'(T_x)\end{array}}{\mathcal{E} \vdash z.m(z') : \sigma(T_r), \texttt{invokes}\ C.m\ \texttt{with}\ \sigma(E_m)}$$

$$\frac{z \mapsto T \in \mathcal{E}}{\mathcal{E} \vdash z : T, \emptyset} \qquad \frac{}{\mathcal{E} \vdash \texttt{null} : \mathcal{N}, \emptyset} \qquad \frac{\mathcal{E} \vdash T}{\mathcal{E} \vdash \texttt{new}\ T : T, \emptyset}$$

Figure 11. Typing expressions. $\sigma = \{a \mapsto b\}$ means that σ is a substitution taking a to b; def(C) is the definition of class C; and param(C) is the declared region parameter in def(C).

Figure 8(b) governs nesting and inclusion of RPLs, subtyping, and subeffects. The rules are shown in Figure 10. The nesting rules in Figure 10(a) say that we can create nested regions by appending names r; that variable z is under the region of its type; and that R is under $R : *$. The inclusion rules in Figure 10(b) say that $R : *$ refers to all regions under R; that inclusion still holds after appending identical names; that we can use the parameter inclusion information from the environment; and that inclusion in a fully specified RPL implies equality. The subtyping rules in Figure 10(c) say that the null type $\mathcal{N}$ is a subtype of any type, and that $C\langle R'\rangle$ is a subtype of $C\langle R\rangle$ if $R' \subseteq R$. The subeffect rules in Figure 10(d) include the standard rules [10, 21] for reads, writes, and effect unions, as well as two new rules for invocation effects. The first rule says that we can conservatively summarize an invocation effect by reporting just its underlying effect. The second rule says that two invocations of the same method are subeffects if their underlying effects are.

The judgment of Figure 8(c) allows us to conclude that an expression e type-checks with type T and effect E. Figure 11 shows the rules for this judgment. In the rule for $\texttt{let}\ x = e\ \texttt{in}\ e'$, 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 $R : *$ to generate a type and effect for the whole expression that is valid in the outer scope. In the rules for field access and assignment, we check that everything is well formed according to the class definition, enforce subtyping for the assignment, and report the read or write effect on the declared region. In the rule for method invocation $z.m(z')$, 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 [3] for an explanation and an example.

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. The rules for variables, $\mathcal{N}$, and `new` are straightforward.

4.2 Dynamic Semantics

The syntax for entities used in the dynamic semantics is shown in Figure 12. 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

Meaning	Symbol	Definition
RPLs	dR_f	$\texttt{Root} \mid o \mid dR_f : r$
	dR	$dR_f \mid dR : * \mid dR : r$
Types	dT	$C\langle dR\rangle$
Effects	dE	$\emptyset \mid \texttt{reads}\, dR \mid \texttt{writes}\, dR \mid$
		$\texttt{invokes}\; C.m\; \texttt{with}\; dE \mid dE \cup dE$
Values	v	$\texttt{null} \mid o$

Figure 12. Dynamic syntax of Core DPJ.

$$\frac{(e, d\mathcal{E}, H) \rightarrow (v, H', dE) \quad (e', d\mathcal{E} \cup \{x \mapsto v\}, H') \rightarrow (v', H'', dE')}{(\texttt{let}\; x = e\; \texttt{in}\; e', d\mathcal{E}, H) \rightarrow (v', H'', dE \cup dE')}$$

$$\frac{\texttt{this} \mapsto o \in d\mathcal{E} \quad H \vdash o : C\langle dR\rangle \quad T\; f\; \texttt{in}\; R_f \in \mathrm{def}(C)}{(\texttt{this}.f, d\mathcal{E}, H) \rightarrow (H(o)(f), H, \texttt{reads}\; d\mathcal{E}(R_f))}$$

$$\frac{\{\texttt{this} \mapsto o, z \mapsto v\} \subseteq d\mathcal{E} \quad H \vdash o : C\langle dR\rangle \quad T\; f\; \texttt{in}\; R_f \in \mathrm{def}(C)}{(\texttt{this}.f = z, d\mathcal{E}, H) \rightarrow (v, H \cup \{o \mapsto (H(o) \cup \{f \mapsto v\})\}, \texttt{writes}\; d\mathcal{E}(R_f))}$$

