# AN EFFECT SYSTEM AND LANGUAGE FOR DETERMINISTIC-BY-DEFAULT PARALLEL PROGRAMMING

BY

ROBERT L. BOCCHINO

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Doctoral Committee:

Associate Professor Vikram Adve, Chair and Director of Research
Professor David Padua
Associate Professor Grigore Roşu
Professor Marc Snir
Bradford Chamberlain, Cray Inc.
Associate Professor Dan Grossman, University of Washington

# Abstract

This thesis presents a new, Java-based object-oriented parallel language called Deterministic Parallel Java (DPJ). DPJ uses a novel *effect system* to guarantee *determinism by default*. That means that parallel programs are *guaranteed* to execute deterministically unless nondeterminism is explicitly requested. This is in contrast to the shared-memory models in widespread use today, such as threads and locks (including threads in ordinary Java). Those models are inherently nondeterministic, do not provide any way to check or enforce that a computation is deterministic, and can even have unintended data races, which can lead to strange and unexpected behaviors. Because deterministic programs are much easier to reason about than arbitrary parallel code, determinism by default simplifies parallel programming.

This thesis makes several broad contributions to the state of the art in programming languages and effect systems. *First*, it presents a comprehensive research agenda for achieving determinism by default in parallel languages with reference aliasing and shared mutable state. It argues that an object-oriented effect system is a good approach to managing shared memory conflicts. It also raises several technical challenges, many of which are taken up in the rest of the thesis.

*Second*, this thesis presents an effect system and language for deterministic parallel programming using a fork-join model of parallel control. With simple modular checking, and with no runtime checking overhead, the effect system guarantees at compile time that there are no conflicting memory accesses between any pairs of parallel tasks. The effect system supports several important patterns of deterministic parallelism that previous systems cannot express. We describe the effect system and language both formally and informally, and prove soundness for the formal language. We also describe our evaluation showing that the language can express a range of parallel programming patterns with good performance.

*Third*, this thesis extends the effect system and language for determinism to support a controlled form of nondeterminism. Conflicting accesses are allowed only for an explicitly identified nondeterministic parallel construct, so the language is deterministic by default. A transactional runtime provides isolation for atomic

statements, while the extended effect system provides stronger compile-time safety guarantees than any system we know of. In addition to determinism by default, the language guarantees race freedom; strong isolation for atomic statements *even if the runtime guarantees only weak isolation*; and an elegant way of composing deterministic and nondeterministic operations that preserves local reasoning about deterministic operations. Again we give an informal treatment, a formal treatment, and soundness proofs. We describe an evaluation showing that the extended language can express realistic nondeterministic algorithms in a natural way, with reasonable performance given the transactional runtime we used. Further, by eliminating unnecessary synchronization, the effect system enables a significant reduction in the software runtime overhead.

*Fourth*, this thesis describes programming techniques and further extensions to the effect system for supporting object-oriented parallel frameworks. Frameworks represent an important tool for parallel programming in their own right. They can also express some operations that the language and effect system alone cannot, for example pipeline parallelism. We show how to write a framework API using the DPJ effect system so that the framework writer can guarantee correctness properties to the user, assuming the user's code passes the DPJ type checker. We also show how to extend the DPJ effect system to add generic types and effects, making the frameworks more general and useful. Finally, we state the requirements for a correct framework implementation. These requirements may be checked with a combination of DPJ's effect system and external reasoning. Again we give an informal treatment, a formal treatment, and soundness proofs. We also describe the results of an evaluation showing that the techniques described can express realistic frameworks and parallel algorithms.

# Acknowledgments

I am grateful to many people for supporting and contributing to the content of this thesis:

- My advisor, Vikram Adve, unwaveringly encouraged, supported, and guided this work. He also contributed many ideas to the work.

- Sarita Adve and Marc Snir contributed many useful discussions about the benefits of deterministic programming, the interaction between deterministic and nondeterministic code, and the various mechanisms for expressing determinism and nondeterminism. Their work on our paper in HotPar 2009, together with Vikram and myself, defined the agenda for this thesis.

- Danny Dig was extremely helpful in evaluating DPJ, and in figuring out how to explain the type system details to a general audience. He also encouraged the use of `ForkJoinTask` for the DPJ runtime.

- Dan Grossman, Grigore Roşu, Chris Rodrigues, and Madhusudan Parthasarathy gave invaluable feedback on the formal aspects of the work.

- Brad Chamberlain, David Padua, and Ralph Johnson gave invaluable feedback on the language aspects of the work, including usability. Brad also worked closely with me on a related project involving software transactional memory (STM) for large-scale clusters. While not directly appearing in this thesis, that work gave me lots of insight into both STMs and parallel programming that I brought to bear on this work.

- Adam Welc, Tatiana Shpeisman, and Yang Ni contributed insights into the performance characteristics and semantic issues raised by software transactional memory. Adam also helped build the transactional runtime for DPJ.

- Rakesh Komuravelli, Stephen Heumann, Nima Honarmand, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian were my "users." They learned how to program in DPJ, and they wrote, tuned, and measured the performance of many DPJ programs. Their efforts contributed greatly to the evaluation of DPJ as a language. Patrick and Stephen also helped implement the compiler code generation. Finally, Mohsen caught and forced me to fix many tricky bugs in the compiler.

- Jeff Overbey got us on the right path using `javac` for the DPJ compiler. He also helped get the compiler implementation started.

- Maurice Herlihy suggested that we use the Deuce STM for the DPJ transactional runtime.

I am also grateful to the following institutions for directly or indirectly supporting the work:

- Microsoft Corporation and Intel Corporation directly funded my Research Assistantship (RA), under the auspices of the Universal Parallel Computing Research Center (UPCRC) at the University of Illinois.

- NSF also directly funded my RA.

- My internship at Cray Inc. supported me for a summer and helped me learn a lot about STM, parallel computing, and language design.

- Motorola supported me as an RA for several semesters and as a summer intern. While the work supported by Motorola does not directly appear in this thesis, it helped me learn a lot about parallel computing (specifically vector computing).

Finally, thanks to Sonia, my family, and my friends for their caring and support.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

This thesis presents a new, Java-based object-oriented parallel language called Deterministic Parallel Java (DPJ). DPJ uses a novel *effect system* to guarantee *determinism by default* at compile time. That means that parallel programs are *guaranteed* to execute deterministically unless nondeterminism is explicitly requested. Further, in DPJ, nondeterminism is carefully controlled and subject to strong compile-time safety guarantees, including freedom from data races. This is in contrast to the shared-memory models in widespread use today, such as threads and locks (including threads in ordinary Java). Those models are inherently nondeterministic, do not provide any way to check or enforce that a computation is deterministic, and can even have unintended data races, which can lead to strange and unexpected behaviors. Finally, DPJ can check that the uses of object-oriented frameworks correspond to their effect specifications, ensuring determinism and other safety guarantees for the framework uses. Frameworks represent an important tool for parallel programming in their own right, and can also express some operations that the language and effect system alone cannot.

This thesis argues that if determinism by default becomes a feature of mainstream programming languages, then parallel programming will be much easier. It outlines the major technical challenges to achieving determinism by default for shared memory programs, and proposes a comprehensive research agenda for addressing these challenges. Finally, this thesis describes solutions to three of the major challenges, using DPJ as a prototype: (1) designing a language and effect system for expressing deterministic computations that are guaranteed at compile time to have no conflicting accesses between parallel tasks, without any runtime overhead for checking determinism; (2) supporting controlled nondeterminism while retaining strong compile-time safety guarantees, including determinism by default; and (3) checking that the uses of object-oriented frameworks correspond to their effect specifications.

## 1.1 The Need for Determinism by Default

**Determinism:** Single-core processors have reached the limit of scaling, and multicore processors are now prevalent, with the number of cores growing according to Moore's law. As a result, parallel programming — once a highly specialized activity — is becoming mainstream. Parallel programming languages, libraries and tools must enable programmers to write parallel programs without a major loss of programmer productivity compared to the sequential programs that they are used to writing. In particular, programmers must be able to write correct programs, i.e., without a major increase in bugs due to the introduction of parallelism. And programmers must be able to understand parallel programs, debug them, and tune them for performance.

This situation presents a challenging problem for languages and related tools such as compilers and runtime systems. Most programmers are used to thinking sequentially. In its most general form, however, parallel programming forces them to consider interactions between different concurrent tasks, usually expressed as *interleavings* of memory operations. Further, unlike the sequential case, multiple interleavings must be considered: different interleavings can produce different results, and the precise interleaving depends on the parallel schedule, which can differ from run to run. As a result, general parallel programming causes an explosion in complexity, both for reasoning about programs, and for the state space that needs to be explored through testing or model checking. Finally, general parallel programming leads to bugs such as data races, deadlocks, and memory consistency violations that can be difficult to find and correct. These bugs are totally unfamiliar, and even bizarre, to the sequential programmer.

We believe that one important response to this challenge is to focus on a property of parallel languages and programs called *determinism*. We say that a *program* is deterministic if it produces the same externally visible output on every execution with a given input, regardless of the parallel schedule chosen for execution. We say that a *language* is deterministic if any legal program written in the language is deterministic. A deterministic language has significant advantages:

- A deterministic program that has an obvious sequential equivalent can be understood without concern for execution interleavings, data races, or complex memory consistency models: the program behavior is completely defined by its sequential equivalent.

- Programmers can reason about programs, debug them during development, and diagnose error reports

after deployment using techniques and tools similar to those currently used for sequential programs.

- Independent software vendors can test codes as they do for sequential programs, without being concerned about the need to cover multiple possible executions for each input. The same test suites developed for the sequential code can be used for the parallel code.

- Programmers can use an incremental parallelization strategy, progressively replacing sequential constructs with parallel constructs, while preserving program behavior.

- Two separately developed but deterministic parallel components should be far easier to compose than more general parallel code, because a deterministic component should have the same behavior regardless of the external context within which it is executed.

Deterministic semantics can also help with parallel performance modeling. In particular, an explicitly parallel loop has *sequential semantics* with a *parallel performance model*: its performance will be what one would expect by assuming that parallel loop iterations do execute in parallel. In effect, both the semantic model and the performance model for such a program can be defined using obvious composition rules [20]. Further, deterministic programming models can enable programmers to spend more time on performance tuning (often the determining factor in performance for real-world software) and less time finding and eliminating insidious concurrency bugs.

In general, shared-memory parallel programs are not deterministic, for the reason stated above: different interleavings produced by different parallel schedules can produce different results. However, *many parallel algorithms are, in fact, intended to behave deterministically*. Typically these are compute-intensive algorithms that accept some input and are intended to produce a single (deterministic) output. Examples can be found in a wide variety of domains, including scientific computations, graphics, voice, video, and artificial intelligence. For these applications, a deterministic language can simplify the writing and maintenance of parallel programs, for the reasons stated above.

In this context it is important to note the following conventions that we adopt regarding the term "determinism":

1. In this thesis, we focus our attention on *concurrency determinism*. Other sources of nondeterminism that carry over from sequential programming (for example, calls to `random` or `gettimeofday`)

are not considered, on the assumption that sequential programmers already know how to cope with this kind of nondeterminism.

2. We mean determinism of the final result; intermediate results need not be deterministic. For example, an integer sum reduction may produce nondeterministic intermediate results on the way to producing a deterministic final result.

3. Unless otherwise noted, throughout this thesis, "the same result" means bitwise equivalence of the visible program output. In some cases (e.g., floating point reductions, or comparing the contents of two set data structures) bitwise equivalence may be too restrictive, and in those cases we will explain more precisely what we mean by "the same result." In particular, this issue will arise in Section 3.4, in connection with specifying commutative updates to sets, counters, and other shared state.

**Controlled nondeterminism:** Of course, not all parallel programs are intended to be deterministic. Some algorithms produce many acceptable answers, all of which satisfy some criterion of correctness. Examples include branch-and-bound search and graph clustering algorithms. It is usually possible to write deterministic versions of these algorithms (by simply picking one possible answer and excluding the others). However, it may not be desirable to *require* determinism for all such algorithms, particularly where fixing a deterministic schedule would cause performance to suffer.

Therefore, we believe that while parallel programming should be deterministic in most cases, some form of controlled nondeterminism should be allowed as well. We believe that any such nondeterminism should have the following safety properties:

1. *Race freedom and sequential consistency.* No execution of a valid program should ever produce a data race. This property is very important, even for nondeterministic codes, because it facilitates reasoning about program semantics. For example, in the Java memory model, race freedom implies sequential consistency, which makes parallel programs much easier to reason about. The Java memory model has a defined semantics in the presence of data races, but it is hard to understand. In the C++ memory model, the program semantics is not even defined in the presence of data races. So in some sense it is impossible to reason correctly about a C++ program that contains a data race!

2. *Strong isolation.* The language should provide *strong isolation* (i.e., isolation with respect to *all* concurrent operations) for sections of code identified as isolated (e.g., statements marked `atomic`). So-

called “weak isolation” (i.e., isolation provided that all conflicts occur between `atomic` statements) is not enough, because it leads back to the same concurrency problems that transactions are trying to eliminate in the first place, i.e., unintended conflicting memory accesses that silently invalidate the programmer’s assumptions about what the program is doing.

3. *Composition of deterministic and nondeterministic operations.* It should be easy to reason about compositions of deterministic and nondeterministic constructs. In particular, we argue that a deterministic computation should always behave as an isolated, sequential composition of its component tasks, even inside a nondeterministic parallel operation. This requirement leads to novel features of our effect system, as discussed in Chapters 5 and 6.

4. *Determinism by default.* Nondeterminism occurs only where requested by an explicit nondeterministic operation. Thus, nondeterminism cannot occur by accident, as it can in arbitrary threaded code. We agree with Lee [76] that this is a critical property for reasoning about parallel code.

We call any language that satisfies property 4 above (nondeterminism must be explicit) a *deterministic-by-default language*. We believe that parallel programming languages should be deterministic by default, and that such languages should also provide properties 1–3 listed above. Note that today’s widely used parallel programming models provide *none* of these four properties. Instead they are based on a “wild” form of shared memory parallelism, where all shared-memory interactions are allowed, and it is up to the programmer (via testing, code inspection, or some other method) to exclude races and other undesirable behaviors.

**Supporting object-oriented frameworks:** Object-oriented frameworks are an important part of the solution for making parallel programming easier. In the framework approach, the framework writer provides most of the code for parallel construction and manipulation of generic data structures; for generic parallel algorithms such as map, reduce, or scan; or for generic parallel coordination patterns such as pipelines. The user fills in the missing pieces with code (in most cases, sequential code) that is applied in parallel by the framework. Examples include the algorithm templates in Intel’s Threading Building Blocks (TBB) [101] and Java’s `ParallelArray` framework [1]. Such frameworks are usually easier to reason about than general parallel programming because the user only has to write sequential code, letting the framework orchestrate the parallelism.

For frameworks, the property that corresponds to determinism by default for general language mechanisms is checking the effect specifications of the framework API. For example, `ParallelArray`'s `apply` method applies an arbitrary user-specified method in parallel to each element of the array. If that method performs an unsynchronized update to a global variable, then an unexpected race will result when the framework applies the function. This kind of race can be excluded if (1) the framework developer gives the API an effect specification (for example, that the function provided to `apply` has no conflicting effects on shared state); and (2) the compiler checks that the specification is met by all code supplied by the user to the framework. Frameworks like this can supplement a deterministic-by-default language by adding new operations that carry the same strong guarantees, such as determinism by default, as the underlying language. For example, a `ParallelArray` framework that *guarantees* that no user-supplied `apply` function has interfering effects on shared state can *guarantee* deterministic behavior for all uses of `apply`.

## 1.2 Technical Challenges

**Determinism:** Determinism is available today for certain restricted styles of parallel programming, such as data parallel or purely functional programs. We discuss these in more detail in Chapter 2. However, defining an expressive and efficient language that guarantees determinism in the presence of general aliasing of shared mutable data remains unsolved. The problem is difficult because techniques such as references and encapsulated updates to shared state, which are expressive and efficient for sequential programs, hide the data dependence relationships between potentially parallel sections of code.

As a result, mainstream parallel programming models today provide no special assistance for programming deterministic algorithms. Parallel applications today primarily use threads and shared memory, whether through libraries like `pthreads`, Intel's Threading Building Blocks (TBB), and OpenMP; or multithreaded languages like Java, C#, and Cilk++. Programs written in these models can be extremely difficult to understand and debug. A correctly written program may be deterministic, but this property is difficult to check. This is a very important deficiency, as many applications that will need to be ported to emerging parallel architectures are written in imperative, object-oriented languages such as C++ and Java.

One recent approach to solving this problem is to add *speculative parallelism* [51, 93, 50, 128, 18, 99, 114, 124] to a language such as Java. This approach can guarantee deterministic semantics, but it either incurs significant runtime overheads [51, 93, 50, 128], or it requires special hardware [99, 114, 124], or it

works well only for coarse-grain sharing [18]. Further, speculation does not solve the fundamental difficulty of hidden updates in parallel programs: while the program will be correct, it will not perform well unless it is tuned to avoid synchronization or speculative conflicts; and tuning requires the programmer to understand the patterns of sharing, interference, and synchronization in the code.

Another approach is to use a combination of static and dynamic checks. Both Jade [105] and Prometheus [11] adopt this approach. However, in both languages, the static type system is relatively weak, and many checks are left to runtime. Further, there is no speculation, so if a runtime check fails, the program aborts. Thus, it is generally not possible to guarantee at compile time or even testing time that the program has a deterministic sequential equivalent. Multiphase Shared Arrays [42] and PPL1 [112] adopt a similar approach.

**Nondeterminism:** For the same reasons as discussed in connection with determinism (i.e., hidden conflicting updates to shared state), parallel codes with intentional nondeterminism can suffer from problems like data races and deadlocks. Again, mainstream parallel programming models provide no particular assistance with avoiding these problems. The most common approach in use today is to use a race and/or deadlock checker, such as Intel's Thread Checker. While such checking is effective in many cases, it is slow and is not guaranteed to find all the races and deadlocks in a program.

Several experimental approaches exist for adding safety guarantees to nondeterministic code. Transactional memory [63] provides isolation and deadlock freedom, but it still permits a race if one or both of the racing memory accesses occur outside a transaction. Further, because of overhead concerns, transactional memory implemented in software (software transactional memory, or STM) typically guarantees only *weak isolation* — i.e., the isolation holds only if there are no conflicts outside of transactions. Even worse, if there *are* conflicts outside of transactions, then (again, for efficiency reasons) many STM implementations produce behaviors that can be very difficult to reason about [108].

Several researchers have described effect systems for enforcing a locking discipline in nondeterministic programs, to prevent data races and deadlocks [24, 6, 68, 86], or to guarantee isolation for critical sections [52]. Each of these efforts provides some subset of the four guarantees stated above for nondeterministic code, but none provides all of them in the same language. Further, none of this work explores the interaction between deterministic and nondeterministic code, or attempts to design a language for determinism by default.

**Checking framework uses:** Current frameworks give no guarantee against conflicting memory operations,

and this a serious deficiency in terms of correctness and program understanding. For example, there is nothing in the `ParallelArray` API that prevents a user from writing an `apply` function that does an unsynchronized write to a global variable, causing a race when the framework applies it in parallel to the array, as discussed above. Today's frameworks only issue a set of informal guidelines for how to use the API safely, which is unsatisfactory.

While several tools and techniques exist that support writing and checking assertions at interface boundaries [119, 75, 89], these ideas have not yet been applied to prohibit interfering effects, i.e., concurrent conflicting memory operations. Doing so involves several open challenges:

1. *Maintaining disjointness:* Useful parallel frameworks need to support parallel updates on contained objects. For example, we would like a `ParallelArray` of distinct objects, where the user can provide an `apply` function that updates an element, and ask the framework to apply it to each distinct object in parallel. To do this safely, the framework must ensure that the objects are really distinct; otherwise the same object could be updated in two parallel iterations, causing a race. For a language like Java with reference aliasing, disjointness of reference is a nontrivial property.

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

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

## 1.3 Deterministic Parallel Java

In this thesis, we present the design and evaluation of *Deterministic Parallel Java* (DPJ). DPJ extends the sequential subset of Java with an effect system that (1) allows deterministic algorithms to be written with a compile-time guarantee of determinism; (2) supports the composition of deterministic and nondeterministic code with strong compile-time guarantees, including determinism by default; and (3) supports the specification and checking of effects on framework APIs. In DPJ, the programmer partitions the heap using *regions*, which are names for sets of memory locations, and writes *method effect summaries* saying which regions are read and written by the method. The compiler checks that the method summaries are correct (i.e., the summary includes all the actual effects of the method) and that parallel tasks are *noninterfering* (i.e., if any two parallel tasks access the same region, then all such accesses are operations that commute with each other, such as reads). DPJ builds on early work in types and effects [61, 56] together with recent advances in object-oriented effect systems [59, 25, 77, 37, 80].

DPJ has the following advantages over previous approaches:

1. It can express a wide range of deterministic algorithms in such a way that the compiler can statically guarantee determinism by default, as well as the safety properties for nondeterministic code mentioned above.

2. Unlike speculative methods, Jade, or Prometheus, purely deterministic code requires no complicated runtime support or extra runtime overhead. DPJ does use a transactional runtime, but that runtime is invoked only if the code uses nondeterministic constructs.

3. DPJ's effect system includes novel support for important patterns of parallelism — such as field-granularity updates on nested data structures and parallel array updates — that previous effect systems do not support.

4. DPJ supports the implementation of object-oriented frameworks that provide correctness guarantees to their users. The framework author can check correctness properties of the framework, including determinism, without seeing the user's code. In any use of the framework that passes the DPJ type checker, the properties hold. The effect system introduces novel features to support generic parallel frameworks, while retaining sound reasoning about effects.

These benefits do come at some cost in terms of programmer annotation: in particular, the programmer must annotate types with region and/or effect information, and methods with effect information. In the technical sections of this thesis, we include a quantitative evaluation of the annotation burden, in terms of the number of annotations required. As discussed in Chapter 2, one of the long-term goals of this work is to explore ways to reduce the annotation burden. We have considered two ways:

1. Developing algorithms and tools for inferring some or all of the type and effect annotations [122].

2. Supplementing or replacing some of the static checking with runtime checking. This would weaken the guarantee and/or add overhead, but it could also simplify the annotations and/or make the language more expressive.

Both of these issues are the subject of active research in our group, but are beyond the scope of the present thesis. The focus of this thesis is on designing an effect system that is highly expressive, while still being usable.

Finally, while this thesis focuses on extending Java, many of the ideas should apply to other object-oriented languages, such as C# and C++. C++ is not a type-safe language; in particular, there is no guarantee that dereferencing a pointer will access an object of the type specified by the pointer. Therefore, to apply these ideas soundly to C++, one must do some additional work. One possible approach is to provide deterministic semantics for type-safe programs, without providing any guarantee for programs that violate type safety.Our research group is actively working on this problem, but it is beyond the scope of this thesis.

## 1.4 Thesis Contributions and Outline

The broad contribution of this thesis is to present a realistic language for shared-memory, object-oriented parallel programming that (1) guarantees determinism by default at compile time, with the safety guarantees stated in Section 1.1; (2) uses speculation only for nondeterministic computations, while adding negligible overhead for enforcing determinism; and (3) allows the uses of object-oriented frameworks to be checked against their effect specifications. This broad contribution subsumes several specific technical contributions. These are stated below, in the order that they appear in the rest of this thesis.

**Research agenda for determinism by default:** Chapter 2 presents a comprehensive research agenda for achieving determinism by default in imperative languages. This chapter argues that an *effect system* (as used

in DPJ) is a good solution to the problem, and it describes contrasting approaches. It also discusses several technical challenges raised by effect systems that are taken up in the rest of the thesis. Finally, this chapter discusses two open issues that are not addressed in the rest of the thesis: (1) reducing programmer burden by inferring effect annotations; and (2) supplementing static effect checking with runtime checks. Other members of our research group are working on these problems.

**Effect system and language for determinism:** Chapters 3 and 4 describe an effect system and language for determinism. The language provides explicit, fork-join parallelism using `foreach` for parallel loops and `cobegin` for parallel statement blocks. Data-dependent synchronization patterns (e.g., a pipelined parallel loop [72]) cannot be expressed by this language, but these patterns can be expressed by object-oriented frameworks, discussed below. The effect system ensures determinism at compile time by checking that no conflicting memory accesses can occur in parallel branches of the same `foreach` or `cobegin`. Chapter 3 gives an informal description of the entire language, and Chapter 4 gives a more formal treatment for a simplified version of the language. This part of the thesis makes the following contributions:

1. We introduce a novel mechanism called a *region path list*, or RPL, for hierarchically partitioning the heap. Hierarchical partitioning is vital for expressing effects. For example, divide-and-conquer parallel computations on trees naturally generate sets of effects 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." RPLs can express several patterns of effects that previous systems [61, 56, 59, 37] cannot. RPLs also allow more flexible subtyping than previous work.

2. To support parallel update computations on arrays, 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.

3. To support in-place parallel divide and conquer operations on arrays, we introduce the notion of *subarrays* (i.e., one array that shares storage with another) and a *partition operation*. Subarrays and partitioning provide a natural object-oriented way to encode disjoint segments of arrays, in contrast to lower-level mechanisms like separation logic [95] that specify array index ranges directly.

4. We introduce an *invocation effect*, together with simple *commutativity annotations*, to permit the

parallel invocation of operations that interfere at the level of reads and writes, but produce the same high-level behavior for any concurrent schedule. This mechanism supports common read-modify-write patterns such as reduction accumulations. It also allows concurrent data structures, such as concurrent sets and hash maps, to interoperate with the language.

5. For a core subset of the type system, we present a formal definition of the static and dynamic semantics. We also prove that our system allows sound static inference about noninterference of effects.

6. We describe a prototype compiler for DPJ that performs the effect checking as described in this thesis and then maps parallelism down to the ForkJoinTask dynamic scheduling framework.

7. We describe an evaluation using 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 three out of six 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.

**Effect system and language for determinism by default:** Chapters 5 and 6 describe extensions to the effect system and language for determinism to add controlled nondeterminism, with the safety guarantees stated in Section 1.1. Again we give an informal description (Chapter 5) followed by a formal treatment (Chapter 6). This part of the thesis makes the following contributions:

1. We present a language that provides the four guarantees stated in Section 1.1 *at compile time*. Our language distinguishes deterministic and nondeterministic parallel tasks. Interference is allowed only inside tasks explicitly identified as nondeterministic, so the language is deterministic by default. As in previous work on languages supported by transactional runtimes [62, 64], inside a nondeterministic composition, the programmer can write a statement `atomic` $S$ that runs the statement $S$ in isolation. However, our effect system guarantees strong isolation even if the underlying TM guarantees only weak isolation. It also guarantees race freedom, which is not guaranteed by any TM systems. To our knowledge, our language is the first to provide all four properties stated above for a shared-memory parallel language.

2. We add a new kind of effect called an *atomic effect* for tracking when memory accesses occur inside an atomic statement. The atomic effects allow the compiler to guarantee both race freedom (property 1) and strong isolation (property 2), by prohibiting conflicting memory operations unless each operation is in an atomic statement.

3. We introduce new effect checking rules to enforce composition of operations (property 3) and determinism by default (property 4). For composition of operations, the extended effect system disallows interference between a deterministic operation and any other concurrent operation unless the whole deterministic operation is enclosed in an atomic statement. For determinism by default, the interference is disallowed for deterministic parallel operations, but allowed for nondeterministic parallel operations.

4. We introduce *atomic regions*, so that the programmer can identify which regions (i.e., sets of memory locations) are allowed to be accessed in an interfering manner. For operations to other regions, the compiler can remove or simplify the STM synchronization, because such operations never cause conflicts.

5. We formalize our ideas using three formal languages: the first has only deterministic parallel operations, the second adds nondeterministic parallel operations, and the third adds atomic regions. We have developed a full syntax, static semantics, and dynamics semantics for all three languages. Further, we have formally stated the soundness properties given informally above, and prove that the properties follow from the semantic definitions.

6. We describe our experience using the language to implement three nondeterministic algorithms: *Delaunay mesh refinement* from the Lonestar benchmarks [2], the *traveling salesman problem* (TSP), and *OO7* [31], a synthetic database benchmark. Our experience shows that porting these algorithms from plain Java into DPJ was relatively straightforward and required neither redesign of existing data structures nor restructuring of the algorithms themselves. Additionally, judicious use of atomic regions eliminated a large fraction of the STM-related overhead in two out of three benchmarks.

**Support for object-oriented frameworks:** Chapters 7 and 8 show how to extend the DPJ effect system to support object-oriented parallel frameworks, as discussed in Section 1.3. Again we give an informal

description (Chapter 7) followed by a formal treatment (Chapter 8). This part of the thesis makes the following contributions:

1. We show how to write a framework API using the DPJ effect system as described in previous chapters so that the framework writer can guarantee disjointness of reference and sound effects for user-supplied methods, assuming the user's code passes the DPJ type checker.

2. We show how to extend the DPJ effect system to add generic effects and generic types, making the frameworks more general and useful. For the effects, we add *constrained effect variables* to enforce disjointness of effect. For generic types, we introduce *type region parameters*, which give the framework author enough information about the types bound to generic type variables to guarantee disjointness and soundness of effect, without knowing the exact type.

3. We give the formal semantics of a core subset of the extended language and formally state the soundness results. We also prove soundness for the formal language.

4. We state the requirements for a correct framework implementation, meaning that if these requirements hold, then noninterference is guaranteed for the entire program. We also show how to use a combination of the DPJ type system and external reasoning to check these requirements informally. We leave as future work the formal verification of the requirements.

5. We describe an evaluation in which we used our language mechanisms to write three parallel frameworks (representing a parallel array, tree, and pipeline) and by writing applications using these frameworks. We found that the language mechanisms are able to capture realistic parallel algorithms. In particular, the pipeline framework expresses a pipeline pattern that cannot be expressed directly using DPJ's fork-join parallel constructs. We also found that the extra annotations required by the system are fairly simple for framework users and, while more complicated for framework writers, are not unduly burdensome.

Chapter 9 concludes by summarizing what has been accomplished here and what remains for future research.

# Chapter 2

# A Research Agenda for Determinism by Default

As discussed in Chapter 1, this thesis argues that to simplify parallel programming, mainstream object-oriented programming languages must become deterministic by default. In this chapter, we lay the groundwork for the following technical chapters by outlining a broad research agenda in support of that goal. The agenda consists of four parts:

**1. How to guarantee determinism in a modern object-oriented language?** For reasons discussed in Section 2.1, our philosophy is to provide *static* guarantees through a combination of a type system and simple compiler analysis when possible, and to fall back on runtime checks only when compile-time guarantees are infeasible. The key is to determine when concurrent tasks make conflicting accesses. The language can help provide this capability in two ways. First, structured parallel control flow makes it easy to analyze which tasks can execute concurrently. Second, language annotations can convey explicitly what data is accessed or updated by a specific task.

**2. How to provide sound guarantees when parts of the program either cannot be proved deterministic or have "harmless" nondeterminism?** Libraries and frameworks written by expert programmers tend to be widely reused, carefully designed, and thoroughly tested. Such code may include components that are not deterministic in isolation, yet can be combined to provide deterministic results. For example, a sequence of commutative inserts to a concurrent search tree within a parallel loop can be executed in arbitrary order and yet give deterministic results, as long as no other operation (e.g., a find) is interleaved between those inserts. Languages should enable such libraries to express contracts that can be enforced by the compiler. The system can then ensure that a client application using the library is deterministic so long as the library implementation meets its specification.

**3. How to specify explicit nondeterminism when needed?** A deterministic-by-default language may need to support transformational computations that permit more than one acceptable answer. If so, the language must achieve three goals. First, any nondeterministic behavior must be *explicit*, e.g., using nondeterministic

control statements; hence the term "deterministic by default." Second, the nondeterminism should be carefully *controlled* so that programmers can reason about possible executions with relatively few interleavings. Third, nondeterministic code should be *isolated* from deterministic code so that the programmer can reason deterministically about the rest of the application.

**4. How to make it easier to develop and port programs to a deterministic-by-default language?** As advocated in this chapter, determinism by default comes at some cost in terms of language annotations. However, the cost is worth the safety and productivity benefits of determinism by default. Further, the cost can be significantly reduced by tools and techniques that infer annotations or help the user write the annotations.

The following sections of this chapter discuss each of these issues in order. We conclude with a discussion of broadly related work that achieves determinism by limiting or excluding side effects.

## 2.1 Guaranteeing Determinism

In this section, we discuss the problem of *guaranteeing* that a program produces deterministic results. As discussed in Chapter 1, a program that is known to be deterministic is much easier to reason about than one that is not. Determinism is also a fundamental *correctness property* for the implementation of any algorithm with a deterministic specification. First, we classify all deterministic algorithms into three broad patterns. Then we discuss technical approaches to enforcing determinism for the three patterns. Finally, we discuss *effect systems*, which we believe are an important part of the solution to enforcing determinism.

Here we assume the computation happens entirely in memory, and we disregard I/O. There are two reasons for this. First, as discussed in Chapter 1, we are primarily concerned with computations that take an input, compute in parallel, and produce an output. In such computations, I/O typically occurs as a separate phase, before the parallel computation. Second, modeling I/O effects is very similar to modeling memory effects. Therefore, the model is easily extended to concurrent computations (e.g., servers) involving concurrent I/O, although such computations are not the primary focus of this thesis.

### 2.1.1 Patterns of Determinism

All deterministic parallel computations that operate on shared memory can be classified into one of three broad patterns:

1. *No memory conflicts.* A parallel computation is deterministic if, for every pair of memory operations in two different tasks that occur in parallel, either (1) both operations are reads, or (2) the operations access disjoint memory locations. In either case, the order of operations has no effect on the result. Examples include computations that read global memory and write to local storage (so-called "embarrassingly parallel" computations), or computations that write to distinct parts of the same data structure (e.g., a tree or array) in global memory.

2. *Synchronized memory conflicts.* A parallel computation is also deterministic if, for every pair of conflicting memory operations between parallel tasks, the operations happen in a deterministic order. For example, thread-level speculation [114] guarantees this property, by enforcing the order of conflicting operations given by a sequential execution of the program. A speculative thread is aborted and restarted if the sequential conflict ordering is violated by the parallel execution.

3. *Confluent memory conflicts.* Finally, a parallel computation may give deterministic results even if it has parallel conflicting memory operations that occur in a nondeterministic order. This property is sometimes called *confluence*. A simple example is a shared read-modify-write counter protected by a lock. If the counter is incremented by $n$ threads, the end result will be $n$, regardless of the order in which the accesses occurred. A slightly more complex example is an associative reduction, e.g., a parallel reduction of an integer array to its sum, using concurrent read-modify-write operations on a reduction variable. A still more complex example is a data structure such as a set or tree built up by concurrent insert operations. In the case of a set, the same set results regardless of the order of insertions. The same property is true of some trees. For example, the spatial tree used in the Barnes-Hut n-body simulation [109] is uniquely determined by the bodies inserted, and independent of the order in which they are inserted. Inserting the bodies in different orders creates different intermediate trees, but the final tree is always the same.

For all three patterns, a fundamental challenge in guaranteeing determinism in a shared-memory imperative language is for the system to detect potentially conflicting memory operations (also called *effects*) between different parallel computations in the program. We call this ability *effect checking*. Effect checking can directly guarantee the first pattern of determinism stated in above (no conflicts), by detecting and prohibiting such conflicts. Effect checking alone is not *sufficient* for patterns two and three: pattern two needs

some sort of ordered runtime synchronization (either speculative or nonspeculative), and pattern three needs some proof of confluence. However, effect checking is still *necessary* for patterns two and three. In pattern two, the computation is deterministic if the conflicting effects are correctly synchronized; therefore, proving correctness requires knowing where the conflicts are. Similarly, in pattern three, the proof of correctness requires proving that conflicting effects are confluent. Again, we cannot get very far if there are unknown conflicting effects.

### 2.1.2 Approaches for Checking Effects

Broadly, there are four known approaches for checking effects:

- **Language-based approaches** (discussed further in Section 2.1.3) use language extensions, usually an extension of the type system, for detecting and/or prohibiting conflicting effects at compile time.
- **Compiler-based approaches** use parallelizing compiler technology (e.g., [72, 27]) to transform sequential programs (with or without annotations) into parallel form.
- **Software runtime approaches** (e.g., [105, 128, 11, 96, 18, 14]) use software runtime checks to detect, and possibly speculation to recover from, violations of deterministic semantics in the execution of a parallel program.
- **Hardware runtime** approaches (e.g., hardware-supported thread-level speculation (TLS) [113, 99] or DMP [43]) use hardware support to achieve the same goals but with less overhead, at the cost of increased hardware complexity.

The four approaches involve different tradeoffs and can be combined in different ways into a composite solution. A language-based approach has the following benefits:

- It allows a high degree of programmer control over the way that data is shared and the way that properties like determinism are checked and enforced.
- It *documents* the available parallelism for future developers, and makes program behavior and performance characteristics explicit in the code.
- It can specify properties that hold at interface boundaries, enhancing modularity. This specification allows the compiler to check and enforce deterministic uses of libraries and frameworks using only

the API; the source code for the implementation is not needed, as it would be for whole-program compiler analysis.

A compiler approach can reduce the burden of writing parallel code compared to a pure language approach. However, for all but very regular codes auto-parallelization is quite difficult; and even where successful it can be brittle (small code changes can destroy performance) and hard to understand.

A robust runtime can reduce or eliminate the need for the programmer or compiler to get the sharing patterns correct. However, runtime approaches can add unnecessary overhead. For example, thread-level speculation with no language or compiler support needs to check every single access to global memory for a potential conflict. Further, runtime approaches can make performance characteristics opaque: synchronizations and aborts can be major performance bottlenecks, and it may not be clear from the text of the program where those are occurring, or how to alleviate the bottlenecks. Further, runtime techniques based on a fail-stop approach are inherently input-dependent: one input may have no conflicts between parallel tasks, while another input has a conflict which causes the program to terminate. As discussed in Chapter 1, while these approaches can guarantee deterministic behavior (the program will always either fail or not fail on a given input), they generally cannot guarantee that a program will behave according to a sequential semantics on all runs.

Overall, explicitly parallel, language-based approaches are the only ones that provide the benefits of performance control, explicit interfaces, modularity, documentation, and compile-time enforcement. We therefore believe that such an approach is the most attractive in the long term. Such an approach can be combined with limited runtime software and hardware checking to enable greater expressivity, where needed.

### 2.1.3 Effect Systems

We believe that an important part of the solution to controlling sharing is a particular language mechanism called an object-oriented *effect system* [82, 30]. Effect systems provide annotations that partition the heap and declare which parts of the heap are accessed by each task, and in what way (for example, read or write). An effect system can easily show that two distinct objects are being created at every recursive call of a divide and conquer pattern, so the subcomputations do not interfere.

The rest of this thesis, after this chapter, presents our work on Deterministic Parallel Java (DPJ). DPJ

uses a sophisticated effect system that partitions the heap into hierarchical *regions* and uses those regions to disambiguate accesses to distinct objects, as well as distinct parts of the same object, referred to through data structures such as sets, arrays, and trees. DPJ's effect system readily supports the first pattern of deterministic parallelism (no conflicts). A simple *local* type-checker can then ensure that there are no conflicting memory operations between concurrent tasks. In a correct DPJ program, nondeterminism cannot happen by accident: any such behavior must be explicitly requested by the user, and a DPJ program with no such request has an obvious sequential equivalent. DPJ's support for the other two patterns (synchronized conflicts and confluent conflicts) is discussed below, in Section 2.2.

When static checks do not work, either because the analysis is not possible or the annotation burden is not justified by the performance gains, we must fall back on runtime techniques. One approach is to use software speculation [128], with hardware support [99] if it is available to reduce overhead . An alternative approach is a fail-stop model that aborts the program if a deterministic violation occurs [105]. This approach gives a weaker guarantee, but it avoids the overhead of logging and rollback. A combination of the two approaches could also be used. For example, fail-stop checking could be used for production runs, while speculation could be used to simplify the initial porting of programs by producing a guaranteed-correct speculative version on the way to a more efficient version. The extra overhead of speculation can be tolerated more readily during the development process. Speculation could also be used (even in production runs) for algorithms that are *inherently speculative*, where new tasks must be launched speculatively or the entire algorithm would become serial [74]. Here, the overhead of speculation may be more tolerable if it is the only way to express the algorithm at all.

## 2.2 Encapsulating Complex Behaviors

In realistic programs, the guarantee of determinism may have to be weakened for parts of the program, for performance or expressivity. However, if we can encapsulate those parts behind an interface with suitable contracts, and guarantee that client code satisfies those contracts, then we can still provide sound guarantees for the rest of the program. This approach is attractive because the encapsulated code can often be placed in libraries and frameworks written by expert programmers skilled in low-level parallel programming and performance issues.

### 2.2.1 Local Nondeterminism

As discussed in Section 2.1.1, a parallel algorithm may have memory conflicts that exhibit nondeterministic intermediate states, while producing confluent final results. Further (e.g., in a reduction), the nondeterminism in the intermediate states is often necessary for good performance. Experienced programmers should be able to write such computations and encapsulate their code in libraries that have deterministic external behavior, with well-defined properties.

In DPJ, we adopt this idea in the form of a `commuteswith` annotation telling the compiler that two methods commute with each other, even if the effect system reports interference. For example, a concurrent read-modify-write update to a shared counter variable is commutative, but the effect system does not know that: it just sees interfering writes. The `commuteswith` annotation allows an experienced programmer to provide a concurrent counter (or more sophisticated examples, such as a concurrent reduction variable, set, or map). The `commuteswith` annotation is discussed further in chapters 3 and 4 of this thesis.

Commutativity is an important special case of confluence (confluence is more general, because it can include more than two operations). Many confluent sequences of operations (e.g., counter updates, set inserts, reductions) can be built up from operations that are all pairwise commutative. Commutativity cannot handle all cases of local nondeterminism, however. In DPJ, we encapsulate such behaviors entirely behind an interface, as discussed in the next section.

### 2.2.2 Unsoundness

In realistic applications, some parts of the program may in fact be deterministic yet perform operations that cannot feasibly be proved sound by the type system or runtime checks. One example is a tree rebalancing. If a data structure is known to be a tree, then this fact can support sound parallel operations, such as a divide and conquer traversal that updates each subtree in parallel. However, rebalancing the tree in a way that retains the guarantee may be difficult without imposing severe alias restrictions such as unique pointers. It is also difficult for a runtime to check efficiently that the tree structure is maintained during and after a rebalancing.

We believe a practical solution in such cases is to allow unsound operations, i.e., operations that may break the determinism guarantee, but to encapsulate those operations inside well-defined data structures and frameworks using traditional object-oriented encapsulation techniques (private and protected fields and

inner classes) supplemented by effect analysis and/or alias control. The effect and alias restrictions can help keep track of what is happening when references in the rest of the code point to data inside an encapsulated structure [30]. Then the compiler can use the guarantees provided by the data structure or framework interface to provide sound guarantees for the rest of the program.

In DPJ, we have applied this idea by extending the effect system to support parallel frameworks. For example, a tree framework can ensure algorithmically that all API operations on the tree (such as rebalancing) maintain the tree structure. This fact can then be used to provide API operations to the user (such as iterating over the tree in parallel and updating its elements) with sound guarantees of noninterference. Frameworks can also support patterns two and three discussed in Section 2.1.1 above, by encapsulating synchronization patterns or confluent operations that the effect system alone cannot prove deterministic. For example, a parallel algorithm for building a spatial tree could be incorporated in a framework, again with an algorithmic guarantee of confluence. In all these cases, the framework API and implementation cooperate with the effect system to provide the deterministic guarantees. Chapters 7 and 8 of this thesis discuss DPJ's framework support.

## 2.3 Explicit Nondeterminism

As discussed in Chapter 1, some algorithms produce several acceptable answers. In contrast to encapsulated nondeterminism in the context of a deterministic program (Section 2.2), here the visible program behavior is nondeterministic. It is probably not desirable to exclude such algorithms entirely from a parallel programming model.

We wish to express such algorithms while achieving the following goals. First, nondeterminism is explicitly expressed, e.g., using a nondeterministic control statement. As discussed in Chapter 1, this is what we call determinism by default. Second, nondeterminism is carefully controlled, so that the programmer need reason about only relatively few program interleavings. Third, the nondeterministic part of the application should not compromise the ability to reason deterministically for the rest of the application.

In DPJ, we achieve these goals in the following way:

- The language explicitly distinguishes parallel constructs that enforce determinism from those that do not. Specifically, `foreach` (for a parallel loop) and `cobegin` (for a parallel statement block) en-

force noninterference between their component parallel tasks, and so guarantee determinism; while `foreach_nd` and `cobegin_nd` (where "nd" stands for "nondeterministic") allow interference between their component parallel tasks, and therefore allow nondeterminism. The language is deterministic by default, because a statement $S$ executes deterministically *unless* the execution of $S$ encountered a dynamic instance of `foreach_nd` or `cobegin_nd`.

- For nondeterministic computations, the language supports an atomic statement `atomic` $S$ that guarantees *strong isolation* for $S$. As discussed further in Chapter 5, because DPJ allows interference *only* between pairs of `atomic` statements, the *only* interleavings that programmers must reason about is the interleaving of `atomic` statements. All other statement orderings follow from program order (i.e., once the order of `atomic` statements is specified, program execution is deterministic).

- Again as discussed in Chapter 5, `foreach` and `cobegin` always behave deterministically, *even inside a `foreach_nd` or `cobegin_nd`*. In particular, they behave like a sequential composition of their component tasks in the obvious order (i.e., the one that would occur if `foreach` or `cobegin` were elided). This property fosters local, compositional reasoning about parallel constructs. In particular, deterministic constructs behave consistently, no matter where they occur.

Chapters 5 and 6 of this thesis discuss DPJ's support for nondeterministic computations.

The Galois system [74] provides capabilities similar to our `foreach` and `foreach_nd`, except that it is possible to write incorrectly synchronized programs (for example, that have data races) in Galois. Our aim is to leverage the effect system described in this thesis to *guarantee* the properties described above.

## 2.4 Usability

A common concern regarding language-based solutions is the cost to rewrite existing programs and to learn new language features. We believe that (1) the costs tend to be exaggerated and the benefits underestimated; and (2) strong technical solutions can significantly reduce the costs. We discuss both points briefly in turn.

**Costs and benefits of language solutions:** First, we are proposing a small set of extensions to an established base language (such as Java or C#), *not* an entirely new language. This fact should mitigate the up-front cost of both learning the new features and writing code that uses the new features. Further, the extra effort to learn and use new language features is likely to be dwarfed by the effort required to write, port, tune,

and test parallel code. A well-designed language that simplifies the latter tasks can more than justify the learning curve. Note also that because we are extending a base language, porting can be done *incrementally*, e.g., kernel by kernel.

Second, although object-oriented effect notations require some extra effort from the programmer, such effort is not wasted. First, effect annotations on methods provide a compiler-checkable *interface* that allows sound, modular reasoning about program components, even in the absence of all the source code (such as for a library or framework). Thus, the annotations enhance modularity and composability. Second, the reasoning required to introduce regions and effects is exactly the reasoning required to understand the sharing patterns in the code. In fact, the region and effect mechanisms give programmers a concrete guide for how to carry out such reasoning.

Third, nontrivial real-world applications are long-lived, and initial development or porting costs are usually a small fraction of long-term maintenance and enhancement costs. A language that simplifies testing and documents sharing patterns in the code reduces maintenance costs.

Fourth, current thread-based languages have woefully inadequate shared memory models. The only memory model accepted today guarantees sequential consistency for data race-free programs, as for Java and (soon) C++. The difficulty lies in the semantics of data races. C++ does not provide any. Java provides semantics that are complex and fragile. If we are to move towards safe parallel languages with tractable memory models, we *must prohibit data races for all allowed programs*. A type and effect system, as discussed in Section 2.1.3, could accomplish this goal with low runtime overhead.

**Reducing the costs:** Some *technical solutions* can further reduce the cost of using new language features:

- *Inferring annotations*: We are exploring how judicious use of effect inference (inferring region and effect annotations) can reduce the programming burden of a system like DPJ [122].

- *Runtime checks*: The language can provide runtime checks, as described in Section 2.1.3, so that large programs can initially be ported without all the effect annotations needed for compile-time checking. The overheads of runtime checks can then be incrementally tuned away by introducing effect annotations where the benefits justify the effort.

- *Integrated development environment (IDE)*: An IDE can use sophisticated *interactive* compiler parallelization technology, combined with modern refactoring technology, to assist the initial porting

process. Making porting a one-time effort allows such an environment to use powerful, but potentially slow, interprocedural parallelization techniques (the strengths of compilers); while making it interactive allows programmers to influence the process and avoid the problems of poor or brittle performance (the weaknesses).

## 2.5 Related Work: Limiting Side Effects

As stated previously, the broad goal of this thesis is to develop a deterministic-by-default language with expressive side effects. Parallelism is easier to express when there are no side effects, because there are no hidden and potentially conflicting accesses to shared memory. However, the absence of side effects requires a more restrictive programming model than is typical for an imperative, object-oriented language with reference passing and mutable objects. Here we briefly survey contrasting approaches, in which safety guarantees are obtained by restricting or eliminating side effects. Related work that is technically closer to DPJ (e.g., other work on effect systems) will be discussed in the relevant following chapters of this thesis, after the technical discussions.

**Data parallel languages.** Data parallel languages express parallel operations on the elements of regular data structures, such as arrays, containing numerical values. CM Fortran [40] and Fortran 90 [88] enabled a SIMD (single instruction, multiple data) style of data parallel programming in which a single statement can operate on an entire *slice* (rectangular subsection) of an array. Later languages introduced mechanisms for specifying the distribution of data across large distributed-memory machines and operations on more complex data structures such as sparse arrays. These languages include Fortran variants [53, 40, 131, 67, 87]; languages based on C and C++ [66, 83]; NESL [21]; , ZPL [32], and Hierarchically Tiled Arrays [19]. Data parallel language constructs can elegantly express deterministic parallelism and achieve very high performance when the data and operations are regular; but they cannot express, or achieve poor performance for, important features like multiple threads of control, parallelism across pointer-based data structures, and dynamic creation and deletion of tasks or threads.

**Parallel functional languages.** Purely functional languages are side effect free. Parallel functional languages such as Concurrent Haskell [70], ML [102], of parallel functional languages use this fact to express parallelism elegantly [79]. However, the absence of side effects means that many common programming patterns are impractical. For example, even a simple operation such as appending to a list requires making

a new copy of the entire list. There has been work on understanding the *effect interference* that results when side effects are added to functional languages [55, 71, 126]. Recently, Dowse et al. have shown how to model I/O in concurrent Haskell while retaining deterministic guarantees [46]. To our knowledge, none of this work investigates language annotations for non-speculative, deterministic parallelism in the presence of pointer-based sharing of mutable data.

**Dataflow languages.** In the 1970s and 1980s, many researchers proposed *dataflow languages* for programming a *dataflow architecture* in which programs were represented as dataflow graphs, exposing all data dependences at the instruction level [94, 49]. By the 1990s, it became apparent that to achieve efficiency, the graph nodes had to consist of many instructions, leading back to a functional style. Today, dataflow programming is mainly used for digital signal processing and other computations called *stream computations* that are naturally expressed as subcomputations called *kernels* interacting via a dataflow graph [69]. It appears unlikely that this approach can replace traditional imperative programming on von Neumann architectures for general-purpose applications.

**Parallel message passing languages.** There has been some work on guaranteeing deterministic results for programs written in an explicit message passing style on a distributed-memory machine. Compositional C++ [34] provides a *synch variable* that behaves like a single assignment variable in a dataflow language. Fortran M [33] allows explicit communications via *channels* and coordinates reading and writing via *tokens* that can be passed through the channels. While similar in spirit to what we are trying to achieve, these languages use very different mechanisms that appear difficult to extend to shared memory systems.

# Chapter 3

# Effect System and Language for Determinism

This chapter describes DPJ's effect system and language features for supporting deterministic parallelism. As discussed in Chapter 1, the basic strategy is as follows:

1. The programmer explicitly marks which sections of code are to be run deterministically in parallel.

2. The programmer partitions the set of heap memory locations (i.e., object fields and array cells) using *regions*, which are sets of memory locations. The programmer also writes *effect summaries* on methods, indicating which regions are read and written by the method.

3. The compiler checks that the method summaries are correct (i.e., they include all the actual effects of the method) and that parallel tasks are *noninterfering*. That means that if any pair of parallel tasks accesses the same region, then the accesses commute — for example, they are both reads.

The rest of this chapter proceeds as follows. Section 3.1 describes some basic capabilities that DPJ shares with other effect systems. Section 3.2 describes *region path lists*, or RPLs. RPLs represent a novel way to partition the heap hierarchically, and are the key to DPJ's expressive effect specifications and subtyping. Section 3.3 describes DPJ's features for supporting parallel computations on arrays. Section 3.4 describes DPJ's *commutativity annotations* for specifying that two method invocations (e.g., two updates to a shared counter) may be applied in parallel with deterministic results, even though they have interfering reads and writes. Section 3.5 presents an evaluation of the deterministic effect system and language, including expressivity, performance, and usability. Section 3.6 discusses related work.

## 3.1 Basic Capabilities

We begin by summarizing some basic capabilities of DPJ that are similar to previous work [82, 77, 59, 30, 35]. We refer to the example in Figure 3.1, which shows a simple binary tree with three nodes and a

```
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              /* Inferred effect is 'reads Links writes L' */
10             left.mass = mass;
11             /* Inferred effect is 'reads Links writes R' */
12             right.mass = mass;
13         }
14     }
15 }
```

Figure 3.1: Basic features of DPJ. New DPJ syntax is highlighted in bold face. Effects inferred by the compiler are given in comments. Note that method `initTree` (line 7) has no effect annotation, so it gets the default effect summary of "reads and writes the entire heap."

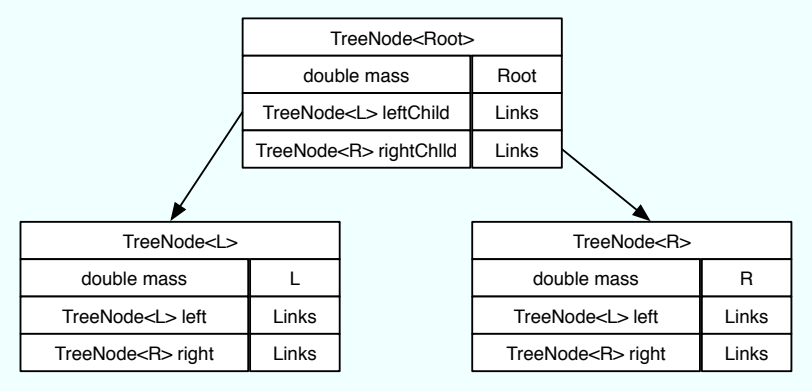


Figure 3.2: Runtime heap typing from Figure 3.1

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.

**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 3.3.1.

**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 *class region declaration* declares a new name $r$ (called a *class 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. A class 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 class 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.

**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 (in DPJ, type parameters always come first and may be preceded by the keyword `type`). 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, each with its data in a different region.

To control aliasing of region parameters, the programmer may write a disjointness constraint [30] of the form $P_1 \ \# \ P_2$, where $P_1$ and $P_2$ are parameters (or regions written with parameters; see Section 3.2) that are required to be disjoint. Disjointness of regions is fully explained in Section 3.2; 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: in a Java program without arrays, like this one, the heap consists of a set of class objects, every class object has a set of fields, and every field has a region. In this simple example, exactly three regions are used for partitioning the heap — `Links`, `L`, and `R` — and every class field at runtime is in one of those regions. See Figure 3.2 for an illustration of the runtime heap typing, assuming the root node has been instantiated with `Root`. In the following sections, we will describe a *nesting relation* on regions that allows hierarchical partitions of the heap, and we will also show how to handle arrays.

**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. Every actual effect of the method must be represented by (sometimes we say “covered by”) an effect in the

summary. A write effect in the summary covers a read effect or a write effect to the same region in the method, but a read effect in the summary covers only a read effect in the method. This rule ensures that the compiler can report all read-write and write-write conflicts between different methods invoked in parallel.

When one method overrides another, the effects of the superclass method must cover 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 [77, 59]. The full DPJ language also includes *effect variables*, 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); effect variables will be discussed in Chapters 7 and 8 of this thesis.

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 3.1. This scheme allows ordinary sequential Java to work correctly, but it requires the programmer to add the annotations in order to introduce safe parallelism. In particular, methods that are never called inside a parallel code section do not require an explicit effect summary, but methods that are called inside a parallel code section do.

**Proving determinism:** To type check the program in Figure 3.1, 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.

## 3.2 Region Path Lists (RPLs)

An important concept in effect systems is *region nesting*. Region nesting supports a hierarchical partitioning of the heap, so the effect system can express that different computations are occurring on different parts of

the heap. For example, to extend the code in Figure 3.1 to a tree of arbitrary depth, we need a tree of nested regions (and the nesting must be unbounded). As discussed in Section 3.3, we can also use nesting to express that (1) two aggregate data structures (like arrays) are in distinct regions and (2) 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 [37, 30] or use explicit declarations that one region is under another [77, 59]. 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.

### 3.2.1 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 $r$,[1] 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 3.1. 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 $R_1 \preceq R_2$ if $R_1$ is under $R_2$. Note that the under relation establishes a *hierarchy of distinct regions*; it does not specify *inclusion of regions*. In particular, `A` and `A:B` are distinct regions (so, for example, `writes A` does *not* imply `writes A:B`). In the next subsection, we will show how to specify *sets of regions* (for example, "all regions under `A`") that can express inclusion relations.

We may also write a region parameter, instead of `Root`, at the head of an RPL, for example `P:A`, where

[1] As noted in Section 3.1, this can be a package- or class-qualified name such as `C.r`; for simplicity, we use $r$ throughout.

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.[2] 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, $\texttt{P:A} \preceq \texttt{P}$, and $\texttt{P:A} \neq \texttt{P:B}$ if and only if $\texttt{A} \neq \texttt{B}$.

Figure 3.3 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 3.1 by 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 from each other (i.e., line 10 vs. line 14 and line 12 vs. line 16). Figure 3.4 shows this situation graphically. The different bindings to P provide a "vertical partition" that distinguishes the two objects, while the different names after the colon provide a "horizontal partition" that distinguishes between the fields. Moreover, DPJ's rules for type comparisons ensure that this kind of reasoning is sound, because the types must match in the left- and right-hand sides of any assignment. For example, it is a compile-time error to attempt to assign a value of type `TreeNode<L>` to a variable of type `TreeNode<R>`.

### 3.2.2 Specifying Sets of Regions

DPJ's regions support recursive algorithms on a tree of unbounded depth. For example, consider the code shown in Figure 3.5. Here we are operating on the same `TreeNode` shown in Figs. 3.1 and 3.3, 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`

[2] As with Java generics, the region parameter information is erased at compile time and not represented at runtime, so there is no runtime cost to this instantiation.

```
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 3.3: Extension of Figure 3.1 showing the use of region nesting and region parameters

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.

Moreover, DPJ's nested regions naturally encode this fact. Notice that we have written the types of fields `left` and `right` `TreeNode<P:L>` and `TreeNode<P:R>` instead of `TreeNode<L>` and `TreeNode<R>` as before. As shown in Figure 3.6, through left-recursive substitution, every node in the runtime tree has a different region bound to the parameter of its type, and the RPL in the region specifies the position in the tree. For example, the root node has type `Tree<Root>`, while the right child of the left child of the root has type `Tree<Root:Left:Right>`. Again, DPJ's type system ensures this reasoning is sound. We just need a way to express effects on the different parts of the tree, which we describe below.

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

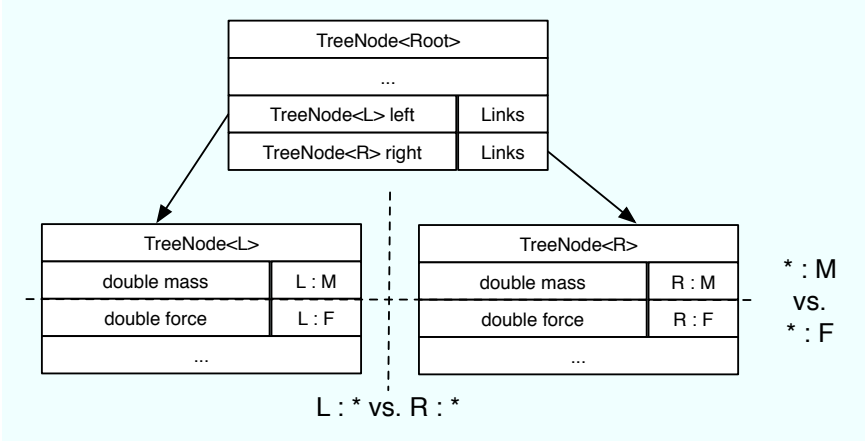


Figure 3.4: Graphical depiction of the distinctions shown in Figure 3.3. The `*` denotes any sequence of RPL elements; this notation is explained further in Section 3.2.2.

**Distinctions from the left:** In lines 11–15 of Figure 3.5, 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, and 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 $n$th 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

```
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 3.5: Using partially specified RPLs for effects and subtyping

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.

### 3.2.3 Subtyping and Type Casts

**Subtyping:** Partially specified RPLs are also useful for subtyping. For example, in Figure 3.5, 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 3.5. 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 $R_1 = R_2$. Note that nesting and inclusion are related: $R_1 \preceq R_2$ implies $R_1 \subseteq R_2\texttt{:*}$. However, nesting alone does *not* imply inclusion of the corresponding sets. For example, $\texttt{A:B} \preceq \texttt{A}$, but $\texttt{A:B} \not\subseteq \texttt{A}$, because `A:B` and `A` denote distinct regions.

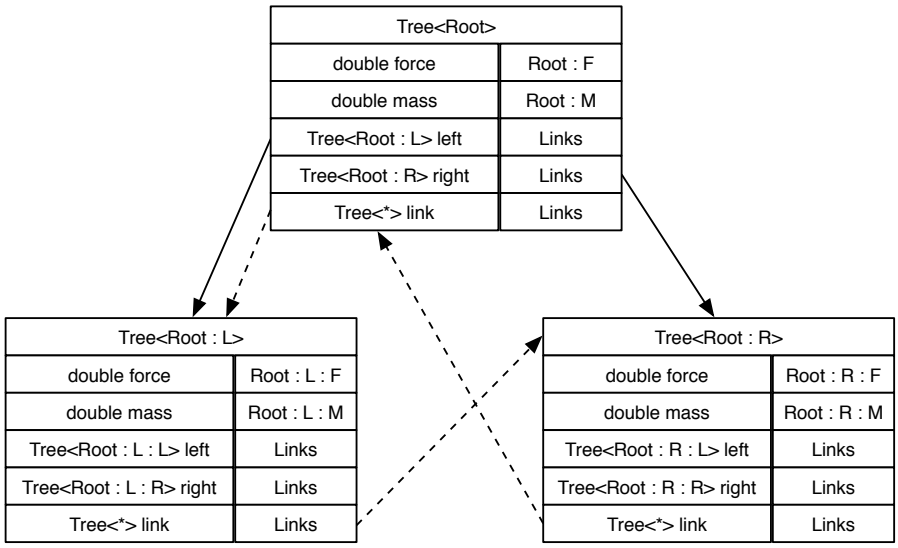


Figure 3.6: Heap typing from Figure 3.5. Reference values are shown by arrows; tree arrows are solid, and non-tree arrows are dashed. Notice that all arrows obey the subtyping rules.

In the next chapter, we discuss the rules for nesting, inclusion, and disjointness of RPLs more formally.

Figure 3.6 illustrates one possible heap typing resulting from the code in Figure 3.5. 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 3.6. 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 potentially 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.

While unchecked downcasts from `Object` can violate determinism, in practice they should not be necessary in a DPJ program: because DPJ fully supports generic types, assigning to and from `Object` is never necessary, except for compatibility with legacy code. In particular, we were able to write all the benchmarks discussed in Section 3.5 without any unchecked downcasts. We could add runtime checking of region compatibility at the point of the cast, as in [24].

## 3.3 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.

### 3.3.1 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. Figure 3.7 illustrates an array of objects violating this property. 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 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

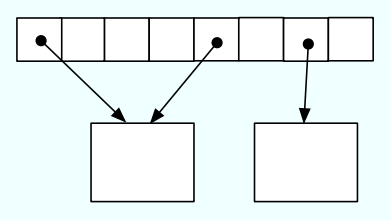

Figure 3.7: An array with duplicate references. The small boxes are array cells, the large boxes are objects, and the arrows are references. Traversing and following the references and performing parallel updates can violate determinism, even if the traversal is unique.

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 3.2.1. Obviously, the compiler's capability to distinguish such types will be limited 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 [54]. 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

```
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 3.8: Example using an index-parameterized array

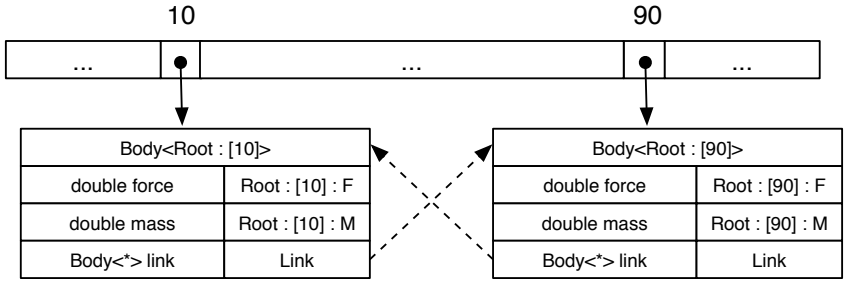


Figure 3.9: Heap typing from Figure 3.8. The type of array cell $i$ is parameterized by $i$. Cross-links are possible, but if any links are followed to access other array cells, the effects are visible.

omit the #$i$ and use an underscore (_) as an implicitly declared variable. For example, `C<[_]>[]<[_]>` is equivalent to `C<[i]>[]<[i]>#i`.

Figure 3.8 shows an example, which is similar in spirit to the Barnes-Hut force computation discussed in Section 3.5. 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 3.9 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 3.9. In that case, in a parallel traversal over the array, either the shared 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 variables (including cells in other index-parameterized arrays), unlike linear types [59, 25, 26]. 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.