$$\frac{\begin{array}{c} H \vdash o : C\langle dR\rangle \quad T_r\; m(T_x\; x)\; E\; \{\; e\; \} \in \mathrm{def}(C) \\ (e, \{\texttt{this} \mapsto o, \mathrm{param}(P) \mapsto dR, x \mapsto v\}, H) \rightarrow (v', H', dE) \end{array}}{(z.m(z'), \{z \mapsto o, z' \mapsto v\} \cup d\mathcal{E}, H) \rightarrow (v', H', \texttt{invokes}\; C.m\; \texttt{with}\; dE)}$$

$$\frac{z \mapsto v \in d\mathcal{E}}{(z, d\mathcal{E}, H) \rightarrow (v, H, \emptyset)}$$

$$\frac{\begin{array}{c} \texttt{this} \mapsto o \in d\mathcal{E} \quad o' \notin \mathrm{Dom}(H) \quad H' = H \cup \{o' \mapsto \mathrm{new}(C)\} \\ \sigma = \{* \mapsto \epsilon\} \quad H' \vdash o' : d\mathcal{E}(C\langle\sigma(R)\rangle) \end{array}}{(\texttt{new}\; C\langle R\rangle, d\mathcal{E}, H) \rightarrow (o', H', \emptyset)}$$

Figure 13. Rules for program evaluation. A program evaluates to value v with heap H and effect dE if its main expression is e and $(e, \emptyset, \emptyset) \rightarrow (v, H, dE)$. If $f : A \rightarrow B$ is a function, then $f \cup \{x \mapsto y\}$ is the function $f' : A \cup \{x\} \rightarrow B \cup \{y\}$ defined by $f'(a) = f(a)$ if $a \neq x$ and $f'(x) = y$. $\mathrm{new}(C)$ is the function taking the fields of class C to `null`.

soundness results [10]. We also have values v, 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 $d\mathcal{E}$, which is a set of elements of the form $z \mapsto v$ (variable z is bound to value v) or $P \mapsto dR$ (region parameter P is bound to region dR). Thus, $d\mathcal{E}$ 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 $d\mathcal{E}(R)$. We extend this notation to types and effects in the obvious way. Notice that we get the syntax of Figure 12 by applying the substitution $d\mathcal{E}$ to the syntax of Figure 7.

A value v is `null` or an object reference $o \in \mathrm{Dom}(H)$. Every value has a type, and we write $H \vdash v : dT$ to mean that the value v has type dT with respect to heap H. The type of `null` is the null type $\mathcal{N}$. The type of an object reference $o \in \mathrm{Dom}(H)$ is $C\langle dR\rangle$, determined when the object is created. An object is a function taking field names to values.

Figure 13 gives the rules for program evaluation. To evaluate $\texttt{let}\; x = e\; \texttt{in}\; e'$, we evaluate e to v, bind v to x, and evaluate e', updating the heap and collecting effects as we go. To evaluate field access and assignment, we use $d\mathcal{E}$ to find the object bound to `this`, and we either read or write its field f, recording the read or write of the declared region after translating it to the dynamic context. To evaluate method invocation $z.m(z')$, we find the bindings of z and z' in $d\mathcal{E}$, create a new environment for the invocation, evaluate the method body, and record the invocation effect. To evaluate a variable expression, we just get the value out of $d\mathcal{E}$. To evaluate $\texttt{new}\; T$, we translate T to dT using dE, after eliminating any $*$ from T, e.g., $\texttt{new}\; C\langle\texttt{Root} : *\rangle$ is the same as $\texttt{new}\; C\langle\texttt{Root}\rangle$; this rule ensures that all object fields are allocated in fully specified regions. We then create a fresh object and a fresh reference of the appropriate type.

$$\frac{r \neq r'}{\mathcal{E} \vdash R : r : \mathcal{R} \# R' : r' : \mathcal{R}}$$

$$\frac{\mathcal{E} \vdash R_f : \mathcal{R} \# R_f : \mathcal{R}'}{\mathcal{E} \vdash R_f : \mathcal{R} : * \# R_f : \mathcal{R}' : *} \qquad \frac{\mathcal{E} \vdash R : * \# R'}{\mathcal{E} \vdash R \# R'}$$

Figure 14. Rules for disjoint RPLs. $\mathcal{R}$ is any sequence of zero or more names r. Disjointness is symmetric (obvious rule omitted).