```
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 3.10: Writing quicksort with the `Partition` operation. `DPJArrayInt` and `DPJPartitionInt` are specializations to `int` values. In line 12, the argument `OPEN` is an `enum` that says to omit the partition index from the subarrays, i.e., they are open intervals.

### 3.3.2 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 3.10 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 3.10, 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 an access to element $i$ into an 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` ob-

jects, 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 3.10 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. Segment 0 is a `DPJArray` representing the elements of the original `DPJArray` array with indices less than $p$, and segment 1 is a `DPJArray` representing the elements of the original array with indices greater than $p$. 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. 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 $v$ (including `this`) of class type to appear at the head of an RPL, for example $v$`:r`. The variable $v$ 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 ensures that different partitions get different regions, and making the variable `final` ensures that it always refers to the same region.

We make these "variable regions" into a tree as follows. If $v$'s type is `C<`$R$`,...>`, then $v$ is nested under $R$; the first region parameter of a class functions like the *owner parameter* in an object ownership system [39, 37]. In the particular case of `DPJPartition`, if the type of $v$ is `DPJPartition<`$R$`>`, then the type of $v$`.get(`$i$`)` is $v$`:[`$i$`]:*`, where $v \preceq R$. Internally, the `get` method uses a type cast to generate a `DPJArray` of type `this:[`$i$`]:*` that points into the underlying array. The soundness of the type cast is not checked by 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 3.10, 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 disjoint, by a distinction from the left (Section 3.2.2). 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 $v_1$ and $v_2$ represent different partitions of the same array, then $v_1$`.get(0)` and $v_2$`.get(1)` could overlap. Therefore, we must not treat them as disjoint. This is why we put `*` at the end of the type $v$`:[`$i$`]:*` of $v$`.get(`$i$`)`; otherwise we could incorrectly distinguish $v_1$`:[0]` from $v_2$`:[1]`, using a distinction from the right. Second, if $v$ has type `DPJPartition<`$R$`>`, then $v$`.get(`$i$`)` has type `DPJArray<`$v$`:[`$i$`]:*>` and points into a `DPJArray<`$R$`>`. Therefore, we must not treat $v$`:[`$i$`]:*` as disjoint from $R$. Here, we simply do not include this distinction in our type system. All we say is that $v$`:[`$i$`]:*` $\preceq R$. See Section 4.3.2 for further discussion of the disjointness rules in our effect system.

## 3.4 Commutativity Annotations

Sometimes to express parallelism we need to look at interference in terms of higher-level operations than read and write [74]. 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.

DPJ contains two features that address this problem. 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 3.11(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 [130] or static analysis [45, 10]) to ensure that methods declared to be commutative really are 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 is indeed

```
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 3.11: Illustration of `commuteswith` and `invokes`

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. 3.11(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 3.11(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 4.1.4 gives the formal rules for covering effects). The invocation effect will be used only in 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 3.11.

## 3.5 Evaluation

We have carried out a preliminary evaluation of the language and type system features presented in this chapter. 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? 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.
- **Usability.** How much programmer effort is required to write a DPJ program, compared to plain Java? Is the effort worth it, given the strong determinism guarantee that DPJ provides over plain Java?

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 [3]. `ForkJoinTask` supports dynamic scheduling of lightweight parallel tasks, using a work-stealing scheduler similar to that in Cilk [22]. 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 multi-threaded 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 [109], k-means clustering from the STAMP benchmarks [90], 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.

### 3.5.1 A Realistic Example

We use the Barnes-Hut force computation to show how to write a realistic parallel program in DPJ. Figure 3.12 shows a simplified version of this code. The main simplification is that the `Vector` objects for representing points in three-dimensional space are immutable, with `final` fields (so there are no effects on these objects), whereas our actual implementation uses mutable objects. Figure 3.13 shows a partial implementation of the `Vector` class used in this code.

In Figure 3.12, 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, 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,

```
/* Abstract class for tree nodes */
abstract class Node<region R> {
    /* Region for mass and position */
    region MP;
    /* Mass */
    double mass in R:MP;
    /* Position */
    Vector pos in R:MP;
}

/* Inner node of the tree */
class InnerNode<region R> extends Node<R> {
    /* Region for children */
    region Children;
    /* Children */
    Node<R:*>[]<R:Children> children in R:Children;
}

/* Leaf node of the tree */
class Body<region R> extends Node<R> {
    /* Region for force */
    region Force;
    /* Force on this body */
    Vector force in R:Force;
    /* Compute force of entire subtree on this body */
    Vector computeForce(Node<R:*> subtree) reads R:*:Children, R:*:MP {
        ...
    }
}

/* Barnes-Hut tree */
class Tree<region R> {
    /* Region for tree */
    region Tree;
    /* Root of the tree */
    Node<R> root in R:Tree;
    /* Leaves of the tree */
    Body<R:[_]>[]<R:[_]> bodies in R:Tree;
    /* Compute forces on all bodies */
    void computeForces() writes R:* {
        foreach (int i in 0, bodies.length) {
            /* reads R:Tree, R:*:InnerNode.Children, R:[i],
               R:*:Node.MP writes R:[i]:Node.Force */
            bodies[i].force = bodies[i].computeForce(root);
        }
    }
}
```

Figure 3.12: Using DPJ to write the Barnes-Hut force computation

```
1  /* Immutable vector representing a point in space */
2  class Vector {
3      /* Coordinates of the vector */
4      final double x, y, z;
5      public Vector(double x, double y, double z) pure {
6          this.x = x; this.y = y; this.z = z;
7      }
8      /* Add two vectors to produce a new vector representing the sum */
9      public static Vector add(Vector a, Vector b) pure {
10         return new Vector(a.x+b.x,a.y+b.y,a.z+b.z);
11     }
12     /* More vector operations not shown */
13     ...
14 }
```

Figure 3.13: `Vector` class for the Barnes-Hut force computation

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 3.2). We also use distinct region names for the force, mass, and children fields of the `Node` classes 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.

### 3.5.2 Expressiveness

We used DPJ to express all available parallelism in the algorithms we studied. 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 [109].

Briefly, we parallelized each of the codes as follows. MergeSort uses subarrays (Section 3.3.2) 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 3.3.1) to generate an array of tasks and compute an array of results, followed by commutativity annotations (Section 3.4) to update 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 3.5.1 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 3.1 summarizes the novel DPJ capabilities used for each code.

Table 3.1: Capabilities used in the benchmarks

| **Capability** | **Merge Sort** | **Monte Carlo** | **IDEA** | **Barnes-Hut** | **Collision Tree** | **K Means** |
|---|---|---|---|---|---|---|
| Index-parameterized array | - | Y | - | Y | - | - |
| Distinctions from the left | Y | Y | Y | Y | Y | - |
| Distinctions from the right | - | - | - | Y | - | - |
| Recursive subranges | Y | - | Y | - | - | - |
| Commutativity annotations | - | Y | - | - | - | 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 [109]. As discussed in Section 3.5.1, we use an index-parameterized array to update the bodies in parallel. As discussed in Section 3.3.1, this requires that we copy each body with the new destination regions at the point of re-insertion. Chapter 7 of this thesis discusses this problem further and proposes one solution, using object-oriented frameworks. We also believe we can ease this restriction by adding a mechanism for disjointness checking at runtime, and this is ongoing work in our research group.

### 3.5.3 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 3.14 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 3.2, for the three codes where we have manually parallelized Java threads versions available, the DPJ versions achieved speedups close to (for IDEA and Barnes Hut), or better than (for 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 thread-based parallel programming model that provides no guarantees of determinism or even race-freedom.

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, apart from 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 3.5.2, is one such example.

| Num | Monte Carlo | | | IDEA | | | Barnes Hut | | |
|---|---|---|---|---|---|---|---|---|---|
| Cores | DPJ | Java | Ratio | DPJ | Java | Ratio | DPJ | Java | Ratio |
| 2 | 2.00 | 1.80 | 1.11 | 1.95 | 1.99 | 0.98 | 1.98 | 1.99 | 0.99 |
| 3 | 2.82 | 2.50 | 1.13 | 2.88 | 2.97 | 0.97 | 2.96 | 2.94 | 1.01 |
| 4 | 3.56 | 3.09 | 1.15 | 3.80 | 3.91 | 0.97 | 4.94 | 3.88 | 1.27 |
| 7 | 5.53 | 4.65 | 1.19 | 6.40 | 6.70 | 0.96 | 6.79 | 7.56 | 0.90 |
| 12 | 8.01 | 6.46 | 1.24 | 9.99 | 11.04 | 0.90 | 11.4 | 13.65 | 0.84 |
| 17 | 10.02 | 7.18 | 1.40 | 12.70 | 14.90 | 0.85 | 15.3 | 19.04 | 0.80 |
| 22 | 11.50 | 7.98 | 1.44 | 18.70 | 17.79 | 1.05 | 23.9 | 23.33 | 1.02 |

Table 3.2: DPJ vs. Java threads performance for Monte Carlo, IDEA encryption, and Barnes Hut. The DPJ and Java numbers are speedups, and Ratio is the DPJ number divided by the Java number.

### 3.5.4 Usability

Table 3.3 shows the number of source lines of code (LOC) 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., class 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. The biggest number of changed lines resulted from writing RPL arguments to 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.

Further, programming tools can alleviate the burden of writing annotations. We have developed an interactive porting tool, DPJIZER [122], 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

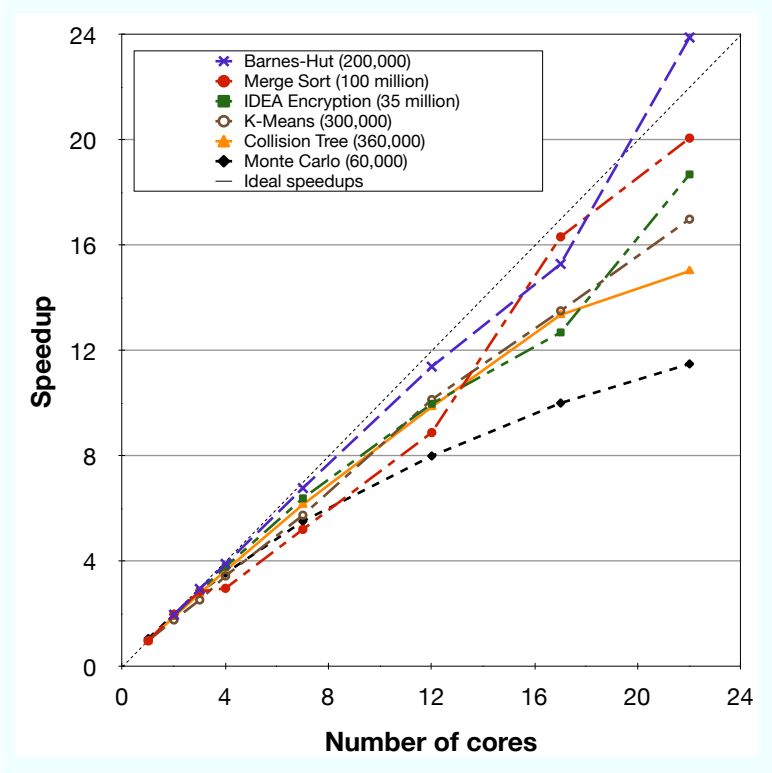


Figure 3.14: Parallel speedups for the six benchmarks. The numbers in the legend are input sizes.

DPJIZER to infer RPLs, assuming that the programmer declares the regions.

In addition, as future work, a good set of defaults could further reduce the amount of manually written annotations. For example, although we have not explored this idea for this work, the design of DPJ could easily be extended as follows. 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. A programmer could override the defaults if further refinements are needed.

## 3.6 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 [82, 61], which adds a region-based type and effect system to a Scheme-like, implicitly parallel language. Leino et al. [77] and

| 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 3.3: Annotation counts for the case studies

Greenhouse and Boyland [59] first added effects to an object-oriented language. None of these systems can represent arbitrarily nested structures or array partitioning, nor can they 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.

Finally, there is extensive literature on using regions for region-based memory management [120, 121, 60]. That work primarily focuses on identifying aliasing, and not noninterference, because their goal is to analyze lifetimes of memory objects.

**Ownership Types:** Though originally designed for alias control [39], 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. Some ownership-based type systems have been combined with effects to enable reasoning about noninterference. Both JOE [37, 111] and MOJO [30] 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 thesis. JOE's `under` effect shape is similar to DPJ's $*$, but it cannot express the equivalent of our distinctions from the right. JOE allows slightly more precision when a type or effect uses a local variable that goes out of scope (our approach is stated precisely in rule LET for typing expressions in the next chapter), 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.

Most ownership systems impose a restriction on the structure of object graphs called *owner dominates*, though some do not [30, 80]. Lu and Potter [80] show how to use effect constraints to break the owner dominates rule in limited ways while still retaining meaningful encapsulation guarantees. The `any` context of [80] 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 [24, 6, 68] have described effect systems for enforcing a locking discipline in non-deterministic 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 [82, 77, 59], 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 is also difficult with ownership. *Ownership domains* [111] represent 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 [26] shows how to use *unique references* extended with *fractional permissions* to guarantee determinism for a simple language with pointers to cells containing values. Terauchi and Aiken [117] have extended this work with a type inference algorithm that simplifies the type annotations and elegantly expresses some simple patterns of determinism. The Plaid language [115] also builds on this work and aims to provide a more robust language for expressing deterministic parallelism.

Unique references are a well-known alternative to effect annotations for reasoning about heap access, and in some cases they can complement effect annotations [59, 25, 24]. Unique references have different strengths and limitations from effect systems. For example, compared to DPJ's indexed parameterized array, an array of unique references can be used to update the objects in parallel and can be reshuffled. However, in the strongest form of unique references, the array is subject to the restriction that no other reference in the entire program may point to any of the objects referred to in the array. This restriction is too severe to be practical for many deterministic programs. While several researchers have examined relaxed uniqueness properties for specific applications [47, 38], designing a realistic deterministic language that uses unique

references is an open problem. As future work, it would be interesting to explore the tradeoffs between DPJ's effect mechanisms and uniqueness constraints for expressing determinism.

**Separation Logic:** Separation logic [103] (SL) is a potential alternative to effect systems for reasoning about shared resources. O'Hearn [95] and Gotsman et al. [58] use SL to check race freedom, though O'Hearn includes some simple proofs of noninterference. Parkinson [98] has extended C# with SL predicates to allow sound inference in the presence of inheritance. Raza et al. [100] 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 [103, 95]. Second, SL approaches generally *either* require heavyweight theorem proving and/or a relatively heavy programmer annotation burden [98] *or* are fully automated, and thereby limited by what the compiler can infer [58, 100].

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 [103], type systems and SL systems have many common goals but have developed largely independently; 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 [105] and Prometheus [11] use runtime checks to guarantee deterministic parallelism for programs that do not fail their checks. Jade also supports a simple form of commutativity annotation [104]. Multi-phase Shared Arrays [42] and PPL1 [112] 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 [18, 51, 128] can achieve determinism with minimal programmer annotations, compared to DPJ. However, speculation generally either incurs significant software overheads or requires special hardware [99, 78, 124]. Grace [18] 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 paralleliza-

tion strategy or show how the code must be rewritten to expose parallelism.

Kendo [96], DMP [43], and CoreDet [17] 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, DMP requires special hardware support, and CoreDet has higher overhead (but stronger guarantees) than Kendo. SharC [13] uses a combination of static and dynamic checks to enforce race freedom, but not necessarily deterministic semantics, in C programs.

Aviram et al. [14] have recently proposed an approach in which a memory consistency model called *deterministic consistency* is enforced by the operating system using runtime mechanisms similar to software distributed shared memory (SDSM). In this approach, each thread receives its own copy of the shared address space at the point where the thread is created, and writes are local by default; merging of writes occurs in a deterministic order only at synchronization points identified in the program (for example, at the end of a parallel loop). While deterministic consistency is a promising approach, the overhead is high for fine-grain sharing patterns (as it is for Grace). Further, it is arguable that copying input state at the beginning of a computation, merging output at the end, and communicating through explicit synchronization points is reminiscent of a functional or message passing style, as discussed in Section 2.5, rather than true shared memory. By contrast, DPJ strives to keep the model close to familiar imperative shared memory.

Burckhardt et al. [29] describe a programming model conceptually similar to that of Aviram et al., but supported by a user-level runtime. Again, each parallel task gets its own copy of shared data, and updates are merged deterministically at task join points. This mechanism seems well suited for some parallel patterns, such as a parallel reduction or a long-running background task that must occasionally interact with the main tasks. However, it does not seem well suited to many of the patterns that DPJ can express, such as disjoint updates on concurrent data structures. Further, it is not clear if this model actually provides any *guarantee* of parallelism: joins can be conditional on reading shared state, and the authors say they used this feature to write a program that produces nondeterministic results.

Finally, a determinism checker [106, 48] 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 coverage of the inputs used for the dynamic analysis. However, it is sound for the observed traces.

# Chapter 4

# Formal Language for Determinism

In this chapter we formalize a core subset of the language described in Chapter 3, called *Core DPJ*. We also prove that the Core DPJ effect system is *sound*, in the sense that the static type and effect annotations allow sound reasoning about noninterference at runtime. To make the formal 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 DPJ 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 numbers.

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 informally in Section 4.4. Removing simplifications 3 and 4 is purely a matter of bookkeeping. We have chosen to make Core DPJ a sequential language, in order to focus on our mechanisms for expressing effects and noninterference. In Section 4.4, we discuss how to extend the formalism to model the `cobegin` and `foreach` constructs of DPJ. In Chapter 6, we also describe such an extension formally (for `cobegin`) for a variant language with simpler mechanisms for expressing noninterference, that also expresses nondeterministic computations.

The rest of this chapter proceeds as follows. In Section 4.1, we present the syntax and static semantics of Core DPJ. In Section 4.2, we present the dynamic execution semantics and prove the key property of

| | | |
|---|---|---|
| **Programs** | $\mathcal{P}$ | $\mathcal{R}^*\ \mathcal{C}^*\ e$ |
| **Region Names** | $\mathcal{R}$ | `region` $r$ |
| **Classes** | $\mathcal{C}$ | `class` $C$`<`$\rho$`>` $\{\ F^*\ M^*$ *comm*$^*\}$ |
| **RPLs** | $R$ | `Root` $\mid \rho \mid v \mid R$`:`$r \mid R$`:[`$i$`]` $\mid R$`:*` |
| **Fields** | $F$ | $T\ f$ `in` $R_f$ |
| **Types** | $T$ | $C$`<`$R$`>` $\mid T$`[]<`$R$`>#`$i$ |
| **Methods** | $M$ | $T\ m(T\ x)\ E\ \{\ e\ \}$ |
| **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.`$f$`=`$v \mid$ `this.`$f \mid v$`[`$n$`]=`$v \mid v$`[`$n$`] \mid `$v$`.`$m$`(`$v$`) \mid `$v$ `new` $C$`<`$R$`> \mid` `new` $T$`[`$n$`]<`$R$`>#`$i$ |
| **Variables** | $v$ | `this` $\mid x$ |
| **Commutativity** | *comm* | $m$ `commuteswith` $m$ |

Figure 4.1: Static syntax of Core DPJ. $C$, $\rho$, $f$, $m$, $x$, $r$, and $i$ are identifiers, and $n$ is a natural number. $R_f$ denotes a fully specified RPL (i.e., containing no `*`).

*type and effect preservation*, which says that static types and effects bound their dynamic counterparts. In Section 4.3, we define disjointness of regions and noninterference of effect, and we prove the main soundness property of DPJ, namely that expressions with noninterfering static effects can be executed in either order with identical results. Finally, in Section 4.4, we informally describe how to extend the core language to add inheritance and explicit parallelism.

## 4.1 Syntax and Static Semantics

Figure 4.1 defines the syntax of Core DPJ. The syntax consists of the key elements described in Chapter 3 (region path lists, 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. As in Chapter 3, we denote a fully specified region path list (RPL) as $R_f$ and a general RPL as $R$.

The static typing is done with respect to an environment $\Gamma$, where each element of $\Gamma$ is one of the following:

- A binding $(v, T)$ stating that variable $v$ has type $T$. These elements come into scope when a new variable (`let` variable or formal parameter) is introduced.
- A constraint $\rho \subseteq R$ stating that region parameter $\rho$ 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 Section 4.1.5

for more details).

- An integer variable $i$. These elements come into scope when we are evaluating an array type or new array expression.

Formally, we write that as follows:

$$\Gamma ::= (v, T) \mid \rho \subseteq R \mid i \mid \Gamma \cup \Gamma$$

### 4.1.1 Programs and Classes

**Valid programs:** The judgment $\vdash \mathcal{P}$ means that program $\mathcal{P}$ is valid. The judgment holds if the classes of $\mathcal{P}$ are valid, and the main expression of $\mathcal{P}$ is well typed with type $T$ and effect $E$ in the empty environment:

$$\boxed{\vdash \mathcal{P}} \quad \textsc{Program} \quad \frac{\forall \mathcal{C}.(\vdash \mathcal{C}) \quad \emptyset \vdash e : T, E}{\vdash \mathcal{C}^*\ e}$$

**Valid classes:** The judgment $\vdash \mathcal{C}$ means that class definition $\mathcal{C}$ is valid. The judgment holds if the fields and methods of $\mathcal{C}$ are valid. We check these facts in the environment that binds `this` to the enclosing class.

$$\boxed{\vdash \mathcal{C}} \quad \textsc{Class} \quad \frac{\Gamma = (\texttt{this}, C\texttt{<}\rho\texttt{>}) \quad \forall F.(\Gamma \vdash F) \quad \forall M.(\Gamma \vdash M)}{\vdash \texttt{class}\ C\texttt{<}\rho\texttt{>}\ \{\ F^*\ M^*\ \}}$$

**Valid fields:** The judgment $\Gamma \vdash F$ means that field $F$ is valid in environment $\Gamma$. The judgment holds if the type and region of $F$ are valid in $\Gamma$.

$$\boxed{\vdash F} \quad \textsc{Field} \quad \frac{\Gamma \vdash T \quad \Gamma \vdash R}{\Gamma \vdash T\ f\ \texttt{in}\ R}$$

**Valid methods:** The judgment $\Gamma \vdash M$ means that method $M$ is valid in environment $\Gamma$. The judgment holds if the method's return type, formal parameter type, and effect are valid in $\Gamma$; its body type-checks in $\Gamma \cup \{(x, T_x)\}$; and the body's type and effect are, respectively, a subtype of the declared return type and a

subeffect of the declared effect:

$$\boxed{\vdash M} \quad \textsc{Method} \quad \frac{\Gamma \vdash T_r, T_x, E \quad \Gamma' = \Gamma \cup \{(x, T_x)\} \quad \Gamma' \vdash e : T', E' \quad \Gamma' \vdash T' \preceq T_r \quad \Gamma' \vdash E' \subseteq E}{\Gamma \vdash T_r\ m(T_x\ x)\ E\ \{\ e\ \}}$$

**Valid commutativity annotations:** The judgment $\Gamma \vdash$ *comm* means that commutativity annotation *comm* is valid in environment $\Gamma$. The judgment holds if the methods named in the annotation are both defined methods of the enclosing class.

$$\boxed{\Gamma \vdash \textit{comm}} \quad \textsc{Comm} \quad \frac{(\texttt{this}, C\texttt{<}\rho\texttt{>}) \in \Gamma \quad \mathsf{defined}(C\,.\,m) \quad \mathsf{defined}(C\,.\,m')}{\Gamma \vdash m\ \texttt{commuteswith}\ m'}$$

Here $\mathsf{defined}(C\,.\,m)$ means that a method named $m$ is defined in class $C$.

### 4.1.2 RPLs

**Valid RPLs:** The judgment $\Gamma \vdash R$ says that RPL $R$ is valid in environment $\Gamma$. The rules for making the judgment formally define RPLs, as described informally in Section 3.2. First, an RPL is valid if it is `Root`, a variable $v$ in scope, or a parameter $\rho$ in scope:

$$\boxed{\Gamma \vdash R} \quad \text{RPL-}\textsc{Root} \quad \frac{}{\Gamma \vdash \texttt{Root}} \qquad \text{RPL-}\textsc{Var} \quad \frac{(v, C\texttt{<}R\texttt{>}) \in \Gamma}{\Gamma \vdash v}$$

$$\text{RPL-}\textsc{Param-1} \quad \frac{(\texttt{this}, C\texttt{<}\rho\texttt{>}) \in \Gamma}{\Gamma \vdash \rho} \qquad \text{RPL-}\textsc{Param-2} \quad \frac{\rho \subseteq R \in \Gamma}{\Gamma \vdash \rho}$$

Second, a declared region name $r$, and array index element `[`$i$`]`, or a star `*` may be appended to a valid RPL to make another valid RPL:

$$\boxed{\Gamma \vdash R} \quad \text{RPL-}\textsc{Index} \quad \frac{\Gamma \vdash R \quad i \in \Gamma}{\Gamma \vdash R\texttt{:[}i\texttt{]}} \qquad \text{RPL-}\textsc{Name} \quad \frac{\Gamma \vdash R \quad \texttt{region}\ r \in \mathcal{P}}{\Gamma \vdash R\texttt{:}r} \qquad \text{RPL-}\textsc{Star} \quad \frac{\Gamma \vdash R}{\Gamma \vdash R\texttt{:*}}$$

The basic idea here is that all the rules but RPL-STAR define fully specified RPLs $R_f$; and adding a star to an RPL refers to all RPLs under that one (see the next subsection).

**Nesting Relation:** The judgment $\Gamma \vdash R \preceq R'$ says that $R$ is nested under $R'$; it establishes the tree structure of RPLs. More information on the tree structure is given in Section 4.3.1.

We formally define the nesting relation as follows. First, the relation is reflexive and transitive:

$$\boxed{\Gamma \vdash R \preceq R'} \quad \textsc{Under-Reflexive}\ \frac{}{\Gamma \vdash R \preceq R} \qquad \textsc{Under-Transitive}\ \frac{\Gamma \vdash R \preceq R' \quad \Gamma \vdash R' \preceq R''}{\Gamma \vdash R \preceq R''}$$

Next, every RPL is nested under `Root`:

$$\boxed{\Gamma \vdash R \preceq R'} \quad \textsc{Under-Root}\ \frac{}{\Gamma \vdash R \preceq \texttt{Root}}$$

If variable $v$ is bound to type $C\texttt{<}R\texttt{>}$, then $v$ is nested under $R$:

$$\boxed{\Gamma \vdash R \preceq R'} \quad \textsc{Under-Var}\ \frac{(v, C\texttt{<}R\texttt{>}) \in \Gamma}{\Gamma \vdash v \preceq R}$$

Adding $r$, `[`$i$`]`, or `*` to the end of an RPL preserves nesting:

$$\boxed{\Gamma \vdash R \preceq R'} \quad \textsc{Under-Name}\ \frac{\Gamma \vdash R \preceq R'}{\Gamma \vdash R\texttt{:}r \preceq R'} \qquad \textsc{Under-Index}\ \frac{\Gamma \vdash R \preceq R'}{\Gamma \vdash R\texttt{:[}i\texttt{]} \preceq R'}$$

$$\textsc{Under-Star}\ \frac{\Gamma \vdash R \preceq R'}{\Gamma \vdash R\texttt{:*} \preceq R'}$$

Finally, inclusion implies nesting:

$$\boxed{\Gamma \vdash R \preceq R'} \quad \textsc{Under-Include}\ \frac{\Gamma \vdash R \subseteq R'}{\Gamma \vdash R \preceq R'}$$

**Inclusion Relation:** The judgment $\Gamma \vdash R \subseteq R'$ says that $R$ is included in $R'$. That means that the set of dynamic regions represented by the static RPL $R$ is included in the set of dynamic regions represented by $R'$. More information on the set inclusion interpretation is given in Section 4.3.1. We formally define the

relationship on static RPLs as follows. First, the relationship is reflexive and transitive:

$$\boxed{\Gamma \vdash R \subseteq R'} \quad \textsc{Include-Reflexive} \ \frac{}{\Gamma \vdash R \subseteq R} \qquad \textsc{Include-Transitive} \ \frac{\Gamma \vdash R \subseteq R' \quad \Gamma \vdash R' \subseteq R''}{\Gamma \vdash R \subseteq R''}$$

Next, if $R$ is nested under $R'$, then $R$ is included in $R'\texttt{:*}$:

$$\boxed{\Gamma \vdash R \subseteq R'} \quad \textsc{Include-Star} \ \frac{\Gamma \vdash R \preceq R'}{\Gamma \vdash R \subseteq R'\texttt{:*}}$$

If $R$ is included in $R'$, then $R\texttt{:}r$ and $R\texttt{:[}i\texttt{]}$ are both included in $R'\texttt{:}r$:

$$\boxed{\Gamma \vdash R \subseteq R'} \quad \textsc{Include-Name} \ \frac{\Gamma \vdash R \subseteq R'}{\Gamma \vdash R\texttt{:}r \subseteq R'\texttt{:}r} \qquad \textsc{Include-Index} \ \frac{\Gamma \vdash R \subseteq R'}{\Gamma \vdash R\texttt{:[}i\texttt{]} \subseteq R'\texttt{:[}i\texttt{]}}$$

If the environment says that $\rho$ is included in $R$, then $\rho$ is in fact included in $R$.

$$\boxed{\Gamma \vdash R \subseteq R'} \quad \textsc{Include-Param} \ \frac{\rho \subseteq R \in \Gamma}{\Gamma \vdash \rho \subseteq R}$$

Finally, if $R$ is included in a fully specified RPL $R_f$, then $R_f$ is also included in $R$:

$$\boxed{\Gamma \vdash R \subseteq R'} \quad \textsc{Include-Full} \ \frac{\Gamma \vdash R \subseteq R_f}{\Gamma \vdash R_f \subseteq R}$$

### 4.1.3 Types

**Valid Types:** The judgment $\Gamma \vdash T$ says that type $T$ is valid in environment $\Gamma$. If $C$ is a defined class and $R$ is a valid RPL, then $C\texttt{<}R\texttt{>}$ is a valid type:

$$\boxed{\Gamma \vdash T} \quad \textsc{Type-Class} \ \frac{\mathsf{defined}(C) \quad \Gamma \vdash R}{\Gamma \vdash C\texttt{<}R\texttt{>}}$$

To check an array type $T$`[]<`$R$`>#`$i$ in $\Gamma$, we check $T$ and $R$ in the environment plus $i$:

$$\boxed{\Gamma \vdash T} \quad \textsc{Type-Array} \quad \frac{\Gamma \cup \{i\} \vdash T, R}{\Gamma \vdash T\texttt{[]<}R\texttt{>\#}i}$$

**Subtyping:** The judgment $\Gamma \vdash T \preceq T'$ says that $T$ is a subtype of $T'$. If $R$ is included in $R'$, then $C$`<`$R$`>` is a subtype of $C$`<`$R'$`>`:

$$\boxed{\Gamma \vdash T \preceq T'} \quad \textsc{Subtype-Class} \quad \frac{\Gamma \vdash R \subseteq R'}{\Gamma \vdash C\texttt{<}R\texttt{>} \preceq C\texttt{<}R'\texttt{>}}$$

If $R$ is included in $R'$ and $T$ equals $T'$ up to differing names of integer variables, then $T$`[]<`$R$`>#`$i$ is a subtype of $T'$`[]<`$R'$`>#`$i'$:

$$\boxed{\Gamma \vdash T \preceq T'} \quad \textsc{Subtype-Array} \quad \frac{\Gamma \cup \{i\} \vdash R \subseteq R'[i' \leftarrow i] \quad T \equiv T'}{\Gamma \vdash T\texttt{[]<}R\texttt{>\#}i \preceq T'\texttt{[]<}R'\texttt{>\#}i'}$$

Here $\equiv$ means identity up to the names of integer variables $i$. It follows from the reflexivity and transitivity of the inclusion relation on RPLs (Section 4.1.2) that subtyping is reflexive and transitive.

### 4.1.4 Effects

**Valid Effects:** The judgment $\Gamma \vdash E$ means that $E$ is a valid effect in environment $\Gamma$. The empty effect is valid:

$$\boxed{\Gamma \vdash E} \quad \textsc{Effect-Empty} \quad \frac{}{\Gamma \vdash \emptyset}$$

If $R$ is a valid region, then `reads` $R$ and `writes` $R$ are both valid effects:

$$\boxed{\Gamma \vdash E} \quad \textsc{Effect-Reads} \quad \frac{\Gamma \vdash R}{\Gamma \vdash \texttt{reads}\, R} \qquad \textsc{Effect-Writes} \quad \frac{\Gamma \vdash R}{\Gamma \vdash \texttt{writes}\, R}$$

If $m$ is a defined method of $C$ and $E$ is a valid effect, then `invokes` $C$`.`$m$ `with` $E$ is a valid effect:

$$\boxed{\Gamma \vdash E} \quad \textsc{Effect-Invokes} \quad \frac{\mathsf{defined}(C.m) \quad \Gamma \vdash E}{\Gamma \vdash \texttt{invokes}\, C.m\, \texttt{with}\, E}$$

Finally, the union of two valid effects is a valid effect:

$$\boxed{\Gamma \vdash E} \quad \textsc{Effect-Union} \quad \frac{\Gamma \vdash E \quad \Gamma \vdash E'}{\Gamma \vdash E \cup E'}$$

**Subeffects:** The judgment $\Gamma \vdash E \subseteq E'$ says that $E$ is a *subeffect* of $E'$. Intuitively, that means $E'$ contains all the effects of $E$, i.e., we can use $E'$ as a (possibly conservative) summary of $E$. The subeffect relation is reflexive and transitive:

$$\boxed{\Gamma \vdash E \subseteq E'} \quad \textsc{SE-Reflexive} \quad \frac{}{\Gamma \vdash E \subseteq E} \qquad \textsc{SE-Transitive} \quad \frac{\Gamma \vdash E \subseteq E' \quad \Gamma \vdash E' \subseteq E''}{\Gamma \vdash E \subseteq E''}$$

The empty effect is a trivial subeffect of every effect:

$$\boxed{\Gamma \vdash E \subseteq E'} \quad \textsc{SE-Empty} \quad \frac{}{\Gamma \vdash \emptyset \subseteq E}$$

If $R$ is included in $R'$, then $\texttt{reads}\ R$ is a subeffect of $\texttt{reads}\ R'$, and similarly for $\texttt{writes}\ R$ and $\texttt{writes}\ R'$:

$$\boxed{\Gamma \vdash E \subseteq E'} \quad \textsc{SE-Reads} \quad \frac{\Gamma \vdash R \subseteq R'}{\Gamma \vdash \texttt{reads}\ R \subseteq \texttt{reads}\ R'} \qquad \textsc{SE-Writes} \quad \frac{\Gamma \vdash R \subseteq R'}{\Gamma \vdash \texttt{writes}\ R \subseteq \texttt{writes}\ R'}$$

Also, writes cover reads:

$$\boxed{\Gamma \vdash E \subseteq E'} \quad \textsc{SE-Reads-Writes} \quad \frac{\Gamma \vdash R \subseteq R'}{\Gamma \vdash \texttt{reads}\ R \subseteq \texttt{writes}\ R'}$$

Next we have two rules for the invocation effect. First, if $E'$ covers $E$, then an invocation of some method with $E'$ covers an invocation of the same method with $E$:

$$\boxed{\Gamma \vdash E \subseteq E'} \quad \textsc{SE-Invokes-1} \quad \frac{\Gamma \vdash E \subseteq E'}{\Gamma \vdash \texttt{invokes}\ C.m\ \texttt{with}\ E \subseteq \texttt{invokes}\ C.m\ \texttt{with}\ E'}$$

Second, we can conservatively summarize the effect $\texttt{invokes}\, C\texttt{.}\, m\, \texttt{with}\, E$ as just $E$:

$$\text{SE-INVOKES-2}\ \frac{}{\Gamma \vdash \texttt{invokes}\, C\texttt{.}\, m\, \texttt{with}\, E \subseteq E}$$

Finally, we have the obvious rules for subeffects of unions and unions of subeffects:

$$\boxed{\Gamma \vdash E \subseteq E'} \quad \text{SE-UNION-1}\ \frac{\Gamma \vdash E \subseteq E'}{\Gamma \vdash E \subseteq E' \cup E''} \qquad \text{SE-UNION-2}\ \frac{\Gamma \vdash E' \subseteq E \quad \Gamma \vdash E'' \subseteq E}{\Gamma \vdash E' \cup E'' \subseteq E}$$

### 4.1.5 Typing Expressions

Every well-typed expression has a *type* and an *effect*. The type is the familiar static type from Java, plus the region information in the class types. The effect summarizes the heap effects that may occur when the expression is evaluated. The following rules govern the typing of expressions.

**Let expressions:** To type $\texttt{let}\ x\texttt{=}e\ \texttt{in}\ e'$, we type $e$, bind $x$ to the type of $e$, and type $e'$. If $x$ appears in the type or effect of $e'$, we weaken it to $R\texttt{:*}$ to generate a type and effect for the whole expression that are valid in the outer scope but still cover the actual type and effect. The type of the expression is the type of $e'$, and the effect is the union of the effects of evaluating $e$ and $e'$:

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

**Field access:** To type field access, we look in the environment to get the type of the enclosing class, check that everything is well formed according to the class definition, and report the read effect on the declared region of the field:

$$\text{FIELD-ACCESS}\ \frac{T\ f\ \texttt{in}\ R_f \in \mathsf{def}(C) \quad (\texttt{this}, C\texttt{<}\mathsf{param}(C)\texttt{>}) \in \Gamma}{\Gamma \vdash \texttt{this}\texttt{.}\, f : T, \texttt{reads}\, R_f}$$

Here $\mathsf{param}(C)$ means the region parameter $\rho$ in the definition of class $C$. Note that there is no need to substitute for the class formal parameter in the resulting type or effect, because in this simple language we allow field access only through `this`.

**Field assignment:** Field assignment is similar, except that we enforce subtyping for the assignment, and we report a write effect:

$$\textsc{Field-Assign}\quad \frac{(\texttt{this}, C\texttt{<}\mathsf{param}(C)\texttt{>}) \in \Gamma \quad (v, T) \in \Gamma \quad T'\ f\ \texttt{in}\ R_f \in \mathsf{def}(C) \quad \Gamma \vdash T \preceq T'}{\Gamma \vdash \texttt{this.}f\texttt{=}z : T, \texttt{writes}\ R_f}$$

**Array access and assignment:** Array access and assignment are almost identical to field access and assignment, except that we substitute the index for the index variable in computing the type and effect:

$$\textsc{Array-Access}\quad \frac{(v, T\texttt{[]<}R\texttt{>\#}i) \in \Gamma}{\Gamma \vdash v\texttt{[}n\texttt{]} : T[i \leftarrow n], \texttt{reads}\ R[i \leftarrow n]}$$

$$\textsc{Array-Assign}\quad \frac{\{(v, T\texttt{[]<}R\texttt{>\#}i), (v', T')\} \subseteq \Gamma \quad \Gamma \vdash T' \preceq T[i \leftarrow n]}{\Gamma \vdash v\texttt{[}n\texttt{]=}v' : T', \texttt{writes}\ R[i \leftarrow n]}$$

**Method invocation:** To type method invocation $v\texttt{.}m\texttt{(}v'\texttt{)}$, we find the class $C$ bound to $v$ in the environment and find the definition of $m$ in $C$ (if $m$ is undefined, then the typing fails). Next we need to translate the declared types and effect of $m$ to the current environment. We do this with two substitutions. The substitution $\sigma$ takes `this` to $v$ and the declared parameter of $C$ to the RPL $R$ appearing in the type of $v$. We use $\sigma$ to translate the declared return type and effect of $m$, reporting an invocation effect with that type and underlying effect. The substitution $\sigma'$ is similar, but it takes $\mathsf{param}(C)$ to a fresh parameter $\rho$, called the *capture parameter*. We use $\sigma'$ to translate the type of $m$'s formal argument $x$ to the current environment, and we check that the type of $v'$ is a subtype of this translated type, in an environment in which the capture parameter is contained in $R$.

$$\textsc{Invoke}\quad \frac{\begin{array}{c}\{(v, C\texttt{<}R\texttt{>}), (v', T)\} \subseteq \Gamma \quad T_r\ m(T_x\ x)\ E\ \{\ e\ \}\ \in \mathsf{def}(C) \quad \sigma = \{\texttt{this} \mapsto v, \mathsf{param}(C) \mapsto R\} \\ \sigma' = \{\texttt{this} \mapsto v, \mathsf{param}(C) \mapsto \rho\} \quad \Gamma \cup \{\rho \subseteq R\} \vdash T \preceq \sigma'(T_x)\end{array}}{\Gamma \vdash v\texttt{.}m\texttt{(}v'\texttt{)} : \sigma(T_r), \texttt{invokes}\ C\texttt{.}m\ \texttt{with}\ \sigma(E)}$$

The capture parameter represents that the actual type of $v$ at runtime is $C\texttt{<}\rho\texttt{>}$, where all we know about $\rho$ is $\rho \subseteq R$. This technique is similar to how Java handles the capture of a generic wildcard [57]. Note that simply using $\sigma$ to translate the formal parameter type would not be sound. To see why, consider the example

```
1  region r
2  class C<P> {
3    C<P> f in Root;
4    C<Root:*> weaken(C<P> x) { x }
5    C<P> assign(C<P> x) writes Root { f = x }
6    C<Root:*> unsound(C<Root:*> x) writes Root {
7      // Inferred type of x1 is C<Root:*>
8      let x1 = weaken(new C<Root>) in
9        // Whoops!  Assigning C<Root:r> to C<Root>
10       x1.assign(new C<Root:r>)
11   }
12 }
```

Figure 4.2: Example showing why we must capture partially specified RPLs

shown in Figure 4.2:

Without the capture parameter, the real type `C<Root>` of the formal parameter `x` of `assign` is weakened too far to `C<Root:*>` in line 8, and the assignment of `C<Root:r>` to `x` is (erroneously) allowed. With the capture parameter, we can see that we are trying to assign `C<Root:r>` to `C<P>`, where all we know about `P` is `P` $\subseteq$ `Root:*`. This unsound assignment is disallowed by the rule given above.

**Variables and new objects:** The rules for typing $v$ and `new` $T$ are obvious:

$$\textsc{Var}\ \frac{(v,T)\in\Gamma}{\Gamma\vdash v:T,\emptyset} \qquad \textsc{New-Class}\ \frac{\Gamma\vdash C\texttt{<}R\texttt{>}}{\Gamma\vdash \texttt{new}\ C\texttt{<}R\texttt{>}:C\texttt{<}R\texttt{>},\emptyset}$$

$$\textsc{New-Array}\ \frac{\Gamma\vdash T\texttt{[]<}R\texttt{>\#}i}{\Gamma\vdash \texttt{new}\ \ T\texttt{[}n\texttt{]<}R\texttt{>\#}i:T\texttt{[]<}R\texttt{>\#}i,\emptyset}$$

## 4.2 Dynamic Semantics

### 4.2.1 Execution State

The syntax for entities used in the dynamic semantics is shown in Figure 4.3. At runtime, we have dynamic RPLs ($dR$), dynamic types ($dT$) and dynamic effects ($dE$), corresponding to static RPLs ($R$), types ($T$) and effects ($E$) respectively. Dynamic RPLs 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 [37]. 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 partial function taking object references

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

Figure 4.3: Dynamic syntax of Core DPJ

to objects; and (2) a dynamic environment $\Sigma$, which is a set of elements of the form $(v, o)$ (variable $v$ is bound to reference $o$) or $(\rho, dR)$ (region parameter $\rho$ is bound to RPL $dR$). The dynamic environment $\Sigma$ defines a natural substitution on RPLs, where we replace the variables with references and the region parameters with RPLs as specified in the environment. We denote the application of this substitution to RPL $R$ as $\Sigma(R)$. We extend this notation to types and effects in the obvious way. Notice that we get the syntax of Figure 4.3 by applying the substitution $\Sigma$ to the syntax of Figure 4.1.

An object is a function taking field names (for class objects) or nonnegative integers (for arrays) to object references. Every object reference $o \in \mathsf{Dom}(H)$ has a type, and we write $H \vdash o : dT$ to mean that the reference $o$ has type $dT$ with respect to heap $H$. An object reference $o$ either maps to an object, in which case $H(o)$ is an object corresponding to the type of $o$, or it does not map to any object, in which case $H(o)$ is undefined. In the latter case, we say that $o$ is a *null reference*. We define null references this way because we need to track the actual types of different references to establish soundness. In an actual implementation that does not do this tracking, we can use a single value $\mathsf{null}$ for every null reference. Dereferencing a null reference at runtime causes the execution to fail by getting stuck. We do not explicitly model null dereference errors or exceptions.

### 4.2.2 Evaluating Programs

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

$$(e, \Sigma, H) \rightarrow (o, H', dE),$$

where $e$ is an expression to evaluate, $\Sigma$ 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. The program $\mathcal{P}$ is an implicit parameter that we omit for conciseness and readability. Notice that $\Sigma$ does not appear on the right-hand side, because we do not need to retain the dynamic environment as global state. A program evaluates to reference $o$ with heap $H$ and effect $dE$ if its main expression is $e$ and $(e, \emptyset, \emptyset) \rightarrow (o, H, dE)$.

**Let expressions:** To evaluate `let` $x$`=`$e$ `in` $e'$, we evaluate $e$ to $o$, bind $o$ to $x$, and evaluate $e'$, updating the heap and collecting effects as we go:

$$\textsc{Dyn-Let}\quad \frac{(e,\Sigma,H)\to(o,H',dE)\quad (e',\Sigma\cup\{(x,o)\},H')\to(o',H'',dE')}{(\texttt{let}\ x\texttt{=}e\ \texttt{in}\ e',\Sigma,H)\to(o',H'',dE\cup dE')}$$

**Field access:** To evaluate field access, we look up the object bound to `this` in $\Sigma$, and we read its field $f$, recording the read of the declared region after translating it to the dynamic context:

$$\textsc{Dyn-Field-Access}\quad \frac{(\texttt{this},o)\in\Sigma\quad H\vdash o:C\texttt{<}dR\texttt{>}\quad T\ f\ \texttt{in}\ R_f\in\mathsf{def}(C)}{(\texttt{this.}f,\Sigma,H)\to(H(o)(f),H,\texttt{reads}\ \Sigma(R_f))}$$

**Field assignment:** Field assignment is similar, except that we update the heap and record a write effect:

$$\textsc{Dyn-Field-Assign}\quad \frac{\{(\texttt{this},o),(v,o')\}\subseteq\Sigma\quad H\vdash o:C\texttt{<}dR\texttt{>}\quad T\ f\ \texttt{in}\ R_f\in\mathsf{def}(C)}{(\texttt{this.}f\texttt{=}v,\Sigma,H)\to(o,H[o\mapsto H(o)[f\mapsto o']],\texttt{writes}\ \Sigma(R_f))}$$

$f : A \to B$ is a function, then $f[x \mapsto y]$ denotes the function $f' : A \cup \{x\} \to B \cup \{y\}$ defined by $f'(a) = f(a)$ if $a \neq x$ and $f'(x) = y$.

**Array access and assignment:** Array access and assignment are nearly identical to field access and assignment, except that we use the array index $n$ to access the array, and we substitute $n$ for the index variable $i$ in computing the region:

$$\textsc{Dyn-Array-Access}\quad \frac{(v,o)\in\Sigma\quad H\vdash o:dT\texttt{[]<}dR\texttt{>\#}i}{(v\texttt{[}n\texttt{]},\Sigma,H)\to(H(o)(n),H,\texttt{reads}\ dR[i\leftarrow n])}$$

$$\textsc{Dyn-Array-Assign}\quad \frac{\{(v,o),(v',o')\}\subseteq\Sigma\quad H\vdash o:dT\texttt{[]<}dR\texttt{>\#}i}{(v\texttt{[}n\texttt{]=}v',\Sigma,H)\to(o',H[o\mapsto H(o)[n\mapsto o']],\texttt{writes}\ dR[i\leftarrow n])}$$

**Method invocation:** To evaluate method invocation $v\texttt{.}m\texttt{(}v'\texttt{)}$, we find the bindings of $v$ and $v'$ in $\Sigma$, create

a new environment for the invocation, evaluate the method body, and record the invocation effect:

$$\textsc{Dyn-Invoke}\quad \frac{H \vdash o : C\texttt{<}dR\texttt{>} \quad T_r\ m(T_x\ x)\ E\ \{\ e\ \}\ \in \mathsf{def}(C) \qquad (e, \{(\texttt{this}, o), (\mathsf{param}(C), dR), (x, o')\}, H) \rightarrow (o'', H', dE)}{(v\texttt{.}m\texttt{(}v'\texttt{)}, \{(v, o), (v', o')\} \cup \Sigma, H) \rightarrow (o'', H', \texttt{invokes}\ C\texttt{.}m\ \texttt{with}\ dE)}$$

**Variables:** To evaluate a variable expression, we just get the reference out of $\Sigma$:

$$\textsc{Dyn-Var}\quad \frac{(v, o) \in \Sigma}{(v, \Sigma, H) \rightarrow (o, H, \emptyset)}$$

**New objects:** To evaluate $\texttt{new}\ T$, we translate $T$ to $dT$ using $\sigma$, after eliminating any $\texttt{*}$ from $T$, e.g., $\texttt{new}\ C\texttt{<Root:*>}$ is the same as $\texttt{new}\ C\texttt{<Root>}$; this rule ensures that all object fields are allocated in fully specified RPLs. We then create a fresh object and a fresh reference of the appropriate type.

$$\textsc{Dyn-New-Class}\quad \frac{o \notin \mathsf{Dom}(H) \quad H' = H \cup \{(o, \mathsf{new}(C))\} \quad H' \vdash o : C\texttt{<}\Sigma(R[\texttt{:*} \leftarrow \epsilon])\texttt{>}}{(\texttt{new}\ C\texttt{<}R\texttt{>}, \Sigma, H) \rightarrow (o, H', \emptyset)}$$

$$\textsc{Dyn-New-Array}\quad \frac{o \notin \mathsf{Dom}(H) \quad H' = H \cup \{(o, \mathsf{new}(T\texttt{[}n\texttt{]}))\} \quad H' \vdash o : \Sigma(T)\texttt{[]<}\Sigma(R[\texttt{:*} \leftarrow \epsilon])\texttt{>\#}i}{(\texttt{new}\ \ T\texttt{[}n\texttt{]<}R\texttt{>\#}i, \Sigma, H) \rightarrow (o, H', \emptyset)}$$

$\mathsf{new}(C)$ is the function taking each field of class $C$ with type $T$ to a null reference of type $\Sigma(T)$, and $\mathsf{new}(T\texttt{[}n\texttt{]})$ is the function taking each $n' \in [0, n-1]$ to a null reference of type $\Sigma(T)$.

### 4.2.3 Judgments for Dynamic RPLs, Types, and Effects

To state and prove the preservation result (Section 4.2.4), we need to establish judgments for dynamic RPLs, types, and effects corresponding to the static judgments defined in Sections 4.1.2 through 4.1.4. As before, $\subseteq$ and $\preceq$ are reflexive and transitive.

**Dynamic RPLs:** The rules for valid dynamic RPLs are similar to the rules for static RPLs (Section 4.1.2), except that we do not allow region parameters in dynamic RPLs; and instead of the rule DYN-RPL-VAR,

we have the rule DYN-RPL-REF, which requires a valid reference:

$$\text{DYN-RPL-ROOT}\ \frac{}{H \vdash \texttt{Root}} \qquad \text{DYN-RPL-REF}\ \frac{H \vdash o : T}{H \vdash o} \qquad \text{DYN-RPL-INDEX}\ \frac{H \vdash dR}{H \vdash dR\texttt{:[}i|n\texttt{]}}$$

$$\text{DYN-RPL-NAME}\ \frac{H \vdash dR \quad \texttt{region}\, r \in \mathcal{P}}{H \vdash dR\texttt{:}r} \qquad \text{DYN-RPL-STAR}\ \frac{H \vdash dR}{H \vdash dR\texttt{:*}}$$

In rule DYN-RPL-INDEX, we write $\texttt{[}i|n\texttt{]}$ to indicate that an index variable $i$ or a numerical index $n$ can appear in that position. Index variables $i$ appear in the RPLs $R$ of dynamic array types $T\texttt{[]<}R\texttt{>\#}i$; they are always substituted away when the array is accessed through an index $n$, via rules DYN-ARRAY-ACCESS and DYN-ARRAY-ASSIGN.

The nesting relationship for dynamic RPLs is similar to the corresponding relationship for static RPLs (Section 4.1.2), except that instead of the rule DYN-UNDER-VAR, we have the rule DYN-UNDER-REF, which says that an object reference is under the RPL of its type:

$$\text{DYN-UNDER-ROOT}\ \frac{}{H \vdash dR \preceq \texttt{Root}} \qquad \text{DYN-UNDER-REF}\ \frac{H \vdash o : C\texttt{<}dR\texttt{>}}{H \vdash o \preceq dR} \qquad \text{DYN-UNDER-NAME}\ \frac{H \vdash dR \preceq dR'}{H \vdash dR\texttt{:}r \preceq dR'}$$

$$\text{DYN-UNDER-INDEX}\ \frac{H \vdash dR \preceq dR'}{H \vdash dR\texttt{:[}i\,|\,n\texttt{]} \preceq dR'\texttt{:[}i\,|\,n\texttt{]}} \qquad \text{DYN-UNDER-STAR}\ \frac{H \vdash dR \preceq dR'}{H \vdash dR\texttt{:*} \preceq dR'} \qquad \text{DYN-UNDER-INCLUDE}\ \frac{H \vdash dR \subseteq dR'}{H \vdash dR \preceq dR'}$$

The inclusion relationship for dynamic RPLs is similar to the rules for static RPLs (Section 4.1.2), except that we do not have any region parameters:

$$\text{DYN-INCLUDE-STAR}\ \frac{H \vdash dR \preceq dR'}{H \vdash dR \subseteq dR'\texttt{:*}} \qquad \text{DYN-INCLUDE-NAME}\ \frac{H \vdash dR \subseteq dR'}{H \vdash dR\texttt{:}r \subseteq dR'\texttt{:}r} \qquad \text{DYN-INCLUDE-INDEX}\ \frac{H \vdash dR \subseteq dR'}{H \vdash dR\texttt{:[}i\,|\,n\texttt{]} \subseteq dR'\texttt{:[}i\,|\,n\texttt{]}}$$

**Dynamic Types and Effects:** The rules for dynamic types and effects are nearly identical to their static counterparts. Instead of writing out all the rules, which would be tedious and not all that enlightening, we describe how to generate them via simple substitution from the rules given in Sections 4.1.3 and 4.1.4. For

every rule given in those sections, do the following:

1. Append DYN- to the front of the name.
2. Replace $\Gamma$ with $H$ and $[i]$ with $[n]$.
3. Replace $R$ with $dR$, $T$ with $dT$, and $E$ with $dE$.

Applying this transformation to all the rules in Sections 4.1.3 and 4.1.4 yields the rules for valid dynamic types, dynamic subtypes, valid dynamic effects, and subeffects. For example, here are the rules for valid types and subtypes, generated via the substitution above from the rules stated in Section 4.1.3.

$$\textsc{Dyn-Type-Class}\ \frac{\mathsf{defined}(C) \quad H \vdash dR}{H \vdash C\texttt{<}dR\texttt{>}} \qquad \textsc{Dyn-Subtype-Class}\ \frac{H \vdash dR \subseteq dR'}{H \vdash C\texttt{<}dR\texttt{>} \preceq C\texttt{<}dR'\texttt{>}}$$

The rest of the rules are similar.

### 4.2.4 Preservation of Type and Effect

In this section we show that the static types and effects bound the dynamic types and effects.

**Valid environments:** We first define the concept of a valid environment. An environment $\Gamma$ is valid if its variables are bound to valid types and its parameters are constrained to be under valid RPLs:

$$\textsc{Env}\ \frac{\forall (v, T) \in \Gamma.\Gamma \vdash T \quad \forall \rho \subseteq R \in \Gamma.\Gamma \vdash R}{\vdash \Gamma}$$

Technically, a valid environment should also be well-defined, in the sense that every variable has at most one binding. We omit this requirement from the definition of valid environments, because it obviously holds by the way that environments are constructed in the typing rules.

Next we have a lemma showing that, for a well-typed program, typing an expression in a valid environment yields a valid type and a valid effect:

**Lemma 4.2.1.** *For a well-typed program, let $\Gamma$ be a valid environment, and let $e$ be an expression such that $\Gamma \vdash e : T, E$. Then $\Gamma \vdash T$ and $\Gamma \vdash E$.*

*Proof.* Use induction on the height of the derivation. We can prove the claim directly in the following cases:

VAR, NEW-CLASS, and NEW-ARRAY: Obvious.

FIELD-ACCESS: $\Gamma \vdash T$ and $\Gamma \vdash R_f$ by rule Field. $\Gamma \vdash \texttt{reads}\ R_f$ by rule EFFECT-READS.

FIELD-ASSIGN: $\Gamma \vdash T$ because $\vdash \Gamma$ and $(v, T) \in \Gamma$. $\Gamma \vdash \texttt{writes}\ R_f$ by rules FIELD and EFFECT-WRITES.

ARRAY-ACCESS: Similar to FIELD-ACCESS.

ARRAY-ASSIGN: Similar to FIELD-ASSIGN.

INVOKE: By rule METHOD, we have $\{(\texttt{this}, C\texttt{<}\mathsf{param}(C)\texttt{>})\} \vdash T_r, E_m$. Because $\vdash \Gamma$ and $(v, C\texttt{<}R\texttt{>}) \in \Gamma$, we have $\Gamma \vdash R$. Therefore we have $\Gamma \vdash \sigma(T_r), \sigma(E_m)$, because it is clear from the rules in Section 4.1.2 that a valid RPL results when we replace $\rho$ with a valid RPL in a valid RPL. The invocation effect is valid by rule EFFECT-INVOKES.

Now consider the inductive case:

LET: The claim is true for the first judgment on the top of the rule by the induction hypothesis (IH). Therefore $\Gamma \vdash C\texttt{<}R\texttt{>}$, so $\vdash \Gamma \cup \{(x, C\texttt{<}R\texttt{>})\}$, and the claim is also true for the second judgment on the top of the rule by the IH. Any RPL appearing in $T'$ or $E'$ must either not contain $x$ at all, or must consist of $x$ followed by a sequence of elements $r$ or `*`. Therefore, substituting $R\texttt{:*}$ for $x$ results in a valid RPL, so $\sigma(T')$ and $\sigma(E')$ are valid in $\Gamma$. Finally, $E \cup \sigma(E')$ is valid in $\Gamma$ by rule EFFECT-UNION. □

**Valid dynamic environments:** A valid dynamic environment is the dynamic analog of a valid static environment:

**Definition 4.2.2** (Valid dynamic environments)**.** *A dynamic environment $\Sigma$ is valid with respect to heap $H$ ($H \vdash \Sigma$) if the following hold:*

1. *For every binding $(v, o) \in \Sigma$, $H \vdash o : dT$.*
2. *For every binding $(\rho, dR) \in \Sigma$, $H \vdash dR$.*
3. *If $(\texttt{this}, o) \in \Sigma$, then $H \vdash o : C\texttt{<}dR\texttt{>}$, and $(\mathsf{param}(C), dR) \in \Sigma$.*

This definition says that the bindings are to valid references and RPLs, and that the actual region of the object bound to `this` is consistent with the binding for the class parameter specified in the environment. We can now define valid heaps:

**Definition 4.2.3** (Valid heaps)**.** *A heap $H$ is valid ($\vdash H$) if for each $o \in \mathsf{Dom}(H)$, one of the following holds:*

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

This definition says that every object reference is well typed with a valid type, and every field of every object and every cell of every array contains a reference with a valid type that is bounded by its static type, translated to the dynamic environment.

Next we define $H \vdash \Sigma \preceq \Gamma$ ("$\Sigma$ instantiates $\Gamma$ in $H$"):

**Definition 4.2.4** (Instantiation of static environments)**.** *A dynamic environment $\Sigma$ instantiates a static environment $\Gamma$ ($H \vdash \Sigma \preceq \Gamma$) if $\vdash \Gamma$, $\vdash H$, and $H \vdash \Sigma$; the same variables appear in $\mathsf{Dom}(\Gamma)$ as in $\mathsf{Dom}(\Sigma)$; and for each pair $(v, T) \in \Gamma$ and $(v, o) \in \Sigma$, $H \vdash v : dT$ and $H \vdash dT \preceq \Sigma(T)$.*

This definition specifies a correspondence between static typing environments and dynamic execution environments, such that we can use the typing in the static environment to draw sound inferences about execution in the dynamic environment. Next we need some standard *substitution lemmas*, which say that under the correspondence established above, judgments about static RPLs, types, and effects carry over to their dynamic translations:

**Lemma 4.2.5.** *If $H \vdash \Sigma \preceq \Gamma$ and $\Gamma \vdash R$, then $H \vdash \Sigma(R)$; and similarly for types $T$ and effects $E$.*

*Proof.* Use induction on the height of the derivation $\Gamma \vdash R$. In the base case, we used one of rules RPL-ROOT, RPL-VAR, RPL-PARAM-1, or RPL-PARAM-2. If we used RPL-ROOT, then $R = \Sigma(R) = \texttt{Root}$, and the result follows by DYN-RPL-ROOT. If we used RPL-VAR, then by Definition 4.2.4, $\Sigma$ substitutes a valid reference for $v$, so we can use DYN-RPL-REF to establish the result. If we used an RPL-PARAM rule, then $R = P$, and again by Definition 4.2.4, $\Sigma$ takes $\rho$ to a valid dynamic RPL.

In the inductive case, either (1) $R = R' : r$, $\Sigma(R) = \Sigma(R')\texttt{:}r$, and RPL-NAME is the last rule in the derivation; or (2) $R = R'\texttt{:[}i\texttt{]}$, $\Sigma(R) = \Sigma(R')\texttt{:[}i\texttt{]}$, and RPL-INDEX is the last rule in the derivation; or (3) $R = R'\texttt{:*}$, $\Sigma(R) = \Sigma(R')\texttt{:*}$, and RPL-STAR is the last rule. In any case, the IH gives us $H \vdash \Sigma(R')$,

and we can use DYN-RPL-NAME, DYN-RPL-INDEX, or DYN-RPL-STAR to complete the derivation of $H \vdash \Sigma(R)$.

The result for types and effects follows from the fact that the rules for valid types and effects are identical in the static and dynamic cases, up to substituting valid dynamic RPLs for valid static RPLs. □

**Lemma 4.2.6.** *If $H \vdash \Sigma \preceq \Gamma$ and $\Gamma \vdash R \preceq R'$, then $H \vdash \Sigma(R) \preceq \Sigma(R')$; and similarly for $\Gamma \vdash R \subseteq R'$, $\Gamma \vdash T \preceq T'$ and $\Gamma \vdash E \subseteq E'$.*

*Proof.* It suffices to prove the results for $R \preceq R'$ and $R \subseteq R'$; the results for types and effects then follow from the exact correspondence (Section 4.2.3) between the static and dynamic rules for subtyping and subeffect. Use induction on the height of the derivation $\Gamma \vdash R \preceq R'$ or $\Gamma \vdash R \subseteq R'$.

For nesting, in the base case, we used one of rules UNDER-ROOT, DYN-UNDER-VAR, or reflexivity. In the case of UNDER-ROOT or reflexivity the claim is obvious. In the case of DYN-UNDER-VAR, from the rule we have $\Gamma \vdash v \preceq R$ and $(v, C\texttt{<}R\texttt{>}) \in \Gamma$; and by $H \vdash \Sigma \preceq \Gamma$, we have $(v, o) \in \Sigma$ with $H \vdash o : C\texttt{<}dR\texttt{>}$ and $H \vdash dR \subseteq \Sigma(R)$. The result follows by rules DYN-UNDER-REF and DYN-UNDER-INCLUDE. For inclusion, in the base case we used either reflexivity or rule INCLUDE-PARAM. For reflexivity, the claim is obvious, and for INCLUDE-PARAM, the claim follows from Definition 4.2.4.

Now consider the inductive case. For nesting, we used UNDER-NAME, UNDER-INDEX, UNDER-STAR, or UNDER-INCLUDE as the last rule in the derivation, and the claim follows straightforwardly from the IH and the corresponding rule for dynamic RPLs. Similarly for inclusion using INCLUDE-STAR, INCLUDE-NAME, INCLUDE-INDEX, or INCLUDE-FULL as the last rule. In the case of INCLUDE-FULL, we must have $dR = dR'$, so the result follows by the reflexivity of the inclusion relation. □

Finally, we state and prove the type and effect preservation result. Note that the initial heap $H$ is valid by Definition 4.2.4 and the assumption $H \vdash \Sigma \preceq \Gamma$.

**Theorem 4.2.7** (Preservation)**.** *For a well-typed program, if $\Gamma \vdash e : T, E$ and $H \vdash \Sigma \preceq \Gamma$ and $(e, \Sigma, H) \rightarrow (o, H', dE)$, then (a) $\vdash H'$; and (b) $H' \vdash dT \preceq \Sigma(T)$, where $H' \vdash o : dT$; and (c) $H' \vdash dE$; and (d) $H' \vdash dE \subseteq \Sigma(E)$.*

*Proof.* The derivation of $(e, \Sigma, H) \rightarrow (o, H', dE)$ is by the rules given in Section 4.2.2. Consider each possibility for the last rule in the derivation. We can show the claim directly in the following cases:

DYN-VAR: (a) holds because the heap does not change. (b) holds because of rule Var, and by Definition 4.2.4. (c) and (d) trivially hold.

DYN-NEW: By rule New, we have $\Gamma \vdash C\texttt{<}R\texttt{>}$. Therefore $H' \vdash C\texttt{<}\Sigma(R)\texttt{>}$ by Lemma 4.2.5; and because omitting stars from a valid RPL yields a valid RPL by rule RPL-STAR, $H' \vdash C\texttt{<}\Sigma(\sigma(R))\texttt{>}$. Further, we are extending the heap with a valid reference, and we initialize all the object fields with null references of the correct type. This establishes (a). (b) holds because $H \vdash \Sigma(\sigma(R)) \subseteq \Sigma(R)$ by repeated applications of rules DYN-INCLUDE-STAR and DYN-INCLUDE-NAME. (c) and (d) trivially hold.

DYN-FIELD-ACCESS: (a) holds because the heap does not change. (b) holds because $\vdash H$. (c) holds because $\Gamma \vdash R_f$ by rule FIELD, and by Lemma 4.2.5. (d) holds by comparing the reported effect in rule FIELD-ACCESS with the actual effect in rule DYN-FIELD-ACCESS.

DYN-FIELD-ASSIGN: (b) holds by rule FIELD-ASSIGN and by Definition 4.2.4. (a) holds because rule FIELD-ASSIGN requires $\Gamma \vdash T \preceq T'$, and by the transitivity of subtyping. (c) holds because $\Gamma \vdash R_f$ by rule Field. (d) holds by comparing the reported effect in rule FIELD-ASSIGN with the actual effect in rule DYN-FIELD-ASSIGN.

DYN-ARRAY-ACCESS: Similar to DYN-FIELD-ACCESS.

DYN-ARRAY-ASSIGN: Similar to DYN-FIELD-ASSIGN.

Now consider the possibilities for the inductive case:

DYN-LET: First, apply the IH to the left-hand reduction on the top of rule LET. This yields $H' \vdash dT \preceq T$, where $dT$ is the dynamic type of $o$, and $T$ is the static type of $e$. That result implies $H' \vdash \Sigma \cup \{(x, o)\} \preceq \Gamma \cup T$, which allows us to apply the IH to the right-hand reduction on the top of LET. Now (a) and (c) hold by the IH and the correspondence between the top of rules LET and DYN-LET. Further, by the IH, (b) holds for the type of $e'$, so it also holds for the weaker type obtained by substituting $R\texttt{:*}$ for $x$ in rule LET. A similar argument for the effects establishes (d).

DYN-INVOKE: Let $\Sigma'$ be the dynamic environment we used to evaluate the method body $e$ in rule DYN-INVOKE, and let $\Gamma'$ be the environment we used to type $e$ in rule METHOD. We need to show $H \vdash \Sigma' \preceq \Gamma'$. The only hard part is showing $H \vdash dT' \preceq \Sigma'(T_x)$, where $H \vdash o' : dT'$; we do this as follows. By hypothesis, $H \vdash dT' \preceq \Sigma(T')$, where $\Sigma$ is the dynamic environment appearing on the bottom of rule DYN-INVOKE, and $T'$ is the type of variable $v'$ in the environment $\Gamma$ appearing on the bottom of rule INVOKE. Now construct the dynamic environment $\Sigma'' = \Sigma \cup \{(\rho, dR)\}$, where $\rho$ is the capture parameter

appearing in rule INVOKE, and $dR$ is the RPL in the type of $o$ as shown in rule DYN-INVOKE. Then $H \vdash \Sigma'' \preceq \Gamma \cup \{\rho \subseteq R\}$, so from rule INVOKE and by Lemma 4.2.6, we have $\Sigma''(T') \preceq \Sigma''(\sigma'(T_x))$. Because $\rho$ is a fresh parameter that does not appear in $T'$, on the LHS we have $\Sigma(T') = \Sigma''(T')$. And because $\sigma'$ takes $\mathsf{param}(C)$ to $\rho$ and $\Sigma''$ takes $\rho$ to $dR$, while $\Sigma'$ takes $\mathsf{param}(C)$ to $dR$, on the RHS we have $\Sigma''(\sigma'(T_x)) = \Sigma'(T_x)$. Putting all this together yields $H \vdash dT' \preceq \Sigma(T') \preceq \Sigma'(T_x)$, which is the result we wanted.

Now by the induction hypothesis, rule METHOD, and Lemma 4.2.6, we have (a) $\vdash H'$; (b) $H' \vdash dT'' \preceq \Sigma'(T_r)$, where $H' \vdash o'' : dT''$; (c) $H' \vdash dE$; and (d) $H' \vdash dE \subseteq \Sigma'(E)$, where $T_r$ is the return type of $m$ and $E$ is the declared effect of $m$. We just need to show $H' \vdash \Sigma'(T_r) \preceq \Sigma(\sigma(T_r))$ and $H' \vdash \Sigma'(E) \subseteq \Sigma(\sigma(E))$, where $\sigma$ is the substitution specified in rule INVOKE. Because neither $T_r$ nor $E$ contains the variable $x$ (see rule METHOD), the substitution $\Sigma'$ is effectively $\{(\texttt{this}, o), (\mathsf{param}(C), dR)\}$, while the substitution $\Sigma \circ \sigma$ is $\{(\texttt{this}, o), (\mathsf{param}(C), \Sigma(R))\}$ Further, because $H' \vdash \Sigma \preceq \Gamma$, we have $H' \vdash dR \subseteq \Sigma(R)$. Therefore the types and effects are the same up to substituting a covering RPL for $dR$ on the RHS, so the required subtyping and subeffect relations hold. □

## 4.3 Noninterference

In this section we establish the main soundness results of Core DPJ. First we explain the set interpretation of dynamic RPLs, which is essential to reasoning about disjointness. Then we define a *disjointness relation* on RPLs, and we show that effects on disjoint RPLs imply disjoint effects on the heap. Then we define *noninterference of effect*, which is the essential criterion for checking that two sections of code may be safely executed in parallel. Finally, we prove that if two expressions have noninterfering static effect summaries, then their order may be interchanged without affecting the final result.

### 4.3.1 Set Interpretation of Dynamic RPLs

In this section we explain the set interpretation of dynamic RPLs. A fully specified RPL $R_f$ names a region of the heap. Given a heap $H$ and a valid dynamic RPL $dR$, we define the *set of regions* associated with $dR$ as follows:

**Definition 4.3.1.** *Let $\vdash H$ and $H \vdash dR$. Then $S(dR, H)$ is defined as follows:*

1. $S(dR_f, H) = \{dR_f\}$.

2. $S(dR\texttt{:}r, H) = \{dR_f\texttt{:}r \mid dR_f \in S(dR, H)\}$.

3. $S(dR\texttt{:[}n\texttt{]}, H) = \{dR_f\texttt{:[}n\texttt{]} \mid dR_f \in S(dR, H)\}$.

4. $S(dR\texttt{:}\star, H) = \{dR_f \mid H \vdash dR_f \preceq dR\}$.

Intuitively, the definition says that a fully specified RPL names a single region; $dR\texttt{:}r$ appends $r$ to all the regions of $dR$; $dR\texttt{:[}n\texttt{]}$ appends $\texttt{[}n\texttt{]}$ to all the regions of $dR$; and $dR\texttt{:}\star$ names all the regions under $dR$.

Now we establish some essential properties of RPLs interpreted as sets. The first property says that the set of RPLs under $dR_f\texttt{:}r$ is distinct from the set of RPLs under $dR_f\texttt{:}r'$; and similarly for $dR_f\texttt{:[}n\texttt{]}$ and $dR_f\texttt{:[}n'\texttt{]}$. This establishes the tree structure of dynamic RPLs.

**Lemma 4.3.2.** *If $r \neq r'$, then $S(dR_f\texttt{:}r\texttt{:}\star, H) \cap S(dR_f\texttt{:}r'\texttt{:}\star, H) = \emptyset$. If $n \neq n'$, then $S(dR_f\texttt{:[}n\texttt{]:}\star, H) \cap S(dR_f\texttt{:[}n'\texttt{]:}\star, H) = \emptyset$.*

*Proof.* We prove the first statement; the proof of the second statement is identical, using $\texttt{[}n\texttt{]}$ instead of $r$. By part 4 of Definition 4.3.1, it suffices to show that if $H \vdash dR'_f \preceq dR_f\texttt{:}r$, then it is impossible to derive $H \vdash dR'_f \preceq dR_f\texttt{:}r'$. From the rules in Section 4.2.3, there are three ways to do the first derivation: (1) $dR'_f = dR_f\texttt{:}r$; or (2) $dR'_f = o$ and $H \vdash o : C\texttt{<}dR_f\texttt{:}r\texttt{>}$; or (3) $R'_f$ satisfies one of the first two possibilities with names $r$ appended. In the first case it is clear from the RPL syntax that we cannot have $H \vdash dR'_f \preceq dR_f\texttt{:}r'$. In the second case, since $o$ has only one type, we could only have $H \vdash dR'_f \preceq dR_f\texttt{:}r$ if $H \vdash dR_f\texttt{:}r \preceq dR_f\texttt{:}r'$. To get this we would have to apply rule UNDER-NAME repeatedly until we got to $o'$ such that $dR_f$ begins with $o'$ and $H \vdash o' \preceq dR_f\texttt{:}r'$. But this would mean that $o'$ would have to appear in its own type, which is impossible from the way we construct dynamic types (Section 4.2.2). Finally, in case (3), appending names $r$ would just require that we use rule DYN-UNDER-NAME until we got back to case (1) or (2). □