4.3 Soundness

We briefly summarize the soundness results for Core DPJ. The main result is that we can define and check a static property of noninterference of effect between expressions in the language, and that static noninterference implies dynamic noninterference, i.e., that the expressions may be safely executed in parallel.

4.3.1 Type and Effect Preservation

First we extend the static judgments of Figure 8(a) and (b) to dynamic judgments using the dynamic translation implied by the environment $d\mathcal{E}$. We say that dR is valid with respect to H ($H \vdash dR$) if there exist R, $\mathcal{E}$, and $d\mathcal{E}$ such that $\vdash \mathcal{E}$, $\mathcal{E} \vdash R$, $H \vdash d\mathcal{E}$, and $d\mathcal{E}(R) = dR$. We define $H \vdash dT$ and $H \vdash dE$ similarly. We say $H \vdash dR\, \alpha\, dR'$, where α is one of $\leq$ and $\subseteq$, if there exist R, R', $\mathcal{E}$, and $d\mathcal{E}$ such that $\mathcal{E} \vdash R\, \alpha\, R'$, $d\mathcal{E}(R) = dR$, and $d\mathcal{E}(R') = dR'$. We define $H \vdash T\, \alpha\, T'$ and $H \vdash E\, \alpha\, E'$ similarly, for the relational judgments α given in Figure 8(b).

Next we formulate the claim that the execution state is always valid for well-typed programs:

Definition 1. *A dynamic environment $d\mathcal{E}$ is valid with respect to heap H ($H \vdash d\mathcal{E}$) if for every binding $z \mapsto v \in d\mathcal{E}$, $H \vdash v : dT$; and if $\texttt{this} \mapsto v \in d\mathcal{E}$, then $H \vdash v : C\langle dR\rangle$, and $\mathrm{param}(C) \mapsto dR \in d\mathcal{E}$.*

Definition 2. *A heap H is valid ($\vdash H$) if for each $o \in \mathrm{Dom}(H)$,*

1. *$H \vdash o : C\langle dR\rangle$; and*
2. *For each field $T\; f\; \texttt{in}\; R_f \in \mathrm{def}(C)$, $H \vdash H(o)(f) : dT$, and $H \vdash dT \leq \{\texttt{this} \mapsto o, \mathrm{param}(C) \mapsto dR\}(T)$.*

Claim 1 (Valid Execution)**.** *For a well-typed program, all heaps and dynamic environments appearing in the execution are valid.*

The proof is via a straightforward induction on the structure of execution. We define $H \vdash d\mathcal{E} \leq \mathcal{E}$ ("$d\mathcal{E}$ instantiates $\mathcal{E}$ in H") as follows:

Definition 3. *$H \vdash d\mathcal{E} \leq \mathcal{E}$ if $\vdash \mathcal{E}$, $\vdash H$, and $H \vdash d\mathcal{E}$; the same variables appear in $\mathrm{Dom}(\mathcal{E})$ as in $\mathrm{Dom}(d\mathcal{E})$; and for each pair $z \mapsto T \in \mathcal{E}$ and $z \mapsto v \in d\mathcal{E}$, $H \vdash v : dT$ and $H \vdash dT \leq d\mathcal{E}(T)$.*

Claim 2 (Preservation)**.** *For a well-typed program, if $\mathcal{E} \vdash e : T, E$ and $H \vdash d\mathcal{E} \leq \mathcal{E}$ and $H \vdash (e, d\mathcal{E}, H) \rightarrow (v, H', dE)$ and $H \vdash v : dT$, then $H' \vdash dT \leq d\mathcal{E}(T)$ and $H' \vdash dE \subseteq d\mathcal{E}(E)$.*

Again the proof is via a straightforward induction. This claim establishes that the static types and effects bound the dynamic types and effects.