Next we establish that the RPL inclusion relation agrees with set inclusion:

**Lemma 4.3.3.** *If $H \vdash dR \subseteq dR'$, then $S(dR, H) \subseteq S(dR', H)$, where the second occurrence of $\subseteq$ represents set inclusion.*

*Proof.* Use induction on the height of the derivation $H \vdash dR \subseteq dR'$. In the base case, we used reflexivity, and the claim is obvious. Otherwise, the last rule was one of transitivity, DYN-INCLUDE-STAR, DYN-INCLUDE-NAME, or DYN-INCLUDE-INDEX. In the case of transitivity, the result follows by the IH together with the fact that set inclusion is transitive. In the case of DYN-INCLUDE-STAR, we need to show that if $H \vdash dR \preceq dR'$, then $S(dR, H) \subseteq S(dR' : *, H)$; but this follows by Definition 4.3.1 and the transitivity of the nesting relation. Finally, in the case of DYN-INCLUDE-NAME or DYN-INCLUDE-INDEX, the result follows directly from the IH and Definition 4.3.1. □

### 4.3.2 Disjointness

We define the disjointness relation for static RPLs ($\Gamma \vdash R \# R'$) as follows. First, we have "distinctions from the left": given two fully specified RPLs that start with the same elements then diverge at the last element, any RPL under the first is disjoint from any RPL under the second. These rules reflect the tree structure of RPLs:

$$\textsc{Disjoint-Left-Name}\quad \frac{r \neq r' \quad \Gamma \vdash R \preceq R_f : r \quad \Gamma \vdash R' \preceq R_f : r'}{\Gamma \vdash R \# R'}$$

$$\textsc{Disjoint-Left-Index}\quad \frac{i \neq i' \quad \Gamma \vdash R \preceq R_f : [i] \quad \Gamma \vdash R' \preceq R_f : [i']}{\Gamma \vdash R \# R'}$$

$$\textsc{Disjoint-Left-Name-Index}\quad \frac{\Gamma \vdash R \preceq R_f : r \quad \Gamma \vdash R' \preceq R_f : [i]}{\Gamma \vdash R \# R'}$$

Second, we have "distinctions from the right": any two RPLs that differ in the same position before a star from the right are disjoint:

$$\textsc{Disjoint-Right-Name}\quad \frac{r \neq r'}{\Gamma \vdash R : r \# R' : r'} \qquad \textsc{Disjoint-Right-Index}\quad \frac{i \neq i'}{\Gamma \vdash R : [i] \# R' : [i']}$$

$$\textsc{Disjoint-Right-Name-Index}\quad \frac{}{\Gamma \vdash R : r \# R' : [i]}$$

Disjointness is symmetric, as may be seen from the symmetry of the rules. We extend the relation to dynamic RPLs (generating rules DYN-DISJOINT-LEFT-NAME, etc.) in the same way described in Section 4.2.3.

We can now establish that RPL disjointness implies disjointness of the region sets.

**Proposition 4.3.4.** *If $H \vdash dR \# dR'$, then $S(dR, H) \cap S(dR', H) = \emptyset$.*

*Proof.* If we used one of the DYN-DISJOINT-RIGHT rules to prove disjointness, then there can be no element in the intersection because of the syntactic difference between the elements in the two sets. Otherwise, we a used DYN-DISJOINT-LEFT rule. A simple induction shows that if $H \vdash dR \preceq dR_f$, then for all $dR'_f \in S(dR, H)$, $H \vdash dR'_f \preceq dR_f$. The result then follows from Lemma 4.3.2. □

Next we define $\mathsf{region}(o, f, H)$, the region of field $f$ of class object $o \in \mathsf{Dom}(H)$, and $\mathsf{region}(o, n, H)$ then region of cell $n$ of array $o \in \mathsf{Dom}(H)$. This definition formalizes the idea that regions $R$ in the field declarations $T\ f$ `in` $R$ partition the heap:

**Definition 4.3.5** (Region of a field or array cell)**.** *If $H \vdash o : C\texttt{<}dR\texttt{>}$ and $T\ f$ `in` $R_f \in \mathsf{def}(C)$, then $\mathsf{region}(o, f, H) = R_f[\texttt{this} \leftarrow o][\mathsf{param}(C) \leftarrow dR]$. If $H \vdash o : dT\texttt{[]<}dR\texttt{>\#}i$, then $\mathsf{region}(o, n, H) = dR[i \leftarrow n]$.*

Note that $\mathsf{region}(o, f, H)$ is fully specified (i.e., it is a region), because only fully specified RPLs are allowed in evaluating `new` expressions (rule DYN-NEW-CLASS). Similarly for $\mathsf{region}(o, n, H)$.

**Proposition 4.3.6.** *At runtime, disjoint regions imply disjoint locations. That is, if*

$$H \vdash \mathsf{region}(o, f, H) \# \mathsf{region}(o', f', H),$$

*then either $o \neq o'$ or $f \neq f'$; and if $H \vdash \mathsf{region}(o, n, H) \# \mathsf{region}(o', n', H)$, then either $o \neq o'$ or $n \neq n'$.*

This claim follows directly from Proposition 4.3.4 and the fact that the class definition together with the region binding in the type of `new` specifies exactly one fully specified region for each object field at the time the object is created.

### 4.3.3 Noninterference of Effect

We define the noninterference relation for static effects ($\Gamma \vdash E \# E'$) as follows. The noninterference relation is symmetric:

$$\text{NI-SYMMETRIC}\quad \frac{\Gamma \vdash E \# E'}{\Gamma \vdash E' \# E}$$

Pairs of reads always commute:

$$\text{NI-READ}\quad \frac{}{\Gamma \vdash \texttt{reads}\, R \# \texttt{reads}\, R'}$$

Read-write and write-write pairs commute only if the locations are disjoint:

$$\text{NI-READ-WRITE}\quad \frac{\Gamma \vdash R \# R'}{\Gamma \vdash \texttt{reads}\, R \# \texttt{writes}\, R'} \qquad \text{NI-WRITE}\quad \frac{\Gamma \vdash R \# R'}{\Gamma \vdash \texttt{writes}\, R \# \texttt{writes}\, R'}$$

An invocation effect commutes with another effect if the underlying effect of the invocation commutes with that effect:

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

Two invocation effects commute if they are declared to commute, regardless of their underlying effects:

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

Finally, we have the obvious rules for empty sets and set unions:

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

We extend the relation to dynamic effects as described in Section 4.2.3.

Now we can prove that expressions with noninterfering effects commute. First we define *basic effects*, which are the actual effects produced by program execution:

**Definition 4.3.7.** *A basic effect is one of the following:* `reads` $R_f$, `writes` $R_f$, *or* `invokes` $C.m$ `with` $dE''$, *where* $dE''$ *is a possibly empty union of basic effects.*

**Lemma 4.3.8.** *If* $(e, \Sigma, H) \rightarrow (o, H', dE)$, *then* $dE$ *is a possibly empty union of basic effects.*

*Proof.* Obvious from the rules given in Section 4.2.2. □

Next we formally define what it means for a commutativity annotation $m$ `commuteswith` $m'$ to be correct:

**Definition 4.3.9.** *For a program* $\mathcal{P}$, *an annotation* $m$ `commuteswith` $m'$ *appearing in class* $\mathcal{C}$ *of* $\mathcal{P}$ *is* ***correct*** *if for every pair of heaps* $H$ *and* $H'$ *and objects* $o$ *and* $o'$ *in* $H$ *such that executing* $o.m$ *and then* $o'.m'$ *with initial heap* $H$ *produces heap* $H'$, *executing the methods in reverse order (i.e.,* $o'.m'$ *then* $o.m$*) in initial heap* $H$ *also produces heap* $H'$.

Informally, $m$ and $m'$ commute if the order in which they appear in the program execution is irrelevant. Note that we define commutativity in terms of physical heap state, which is somewhat stronger than we might want in a real application: for example, we might want to treat two methods as commutative if either order of execution produces the same abstract data structure, with possibly different internal representations. The formalism given here could easily be extended to represent higher-level notions of equivalent heap state, such as equivalent data structures with different internal representations.

Now we can state a proposition about the dynamic effects produced by program execution: if we evaluate $e$ then $e'$ with the same dynamic environment $\Sigma$, and if the dynamic effects produced by the two evaluations are noninterfering, then $e$ and $e'$ are *commutative*, i.e., we get the same result as if we evaluate $e'$ then $e$:

**Proposition 4.3.10.** *If all annotations* $m$ `commuteswith` $m'$ *appearing in* $\mathcal{P}$ *are correct and* $(e, \Sigma, H) \rightarrow (o, H', dE)$ *and* $(e', \Sigma, H') \rightarrow (o', H'', dE')$ *and* $H'' \vdash dE \# dE'$, *then there exists* $H'''$ *such that* $(e', \Sigma, H) \rightarrow (o', H''', dE')$ *and* $(e, \Sigma, H''') \rightarrow (o, H'', dE)$.

*Proof.* First, note that the rules in Section 4.2.2 faithfully record the heap effects of expression evaluation: whenever we read a region, we record a read effect to that region (DYN-FIELD-ACCESS); whenever we write a region, we record a write effect to that region (DYN-FIELD-ASSIGN); and whenever we invoke a method, we record an invocation with all the effects that occurred during the evaluation of the method body (DYN-INVOKE). Further, by rules NI-EMPTY and NI-UNION, it is clear that two effects are noninterfering if and

only if their component basic effects are pairwise noninterfering. Thus it suffices to prove the proposition in the case where each of $dE$ and $dE'$ is a basic effect.

Note that the invocation effect is recursive (invocation effects can appear in the effect $dE''$ in `invokes` $C.m$ `with` $dE''$). So use induction on the total number of times that `invokes` appears in $dE$ and $dE'$. In the base case of zero times, the result is obvious from the noninterference rules, Proposition 4.3.6, and the semantics of read and write. In the inductive case, either $dE$ or $dE'$ (or both) is an invocation effect, and we must have used either DYN-NI-INVOKES-1 or DYN-NI-INVOKES-2 to prove $H \vdash dE \# dE'$. Assume without loss of generality that $dE = \texttt{invokes}\, C.m\, \texttt{with}\, dE''$. If we used DYN-NI-INVOKES-1, then we have $H \vdash dE'' \# dE'$, and by the IH all the operations that created $dE''$ can be interchanged with the operation that created $dE'$. Therefore the method invocation that produced $dE$ can be interchanged with the operation that produced $dE'$. If we used DYN-NI-INVOKES-2, then by Definition 4.3.9, the operations can be interchanged without changing the resulting heap state. □

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

**Theorem 4.3.11.** *If* $\Gamma \vdash e : T, E$ *and* $\Gamma \vdash e' : T', E'$ *and* $\Gamma \vdash E \# E'$ *and* $H \vdash \Sigma \preceq \Gamma$ *and* $(e, \Sigma, H) \to (o, H', dE)$ *and* $(e', \Sigma, H') \to (o', H'', dE')$*, then there exists* $H'''$ *such that* $(e', \Sigma, H) \to (o', H''', dE')$ *and* $(e, \Sigma, H''') \to (o, H'', dE)$.

*Proof.* By Proposition 4.3.10, it suffices to prove that $H'' \vdash dE \# dE'$, and again we can assume that $dE$ and $dE'$ are basic. By Theorem 4.2.7, we have (a) $H'' \vdash dE \subseteq \Sigma(E)$ and $H'' \vdash dE' \subseteq \Sigma(E')$, and by Lemma 4.2.6, we have (b) $H'' \vdash \Sigma(E) \# \Sigma(E')$. Now consider the possibilities for $dE$ and $dE'$. Again we use induction on the total number of times that `invokes` appears in $dE$ and $dE'$. In the base case of zero times, there are three possibilities, up to reordering:

1. $dE = \texttt{reads}\, dR_f$, $dE' = \texttt{reads}\, dR'_f$: Obvious because pairs of reads always commute.

2. $dE = \texttt{reads}\, dR_f$, $dE' = \texttt{writes}\, dR'_f$: By (a) together with rules DYN-SE-READS, DYN-SE-READS-WRITES, and DYN-SE-WRITES, there exists an effect `reads` $dR$ or `writes` $dR$ in $\Sigma(E)$ such that $H \vdash dR_f \subseteq dR$, and there exists an effect `writes` $dR'$ in $\Sigma(E')$ such that $H \vdash dR'_f \subseteq dR'$. By (b) and the rules for noninterference of effect, we have $H'' \vdash dR \# dR'$. Lemma 4.3.3 and Proposition 4.3.4 then establish that $dR_f$ and $dR'_f$ are distinct regions.

3. $dE = \texttt{writes}\, dR_f$, $dE' = \texttt{writes}\, dR'_f$: Nearly identical to case 2, except that only DYN-SE-WRITES is used, and both effects are covered by a write.

In the inductive case, there are a further three possibilities, again up to reordering:

4. $dE = \texttt{reads}\, R_f$, $dE' = \texttt{invokes}\, C.m\, \texttt{with}\, dE''$: By the rules for subeffects, together with (a), $dE$ is covered by either $\texttt{reads}\, dR$ or $\texttt{writes}\, dR$ in $\Sigma(E)$; and either $\texttt{invokes}\, C.m\, \texttt{with}\, dE'''$ appears in $\Sigma(E')$ with $H'' \vdash dE' \subseteq dE'''$ (rule Dyn-SE-Invokes-1), or $H'' \vdash dE'' \subseteq \Sigma(E')$ (rule Dyn-SE-Invokes-2). In the first case, by rule DYN-NI-INVOKES-1, we have that the covering read or write effect is disjoint from $dE'''$, and by the IH we have that $\texttt{reads}\, R_f$ is disjoint from $dE''$, which gives the result, again by rule DYN-NI-INVOKES-1. In the second case, the IH gives the result directly.

5. $dE = \texttt{writes}\, R_f$, $dE' = \texttt{invokes}\, C.m\, \texttt{with}\, dE''$: Same as case 4, except that $dE$ is covered only by a write effect.

6. $dE = \texttt{invokes}\, C.m\, \texttt{with}\, dE''$, $dE' = \texttt{invokes}\, C'.m'\, \texttt{with}\, dE'''$: If we used rule NI-Invokes-1 to prove that the two covering effects are noninterfering, then this case reduces to the IH. Otherwise, each of $dE$ and $dE'$ is covered by an invocation effect, and the two effects are noninterfering by rule NI-Invokes-2. In this case, we have $C = C'$, $m\, \texttt{commuteswith}\, m' \in \mathsf{def}(C)$, and $H'' \vdash E'' \# E'''$. Therefore $H'' \vdash dE \# dE'$, again by rule NI-Invokes-2.

□

Theorem 4.3.11 says that if two expressions have noninterfering *static* effects, then their actual runtime effects are noninterfering as well. Therefore, we can use the static effect information to reason soundly about noninterference at runtime.

## 4.4 Extending the Language

In this section, we informally discuss how to extend Core DPJ as follows: (1) adding parallel constructs to the sequential language; and (2) adding inheritance.

### 4.4.1 Adding Parallel Constructs

As discussed in Chapter 3, the actual DPJ language includes `foreach` for parallel loops and `cobegin` for a block of parallel statements. We can easily add an expression `cobegin(`$e$`,`$e'$`)` that 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 composing expressions $e_i$ in parallel and evaluating each one in environment $dE_i$ containing the binding $(I, i)$, where $I$ is an induction variable defined in the scope of the `foreach`, and $i$ is a natural number. In all cases we 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 (Section 4.3.3). It follows directly from Theorem 4.3.11 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, which is exactly the result we wanted.

However, there is one subtlety here that we should not overlook. Because we used large-step semantics to define the dynamic semantics (Section 4.2.2), there are *by definition* no interleavings between the executions of two expressions $e$ and $e'$: we either execute $e$ then $e'$ or $e$ then $e'$. This means we essentially get atomicity of expression execution "for free." In a real program execution, there may be arbitrary interleavings between the effects of parallel code sections. To simulate these interleavings, we could use a more complicated execution model, such as small-step semantics, to model the global effect of each individual step of expression execution, instead of treating the entire expression as a unitary effect.

Notice, however, what we actually showed in the course of proving Theorem 4.3.11: that if expressions $e$ and $e'$ have noninterfering static effects, then the individual basic effects generated by their executions are pairwise commutative. It is therefore straightforward to show that if we adopted a more fine-grain execution model (such as small-step semantics), then we could use the commutativity of the individual basic effects to establish the commutativity of the groups of basic effects generated by executing $e$ and $e'$. In fact, this is essentially what we did in proving Theorem 4.3.11, just without explicitly modeling the interleavings in the rules for dynamic execution.

Finally, in Chapter 6, we give just such a small-step semantics model for a simpler deterministic language. There we explore the interactions between deterministic and nondeterministic execution, so it is important to model the parallel execution. Here, by contrast, our purpose is to develop type and effect mechanisms for noninterference. So we defer the detailed model of parallel execution to the later chapter.

### 4.4.2 Adding Inheritance

**Syntax and Static Semantics:** We add inheritance to the syntax and static semantics of Core DPJ as follows. First, we extend the definition of valid types to account for the inheritance hierarchy. We add the type `Object`, and we change the class definitions to the form `class` $C$<$\rho$> `extends` $T$, where $T$ is a type that is valid in the environment containing $\rho \subseteq$ `Root:*`. We put a relation $\preceq$ on class names that represents the class hierarchy in the obvious way; formally, it is the reflexive, transitive closure of the relation given by $C \preceq C'$ if `class` $C$<$\rho$> `extends` $C'$<$R$> $\in \mathcal{P}$.

Next, in order to describe the semantics of inheritance, we need to be able to translate types, effects, and RPLs written in terms of the parameters of a superclass down to the subclass. To do this, we use the parameter substitutions implied by the `extends` clauses to define a *context translation operator* $\mathsf{trans}_{C \leftarrow C'}$ that rewrites $\mathsf{param}(C)$ in terms of $\mathsf{param}(C')$ if $C \preceq C'$: First, if $C = C'$, then $\mathsf{trans}_{C \leftarrow C'}(\rho) = \rho$. Second, if `class` $C$<$\rho$> `extends` $C'$<$R$> $\in \mathcal{P}$, then $\mathsf{trans}_{C \leftarrow C'}(\mathsf{param}(C')) = R$. Third, if $C \preceq C' \preceq C''$, then

$$\mathsf{trans}_{C \leftarrow C''}(\mathsf{param}(C'')) = \mathsf{trans}_{C \leftarrow C'}(\mathsf{trans}_{C' \leftarrow C''}(\mathsf{param}(C''))),$$

where we extend $\mathsf{trans}_{C \leftarrow C'}$ in the obvious way to an operation on RPLs. We also extend $\mathsf{trans}_{C \leftarrow C'}$ in the obvious way to an operation on types and effects.

Now we just have to amend the rules for the static semantics to account for the fact that RPLs, types, and effects may be inherited from superclasses, and that inherited entities must be translated to the context in which they are used:

1. To the rules for subtyping (Section 4.1.3), add the following rules. First, every type is a subtype of `Object`. Second, if $C' \preceq C$, then $C'$<$R$> is a subtype of $C$<$\sigma(\mathsf{trans}_{C' \leftarrow C}(\mathsf{param}(C)))$>, where $\sigma = \{\mathsf{param}(C') \mapsto R\}$.

2. Methods are inherited or overridden by subclasses as in Java. In the rule METHOD (Section 4.1.1), check that if $C'$`.`$m$ overrides $C$`.`$m$, then (1) the parameter and return types match after applying $\mathsf{trans}_{C' \leftarrow C}$ to the types appearing in $C$`.`$m$; and (2) the declared effect of $C'$`.`$m$ is a subeffect of $\mathsf{trans}_{C' \leftarrow C}(E)$, where $E$ is the declared effect of $C$`.`$m$.

3. In rule FIELD-ACCESS, look for $T$ $f$ `in` $R_f \in \mathsf{def}(C')$, with $C \preceq C'$, and record effect `reads` $\mathsf{trans}_{C \leftarrow C'}(R_f)$; and similarly for FIELD-ASSIGN. In rule INVOKE, look for $T_r$ $m(T_x\ x)$ $E$ { $e$ } $\in$

$\mathsf{def}(C')$, with $C \preceq C'$. Use $\mathsf{trans}_{C \leftarrow C'}(T_r)$ instead of $T_r$; and similarly for $T_x$ and $E$.

**Dynamic Semantics:** To extend the dynamic semantics, we add `Object` as a valid dynamic type. We also extend the rules for valid dynamic types and subtypes as described in Section 4.2.3, using the extensions to the static type rules described above. Again, we have to account for the fact that RPLs, types, and effects may be inherited from superclasses:

1. In rule DYN-FIELD-ACCESS, look for $T\ f$ `in` $R_f \in \mathsf{def}(C')$, with $C \preceq C'$, and record effect `reads` $\Sigma(\mathsf{trans}_{C \leftarrow C'}(R_f))$; and similarly for rule DYN-FIELD-ASSIGN.

2. In rule INVOKE, look for $T_r\ m(T_x\ x)\ E\ \{\ e\ \} \in \mathsf{def}(C')$, for $C \preceq C'$, and record effect `invokes` $C'$`.`$m$ `with` $dE$.

3. In rule New, the type of $o'$ is $C$<$\Sigma(\sigma(\mathsf{trans}_{C' \leftarrow C''}(R)))$>, where $C'$ is the class in the type of $o$, and $C''$ is the class in whose definition the `new` expression actually appears.

**Noninterference and Soundness:** We extend rule NI-Invokes-2 (Section 4.3.3) so that two invocation effects commute if the methods $C$`.`$m$ and $C$`.`$m'$ are declared to commute *or either of the methods overrides a method that is declared to commute.* Formally, if $m$ `commuteswith` $m' \in \mathsf{def}(C)$, then it is assumed that $C'$`.`$m$ commutes with $C''$`.`$m'$ for all $C' \preceq C$ and $C'' \preceq C$, and these facts must be guaranteed by the implementations of subclasses of $C$. In this sense the commutativity annotation is "inherited." This rule is necessary to ensure sound inference about commutativity in the presence of polymorphic method invocation.

We extend the operator $\mathsf{trans}_{C' \leftarrow C}$ to static environments as follows. If $(\texttt{this}, C\texttt{<}\mathsf{param}(C)\texttt{>}) \in \Gamma$, then $\mathsf{trans}_{C' \leftarrow C}(\Gamma)$ is defined to be

$$\{(\texttt{this}, C'\texttt{<}\mathsf{param}(C')\texttt{>})\} \cup \{(v, \mathsf{trans}_{C' \leftarrow C}(T)) | (v, T) \in \Gamma \wedge v \neq \texttt{this}\} \cup \{\rho \subseteq \mathsf{trans}_{C' \leftarrow C}(R) | \rho \subseteq R \in \Gamma\}.$$

Intuitively, we replace $C$<$\mathsf{param}(C)$> with $C'$<$\mathsf{param}(C')$> in the binding of `this` and translate the RHS of all bindings and constraints to the environment of $C'$. It is straightforward to show that for every essential relationship (valid RPLs, types, and effects; nesting, inclusion, subtyping, etc.) that holds in $\Gamma$,

the same is true in $\mathsf{trans}_{C' \leftarrow C}(\Gamma)$ for the entities after translation by $\mathsf{trans}_{C' \leftarrow C}$. Lemma 4.2.1 then follows immediately for the extended language.

Next we extend Definition 4.2.4 (instantiation of static environments by dynamic environments) as follows: $H \vdash \Sigma \preceq \Gamma$ if `this` is bound to class $C$ in $\Gamma$, `this` is bound to class $C'$ in $\Sigma$, and $H \vdash \Sigma \preceq \mathsf{trans}_{C' \leftarrow C}(\Gamma)$ according to Definition 4.2.4 as given in Section 4.2.4. The extended definition accounts for the fact that the class of the actual object bound to `this` at runtime may be a subclass of the statically enclosing class of the method we are executing. It is straightforward to extend the proofs of Lemmas 4.2.5 and 4.2.6 and Theorem 4.2.7 under this extended definition.

The definitions and results in Section 4.3.1 and 4.3.2 do not depend on the extended language, so those carry through unchanged. The proof of Proposition 4.3.6 is obvious once we extend rule NI-Invokes-2 as described above. The proof of Theorem 4.3.11 then goes through exactly as stated in Section 4.3.3.

# Chapter 5

# Effect System and Language for Determinism by Default

This chapter describes DPJ's mechanisms for controlled nondeterminism, building on the effect system and language described in the previous chapters. Section 5.1 explains the new language mechanisms for creating parallel tasks with possibly nondeterministic results, using the Traveling Salesman Problem algorithm as an example. Section 5.2 explains the new effect system features that cooperate with a *weakly isolated* transactional runtime (in our implementation, an STM) to enforce the safety properties described in Chapter 1. Section 5.3 describes additional effect system features that improve the performance of the transactional runtime by allowing the compiler to remove unnecessary synchronization overhead. Section 5.4 describes a prototype compiler implementing the techniques. Section 5.5 describes an evaluation of the techniques using three nondeterministic parallel benchmarks. Finally, Section 5.6 discusses related work.

## 5.1 Expressing Nondeterminism

To express nondeterministic computations, we introduce the following new language mechanisms:

1. *`foreach_nd`:* We add a nondeterministic parallel loop, denoted `foreach_nd`. The `nd` denotes "nondeterministic." It is the same as `foreach` discussed in previous chapters, except that interference is allowed between the loop iterations of `foreach_nd`. Executing these loop iterations in parallel therefore may permit nondeterministic results.

2. *`cobegin_nd`:* We add a nondeterministic parallel statement composition, denoted `cobegin_nd`. It is the same as `cobegin` except that again, interference is allowed between the tasks.

3. *Atomic statements:* We introduce an *atomic statement* with the syntax `atomic` $S$, where $S$ is a statement. A statement `atomic` $S$ indicates that $S$ is to be run as if *all other concurrent execution* were suspended while $S$ is executing. This is called strong isolation [84, 108].

We also extend the effect system to enforce the safety properties stated above (including strong isolation). The safety properties and the effect system extensions are discussed further in the next section.

We illustrate the new language features with a running example of the traveling salesman problem, or TSP. TSP is the well-known problem of finding a shortest cycle in a weighted graph that visits all the nodes once (i.e., a Hamiltonian cycle). TSP can be solved by *branch and bound search*, a common algorithm for solving optimization problems and a classic example for nondeterministic computation. A branch and bound algorithm divides the search space into sets of possibilities (the "branch") and "bounds" each set by estimating how far it is from the optimization goal. Branches that are no better than previously explored ones are discarded, while ones that are potentially better are explored.

Figures 5.1–5.3 show simplified Java-like pseudocode for TSP. For now we discuss only the mechanisms for task creation and synchronization; the region and effect annotations for proving safety properties are discussed in the next section. The global data structures (lines 1–13) include a weighted graph that is the input to the program; a priority queue for storing the paths being explored; and a "best" (i.e., shortest) tour, which is refined as the computation progresses, eventually storing the answer. The main computation loop (lines 15–23) illustrates `foreach_nd`. It iterates in parallel over several worker tasks. Each task generates a prefix to search (using the pseudocode in Figure 5.2) and searches it (using the pseudocode in Figure 5.3), until there are no more prefixes to search. The prefix generation occurs in isolation because of the `atomic` statement at lines 18–20 of Figure 5.1.

Figure 5.2 shows the code for generating a prefix. If there is any useful work remaining on the priority queue, the worker task removes a prefix from the queue. If the prefix already contains enough edges, it returns that prefix to be solved. Otherwise, it generates new prefixes by adding one edge to that prefix, putting it on the priority queue, and repeating (lines 8–13). Notice that all the worker threads are calling `generateNextPrefix` (and therefore reading and writing the priority queue) concurrently. Atomic access to the queue is therefore essential for correctness.

Note that while the calls to `generateNextPrefix` are effectively serialized, each worker can start its call to `searchAllToursWithPrefix` as soon as its call to `generateNextPrefix` is done, in a pipelined manner. This pattern can achieve good speedups because most of the work in this code is done in `searchAllToursWithPrefix`.

```
/* Regions for partitioning data */
region ReadOnly, atomic Mutable;

/* Graph we are working on; immutable */
Graph<ReadOnly> graph in ReadOnly = the TSP graph;

/* Priority queue for tour prefix paths */
final PriorityQueue<Path<ReadOnly>, Mutable> priorityQueue =
    new PriorityQueue<Path<ReadOnly>, Mutable>();
priorityQueue.add(new Path<ReadOnly>(startNode));

/* The answer */
Path<ReadOnly> bestTour in Mutable = infinite path;

foreach_nd(int i in 0, NWORKERS) {
    Path<ReadOnly> prefix = null;
    while (true) {
        atomic { prefix = generateNextPrefix(); }
        if (prefix == null) break;
    }
}
```

Figure 5.1: Global data and main computation for the Traveling Salesman Problem

```
Path generateNextPrefix() reads ReadOnly writes Mutable {
    while (!priorityQueue.isEmpty() &&
           priorityQueue.best().length() < bestTour.length()) {
        Path<ReadOnly> prefix = priorityQueue.removeBest();
        if (prefix.nodeCount() > PREFIX_CUTOFF) {
            return prefix;
        } else {
            for (each edge edge that can be added to prefix
                 while staying under bestTour.length()) {
                Path<ReadOnly> newPrefix =
                    new Path<ReadOnly>(prefix, edge);
                priorityQueue.add(newPrefix);
            }
        }
    }
    return null;
}
```

Figure 5.2: Generating the next tour prefix

```
1  void searchAllToursWithPrefix(Path<ReadOnly> prefix)
2      reads ReadOnly writes atomic Mutable {
3      for (each Hamilton cycle tour in graph with prefix prefix) {
4          atomic {
5              if (tour.length() < bestTour.length()) {
6                  bestTour = tour;
7              }
8          }
9      }
10 }
```

Figure 5.3: Searching all tours with a given prefix

Figure 5.3 shows the code for searching tours starting with a given prefix. The construction of the tours is read-only on the graph, so no synchronization is needed for access to the graph. The only synchronization needed is for the concurrent read-modify-write access to `bestTour`, at lines 5–7, which also must be atomic.

## 5.2 Enforcing Safety Properties

As discussed in Chapters 1 and 2, our goal is to enforce, at compile time, four safety guarantees for the extended language with nondeterminism: (1) freedom from data races; (2) strong isolation for statements marked `atomic`; (3) sequential equivalence for deterministic parallel constructs; and (4) determinism by default. We now discuss several extensions to the deterministic effect system described in Chapters 3 and 4 that allow these four properties to be enforced.

**Data race freedom and strong isolation:** We use the following strategy to ensure both data race freedom and strong isolation:

1. A transactional runtime guarantees at least weak isolation of `atomic` statements.

2. The effect system ensures that for any pair of conflicting memory accesses, both of the accesses occur inside `atomic` statements. This requirement ensures strong isolation, because no conflicts between unguarded memory accesses and `atomic` statements are allowed. It also ensures race freedom, because no conflicts between pairs of unguarded accesses are allowed.

Part 1 of our strategy is familiar from previous work on language mechanisms supported by transactional runtimes [62, 8, 65, 44]. Part 2 is new, and it leads to two significant advantages. First, our strategy guaran-

tees strong isolation even if the underlying implementation guarantees only weak isolation. This property is very important, because it allows our language to be built on top of a standard software transactional memory (STM) implementation. Typically, STM guarantees weak isolation, because the overhead of guaranteeing strong isolation in software is prohibitive. Second, our strategy prohibits all data races. Even TM systems with strong isolation generally allow data races between pairs of accesses, both of which occur outside any transaction.

*Effect system extensions.* To ensure that pairs of conflicting accesses both occur inside `atomic` statements (part 2 above), we extend the DPJ effect system described in Chapter 3 as follows:

1. Internally, the compiler distinguishes read and write effects as either *atomic* (meaning the effect occurred inside an `atomic` statement) or *non-atomic* (meaning the effect occurred outside any `atomic` statement).

2. To support sound reasoning about atomic effects across method invocations, we extend the syntax of DPJ's method effect summaries to denote whether effects are atomic. An atomic effect is denoted by writing the keyword `atomic` before the RPL. For example, an effect `writes` $R_1$`, atomic` $R_2$ denotes a non-atomic write to $R_1$ and an atomic write to $R_2$.

3. We extend DPJ's rules for subeffects so a non-atomic effect covers an atomic effect, but not vice versa.

4. In checking a `foreach_nd` or `cobegin_nd` statement, interference is allowed between the component parallel tasks only between pairs of effects that are either mutually noninterfering or both `atomic`.

Together, these rules ensure that any pair of conflicting accesses both occur in `atomic` statements. The first three rules ensure that atomic effects are reported only where there is, in fact, an enclosing `atomic` statement. The fourth rule disallows interference without atomic effects, i.e., without enclosing `atomic` statements.

*TSP example.* The TSP example discussed in the previous section illustrates the effect system extensions. As shown in line 2 of Figure 5.1, two regions are used to hold the data: `ReadOnly` for fields that will not

be modified during the computation, and `Mutable` for those that will be. The type

```
PriorityQueue<Path<ReadOnly>, Mutable>
```

of the priority queue indicates that the queue contains objects of type `Path<ReadOnly>`, and that the internal data used to represent the queue itself is in region `Mutable`. Here are examples of how the four rules stated above are used to check the correctness of the TSP example:

1. In Figure 5.3, the read effect on region `ReadOnly` is not atomic, because it is generated in the test condition of the `for` loop, outside the `atomic` statement. However, the write effect on region `Mutable` is atomic, because it is generated by the assignment to the variable `bestTour` at line 6, inside the `atomic` statement.

2. The effect summary in line 2 of Figure 5.3 says that the write effect on `Mutable` is atomic (but the read effect on `ReadOnly` is not).

3. It would be permissible (but conservative) to rewrite the effect summary in line 2 of Figure 5.3 as

   ```
   reads ReadOnly writes Mutable,
   ```

   because `writes Mutable` covers `writes atomic Mutable`, which is the actual effect generated in line 6. However, it would be a compile-time error to write `reads atomic ReadOnly`, because the effect on `ReadOnly` is not atomic.

4. In the `foreach_nd` at lines 15–23 of Figure 5.1, the effects on `ReadOnly` are all reads, and `Mutable` and `ReadOnly` are distinct regions. Therefore, the only interfering effects across iterations of the loop are the atomic writes to `Mutable` generated by the calls to `generateNextPrefix` and `searchAllToursWithPrefix`. The first call occurs inside an `atomic` section, so it generates atomic effects. The second call occurs outside an atomic section, but as noted above, the write effect in the signature of `searchAllToursWithPrefix` is marked atomic (line 2 of Figure 5.3. On the other hand, if the `atomic` statement at line 18 of Figure 5.1 were removed, or the write effect in line 2 of Figure 5.3 were made non-atomic, then a compile-time error would occur in checking the `foreach_nd` loop.

*Programmer benefit.* Together, race-freedom and strong isolation convey the following benefits to the programmer in this language:

1. *Sequential consistency.* Because there are no data races, the Java memory model guarantees a sequentially consistent execution. That means the observed execution is consistent with program order (i.e., the order of execution defined by the program text).

2. *Reduced interleavings.* Because code occurring outside an `atomic` statement is either noninterfering or not parallel by definition in this language, a program's execution is determined by only two things: (1) the actual order of execution of concurrent, interfering `atomic` statements, and (2) program order. Further, program order does not introduce any schedule dependence over and above (1). Therefore, *the only source of non-equivalent interleavings is from different orderings of concurrent, interfering* `atomic` *statements*.

The first property is important because it is well known that sequentially consistent executions are much easier to reason about than non-sequentially consistent ones. The second property is important because many programmers (and testing tools) analyze program behavior by reasoning about the possible interleavings (or schedules) of parallel operations. Reducing the effective number of interleavings makes such reasoning easier.

**Sequential equivalence for deterministic constructs:** As discussed more formally in the next chapter, a basic property of DPJ as described in Chapter 3 is that `cobegin` and `foreach` behave exactly like a program-ordered sequential composition of their component tasks. We wish to preserve this property for the extended language that includes `cobegin_nd` and `foreach_nd`. We view this property as essential for allowing local, compositional reasoning about the interactions between deterministic and nondeterministic operations.

To guarantee this property, we incorporate the following two rules in the type system:

1. Interference is allowed only between parallel branches of a `foreach_nd` or `cobegin_nd`. No interference is allowed between parallel branches of a `cobegin` or `foreach`, even if the interfering accesses are both guarded by `atomic` sections.

2. Inside a `foreach_nd` or `cobegin_nd`, interference is allowed between a `cobegin` statement and other parallel code only if the entire `cobegin` statement is enclosed in an `atomic` statement. This

ensures that every `cobegin` executes as if it were an isolated, sequential statement, even inside a `foreach_nd` or `cobegin_nd`.

The motivation for the first rule is fairly obvious. For example, we wish to disallow code like this:

```
x = 0
cobegin {
    atomic x = 1; // S1
    atomic x = 2; // S2
}
```

Even though the writes to variable `x` are enclosed in `atomic` statements (so there is no data race), the order of execution of the statements `S1` and `S2` is nondeterministic, as is the final result. The point of `cobegin` is to indicate deterministic composition, so we just disallow such interference. In the typing rules, the rule for checking `cobegin`/`foreach` is then exactly the same as in Chapter 3, while the rule for `cobegin_nd`/`foreach_nd` is as stated above (interference is allowed, but only where guarded by `atomic` statements). The next chapter gives these rules formally.

The motivation for the second rule is perhaps more surprising. To see the motivation, consider the following program:

```
z = 0;
cobegin_nd {
    cobegin {
        atomic x = z; // S1
        atomic y = z; // S2
    }
    atomic z = 1;     // S3
}
```

In our view, this program has a serious problem: it destroys the property we want to carry over from the deterministic language, i.e., that `cobegin` behaves like a program-ordered sequential composition of its component tasks. According to the semantics of `cobegin_nd` and `cobegin`, the statements could be executed in the order `S2`, `S3`, `S1`, producing the result $\texttt{x} = 1$, $\texttt{y} = 0$. This result is impossible for a sequentially consistent execution of the program obtained by erasing the `cobegin`: for that program, `S1` must occur before `S2`; and so if `x` equals 1, then `S3` must have executed before `S1` *and* `S2`, and `y` must equal 1 as well. In effect, the presence of the interfering write to `z` in the other `cobegin_nd` branch exposes the fact that the order of execution of `S1` and `S2` was different for `cobegin` than it could be for ordinary sequential composition.

Because we want programmers to be able to reason about `cobegin` as program-ordered sequential composition, we disallow this program. Specifically, we require that the *entire* `cobegin` statement execute as an isolated operation (as if it were surrounded entirely by `atomic`) whenever it occurs inside `cobegin_nd`. We express this requirement in the effect system by simply "ignoring" any `atomic` statements occurring inside the `cobegin`, for purposes of computing the effect of the entire `cobegin`. For example, in the fragment above, the `cobegin` branches generate atomic effects on `x`, `y`, and `z` at the point of the assignments. However, when those component effects are accumulated into the effect of the entire `cobegin`, they are transformed into non-atomic effects, so the effect of the entire cobegin is a *non-atomic* read of `z` and a pair of *non-atomic* writes to `x` and `y`. In particular, because the read of `z` is non-atomic in the first branch of the `cobegin_nd`, the conflicting write to `z` in the second branch is disallowed (even though it is atomic). Again, these rules are stated more formally in the next chapter.

Notice the following (slightly different) programs that a programmer *may* write in our language. First, if the entire `cobegin` is enclosed in an `atomic` statement, then the effects of the `cobegin` are made atomic again, and the composition is allowed:

```
z = 0;
cobegin_nd {
    atomic cobegin { x = z; y = z; }
    atomic z = 1;
}
```

This code is permissible, because the `atomic` statement in the first branch of the `cobegin_nd` guarantees that the `cobegin` executes in isolation, despite the interference with the second branch.

Second, a programmer who truly wants both interference and interleaving can write a `cobegin_nd` instead of a `cobegin`:

```
z = 0;
cobegin_nd {
    cobegin_nd { atomic x = z; atomic y = z; }
    atomic z = 1;
}
```

Because the inner parallelism is created by `cobegin_nd`, and not `cobegin`, the effects on `x`, `y`, and `z` are reported as atomic effects to the outer `cobegin_nd`, and so the interference is allowed.

**Determinism by default:** The typing rules discussed above also guarantee the following property: evaluation of any isolated statement that does not dynamically execute a `foreach_nd` or `cobegin_nd` yields

a fixed output heap state for a fixed input heap state, up to inessential details like the addresses of objects on the heap. From the discussion above, an isolated statement is (1) any `atomic` statement, `cobegin`, or `foreach`; or (2) any statement not occurring in `foreach_nd` or `cobegin_nd` (including the entire program); or (3) any statement that does not dynamically execute an `atomic` statement (because if a statement executes no `atomic` statement, then the type system ensures that it runs with no interference from any other parallel statement). We call this property *determinism by default*, because it says that nondeterminism occurs only where explicitly requested via `foreach_nd` or `cobegin_nd`. We view this property as essential to any language that allows the composition of deterministic and nondeterministic parallel constructs. This property is stated and proved formally in the next chapter.

## 5.3 Performance: Removing Unnecessary Barriers

To enforce isolation of `atomic` statements, we elect to use a Software Transactional Memory (STM) runtime system [63] because it provides weak atomicity with simple programmer annotations, better composability than locks, and potentially better scalability than locks because of optimistic rather than pessimistic synchronization. One key drawback of STMs is the overhead due to *transactional read and write barriers* for every load or store to shared data (e.g., see [129]). These barriers are small sections of code, often automatically inserted by a transaction-aware compiler, that invoke the STM runtime to implement some transactional concurrency control protocol. The barriers can either read and write shared memory directly (so-called *in-place update* STM) and *undo* all transactional operations when a transaction aborts; or they can buffer updates into a private data structure (so-called *write buffering* STM) and apply all the buffered changes into shared memory when a transaction successfully commits. In both cases, barriers can incur significant overhead, and minimizing them is essential for performance.

The DPJ effect system can help with this problem. Although the purpose of the system to this point has been to enforce safety guarantees such as determinism and race freedom, we observe that we can leverage the DPJ effect system to remove STM barriers that are not necessary, because the accesses guarded by the barriers can never cause a transactional conflict. To do this, we need some slight extensions to the DPJ effect system. As described to this point, the effect system is designed to report interference *at the points where parallelism is created*, e.g., at a `cobegin` or `cobegin_nd`. Effects are generated at the point of use, then propagated back to the parallel task creation via the method effect summaries. However, to safely

remove TM synchronization, we must ensure that the code never causes interference in any use. This is a slightly different property, as illustrated in Figure 5.4. That figure shows a simple program with a method `setX` that atomically updates a variable `x`, a method `setXY` that atomically updates variables `x` and `y`, and a main method that calls `setX` and `setY` in parallel. If the whole program is represented in the figure, then the write to `y` in line 16 needs no synchronization, because it does not interfere with anything in the `cobegin_nd` at lines 5–8. However, there is no way to tell that from the body of method `setXY`, because it is a property of the *uses* of the method. Further, nothing in the effect system described to this point encodes this property.

```
1  class BarrierRemoval {
2      region X, Y;
3      int x in X, y in Y;
4      void work() {
5          cobegin_nd {
6              setX(1);
7              setXY(0);
8          }
9      }
10     void setX(int x) writes atomic X {
11         /* This write needs synchronization */
12         atomic this.x = X;
13     }
14     void setXY(int x, int y) writes atomic X, Y {
15         atomic {
16            /* This write needs synchronization */
17            this.x = x;
18            /* This write does not */
19            this.y = y;
20        }
21     }
22 }
```

Figure 5.4: Illustration of the problem of barrier removal. Whether the writes in lines 11, 17, and 19 need synchronization depends on how they are used. Here, the uses in lines 6–7 dictate that lines 11 and 17 need synchronization, but line 19 does not.

**Atomic regions:** Fortunately, we can encode this information with some simple extensions to the effect system. We extend the effect system to distinguish between two kinds of regions: those that may interfere (and so need barriers everywhere) and those that cannot (and so do not need synchronization barriers anywhere). We call the first kind of region an *atomic region*, and the second kind a *non-atomic region*. Atomic regions are not limited to access inside an atomic statement: both kinds of regions can be accessed either inside or outside an atomic statement. However, only access to atomic regions is guarded by the transactional runtime

```
1  class AtomicRegions {
2      region atomic X, Y;
3      int x in X, y in Y;
4      void work() {
5          cobegin_nd {
6              setX(1);
7              setXY(0);
8          }
9      }
10     void setX(int x) writes atomic X {
11         atomic this.x = x;
12     }
13     void setXY(int x, int y) writes atomic X, atomic Y {
14        atomic {
15            this.x = x;
16            this.y = y;
17        }
18     }
19 }
```

Figure 5.5: Illustration of atomic regions.

inside an atomic statement.

We extend the syntax of region declarations so that the programmer can put the keyword `atomic` before the region name, indicating that the declared region is atomic. This syntax is illustrated in line 2 of Figure 5.5 (for region `X`) as well as line 2 of Figure 5.1 for region `Mutable` in the TSP example.

We also slightly change the rules for atomic effects from what is described in the previous section: now, only operations on atomic regions generate atomic effects. Operations on non-atomic regions never generate atomic effects, even inside a transaction. For example, in Figure 5.5, only region `X` is declared atomic (line 2), so the write to region `Y` in line 16 generates a non-atomic effect, even though it occurs inside an atomic statement. Notice that the effect summary in line 13 now reports the write effect on `Y` as non-atomic, as it must according to the rules stated in the previous section. Similarly, in the TSP example, the read of region `ReadOnly` in Figure 5.3 generates a non-atomic effect, even though it is inside the `atomic` block at line 4. The write to region `Mutable` generates an atomic effect.

These rules allow the compiler to perform the following optimizations for transactional operations on non-atomic regions:

1. A read operation on a non-atomic region never interferes with anything, and has no effect on the heap. Therefore, it needs no special TM code generation at all; it can be implemented as an ordinary

read operation. This optimization completely eliminates all STM overhead for transactional reads on non-atomic regions.

2. A write operation on a non-atomic region also never interferes with anything, so it cannot cause a transaction abort and does not need any synchronization. For example, in a TM that uses write-versioning, the version for that variable need not be updated. And in a TM that acquires locks before writing, no lock need be acquired. However, because the enclosing transaction could be aborted, in a TM implementation that uses in-place writes and undo logging, the old value must still be logged, so that it can be restored on abort. In a TM that uses write buffering, the new value must still be written to the write buffer.

For example, in Figure 5.5, the compiler can avoid updating a version or taking a lock for the write in line 16. In the TSP example, no TM overhead at all is generated by any read to region `ReadOnly` (for example the read access to the TSP graph in line 3 of Figure 5.3). While this optimization does not remove all TM overhead for writes, it still produces substantial savings because (1) transactional reads often outnumber transactional writes; (2) locking and versioning represent a substantial part of the TM overhead on writes; and (3) even in cases where there are no actual conflicts, unnecessary TM locking or versioning can lead to false conflicts due to hash collisions, causing more aborts and impeding scalability. We will say more about the performance impact of the optimizations in Section 5.5.

**Atomic region parameters:** Because region parameters function as RPLs, we also let them be declared atomic or non-atomic. Then the same code generation rules apply as discussed above: reads and writes on non-atomic region parameters generate non-atomic effects, and transactional barriers are removed or simplified for accesses to non-atomic region parameters.

However, to ensure soundness, we must be careful about the interaction between region names and region parameters. To see the problem, consider the code in Figure 5.6. Region `r` is declared atomic (line 2), so the effects on `r` in lines 7–8 are also atomic, and are allowed to interfere. However, inside the body of `setX` (lines 11–13), parameter `R` is not declared atomic, so the write in line 12 does not get any synchronization barrier, even when `setX` is used in a transaction, as in lines 7–8. (As explained more fully in the next section, when a method is used in a transaction, it is cloned, and barriers are inserted for all its accesses to atomic regions.) Therefore, this code cannot be correctly compiled according to the rules stated above.