4.3.2 Disjointness

We define the disjointness relationship for static RPLs ($\mathcal{E} \vdash R \# R'$) as shown in Figure 14. The first rule expresses "distinctions from the right": two RPLs must be disjoint if they end in different sequences of basic names. The left-hand rule in the bottom row expresses "distinctions from the left": if two RPLs proceed along the same path from `Root` or a parameter P or a variable z then diverge, then everything under the first must be disjoint from everything under the second. In the case of RPLs beginning with the same parameter or variable, the rule is sound because the same

$$\frac{}{\mathcal{E} \vdash \emptyset \# E} \qquad \frac{}{\mathcal{E} \vdash \texttt{reads}\ R \# \texttt{reads}\ R'} \qquad \frac{\mathcal{E} \vdash R \# R'}{\mathcal{E} \vdash \texttt{reads}\ R \# \texttt{writes}\ R'}$$

$$\frac{\mathcal{E} \vdash R \# R'}{\mathcal{E} \vdash \texttt{writes}\ R \# \texttt{writes}\ R'} \qquad \frac{\mathcal{E} \vdash E \# E'}{\mathcal{E} \vdash \texttt{invokes}\ C.m\ \texttt{with}\ E \# E'}$$

$$\frac{m\ \texttt{commuteswith}\ m' \in \text{def}(C)}{\mathcal{E} \vdash \texttt{invokes}\ C.m\ \texttt{with}\ E \# \texttt{invokes}\ C.m'\ \texttt{with}\ E'}$$

$$\frac{\mathcal{E} \vdash E \# E'' \quad \mathcal{E} \vdash E' \# E''}{\mathcal{E} \vdash E \cup E' \# E''}$$

Figure 15. The noninterference relation on effects. Noninterference is symmetric (obvious rule omitted).

dynamic RPL will be bound at runtime to the parameter or variable in both RPLs. Finally, if $R : *$ and R' are disjoint, then R and R' are disjoint as well. We extend the relation to dynamic RPLs in the same way described in Section 4.3.1.

We define region(o, f, H), the region of field f of object $o \in \text{Dom}(H)$ as follows.

Definition 4. *If $H \vdash o : C\langle dR \rangle$ and $T\ f\ \textit{\textbf{in}}\ R_f \in$ def(C), then region$(o, f, H) = \{\textit{\textbf{this}} \mapsto o, \textit{param}(C) \mapsto dR\}(R_f)$.*

Claim 3. *region(o, f, H) is fully specified.*

This claim follows from the fact that only fully specified RPLs are allowed in the dynamic types of objects (Section 4.2).

Claim 4. *Disjoint regions imply disjoint locations.*

This claim follows from the fact that the class definition together and the region binding in the type of `new` specify exactly one region for each object field at the time the object is created.

Claim 5. *Disjoint RPLs imply disjoint sets of fully specified regions, i.e., disjoint sets of locations.*

This claim follows from the tree structure of RPLs. It is true for both static and dynamic RPLs. In the static case, disjoint static RPLs imply disjoint dynamic RPLs, which imply disjoint regions. (locations).

4.3.3 Noninterference of Effect

We define the noninterference relationship for static effects ($\mathcal{E} \vdash E \# E'$) as shown in Figure 15. 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. We extend the relation to dynamic effects as described in Section 4.3.1.

Claim 6. *Expressions with noninterfering effects commute.*

The claim is true for dynamic effects from the commutativity of reads, the disjointness results of Section 4.3.2, and the assumed correctness of the commutativity specifications for methods. The claim is true for static effects by the type and effect preservation property of Section 4.3.1.

5. Evaluation

We have done a preliminary evaluation of the type system features presented in this paper. Our goal was to answer the following questions: (1) First, can the type system express important parallel algorithms on object-oriented data structures? When does it fail to capture parallelism and why? (2) Second, are each of the *new* features in the DPJ type system important to express one or more of these algorithms? (3) Third, for each of the algorithms, how much speedup is realized in practice?

We extended Sun's *javac* compiler so that it compiles DPJ into ordinary Java source. We built a run-time system for DPJ using the *ForkJoinTask* framework that is expected to be added to the `java.util.concurrent` library in Java 1.7. This framework supports dynamic scheduling of lightweight parallel tasks, using a work-stealing scheduler similar to that in Cilk [2]. The DPJ compiler automatically translates `foreach` to a recursive pattern that successively divides the iteration space, to a depth that is tunable by the programmer. For a cobegin, the compiler creates one task for each statement in the cobegin. The scheduling and synchronization overheads of the *ForkJoinTask* framework generally allow tasks as small as a few thousand instructions to be executed efficiently.