```
1  class AtomicRegionParams<region R> {
2      atomic region r;
3      int x in R;
4      void work(AtomicREgionParams<r> arp) {
5          cobegin_nd {
6              /* writes atomic r */
7              atomic arp.setX(0);
8              /* writes atomic r */
9              atomic arp.setY(1);
10         }
11     }
12     void setX(int x) writes R {
13         /* writes R */
14         this.x = x;
15     }
16 }
```

Figure 5.6: Inconsistent bindings of region names to region parameters. As explained in the text, this code is disallowed.

The solution we adopt is to disallow this code. Specifically, we require that bindings of regions to parameters be consistent: only atomic regions may be bound to atomic region parameters, and similarly for non-atomic regions and parameters. This rule ensures that the actual runtime region bound to a parameter is atomic if and only if the parameter is declared atomic. Therefore, the compiler can soundly use the parameter declaration to do code generation.

This solution does impose one important limitation. If a programmer wishes to use a class region parameter as an atomic region in some context and a non-atomic region in some other context, then the class must be *cloned*: the programmer must create two copies of the class, one with the atomic parameter and one with the non-atomic parameter. A similar limitation applies to method region parameters. An alternative approach is to have the compiler automatically clone the classes and methods, similarly to what C++ already does for templates. This would complicate the implementation, and we have not done it for the prototype implementation discussed in the next section, but it does not raise any significant technical issues. An alternative would be to introduce polymorphism over whether a class region parameter is atomic.

## 5.4 Prototype Implementation

To implement the nondeterminism support, we extended the DPJ compiler described in Section 3.5. We implemented atomic blocks using the Deuce STM library [4]. We used the well-respected Transactional

Locking II (TL2) algorithm [44]. TL2 is a write-buffering (i.e., lazy versioning) algorithm with optimistic reads. Deuce supports concurrency control at the object field level, and it uses a lightweight custom reflection mechanism to access object fields inside transactions.

We selected this STM system for pragmatic reasons of ease of implementation, and because it implements a well-known high-performance STM algorithm. We have not attempted to maximize absolute performance in our implementation; it could probably be improved by using a different STM system, such as one integrated with the JVM. Our method is applicable to other types of STM systems and algorithms (including those utilizing in-place updates).

For each atomic block, the compiler generates code to execute the body of the atomic block as a transaction, retrying until the transaction commits successfully. Nested atomic blocks are flattened. Methods that are transitively callable within atomic blocks are cloned; versions containing barriers are used when they are called within atomic blocks. Within atomic blocks, the compiler inserts normal read and write barriers for accesses to fields in atomic regions. As discussed in section 5.3, the compiler omits barriers for read accesses to non-atomic regions, and it generates logging-only barriers for write accesses.

We modified the TL2 implementation in Deuce to support these optimized logging-only barriers. However, because TL2 is a write-buffering algorithm, we would have to use read-barriers to obtain correct values in the read-after-write cases. To avoid read barriers entirely, we modified the algorithm to perform in-place updates for these locations, and we maintain a separate undo log to revert the effects of such updates in case the transaction aborts. Reads to such locations do not need barriers because they can now obtain their values directly from the original memory location.

## 5.5 Evaluation

The ideas presented in this thesis raise four key questions for experimental investigation: (1) Can the language express nondeterministic algorithms in a natural way? (2) Can the algorithms expressed in the language give good performance? (3) How effective is the optimization of STM barriers? (4) What is the annotation overhead of the language?

We used four nondeterministic algorithms to evaluate these questions: two different versions of TSP, Delaunay mesh triangulation from the *Lonestar Benchmarks* [2], and *OO7*, a synthetic database benchmark that has been used in previous studies of parallel performance [127, 108]. These codes are discussed further

below.

### 5.5.1 Expressing Parallelism

**Traveling Salesman Problem:** We studied two versions of the TSP algorithm, which we call **TSP-PQ** and **TSP-R**. TSP-PQ is the algorithm described in Section 5.1. As discussed there, the algorithm proceeds in two phases: the first phase breaks the problem up into subproblems and adds them to a priority queue, and the second phase concurrently removes items from the queue and processes each one using sequential recursive search. The priority queue orders the work, so that more promising subtrees are explored first.

TSP-R is a variant that eliminates the priority queue and uses recursion to express the entire algorithm. At each level of the tree, the algorithm computes a bound for each subtree and compares the bound against the global current best tour. Bounds that are definitely no better than the current best are excluded, while bounds that may be better are explored recursively. The recursion occurs in parallel until a specified depth of the tree; in our studies we used a depth varying with the log of the number of threads. TSP-R is a simpler algorithm than TSP-PQ, but it potentially suffers from more contention, as the global best tour must be read before every recursive descent into a subtree to avoid exploring too many bad paths. By contrast, because TSP-PQ uses a priority queue to order the paths, it can read the global best tour less often (once per tree level).

We adapted both versions of TSP from code that was used in previous studies of STM performance [108, 107]. Our TSP-PQ code uses the identical algorithm to the original code, and expresses the parallelism in the same way. The original code had a data race, and we added one extra atomic block to eliminate that race. Our TSP-R code is a transformation of TSP-PQ that eliminates the priority queue, checks the bound at each level of the tree, and parallelizes the recursion.

**Delaunay Mesh Refinement:** This code uses Chew's algorithm [36, 74] to find and eliminate "bad triangles," i.e., those that do not satisfy some quality constraint from a Delaunay triangulation of a mesh of points. The program is nondeterministic since different orders of processing of bad elements lead to different meshes, although all such meshes satisfy the quality constraints [36]. The program uses a `foreach_nd` loop, and each iteration of the loop spawns a new worker thread (at most one per core). Each worker thread has a private worklist of bad triangles. In each iteration of the worklist loop, the worker selects one bad triangle from the work list, forms a *cavity* around it, re-triangulates the cavity, and adds any new bad triangles

back to the worklist. Cavity finding and re-triangulating code sections access the shared mesh data structure and are enclosed in atomic blocks.

**OO7:** OO7 simulates a number of clients, each performing a fixed number of queries on an in-memory database. Each query is enclosed in an atomic block. The performance metric is the throughput (queries per unit time), and we measure how this scales by varying the number of clients while keeping the number of queries performed by each one constant. The program uses a `foreach_nd` loop, with one iteration corresponding to each client. We configured it to use a number of clients equal to the number of worker threads, so there is always one thread per client. Thus, the total amount of work performed is proportional to the number of threads.

**Summary:** We successfully expressed all the parallelism that did not use data races in these four nondeterministic algorithms. As discussed above, we eliminated a race in TSP-PQ that was presumably there to avoid synchronization; we could have also written TSP-R with a similar race. The four codes do not use any deterministic algorithms but such algorithms do not incur any runtime performance overheads in our language; such overheads are dominated by that of atomic sections in nondeterministic components. We studied the performance and expressivity of the language for deterministic algorithms in Chapter 3.

### 5.5.2 Performance

To evaluate performance, we measured the self-relative speedup (i.e., the speedup compared to running the transactional code on one thread) achieved by the three codes. We focused on self-relative speedup rather than absolute speedup because (a) optimizing the code generation for atomic statements has not been a focus of this thesis, and (b) the Deuce STM, although using a good *algorithm*, lacks many many essential performance features of a high performance Java STM [107]. Self-relative speedups have the effect of "factoring out" some of the performance impact of the STM implementation while capturing the scalability of the benchmarks.

We ran and measured the codes on a 24-core system using four Intel Xeon E7450 processors (each with six cores), running Windows Server 2008. Figure 5.7 shows the self-relative speedups with barrier optimizations enabled, using running times for Delaunay and TSP, and throughput scaling for OO7. Because the runtimes are nondeterministic, we averaged 5–10 runs for each data point, using an interquartile method to exclude a few extreme outliers. For both TSP variants, we used the one-thread version of TSP-PQ,

which was the faster of the two, as the baseline. Both versions of TSP show good scaling, and OO7 shows moderately good scaling, throughout the range of numbers of threads $t$ we examined. TSP-R shows better (superlinear) speedup for smaller $t$; this is because the parallel algorithm is very efficient in that range: it rules out subtrees quickly, and so visits only about $1/4$ of the tree nodes at $t = 2$ compared to $t = 1$. However, the scaling curve for TSP-R flattens out as $t$ increases, most likely due to higher contention than TSP-PQ.

The speedup curve for Delaunay is poor: it flattens out and reaches only 3x on 22 threads. We profiled the code to understand the source of this behavior and traced it to the method `System.identityHashcode()` in the JVM. This standard Java function is extensively used in Deuce to index into lock tables. We observed that the time spent in this function grows with the number of threads. In Delaunay, which has large transactions, this overhead negatively affected the speedup curve. This problem can by solved by modifying the JVM, but we leave that (and other optimizations for `atomic`) to future work.

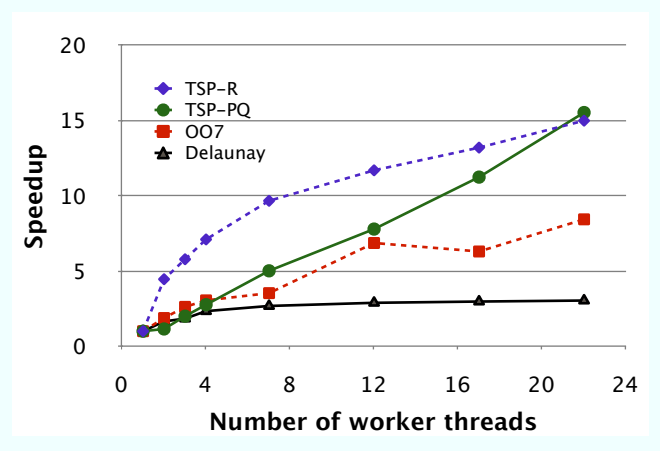


Figure 5.7: Self-relative speedups. For OO7, we scaled the amount of work with the number of worker threads, and measured speedup based on throughput scaling (number of queries done per unit time). The barrier optimization was enabled for all of these benchmarks.

### 5.5.3 Impact of Barrier Elimination

We compared the performance of two versions of the parallel code for each benchmark: with and without the barrier simplification optimization for non-atomic regions. Figure 5.8 shows the improvement in running time for the optimized code compared to the unoptimized code. Figure 5.9 shows the reduction in the number of dynamically-executed barriers due to our optimizations.

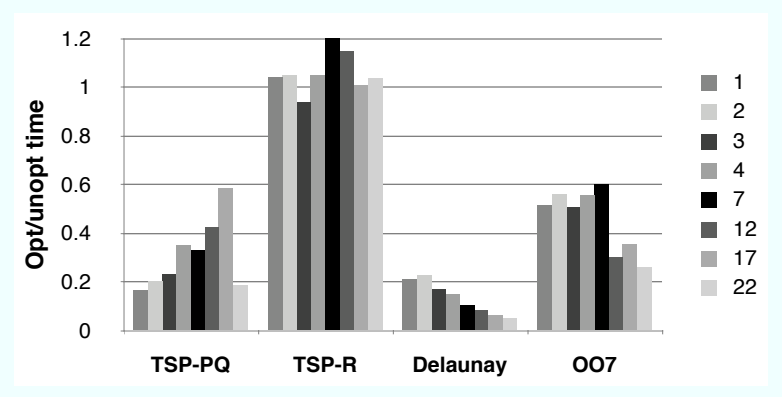


Figure 5.8: Ratio of optimized runtimes (with barrier elimination) to unoptimized runtimes (without barrier elimination). A value lower than 1 means the optimization increased performance.

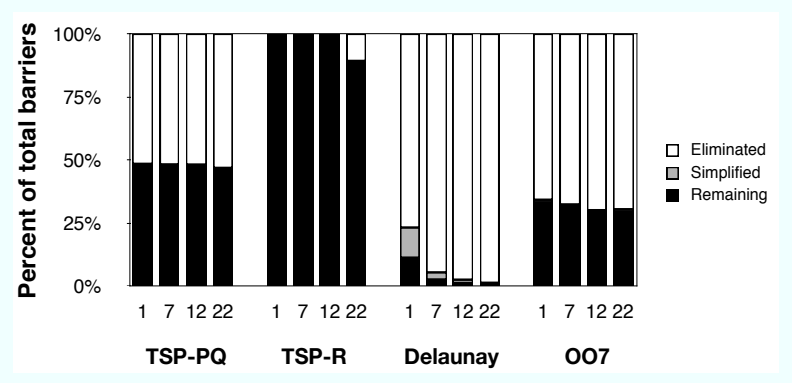


Figure 5.9: Reduction in barriers due to optimizations, showing the proportion of barriers from the unoptimized version that are eliminated entirely, simplified to log-only write barriers, or that remain as full barriers in the optimized version, for each of the three benchmarks with 1, 7, 12, and 22 worker threads.

The optimization has a substantial impact on performance for three of the four benchmarks (TSP-PQ, Delaunay, and OO7). The performance improvements correlate well with the barrier reductions. The optimizations give essentially no improvement for TSP-R, because the transactions are very short (reads and read-modify-write operations on the best tour). As a result (1) there are few if any barriers to remove; and (2) transactional overhead is not a significant component of the overall runtime. On the other hand, TSP-PQ, OO7, and Delaunay use longer transactions, providing more opportunities for reducing overhead.

Our optimizations can eliminate barriers both by actually removing barrier operations on certain statements and also by reducing the number of times that transactions must be retried. The latter effect occurs because removing unnecessary barriers reduces the number of false conflicts incurred by the STM system. As shown in Table 5.1, this effect is more pronounced with larger numbers of worker threads, so our optimizations not only reduce scalar overheads but also improve scalability. For example, in Delaunay, the

| | Delaunay | | OO7 | |
|---|---|---|---|---|
| threads | opt | unopt | opt | unopt |
| 2 | 0.999 | 0.944 | 0.944 | 0.932 |
| 3 | 0.975 | 0.848 | 0.877 | 0.872 |
| 4 | 0.998 | 0.810 | 0.822 | 0.560 |
| 7 | 0.993 | 0.647 | 0.700 | 0.210 |
| 12 | 0.996 | 0.405 | 0.539 | 0.100 |
| 17 | 0.995 | 0.291 | 0.442 | 0.071 |
| 22 | 0.994 | 0.244 | 0.369 | 0.071 |

b

Table 5.1: Ratio of committed transactions to started transactions for Delaunay and OO7. Lower numbers indicate more aborted transactions. For both versions of TSP, all numbers are 1.000.

| **Program** | **Total SLOC** | **Annotated SLOC** | **Region Decls** | **RPLs** | **Params** | **Effect Summ.** |
|---|---|---|---|---|---|---|
| TSP-PQ | 433 | 77 (17.8%) | 2(1) | 101(4) | 6(2) | 14/20 |
| TSP-R | 200 | 34 (17%) | 2(1) | 42(4) | 2(0) | 6/12 |
| OO7 | 1570 | 105 (6.7%) | 4(1) | 76(7) | 6(0) | 52/104 |
| Delaunay | 1994 | 302 (15.1%) | 3(1) | 374(3) | 21(7) | 165/216 |
| **Total** | 4197 | 518 (12.3%) | 11(4) | 593(18) | 35(9) | 237/352 |

Table 5.2: Annotation counts for the four benchmarks. In the middle columns, the numbers in parentheses represent the number of annotations marked `atomic`. In the last column, $x/y$ means of $y$ total method definitions in the program, $x$ were annotated with effect summaries.

optimization changed this ratio from 0.944 to 0.999 on 2 threads but from 0.244 to 0.944 on 22 threads.

### 5.5.4 Annotation Overhead

Table 5.2 provides a quantitative measure of the annotation overhead of writing the four benchmarks in our language. Column 1 after the vertical bar shows the total number of non-blank, non-comment lines of source code, counted by `sloccount`. Column 2 gives the count of annotated lines, as an absolute number and as a percentage of the total lines. The following three columns show the number of region declarations, RPLs (including arguments to `in`, arguments to types and methods, and arguments to effect summaries), and region parameters. The number of annotations marked `atomic` is shown in parentheses after the main number. The last column shows the number of effect summaries before the slash, and the number of method definitions after the slash.

While the average number of annotated lines (12.3%) is nontrivial, we believe it is not unduly high, given the strong safety properties of the programming model. As in our prior work [23], most of the RPL annotations were arguments to types. The overhead could be reduced by inferring some of the annotations [122],

but we leave that for future work.

Our approach does impose the limitation that if a programmer wishes to use a class region parameter as an atomic region in some context and a non-atomic region in some other context, then the class must be *cloned*: the programmer must create two copies of the class, one with the atomic parameter and one with the non-atomic parameter. The cloning is required because different barriers must be generated for methods of the class that operate transactionally on the parameter, depending on whether the region bound to the parameter is atomic. The cloning could be done automatically by the compiler, similarly to what C++ does for templates. While we have not implemented this approach, we believe it does not raise any significant technical issues.

In the benchmarks we studied, only Delaunay required class cloning. In Delaunay, we needed both atomic and non-atomic versions of the list and map structures used in the benchmark.

## 5.6 Related Work

We group the related work into five categories: type and effect systems; language support for STM correctness; compiler and runtime support for STM correctness; compiler and runtime support for optimizing STM overheads; and general work on nondeterministic parallel programming models.

**Type and Effect Systems:** Several researchers have described effect systems for enforcing a locking discipline in nondeterministic programs that prevents data races and deadlocks [24, 6, 68] or guarantees isolation for critical sections [52]. Matsakis et al. [85] have recently proposed a type system that guarantees race-freedom for locks and other synchronization constructs using a construct called an "interval" for expressing parallelism. While there is some overlap with our work in the guarantees provided (race freedom, deadlock freedom, and isolation), the mechanisms are very different (explicit synchronization vs. atomic statements supported by STM). Further, these systems do not provide determinism by default. Finally, there is no other effect system we know of that provides both race freedom *and* strong isolation together.

**STM Correctness (Language):** STM Haskell [64] provides an isolation guarantee, but for a pure functional language that uses monads to limit effects to the transactional store, unlike our imperative shared-memory language. Moore and Grossman [91] and Abadi et al. [5] use types and effects to guarantee strong isolation for an imperative language, but their languages permit races where neither access occurs in a transaction. Finally, none of these languages allows both transactional and non-transactional effects to the

same memory, as our language does.

Beckman et al. [15] show how to use a form of alias control called *access permissions* [26] to verify that the placement of atomic blocks in a threaded program respects the invariants of a specification written by the programmer — for example, that a condition is checked and acted upon atomically. This approach is complementary to ours: we provide guarantees of race freedom, strong isolation, and determinism by default for all programs in our language; on top of that one could check that additional programmer-specified invariants are satisfied.

**STM Correctness (Compiler and Runtime):** Several STMs guarantee strong isolation by preventing interference between transactions and non-transactional accesses at runtime. Most of these systems use a combination of sophisticated static whole-program analysis, runtime optimizations, and other runtime techniques like page protection to optimize strong isolation [108, 107, 28, 7]. While these techniques can significantly reduce the cost of strong isolation, they cannot completely eliminate it. In contrast, our language-based approach provides strong isolation without imposing extra runtime overhead.

**Reducing STM Overheads:** Much research has been devoted to reducing the cost of compiler-generated STM barriers on transactional memory accesses. Early work [8, 65] showed how to eliminate several classes of transactional overhead including redundant barriers, barriers for accesses to provably immutable memory locations, and certain barriers for accesses to objects allocated in a transaction. Recent work by Afek et al. [9] uses the logic of program reads and writes within a transaction to reduce STM overhead: for example, a shared variable that is read several times can be be read once and cached locally. These optimizations complement ours, as they target different kinds of STM overhead from our work.

Beckman et al. [16] show how to use access permissions to remove STM synchronization overhead. While the goals are the same as ours, the mechanisms are different (alias control vs. type and effect annotations). The two mechanisms have different tradeoffs in expressivity and power: for example, Beckman et al.'s method can eliminate write barriers only if an object is accessed through a unique reference, whereas our system can eliminate barriers for access through shared references, so long as the access does not cause interfering effects. However, alias restrictions can express some patterns (such as permuting unique references in a data structure) that our system cannot. As future work, it would be interesting to explore these tradeoffs further.

Finally, several researchers have eliminated STM overhead for accesses to thread-local data using whole-

program static escape analysis [108] and programmer annotations to specify code blocks that do not require instrumentation [129]. Unlike our work, this work either requires whole-program analysis, or it relies on unverified programmer annotations.

**Nondeterministic Parallel Programming:** Several research efforts are developing parallel models for nondeterministic codes with irregular data access patterns, such as Delaunay mesh refinement. Galois [74] provides a form of isolation, but with iterations of parallel loops (instead of atomic statements) as the isolated computations. Concurrency is increased by detecting conflicts at the level of method calls, instead of reads and writes, and using semantic commutativity properties. Lublinerman et al. [81] have proposed *object assemblies* as an alternative model for expressing irregular, graph-based computations.

These models are largely orthogonal to our work. In Galois, strong isolation holds if all shared data is accessed through well-defined APIs, but this property is not enforced, either statically or at runtime. We believe that our type and effect mechanisms could be applied to Galois to ensure this property. The object assemblies model may have stronger isolation guarantees than Galois, but it is very specialized to irregular graph computations, in contrast to the more general fork-join model we present here.

Kulkarni et al. [73] have recently proposed *task types* as a way of enforcing a property they call *pervasive atomicity*. This work shares with ours the broad goal of reducing the number of concurrent interleavings the programmer must consider. However, Kulkarni et al. adopt an actor-inspired approach, in which data is non-shared by default, and sharing musk occur through special "task objects." This is in contrast to our approach of allowing familiar shared-memory patterns of programming, but using effect annotations to enforce safety properties. Finally, none of the work discussed above provides any deterministic-by-default guarantee.

# Chapter 6

# Formal Language for Determinism by Default

To make precise the ideas discussed in the previous chapter, this chapter presents three variants of the same formal language, each one building on the last:

1. The first variant, which we call the *simplified deterministic language*, is the same as Core DPJ discussed in Chapter 4, with two exceptions. First, we have simplified the effect system to focus on the important elements for this Chapter. In particular, we simplify the RPLs, and we omit arrays, variable regions, and commutativity annotations. Second, the simplified deterministic language explicitly models parallel execution.

2. The second variant, which we call the *deterministic-by-default language*, adds nondeterministic parallel composition, atomic expressions, and atomic effects to the simplified deterministic language. These features formalize the new language features introduced in the previous chapter.

3. The third variant, which we call the *atomic regions language*, adds atomic regions for removing or simplifying transactional barriers.

Without loss of generality, we include `cobegin` and `cobegin_nd` only in these simple languages; the treatment for `foreach` and `foreach_nd` is similar.

## 6.1 Overview of Language Variants

We first explain the syntactic structure of all three languages, and we summarize the soundness guarantees that each one provides. In the following subsections, we explain the formal semantics of each language variant, state the soundness guarantees more formally, and prove that the guarantees follow from the semantic definitions.

| | | | |
|---|---|---|---|
| **Programs** | $\mathcal{P}$ | ::= | $\mathcal{R}^*\ \mathcal{C}^*\ e$ |
| **Classes** | $\mathcal{C}$ | ::= | `class` $C$`<`$\rho$`>` $\{\ F^*\ M^*\ \}$ |
| **Region Names** | $\mathcal{R}$ | ::= | `region` $r$ |
| **Fields** | $F$ | ::= | $T\ f$ `in` $R$ |
| **Methods** | $M$ | ::= | $T\ m(T\ x)\ E\ \{\ e\ \}$ |
| **Regions** | $R$ | ::= | $r \mid \rho$ |
| **Types** | $T$ | ::= | $C$`<`$R$`>` |
| **Effects** | $E$ | ::= | $\emptyset \mid$ `reads` $R \mid$ `writes` $R \mid E \cup E$ |
| **Expressions** | $e$ | ::= | `this.`$f \mid$ `this.`$f$`=`$e \mid e$`.`$m$`(`$e$`)` $\mid v \mid$ `new` $T \mid$ `seq(`$e$`,`$e$`)` $\mid$ `cobegin(`$e$`,`$e$`)` |
| **Variables** | $v$ | ::= | `this` $\mid x$ |

Figure 6.1: Syntax of the simplified deterministic language. $C$, $\rho$, $f$, $m$, and $x$ are identifiers.

**Simplified deterministic language:** Figure 6.1 gives the syntax of the simplified deterministic language. A program $\mathcal{P}$ consists of zero or more region declarations, zero or more class definitions, and an expression to evaluate. A class $\mathcal{C}$ consists of a class name $C$, a region parameter $\rho$, zero or more field declarations, and zero or more method declarations. A field $F$ specifies a type, a field name, and a region. A method $M$ consists of a return type, a method name, a formal parameter type, a formal parameter, an effect, and an expression to evaluate. A region $R$ is either a region name $r$ or a region parameter $\rho$. A type $T$ is a class instantiated with a region parameter, $C$<$R$>. An effect $E$ is a possibly empty union of read effects and write effects on regions.

For expressions $e$, we model field access, field assignment, method invocation, variables, new objects, sequential composition (`seq`), and deterministic parallel composition (`cobegin`). A variable $v$ is `this` or a method formal parameter $x$. The operational semantics of the first five expressions in Figure 4.1 is exactly as in Java. The last two expressions evaluate both component expressions (either sequentially or in parallel) and return the value of the second component as the value of the entire expression.

The simplified deterministic language provides the following semantic guarantees, stated more formally as Theorems 6.2.9 and 6.2.10 in Section 6.2. They follow from the fact that the executions of the two branches of any `cobegin` expression are required to be noninterfering:

1. *Equivalence of* `cobegin` *and* `seq`*:* In terms of the final result (final value produced and final heap state), there is no difference between executing `cobegin(`$e$`,`$e'$`)` and `seq(`$e$`,`$e'$`)`. As a consequence, the entire program is guaranteed to behave like a sequential program (the one that results by replacing `cobegin` everywhere with `seq`).

2. *Determinism:* If an expression $e$ evaluates to completion, then the value it produces is deterministic.

| | | | |
|---|---|---|---|
| **Effects** | $E$ | ::= | ... \| `atomic reads` $R$ \| `atomic writes` $R$ |
| **Expressions** | $e$ | ::= | ... \| `cobegin_nd(`$e$`,`$e$`)` \| `atomic` $e$ |

Figure 6.2: Syntax of the deterministic-by-default language (extends Figure 6.1).

Moreover, the final state of the heap locations accessed by $e$ is deterministic, and if $e$ is evaluated in a sequential context (i.e., not inside a `cobegin`), then the entire final heap state is deterministic.[1] In particular, the final heap produced by a terminating execution of the whole program is deterministic.

**Deterministic-by-default language:** Figure 6.2 shows the additional syntax for the deterministic-by-default language. We extend the syntax of effects to record `atomic` effects. We also add (1) `cobegin_nd`, which is the same as `cobegin`, except that it allows interference guarded by atomic expressions; and (2) expressions `atomic` $e$, which signal that expression $e$ should be executed in *isolation*, that is, as if it were executed all at once, with no interleavings from the rest of the execution.

The deterministic-by-default language provides the following semantic guarantees, stated more formally as Theorems 3–6 in Section 6.3:

1. *Race freedom and sequential consistency:* Program execution contains no data race. This result follows because the effect system requires that all parallel interference occur between pairs of accesses guarded by atomic expressions. Further, in the Java memory model, race freedom implies sequential consistency, i.e., one can reason about execution as a *program-ordered* interleaving of memory operations.

2. *Strong isolation:* For the same reason that the program is race free, expressions `atomic` $e$ execute $e$ in isolation, *even if the underlying implementation guarantees only weak isolation*. Moreover, the effect system disallows any interference between the `cobegin` and concurrent operations that would violate isolation of the `cobegin`. Therefore, every `cobegin` expression executes in isolation. Together, race freedom and strong isolation imply that execution is a sequentially consistent interleaving *of isolated expressions*.

3. *Equivalence of `cobegin` and `seq`*: Because `cobegin(`$e$`,`$e'$`)` executes in isolation, it is equivalent

[1] If $e$ is evaluated in a parallel context, then the state of other locations not accessed by $e$ depends on the scheduling of the parallel computations that access those locations. For example, suppose $e$ changes the value of variable `x` from 0 to 1, $e'$ changes the value of variable $y$ from 0 to 1, and $e$ and $e'$ are executed in parallel. Then at the end of executing $e$ the value of `x` will always be 0; but the value of `y` at that point will depend on whether $e'$ has executed yet.

| | | | |
|---|---|---|---|
| **Regions** | $\mathcal{R}$ | ::= | ... \| `atomic region` $r$ |
| **Classes** | $\mathcal{C}$ | ::= | ... \| `class` $C$`<atomic` $\rho$`>` $\{ F^* \; M^* \}$ |

Figure 6.3: Syntax of the atomic regions language (extends Figure 6.2).

to an isolated execution of `seq`, i.e., `atomic seq(`$e$`,` $e'$`)`. For the deterministic-by-default language, we make `cobegin` behave like `atomic seq`, and not just `seq`, to guarantee that `cobegin` executes deterministically, *even inside a* `cobegin_nd`.

4. *Determinism by default:* Both atomic and cobegin expressions execute deterministically in the same sense as discussed for the simplified deterministic language, *even inside a* `cobegin_nd`, *unless* they contain a dynamic instance of `cobegin_nd`.

**Atomic regions language:** The third variant of the formal language allows some regions to be marked `atomic`, and *only operations on those regions generate atomic effects*. Operations on non-atomic regions *never generate atomic effects, even in an atomic expression.* Figure 6.3 shows the new syntax.

The execution semantics of this language variant is identical to that of the deterministic-by-default language, except that the compiler can distinguish, and potentially optimize, operations within an atomic expression that never interfere with concurrent tasks. In Section 5.4, we discussed a prototype compiler that uses these rules to optimize our STM by omitting or simplifying barriers (inside an atomic expression) for such noninterfering operations.

## 6.2 Simplified Deterministic Language

This section gives the formal semantics and soundness results for the simplified deterministic language.

### 6.2.1 Static Semantics

The typing is with respect to an environment $\Gamma$, which is a possibly empty union of elements $(v, T)$ stating that variable $v$ has type $T$:

$$\Gamma ::= \emptyset \mid (v, T) \mid \Gamma \cup \Gamma$$

**Programs and Classes:** The rules for valid programs and classes are nearly identical to those given in

Chapter 4, except that in this simplified language there are no commutativity annotations.

$$\boxed{\vdash \mathcal{P}} \quad \text{PROGRAM}\ \frac{\forall \mathcal{C}.(\vdash \mathcal{C}) \quad \emptyset \vdash e : T, E}{\vdash \mathcal{C}^*\ e} \qquad \boxed{\vdash \mathcal{C}} \quad \text{CLASS}\ \frac{\Gamma = (\texttt{this}, C\texttt{<}\rho\texttt{>}) \quad \forall F.(\Gamma \vdash F) \quad \forall M.(\Gamma \vdash M)}{\vdash \texttt{class}\ C\texttt{<}\rho\texttt{>}\ \{\ F^*\ M^*\ \}}$$

$$\boxed{\Gamma \vdash F} \quad \text{FIELD}\ \frac{\Gamma \vdash T \quad \Gamma \vdash R}{\Gamma \vdash T\ f\ \texttt{in}\ R} \qquad \boxed{\Gamma \vdash M} \quad \text{METHOD}\ \frac{\Gamma \vdash T_r \quad \Gamma \vdash T_x \quad \Gamma \vdash E \quad \Gamma \cup (x, T_x) \vdash e : T_r, E' \quad \Gamma \vdash E' \subseteq E}{\Gamma \vdash T_r\ m(T_x\ x)\ E\ \{\ e\ \}}$$

**Regions and Types:** The rules for valid regions and types are simplified from Chapter 4. There are no rules for subtyping in the static semantics, because in this simplified language, we require identity of types at assignments.

$$\boxed{\Gamma \vdash R} \quad \text{REGION-NAME}\ \frac{\texttt{region}\ r \in \mathcal{P}}{\Gamma \vdash r} \qquad \text{REGION-PARAM}\ \frac{(\texttt{this}, C\texttt{<}\rho\texttt{>}) \in \Gamma}{\Gamma \vdash \rho}$$

$$\boxed{\Gamma \vdash T} \quad \text{TYPE}\ \frac{\texttt{class}\ C\texttt{<}\rho\texttt{>}\ \{\ F^*\ M^*\ \} \in \mathcal{P} \quad \Gamma \vdash R}{\Gamma \vdash C\texttt{<}R\texttt{>}}$$

**Valid effects:** The rules for valid effects, subeffects, and noninterfering effects are also a subset of the rules from Chapter 4. Again, the subeffect relation is reflexive and transitive, and the noninterference relation is symmetric (obvious rules omitted).

$$\boxed{\Gamma \vdash E} \quad \text{EFFECT-EMPTY}\ \frac{}{\Gamma \vdash \emptyset} \qquad \text{EFFECT-READS}\ \frac{\Gamma \vdash R}{\Gamma \vdash \texttt{reads}\ R} \qquad \text{EFFECT-WRITES}\ \frac{\Gamma \vdash R}{\Gamma \vdash \texttt{writes}\ R} \qquad \text{EFFECT-UNION}\ \frac{\Gamma \vdash E \quad \Gamma \vdash E'}{\Gamma \vdash E \cup E'}$$

$$\boxed{\Gamma \vdash E \subseteq E'} \quad \text{SE-EMPTY}\ \frac{}{\Gamma \vdash \emptyset \subseteq E} \qquad \text{SE-READS-WRITES}\ \frac{}{\Gamma \vdash \texttt{reads}\ R \subseteq \texttt{writes}\ R}$$

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

$\boxed{\Gamma \vdash E \# E'}$

NI-EMPTY
$$\frac{}{\Gamma \vdash \emptyset \# E}$$

NI-READS
$$\frac{}{\Gamma \vdash \texttt{reads}\ r \# \texttt{reads}\ r'}$$

NI-WRITES
$$\frac{r \neq r'}{\Gamma \vdash \texttt{writes}\ r \# \texttt{writes}\ r'}$$

NI-READS-WRITES
$$\frac{r \neq r'}{\Gamma \vdash \texttt{reads}\ r \# \texttt{writes}\ r'}$$

NI-UNION
$$\frac{\Gamma \vdash E \# E'' \quad \Gamma \vdash E' \# E''}{\Gamma \vdash E \cup E' \# E''}$$

**Expressions:** As in Chapter 4, every expression has a type and an effect. The judgment $\Gamma \vdash e : T, E$ means that expression $e$ is well typed with type $T$ and effect $E$ in environment $\Gamma$.

*Basic operations from Chapter 4:* Field access and assignment, method invocation, variable access, and new objects all work the same way as described in Chapter 4. There are no arrays or `let` expressions in this language.

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

FIELD-ACCESS
$$\frac{(\texttt{this}, C\texttt{<}\rho\texttt{>}) \in \Gamma \quad F(C, f) = T\ f\ \texttt{in}\ R}{\Gamma \vdash \texttt{this}.f : T, \texttt{reads}\ R}$$

FIELD-ASSIGN