We have ported a number of small kernels as well as more realistic, moderate-sized benchmarks to DPJ; we focus on the latter in this work. These codes include a recursive parallel Merge Sort using divide-and-conquer, two codes from the Java Grande parallel benchmark suite (a Monte Carlo financial simulation and IDEA encryption), the Barnes-Hut force computation in 3 dimensions [29], and one compute-intensive phase of a large, real-world open source game engine called JMonkey. The JMonkey phase is a collision detection algorithm using binary trees, and we refer to it as Collision Tree. Barnes-Hut and Collision Tree have irregular parallelism and sharing patterns. For Barnes-Hut, we focus on the force computation because that phase dominates the running time of the program except for small problem sizes [29]. The two Java Grande codes have explicitly parallel versions using Java threads (as well as equivalent sequential versions), and we can compare our speedups against those. For all the codes, we began with a sequential version and modified it to add the DPJ type annotations. The fraction of code changed was relatively small.

5.1 Expressivity

DPJ is able to extract *all* the available parallelism for Merge Sort, IDEA, and collision detection. Monte Carlo ends with a short parallel sum-reduction, and we have not yet provided any "built-in" reduction operations in DPJ (parallel languages usually provide this in a standard library). This issue is discussed further in Section 5.3. The overall Barnes-Hut program includes four major phases in each time step: tree building; center-of-mass computation; force calculations; and position calculations. DPJ can express the parallelism in all phases. Expressing the force, center of mass, and position calculations is straightforward. Parallelizing the tree-building phase is possible, e.g., with a divide-and-conquer algorithm, but not with an algorithm that inserts leaves from the root of the tree and requires locks, e.g., as in [29].

Table 1. Capabilities Used In The Benchmarks

1. Linear array; 2. Distinctions from the left; 3. Distinctions from the right; 4. Recursive subranges; 5. Effect masking via local regions; 6. Disjoint region parameters.

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

Table 1 shows which capabilities are used in each of our codes. The four capabilities correspond to the novel type system features discussed in Sections 3.1– 3.3. Barnes-Hut uses "distinctions from the right" (#3) to distinguish force updates from read-only operations on the tree, similar to the pattern shown in Fig. 3. The commutativity feature (Section 3.4) is not used by these benchmarks.

5.2 Speedups

We measured the performance of each of the benchmarks on a modern multi-core machine. For all except JMonkey, we used a Dell R900 multiprocessor running Red Hat Linux with 24 cores, comprising four 6-core Xeon processors, and a total of 48GB of main memory. For JMonkey, we needed to run OpenGL routines on a hardware graphics card and so used an 8-core Apple Mac Pro running MacOS 10.5.5, comprising two 4-core Xeon processors and a total of 2GB of memory. For each case, we took the minimum of five runs on an idle machine to get an accurate timing.

We studied multiple inputs for each of the benchmarks and also experimented with different cutoffs for the recursion. For lack of space, we only present the results for the inputs and parameter values that show the best speedups. The sensitivity of the parallelism to input size or cutoff parameters are properties of the algorithm and not consequences of specific DPJ type system features. The input sizes we used are shown in the legend for Figure 16.

Figure 16 shows the speedups of the four programs on the Dell for $P \in \{1, 2, 3, 4, 7, 12, 17, 22\}$ processors, and for Collision Tree on the Mac Pro for $1 \leq P \leq 7$. The graph shows that three of the benchmarks (Merge Sort, IDEA and Monte Carlo) have moderate or good speedups up to 22 processors. The two irregular codes (Barnes Hut and Collision Tree) have lower but still useful speedups on these systems. We expect that the root problem in Barnes Hut is poor locality because the speedup graph of Barnes Hut levels off essentially like the leveling off in bandwidth we have observed on the Dell machine beyond 8 processors (using the standard Stream [22] benchmark). Section 5.3 discusses ways to improve locality in this code. More generally, all these codes have only been minimally tuned so far.

Importantly, the two Java Grande codes achieved speedups as good as (IDEA), or better than (Monte Carlo), the manually parallelized version written to use the base Java threads directly, for the same inputs on the same machines. The manually parallelized Monte Carlo code showed the same leveling off in speedup as the DPJ version beyond about 4 cores. Thus, in these cases at least, DPJ is able to express the available parallelism in non-trivial programs as well as a lower-level parallel programming model.

Our experience so far has shown us that DPJ itself can be very efficient, even though both the compiler and run-time are very preliminary. In particular (except for very small run-time costs for the dynamic partitioning mechanism for subarrays), our type system requires no run-time checks or speculation and therefore *adds negligible run-time overhead for achieving determinism* On the other hand, it is possible that the type system may constrain algorithmic design choices (e.g., prevent certain optimizations or require sub-optimal parallelization strategies), which would be a penalty of the type system. The Barnes-Hut code illustrates one such situation, as explained below.

5.3 Limitations

Our evaluation and experience also showed some interesting limitations of the current implementations. First, Barnes-Hut performance could be improved significantly by reordering the particles according to their proximity in space [29]. Unfortunately, because the particles are stored in a linear array (Section 3.1), to reorder them and retain soundness, we would have to make a copy of the bodies with the new destination regions at the point of assignment. We expect the improved locality will offset the cost of the extra copies, but as future work, we believe we can ease this restriction by moving some of the linearity checking to runtime.

Second, the Monte Carlo code ends in a short reduction phase, which was not parallelized in DPJ though it was parallel in Java Grande. This is simply because DPJ currently lacks any "built-in" reduction operations. The Monte Carlo code was not affected significantly but many other codes require parallel reductions for high performance. We are working on general language mechanisms to *encapsulate trusted library functions* such as associative reductions or commutative operations on concurrent data structures.

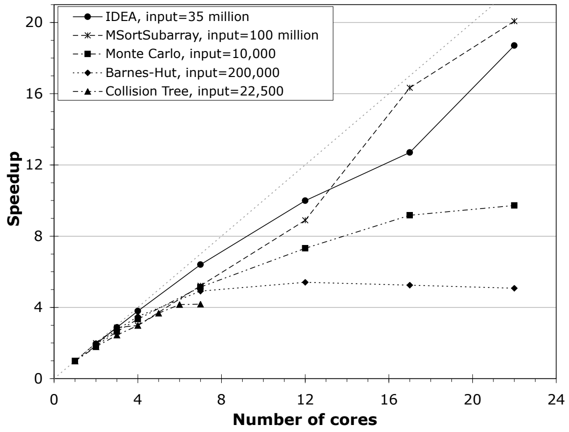


Figure 16. Speedups.

6. Related Work

We divide 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 other related work.

Effect Systems: The seminal work on types and effects for concurrency is FX [21, 16], which adds a region-based type and effect system to a Scheme-like, implicitly parallel language. Apart from the obvious differences due to the base language (for example, no support for method overriding), there is no notion of field region declarations, region nesting, or nested effects; nor is there any (static or dynamic) array partitioning. Thus, FX cannot express the essential patterns of imperative programming, such as in-place updates of trees and arrays, that we address in this paper.

Leino et al. [18] were among the first to add effects to an object-oriented language. Greenhouse and Boyland [15] present a similar system. In both systems, the regions may be nested, unlike in FX. However, both systems lack region parameters, so they cannot express arbitrarily nested structures. There is also no array partitioning. Both systems specify effects less precisely than DPJ, and unlike DPJ, they both rely on alias restrictions and/or supplementary alias analysis for soundness of effect.

Ownership Types: Though originally designed for alias control [11], object ownership has grown far beyond this original purpose, and many variant systems have been proposed. Here we confine our discussion to systems that combine ownership with effects. The systems closest to ours are JOE [10, 30] and MOJO [8]. Both 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 (see Section 3.2). Moreover, because JOE uses sequences of final fields instead of RPLs to establish disjointness, it is not clear how to extend JOE to provide this capability. MOJO has a wildcard region specifier ?, but it pertains to the orthogonal capability of *multiple ownership*. Multiple ownership allows objects to be placed in multiple regions, so the region hierarchy is a DAG, rather than a tree.

Lu and Potter [20] show how to use effect constraints to break the owner dominates rule in limited ways while still retaining meaningful guarantees. Boyapati et al. [5, 4] describe an effect system 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, for example, they cannot express array effects or nested effects.

Finally, an important difference between DPJ and most ownership systems is that we allow *explicit region declarations*, like [21, 18, 15], 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 under traditional ownership. *Ownership domains* [30] is a kind of hybrid between ownership and an explicit-declaration system; this suggests a way that DPJ could be extended if the other features of ownership (such as alias control) are desired.

Unique References: Boyland [7] shows how to use alias restrictions to guarantee determinism for a simple language with pointers. Terauchi and Aiken [32] have recently extended this work with a type inference algorithm that eases the burden of writing the type annotations and can elegantly express 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 [15, 6]. However, alias limitations severely restrict the expressivity of an object-oriented language. It is not clear whether the techniques described in [7, 32] could be applied to a robust object-oriented language.

Separation Logic: Separation logic [27] (SL) is a general mechanism for specifying and checking program properties having to do with shared use of resources, such as the heap. Thus it is a potential alternative to effect systems as a method for making shared memory parallel programming safer. O'Hearn [23] and Gotsman et al. [14] have recently investigated the use of SL for concurrency. The focus of their work is on race freedom, though O'Hearn includes some simple proofs of noninterference. Parkinson [24] has recently extended C# with SL predicates to allow sound inference in the presence of inheritance.

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 have to be defined using SL predicates [27, 23]. Second, existing SL approaches generally *either* require heavyweight theorem proving and/or a relatively heavy programmer annotation burden [24] *or* they are fully automated, and thereby limited by what the compiler can infer [14].

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 [27], 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.

Other Related Work: The Jade language [28] aims for deterministic parallelism in the presence of aliasing of mutable objects, but it uses a much weaker type system than ours, and relies largely on runtime checks. Multiphase Shared Arrays [12] and PPL1 [31] are similar in that they rely on runtime checks that may fail if determinism is violated. SharC [1] uses a combination of static and dynamic checks to enforce race-freedom in C programs.

Speculative parallelism [13, 34] can be used to achieve determinism in an object-oriented language with references. However, speculation either incurs significant run-time overheads, or requires special hardware [26, 19, 33]. Further, speculation does not show how the code must be rewritten to expose parallelism.

7. Conclusion

We have described a novel type and effect system that guarantees noninterference, and uses that to enforce deterministic semantics, for an expressive object-oriented parallel language that supports linked data structures such as arrays of references and trees. Our experience has shown that the novel features in the type system were necessary to parallelize realistic programs, and collectively allow a range of different parallel patterns. The language is able to achieve moderate to good speedups on several of these programs on a 24-processor system. Our immediate goals for future work are to provide more flexibility by adding runtime support for linear data structures; to develop language features for encapsulating low-level libraries and frameworks soundly; and to explore language support for supporting non-deterministic and deterministic algorithms in a program while preserving the safety guarantees for the latter.

References

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

[2] R. D. Blumofe et al. Cilk: An efficient multithreaded runtime system. *PPOPP*, 1995.

[3] R. L. Bocchino and V. S. Adve. Formal definition and proof of soundness for Core DPJ. Technical Report UIUCDCS-R-2008-2980, University of Illinois at Urbana-Champaign, 2008.

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

[5] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. *OOPSLA*, 2001.

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

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

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

[9] P. Charles et al. X10: An object-oriented approach to non-uniform cluster computing. *OOPSLA*, 2005.

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

[11] D. G. Clarke et al. Ownership types for flexible alias protection. *OOPSLA*, 1998.

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

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

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

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

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

[17] M. Kulkarni et al. Optimistic parallelism requires abstractions. *PLDI*, 2007.

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

[19] W. Liu et al. POSH: a TLS compiler that exploits program structure. *PPOPP*, 2006.

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

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

[22] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream, 2004.

[23] P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor.*

Comp. Sci., 2007.

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

[25] A. Potanin et al. Generic ownership for generic Java. *OOPSLA*, 2006.

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

[27] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. *Symp. on Logic in Comp. Sci.*, pages 55–74, 2002.

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

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

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

[31] M. Snir. Parallel Programming Language 1 (PPL1), V0.9 — Draft. Technical Report UIUCDCS-R-2006-2969, University of Illinois at Urbana-Champaign, 2006.

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

[33] C. von Praun et al. Implicit parallelism with ordered transactions. *PPOPP*, 2007.

[34] A. Welc et al. Safe futures for Java. *OOPSLA*, 2005.