$$\frac{(\texttt{this}, C\texttt{<}\rho\texttt{>}) \in \Gamma \quad \Gamma \vdash e : T, E \quad F(C, f) = T\ f\ \texttt{in}\ R}{\Gamma \vdash \texttt{this}.f\texttt{=}e : T, E \cup \texttt{writes}\ R}$$

INVOKE
$$\frac{\Gamma \vdash e : C\texttt{<}R\texttt{>}, E \quad \Gamma \vdash e' : \sigma_{C\texttt{<}R\texttt{>}}(T_x), E' \quad M(C, m) = T_r\ m(T_x\ x)\ E_m\ \{\ e''\ \}}{\Gamma \vdash e.m\texttt{(}e'\texttt{)} : \sigma_{C\texttt{<}R\texttt{>}}(T_r), E \cup E' \cup \sigma_{C\texttt{<}R\texttt{>}}(E_m)}$$

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

NEW
$$\frac{\Gamma \vdash T}{\Gamma \vdash \texttt{new}\ T : T, \emptyset}$$

In rule INVOKE, the substitution $\sigma_{C\texttt{<}R\texttt{>}}$ substitutes $R$ for the region parameter $\rho$ of class $C$.

*Sequential composition:* To type a sequential composition expression `seq(`$e$`,`$e'$`)`, we type the component expressions $e$ and $e'$. The type of the expression is the type $T'$ of $e'$. The effect is the union of the effects $E$ and $E'$ of $e$ and $e'$.

$\boxed{\Gamma \vdash e : T, E}$ SEQ
$$\frac{\Gamma \vdash e : T, E \quad \Gamma \vdash e' : T', E'}{\Gamma \vdash \texttt{seq(}e\texttt{,}e'\texttt{)} : T', E \cup E'}$$

*Deterministic parallel composition:* Typing deterministic parallel composition `cobegin(`$e$`,`$e'$`)` is the same as for sequential composition, except that we require the effects of $e$ and $e'$ to be noninterfering.

$$\boxed{\Gamma \vdash e : T, E} \quad \textsc{Cobegin} \quad \frac{\Gamma \vdash e : T, E \quad \Gamma \vdash e' : T', E' \quad \Gamma \vdash E \# E'}{\Gamma \vdash \texttt{cobegin(}e\texttt{,}e'\texttt{)} : T', E \cup E'}$$

### 6.2.2 Dynamic Semantics

**Execution Environment:** We extend the static expression syntax (Section 6.1), to represent computations. In addition to an expression from the static syntax, an expression can also be an object reference $o$; or a local execution state $(e, \Sigma, E)$, where $\Sigma$ is a dynamic environment (defined below); or $e_i$, where $i$ is a unique identifier:

$$e ::= \ldots \mid o \mid (e, \Sigma, E) \mid e_i$$

The additional expressions have the following meanings:

- Object references $o$ are the values produced by reducing expressions.
- A local execution state $(e, \Sigma, E)$ records an expression $e$ to evaluate, an environment $\Sigma$ containing the bindings for the free variables in $e$, and the effect $E$ of reducing $e$.
- The identifiers $i$ are "tags" that allow us to refer unambiguously to subexpressions that are undergoing reduction.

Reducing local execution state "in place" allows us to retain the recursive structure of an execution history, as opposed to, e.g., flattening everything into threads. This in turn makes it easier to state and prove the desired properties of the execution.

As in Chapter 4, we define a dynamic environment $\Sigma$ that maps variables $v$ to references $o$:

$$\Sigma ::= \emptyset \mid (v, o) \mid \Sigma \cup \Sigma$$

An object $O$ is a mapping from field names $f$ to object references $o$:

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

A heap $H$ is a partial function from object references $o$ to pairs $(O, T)$, where $O$ is an object, and $T$ is the type of $O$:

$$H ::= \mathsf{null} \mid o \mapsto (O, T) \mid H \cup H$$

$\mathsf{null}$ is a special reference that is in $\mathsf{Dom}(H)$ but does not map to an object. Attempting to invoke a method of $\mathsf{null}$ causes the execution to fail.

**Program Execution:** We describe program execution as a small-step operational semantics. The execution state is $(e, H)$, consisting of an expression to evaluate and a heap. Program execution is defined by the transition relation

$$(e, H) \rightarrow (e', H').$$

Informally, a single step transforms an expression into another expression and updates the heap. The rules defining this relation are stated below. Formally, if $\mathcal{P} = \mathcal{C}^*\ e_{\mathcal{P}}$, then an execution of program $\mathcal{P}$ is given by

$$((e_{\mathcal{P}}, \emptyset, \emptyset)_i, \mathsf{null}) \rightarrow^* (e_i, H),$$

for some $i$, $e_i$, and $H$, where $i$ is an arbitrary index denoting the top-level expression in the reduction, $e_i$ is the evolution of expression $(e_{\mathcal{P}}, \emptyset, \emptyset)_i$, and $H$ is the evolved heap (represented as a domain containing `null` plus all object references $o$ added during the execution). A terminating execution has $e_i = (o, \emptyset, E)_i$, where $o$ is the "answer" computed by the program, and $E$ is the union of all effects on $H$ done in the execution.

**Operational Semantics of Expressions:** First, we need a standard rule for evaluating subexpressions:

$$\textsc{Dyn-Subexp}\quad \frac{(e, H) \rightarrow (e', H')}{(e'', H) \rightarrow (e''[e_i \leftarrow e_i'], H')}$$

Rule DYN-SUBEXP says that if we know how to evaluate expression $e$ to $e'$ with respect to heap $H$, and $e$ appears as a subexpression of $e''$ with index $i$, then we can reduce $e''$ by rewriting the subexpression $e_i$ in place and updating the heap. Because there are in general several subexpressions $e_i$ that are eligible for rewriting in this way, we use the indices $i$ to identify which one is being reduced. The choice of which one to reduce next is, in general, nondeterministic.

Next we give the rules for evaluating local state expressions $(e, \Sigma, E)$. For expressions with nontrivial subexpressions, we (1) convert the subexpressions $e'$ of $e$ into the form $(e', \Sigma', \emptyset)_i$, where $i$ is a fresh identifier; (2) evaluate the subexpressions to $(o, \Sigma', E)_i$ using rule DYN-SUBEXP; and (3) use the results to finish evaluating the main expression $e$. Notice that in the subexpression evaluation, $e'$ evaluates to $o$, $\Sigma'$ is unchanged, and the effect changes from $\emptyset$ to some effect $E$. An expression reduction always starts out as $(e, \Sigma, \emptyset)$, where $e$ is an expression in the program text; goes through various transformations; and ends up as $(o, \Sigma, E)$, where $o$ is an object reference, and $E$ collects the effects of the reduction.

*Field access:* For field access `this.`$f$, we read the value of field $f$ out of the heap and record the read effect:

DYN-FIELD-ACCESS

$$\frac{(\texttt{this}, o) \in \Sigma \quad H(o) = (O, C\texttt{<}r\texttt{>}) \quad F(C, f) = T\ f\ \texttt{in}\ R}{((\texttt{this.}f, \Sigma, \emptyset), H) \rightarrow ((O(f), \Sigma, \texttt{reads}\ \sigma_{C\texttt{<}r\texttt{>}}(R)), H)}$$

In computing the effect, rule DYN-FIELD-ACCESS uses the substitution $\sigma_{C\texttt{<}r\texttt{>}}$ defined in Section 6.2.1.

*Field assignment:* For field assignment `this.`$f$`=`$e$, we evaluate $e$, then update the heap and record the write effect:

DYN-FIELD-ASSIGN-EVAL

$$\frac{\mathsf{fresh}(i)}{((\texttt{this.}f\texttt{=}e, \Sigma, \emptyset), H) \rightarrow (\texttt{this.}f\texttt{=}(e, \Sigma, \emptyset)_i, H)}$$

DYN-FIELD-ASSIGN-UPDATE

$$\frac{(\texttt{this}, o) \in \Sigma \quad H(o) = (O, C\texttt{<}r\texttt{>}) \quad F(C, f) = T\ f\ \texttt{in}\ R}{((\texttt{this.}f\texttt{=}(o', \Sigma, E)_i, H) \rightarrow (o', \Sigma, E \cup \texttt{writes}\ \sigma_{C\texttt{<}r\texttt{>}}(R), H[o \mapsto (O[f \mapsto o'], C\texttt{<}r\texttt{>})])}$$

$\mathsf{fresh}(i)$ means that $i$ is a fresh identifier. $f[a \mapsto b]$ denotes the function identical to $f$ everywhere on its domain, except that it maps $a$ to $b$.

*Method invocation:* For method invocation $e.m(e')$, we evaluate $e$ to $o$, evaluate $e'$ to $o'$, execute the method body in the environment with the correct method parameter bindings, and accumulate the results. The aggregate effect of the expression is the union of the effects of evaluating the two arguments and the

method body.

DYN-INVOKE-EVAL-THIS

$$\frac{\mathsf{fresh}(i)}{((e\texttt{.}m\texttt{(}e'\texttt{)}, \Sigma, \emptyset), H) \to ((e, \Sigma, \emptyset)_i\texttt{.}m\texttt{(}e'\texttt{)}, H)}$$

DYN-INVOKE-EVAL-ARG

$$\frac{\mathsf{fresh}(j)}{((o, \Sigma, E)_i\texttt{.}m\texttt{(}e\texttt{)}, H) \to ((o, \Sigma, E)_i\texttt{.}m\texttt{(}(e, \Sigma, \emptyset)_j\texttt{)}, H)}$$

DYN-INVOKE-EVAL-METHOD-BODY

$$\frac{H(o) = (O, C\texttt{<}r\texttt{>}) \quad M(C, m) = T_r\ m(T_x\ x)\ E_m\ \{\ e\ \} \quad \mathsf{fresh}(k)}{((o, \Sigma, E)_i\texttt{.}m\texttt{(}(o', \Sigma, E')_j\texttt{)}, H) \to (((e, (\texttt{this}, o) \cup (x, o'), \emptyset)_k, \Sigma, E \cup E'), H)}$$

DYN-INVOKE-ACCUMULATE

$$\frac{}{(((o, \Sigma, E)_i, \Sigma', E'), H) \to ((o, \Sigma', E \cup E'), H)}$$

Notice that if $e$ evaluates to `null`, then DYN-INVOKE-EVAL-METHOD-BODY cannot be applied. This is effectively a "null dereference." We don't model error state or exceptions explicitly; instead a null dereference just means that we have reached a "stuck state," such that no successful program execution including that state is possible. All of our soundness results (Section 6.2.3) are stated in terms of successful executions.

*Variable access:* For variables $v$, we just read the value out of the environment:

DYN-VARIABLE

$$\frac{(v, o) \in \Sigma}{((v, \Sigma, \emptyset), H) \to ((o, \Sigma, \emptyset), H)}$$

*New objects:* For `new` we create a new object of the correct type, initialize its fields to null, and add a reference to the object to the heap:

DYN-NEW

$$\frac{o \notin \mathsf{Dom}(H) \quad O = \cup_{f \in \mathsf{fields}(C)}(f \mapsto \mathsf{null})}{((\mathsf{new}\ C\texttt{<}R\texttt{>}, \Sigma, \emptyset), H) \to ((o, \Sigma, \emptyset), H \cup o \mapsto (O, \sigma_{\Sigma,H}(C\texttt{<}R\texttt{>})))}$$

$\sigma_{\Sigma,H}$ is the following function that takes regions to regions, types to types, and effects to effects:

1. If $(\texttt{this}, o)$ does not appear in $\Sigma$ for any $o$, then $\sigma_{\Sigma,H}$ is the identity function.
2. Otherwise, if $(\texttt{this}, o) \in \Sigma$ and $H \vdash o : C\texttt{<}r\texttt{>}$, then $\sigma_{\Sigma,H}$ is $\sigma_{C\texttt{<}r\texttt{>}}$, as defined in Section 6.2.1.

Note that $\sigma_{\Sigma,H}$ is undefined if $(\texttt{this}, \texttt{null}) \in \Sigma$, but it is obvious that this cannot happen.

*Sequential composition:* For sequential composition $\texttt{seq(}e\texttt{,}e'\texttt{)}$, we first evaluate $e$ to $o$, and then we evaluate $e'$ to $o'$. We use $o'$ as the result of the whole expression ($o$ is thrown away; the evaluation of $e$ is only for its side effects). The aggregate effect of the expression is the union of the effects of executing each branch.

DYN-SEQ-EVAL-FIRST

$$\frac{\mathsf{fresh}(i)}{((\texttt{seq(}e\texttt{,}e'\texttt{)}, \Sigma), H) \rightarrow (\texttt{seq(}(e, \Sigma, \emptyset)_i\texttt{,}e'\texttt{)}, H)}$$

DYN-SEQ-EVAL-SECOND

$$\frac{\mathsf{fresh}(j)}{(\texttt{seq(}(o, \Sigma, E)_i\texttt{,}e\texttt{)}, H) \rightarrow (\texttt{seq(}(o, \Sigma, E)_i\texttt{,}(e, \Sigma, \emptyset)_j\texttt{)}, H)}$$

DYN-SEQ-ACCUMULATE

$$\frac{}{(\texttt{seq(}(o, \Sigma, E)_i\texttt{,}(o', \Sigma, E')_j\texttt{)}, H) \rightarrow ((o', \Sigma, E \cup E'), H)}$$

*Deterministic parallel composition:* For deterministic parallel composition $\texttt{cobegin(}e\texttt{,}e'\texttt{)}$, we evaluate $e$ to $o$ and $e'$ to $o'$ "in parallel," i.e., we allow the evaluation steps of the two expressions to be arbitrarily interleaved. For the accumulation step, again we use $o'$ as the result of the whole expression.

DYN-COBEGIN-EVAL

$$\frac{\mathsf{fresh}(i) \quad \mathsf{fresh}(j)}{((\texttt{cobegin(}e\texttt{,}e'\texttt{)}, \Sigma, \emptyset), H) \rightarrow (\texttt{cobegin(}(e, \Sigma, \emptyset)_i\texttt{,}(e', \Sigma, \emptyset)_j\texttt{)}, H)}$$

DYN-COBEGIN-ACCUMULATE

$$\frac{}{(\texttt{cobegin(}(o, \Sigma, E)_i\texttt{,}(o', \Sigma, E')_j\texttt{)}, H) \rightarrow ((o', \Sigma, E \cup E'), H)}$$

Notice that all rule applications are deterministic (i.e., there is only one next rule to apply) except in the case of `cobegin`. Note also that `cobegin` still provides deterministic *results*, because the type system guarantees noninterference of heap accesses between the two branches. However, the actual sequence of steps in the execution is nondeterministic.

### 6.2.3 Soundness

**Static Environments:** A static environment $\Gamma$ is valid if its elements are valid with respect to itself:

$$\textsc{Env}\quad \frac{\forall (v,T) \in \Gamma . \Gamma \vdash T}{\vdash \Gamma}$$

Our first theorem says that typing a valid expression in a valid environment produces a valid type and effect:

**Theorem 6.2.1** (Validity of static expression typing)**.** *If* $\vdash \mathcal{P}$ *and* $\vdash \Gamma$ *and* $\Gamma \vdash e : T, E$*, then* $\Gamma \vdash T$ *and* $\Gamma \vdash E$*.*

*Proof.* By induction on the structure of $e$, showing the result for each expression typing rule in Section 6.2.1.

*Base cases:* FIELD-ACCESS: Follows from the typing of class fields in Section 6.2.1.

VARIABLE: Follows from $\vdash \Gamma$.

NEW: Follows from the condition of the rule.

*Inductive cases:* FIELD-ASSIGN: Follows from the induction hypothesis applied to the typing of $e$, plus the same argument as for FIELD-ACCESS.

INVOKE: The induction hypothesis gives that $R$, $E$, and $E'$ are valid. Rule METHOD gives that $T_r$ and $E_m$ are valid in the environment in which `this` is bound to $C\texttt{<}\rho\texttt{>}$. But then $\sigma_{C\texttt{<}R\texttt{>}}(T_r)$ and $\sigma_{C\texttt{<}R\texttt{>}}(E_m)$ are valid in $\Gamma$, because $\Gamma \vdash R$, and $\sigma_{C\texttt{<}R\texttt{>}}$ substitutes $R$ for $\rho$.

SEQ: Follows directly from the induction hypothesis.

COBEGIN: Follows directly from the induction hypothesis. □

**Reference types:** The execution state includes the *null type* $\mathcal{N}$, which is the type of null:

$$T ::= \ldots \mid \mathcal{N}$$

We also add rules for typing references:

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

And because we have a null type, we need simple rules for subtyping:

$$\boxed{\Gamma \vdash T \preceq T'} \qquad \textsc{Subtype-Reflexive}\ \frac{}{\Gamma \vdash T \preceq T} \qquad \textsc{Subtype-Null}\ \frac{}{\Gamma \vdash \mathcal{N} \preceq T}$$

**Heaps:** A heap is valid if its elements are valid:

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

An object-type pair $(O, C\texttt{<}r\texttt{>})$ is valid if (1) $C\texttt{<}r\texttt{>}$ is a valid type in the empty environment; and (2) for every field $f$ of $C$, $O(f)$ is defined, its type is valid and a subtype of the static type of $f$, after substituting $r$ for the parameter of $C$:

$$\boxed{H \vdash (O,T)} \qquad \textsc{Object}\ \frac{\emptyset \vdash C\texttt{<}r\texttt{>} \quad \forall (T\ f\ \mathtt{in}\ R \in \mathsf{fields}(C)).(\emptyset \vdash \sigma_{C\texttt{<}r\texttt{>}}(T) \wedge H \vdash O(f) : T' \wedge H \vdash T' \preceq \sigma_{C\texttt{<}r\texttt{>}}(T))}{H \vdash (O, C\texttt{<}r\texttt{>})}$$

$r$ is a region name. Notice that we check the dynamic type of an object or object field in the empty environment, because all region parameters have been substituted away.

**Dynamic environments:** A dynamic environment $\Sigma$ is valid if all its elements are valid with respect to a heap $H$:

$$\boxed{H \vdash \Gamma} \qquad \textsc{Dyn-Env-Empty}\ \frac{}{H \vdash \emptyset} \qquad \textsc{Dyn-Env-Elt}\ \frac{H \vdash o : T}{H \vdash (v,o)} \qquad \textsc{Dyn-Env-Union}\ \frac{H \vdash \Gamma \quad H \vdash \Gamma'}{H \vdash \Gamma \cup \Gamma'}$$

**Instantiation of environments:** The judgment $H \vdash \Sigma \preceq \Gamma$ says that the dynamic environment $\Sigma$ *instantiates* the static environment $\Gamma$. That means the heap and both environments are valid; the variables appearing in both environments match; and the corresponding types in the environments match, after translation to the dynamic environment. Instantiation lets us use the static typing of expressions to infer that the dynamic

execution of those expressions is well-behaved.

The basic rule for instantiation just checks everything for validity and records the original dynamic environment to the left of the $\vdash$. This makes the original dynamic environment available as we dissect it to compare it to the static environment element by element:

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

Next we have the element-by-element rules:

$$\boxed{\Sigma, H \vdash \Sigma' \preceq \Gamma} \qquad \textsc{Inst-Empty}\ \frac{}{\Sigma, H \vdash \emptyset \preceq \emptyset} \qquad \textsc{Inst-Union}\ \frac{\Sigma, H \vdash \Sigma' \preceq \Gamma' \quad \Sigma, H \vdash \Sigma'' \preceq \Gamma''}{\Sigma, H \vdash \Sigma' \cup \Sigma'' \preceq \Gamma' \cup \Gamma''}$$

$$\textsc{Inst-Elt}\ \frac{H \vdash o : C'\texttt{<}r\texttt{>} \quad \emptyset \vdash C'\texttt{<}r\texttt{>} \preceq \sigma_{\Sigma,H}(C\texttt{<}R\texttt{>})}{\Sigma, H \vdash (v, o) \preceq (v, C\texttt{<}R\texttt{>})}$$

$\sigma_{\Sigma,H}$ is the translation function defined in Section 6.2.2.

**Execution state:** The judgment $\Gamma \vdash ((e, \Sigma, \emptyset), H) : T, E$ states that local execution state $((e, \Sigma, \emptyset), H)$ is valid with respect to static environment $\Gamma$ with type $T$ and effect $E$. That means $\Sigma$ instantiates $\Gamma$, and $e$ is well typed in $\Gamma$ with type $T$ and effect $E$.

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

**Preservation of Type and Effect:** In this section we prove that for successful executions, the static types and effects of expressions computed according to Section 6.2.1 approximate the dynamic types and effects produced during actual execution according to Section 6.2.2. First we need some technical lemmas. Again, $\sigma_{\Sigma,H}$ is the translation function defined in Section 6.2.2.

Assume $\Sigma$ instantiates $\Gamma$ with respect to heap $H$. If region $R$ is valid in $\Gamma$, then region $\sigma_{\Sigma,H}(R)$ is valid in the empty environment; and similarly for types and effects:

**Lemma 6.2.2.** *If $H \vdash \Sigma \preceq \Gamma$ and $\Gamma \vdash R$, then $\emptyset \vdash \sigma_{\Sigma,H}(R)$. The same result holds replacing $R$ with $T$ or $E$.*

*Proof.* For regions, the result is obvious unless $R$ is a region parameter $\rho$. But in that case, we must have `this` bound to $C$<$\rho$> in $\Gamma$ and `this` bound to $o$ in $\Sigma$, where $H \vdash o : C$<$r$> for some valid region name $r$. (The last fact holds because $\vdash H$ implies $H \vdash o : T$ and $\emptyset \vdash T$, so $T$ must be a class type with a region name for its region argument; $o$ cannot be `null`.) Then $\sigma_{\Sigma,H}$ substitutes $r$ for $\rho$, so the result holds. For types and effects, the result holds from the way that types and effects are constructed from regions. □

Again assume $\Sigma$ instantiates $\Gamma$ in heap $H$. If effect $E$ is included in $E'$ with respect to $\Gamma$, then effect $\sigma_{\Sigma,H}(E)$ is included in $\sigma_{\Sigma,H}(E')$ with respect to the empty environment:

**Lemma 6.2.3.** *If $H \vdash \Sigma \preceq \Gamma$ and $\Gamma \vdash E \subseteq E'$, then $\emptyset \vdash \sigma_{\Sigma,H}(E) \subseteq \sigma_{\Sigma,H}(E')$.*

*Proof.* By the rules for unions of effects, it suffices to assume that $E$ and $E'$ are both individual read or write effects. In that case, either reflexivity or rule SE-READS-WRITES must apply, with the same region $R$ appearing in both effects. If $R$ is a region name $r$, then the result holds directly. If $R$ is a region parameter $\rho$, then the result holds because $\sigma_{\Sigma,H}$ substitutes the same region name $r$ for the parameter in both effects. □

Next we have the preservation result. Informally, if a program is well-typed and a local state expression $(e, \Sigma, \emptyset)$ reduces to $(o, \Sigma, E)$ starting with a good state, then the resulting heap is valid; the resulting type is valid and is a subtype of the static type of $e$; $E$ is valid; and $E$ is a subeffect of the static effect of $e$. To state the result precisely, we need a notation to describe the reduction of expressions $(e, \Sigma, \emptyset)$ occurring as a subexpression of the main program expression. We write

$$((e, \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} (e_i, H')$$

to mean that $\mathcal{P}$ is well typed with main expression $e_{\mathcal{P}}$, expression $e$ appears in the text of $\mathcal{P}$, and there exist expressions $e_j$ and $e'_j$ such that

$$((e_{\mathcal{P}}, \emptyset, \emptyset)_j, \mathsf{null}) \rightarrow^* (e_j, H) \rightarrow^* (e'_j, H'),$$

$(e, \Sigma, \emptyset)_i$ is a subexpression of $e_j$, $e_j$ is the first expression in which expression $i$ appears, and $e_i$ is a

subexpression of $e'_j$. $(e_j, H)$ is called the *initial state* of the reduction, and $(e'_j, H')$ is called the *final state* of the reduction.

Note that in general, the reduction of $(e, \Sigma, H)_i$ can occur "under a `cobegin`" (i.e., in reducing one of the two branch subexpressions of a `cobegin` expression); so in general, the reduction $(e_j, H) \rightarrow^* (e'_j, H')$ can contain steps reducing expressions other than expression $i$. These steps are still "part of the computation" of reducing expression $i$, because in theory, their effect on the heap state could affect the reduction of expression $i$. However, the goal of this section is to prove that in fact there is no interference between the reduction of these other expressions and the reduction of expression $i$.

**Theorem 6.2.4** (Preservation)**.** *If $((e, \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_i, H')$, and all steps that do not reduce expression $i$ take a valid heap to another valid heap, then (a) $\Gamma_e \vdash ((e, \Sigma, \emptyset), H) : T_s, E_s$, where $\Gamma_e$ is the environment used to type $e$ in typing $\mathcal{P}$; and (b) $\vdash H'$; and (c) $H' \vdash o : T$; and (d) $\emptyset \vdash T \preceq \sigma_{\Sigma,H}(T_s)$; and (e) $\emptyset \vdash E$; and (f) $\emptyset \vdash E \subseteq \sigma_{\Sigma,H}(E_s)$.*

*Proof.* It suffices to prove that for every reduction $((e, \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_i, H')$, if (a) holds, then the theorem holds for that reduction and all subexpressions reduced during the course of that reduction. This is because every reduction is a subexpression of the whole program reduction, and (a) is clearly satisfied for the whole program. We prove this equivalent result by induction on the structure of $e$.

**Base cases:** *Field access:* Except for steps that do not reduce expression $i$, the reduction occurs in one step via rule DYN-FIELD-ACCESS. (b) holds because reducing expression $i$ does not change $H$, and by assumption all other steps preserve $\vdash H$. (c) and (d) hold by $\vdash H$. (e) holds by rule FIELD and Lemma 6.2.2. (f) holds by construction (comparing DYN-FIELD-ACCESS with FIELD-ACCESS).

*Variable access:* Except for steps that do not reduce expression $i$, the reduction occurs in one step via rule DYN-VARIABLE. (b) holds for the same reason as stated for field access. (c) and (d) hold by $H \vdash \Sigma \preceq \Gamma$. (e) and (f) trivially hold.

*New objects:* Except for steps that do not reduce expression $i$, the reduction occurs in one step via rule DYN-NEW. (c) holds because $\Gamma \vdash C\texttt{<}R\texttt{>}$, so by Lemma 6.2.2, $\emptyset \vdash \sigma_{\Sigma,H}(C\texttt{<}R\texttt{>})$. (b) holds because the other steps preserve $\vdash H$ by assumption and the fields of the newly created object are valid, because `null` is a subtype of every type. (d) holds by construction. (e) and (f) trivially hold.

**Inductive cases:** *Field assignment:* The induction hypothesis applies to the reduction from the application of DYN-FIELD-ASSIGN-EVAL to the application of DYN-FIELD-ASSIGN-UPDATE. (b) holds because

by the induction hypothesis, the dynamic type of $o'$ is a subtype of $\sigma_{\Sigma,H}(T_e)$, where $T_e$ is the static type of $e$; and by rule FIELD-ASSIGN, $T_e$ is equal to the static type of field $f$. (c) and (d) hold by the induction hypothesis and by construction. (e) holds because the region of $f$ is valid in $\Gamma$, and by Lemma 6.2.2. (f) holds by construction.

*Method invocation:* The induction hypothesis applies to the evaluation of the argument to `this` and the argument to the formal parameter $x$. For the evaluation of the method body, we need to show that $H \vdash \Gamma' \preceq \Gamma'$, where $\Gamma' = (\texttt{this}, o) \cup (x, o')$ is the environment in which we execute the method body, and $\Gamma' = (\texttt{this}, C\texttt{<}\rho\texttt{>}) \cup (x, T_x)$ is the environment in which we typed the method $M(C, m)$. The types of the bindings to `this` match by construction. As to the bindings to $x$, from INVOKE we have that the static type of $e'$ matches $\sigma_C\texttt{<}_R\texttt{>}(T_x)$, and by the induction hypothesis, we have that the dynamic type of $o'$ is a subtype of $\sigma_{\Sigma,H}(\sigma_C\texttt{<}_R\texttt{>}(T_x)) = \sigma_C\texttt{<}_{\sigma_{\Sigma,H}(R)}\texttt{>} = \sigma_{\Gamma',H}(T_x)$.

Now we can apply the induction hypothesis to the execution of the method body. This gives (b), (c), and (e) directly. As to (d), the induction hypothesis gives that $o$ (in DYN-INVOKE-ACCUMULATE) is a subtype of $\sigma_{\Gamma',H}(T_r) = \sigma_{\Sigma,H}(\sigma_C\texttt{<}_R\texttt{>}(T_r)$. A similar argument using Lemma 6.2.3 establishes (f).

*Sequential composition:* Follows directly from the induction hypothesis.

*Deterministic parallel composition:* Follows directly from the induction hypothesis. □

**Noninterference:** In this section we prove that the executions of the two branches of a `cobegin` are mutually noninterfering. First we have a technical lemma stating that static disjointness of effect implies dynamic disjointness of effect, under translation by $\sigma_{\Sigma,H}$:

**Lemma 6.2.5.** *If $H \vdash \Sigma \preceq \Gamma$ and $\Gamma \vdash E \# E'$, then $\emptyset \vdash \sigma_{\Sigma,H}(E) \# \sigma_{\Sigma,H}(E')$.*

*Proof.* Obvious for this language, because we form noninterference judgments only for effects that either are (1) both read effects or (2) operate on distinct region names. Neither of these properties is affected by $\sigma_{\Sigma,H}$. □

Next we have the theorem:

**Theorem 6.2.6** (Noninterference of effect for `cobegin` branches)**.** *If*

$$((\texttt{cobegin(}e\texttt{,}\ e'\texttt{)}, \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} (\texttt{cobegin(}(o, \Sigma, E)_j\texttt{,}\ (o', \Sigma, E')_k\texttt{)}_i, H'),$$

*then* $\emptyset \vdash E \# E'$.

*Proof.* By the static typing rule for `cobegin`, the static effects of $e$ and $e'$ are noninterfering. By Theorem 6.2.4, the dynamic effects of executing $e$ and $e'$ are contained in the static effects after translation by $\sigma_{\Sigma,H}$. By Lemma 6.2.5, that translation preserves disjointness of effect. □

**Determinism:** In this section we prove that the execution of `cobegin` is deterministic. First we need a definition formalizing the idea that every object field on the heap resides in exactly one region:

**Definition 6.2.7** (Region of a field)**.** *If* $H \vdash o : C\texttt{<}r\texttt{>}$ *and* $T\ f$ `in` $R \in \mathit{fields}(C)$*, then* $\mathit{region}(o.f, H) = \sigma_{C\texttt{<}r\texttt{>}}(R)$.

It is obvious that if $(e, H) \rightarrow^* (e', H')$ and $\mathsf{region}(o.f, H) = r$, then $\mathsf{region}(o.f, H') = r$, because $\mathsf{Dom}(H)$ only ever grows, not shrinks, and the dynamic type of a reference $o \in \mathsf{Dom}(H)$ never changes.

Next we have a lemma stating that at runtime, disjoint regions imply disjoint locations:

**Lemma 6.2.8.** *If* $\mathit{region}(o.f, H) \neq \mathit{region}(o'.f', H)$*, then either* $o \neq o'$ *or* $f \neq f'$.

*Proof.* Let $R$ and $r$ be as shown in Definition 6.2.7. If $\mathsf{region}(o.f, H) \neq \mathsf{region}(o'.f', H)$, then either $R$ or $r$ must be different when the two regions are computed. If $R$ is different, we must either have a different class in the type of $o'$, implying $o \neq o'$; or we must have a different field of the same class, implying $f \neq f'$. If $r$ is different, we must have a different type for $o'$, implying $o \neq o'$. □

Now we can prove that `cobegin` is semantically identical to `seq`. That is, replacing `cobegin` with `seq` produces identical results:

**Theorem 6.2.9** (Semantic equivalence of `cobegin` and `seq`)**.**

$$(\texttt{cobegin}(e, e'), \Sigma, \emptyset)_j, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_j, H')$$

*with initial state* $(e_i, H)$ *if and only if*

$$(\texttt{seq}(e, e'), \Sigma, \emptyset)_j, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_j, H')$$

*with initial state* $(e_i[(\texttt{cobegin(}e\texttt{,}\ e'\texttt{)}, \Sigma, \emptyset)_j \leftarrow (\texttt{seq(}e\texttt{,}\ e'\texttt{)}, \Sigma, \emptyset)_j], H)$*.*

*Proof.* The if direction is obvious, because the reduction of `seq` is one legal reduction of `cobegin`. For the only if direction, first assume that the reduction of expression $j$ does not occur inside a `cobegin`, so all steps reduce expression $j$. Then by the rules for reducing `cobegin`, the first reduction implies

$$(\texttt{cobegin(}(e', \Gamma)_j\texttt{,}(e'', \Gamma)_k\texttt{)}_i, H') \rightsquigarrow_{\mathcal{P}} (\texttt{cobegin(}(o, E)_j\texttt{,}(o', E')_k\texttt{)}_i, H'').$$

By Theorem 6.2.6, the effects $E$ and $E'$ are noninterfering; and by the rules for noninterfering effects, together with Lemma 6.2.8, all read-write and write-write pairs of accesses, one from each reduction, are disjoint. Further, the rules for field access and assignment faithfully record the effects of every heap access, so there can be no interfering accesses in the two branches. Since the only possible dependences between the two reductions are through the heap, and we just showed there are no such dependences, every step in the reduction of expression $j$ commutes with every step in the reduction of expression $k$. By a simple induction on the length of the execution sequence, we can therefore produce an equivalent result by reducing expression $j$ first and then reducing expression $k$, which is exactly the reduction rule for `seq`.

Now assume that expression $j$ occurs inside a `cobegin`. By a simple induction, it suffices to consider the case of one `cobegin`. In that case, by the same noninterference argument given above, none of the steps reducing $j$ interfere with the steps that don't reduce $j$. So again we can rearrange the $j$-reducing steps so that they first reduce $e$ to $o$, and then reduce $e'$ to $o'$. □

Finally, we can state the determinism result:

**Theorem 6.2.10** (Input-Output Determinism)**.** *If*

$$((e, \Sigma, \emptyset)_j, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_j, H')$$

*and*

$$((e, \Sigma, \emptyset)_j, H) \rightsquigarrow_{\mathcal{P}} ((o', \Sigma, E')_j, H'')$$

*with the same initial state, then $o \cong o'$, where $\cong$ denotes equivalence up to renaming object references, and $E = E'$. Moreover, if $(e, \Sigma, \emptyset)_j$ is not a subexpression of any* `cobegin` *expression, then $H \cong H'$.*

*Proof.* First, consider the case where $(e, \Sigma, \emptyset)_j$ does not occur in any `cobegin` expression. By Theorem 6.2.9, if we replace all instances of `cobegin` in $e_i$ with `seq`, we get exactly the same executions. But

except for the `cobegin` reduction rules, the dynamic semantics rules are entirely deterministic (there is only ever one thing to do next), except for the choice of object identifier in rule DYN-NEW and the choice of fresh expression identifiers when introducing new expressions to be reduced. Clearly nothing in the semantics depends on the choice of these identifiers.

Now suppose that $(e, \Sigma, \emptyset)_j$ occurs in exactly one `cobegin` expression. It is nondeterministic what effects of the other `cobegin` branch are complete at the point where $(e, \Sigma, \emptyset)_j$ is reduced to $(o, \Sigma, E)_j$. However, by the same argument as in the proof of Theorem 6.2.9, none of those effects can interfere with the computation of $o$. A simple induction extends this argument to more than one `cobegin` expression. □

## 6.3 Deterministic-by-Default Language

This section describes the semantics and soundness results for the deterministic-by-default language.

### 6.3.1 Static Semantics

For the extended language, we need to extend the definition of effects and the typing of expressions.

**Valid effects:** We add a rule for valid atomic effects.

$$\boxed{\Gamma \vdash E} \qquad \text{EFFECT-ATOMIC}\ \frac{\Gamma \vdash E}{\Gamma \vdash \texttt{atomic}\ E}$$

**Subeffects:** We add rules stating when atomic effects are subeffects of other effects.

$$\boxed{\Gamma \vdash E \subseteq E'} \qquad \text{SE-ATOMIC-1}\ \frac{\Gamma \vdash E \subseteq E'}{\Gamma \vdash \texttt{atomic}\ E \subseteq E'} \qquad \text{SE-ATOMIC-2}\ \frac{\Gamma \vdash E \subseteq E'}{\Gamma \vdash \texttt{atomic}\ E \subseteq \texttt{atomic}\ E'}$$

Rule SE-ATOMIC-1 formally expresses the idea that non-atomic effects cover atomic effects, i.e., if $E$ occurred in an atomic expression, then we can summarize the effect as either `atomic` $E$ or $E$. Note that the converse is not true, i.e., we cannot summarize $E$ with `atomic` $E$. Rule SE-ATOMIC-2 says that two atomic effects are subeffects if the underlying effects are.

**Noninterfering effects:** We add a rule stating that an atomic effect is interfering with another effect if the

underlying effect is.

$$\boxed{\Gamma \vdash E \# E'} \qquad \text{NI-ATOMIC}\ \frac{\Gamma \vdash E \# E'}{\Gamma \vdash \texttt{atomic}\ E \# E'}$$

**Safe nondeterministic execution:** The judgment $\Gamma \vdash \mathsf{nondet}(E, E')$ states that it is safe to run expressions with effects $E$ and $E'$ nondeterministically in parallel.

$$\boxed{\Gamma \vdash \mathsf{nondet}(E, E')} \qquad \text{NONDET-SYMMETRIC}\ \frac{\Gamma \vdash \mathsf{nondet}(E, E')}{\Gamma \vdash \mathsf{nondet}(E', E)} \qquad \text{NONDET-NI}\ \frac{\Gamma \vdash E \# E'}{\Gamma \vdash \mathsf{nondet}(E, E')}$$

$$\text{NONDET-ATOMIC}\ \frac{}{\Gamma \vdash \mathsf{nondet}(\texttt{atomic}\ E, \texttt{atomic}\ E')} \qquad \text{NONDET-UNION}\ \frac{\Gamma \vdash \mathsf{nondet}(E, E'') \quad \Gamma \vdash \mathsf{nondet}(E', E'')}{\Gamma \vdash \mathsf{nondet}(E \cup E', E'')}$$

The rules formally express the idea that it is safe to let two effects run inside `cobegin_nd` if (1) they are mutually noninterfering (NONDET-NI); or (2) they both occur inside atomic expressions (NONDET-ATOMIC); or (3) they can be decomposed into unions of effects that are safe to run nondeterministically in parallel (NONDET-UNION).

**Expressions:** We add new rules for typing `cobegin_nd` and `atomic` expressions. We also revise the rule for typing `cobegin` expressions.

*Nondeterministic parallel composition:* Rule COBEGIN_ND is similar to COBEGIN, except that noninterference is not required as to the effects of the two branches. Instead, it is sufficient that the effects are safe to run nondeterministically in parallel.

$$\boxed{\Gamma \vdash e : T, E} \qquad \text{COBEGIN\_ND}\ \frac{\Gamma \vdash e : T, E \quad \Gamma \vdash e' : T', E' \quad \Gamma \vdash \mathsf{nondet}(E, E')}{\Gamma \vdash \texttt{cobegin\_nd(}e\texttt{,}\ e'\texttt{)} : T', E \cup E'}$$

*Atomic expressions:* Rule ATOMIC collects the effect $E$ of the expression $e$, then marks all the constituent

read and write effects atomic, to reflect the fact that $E$ occurred inside an atomic expression.

$$\boxed{\Gamma \vdash e : T, E} \qquad \textsc{Atomic}\ \frac{\Gamma \vdash e : T, E \quad \vdash \mathsf{atomic}(E) = E'}{\Gamma \vdash \texttt{atomic}\ e : T, E'}$$

The judgment $\Gamma \vdash \mathsf{atomic}(E) = E'$ says that effect $E'$ is the result of taking $E$ and marking all its constituent effects `atomic`.

$$\boxed{\Gamma \vdash \mathsf{atomic}(E) = E'} \qquad \textsc{Atomic-Empty}\ \frac{}{\Gamma \vdash \mathsf{atomic}(\emptyset) = \emptyset} \qquad \textsc{Atomic-Reads}\ \frac{}{\Gamma \vdash \mathsf{atomic}(\texttt{reads}\ R) = \texttt{atomic reads}\ R}$$

$$\textsc{Atomic-Writes}\ \frac{}{\Gamma \vdash \mathsf{atomic}(\texttt{writes}\ R) = \texttt{atomic writes}\ R} \qquad \textsc{Atomic-Atomic}\ \frac{}{\Gamma \vdash \mathsf{atomic}(\texttt{atomic}\ E) = \texttt{atomic}\ E}$$

$$\textsc{Atomic-Union}\ \frac{\Gamma \vdash \mathsf{atomic}(E) = E' \quad \Gamma \vdash \mathsf{atomic}(E'') = E'''}{\Gamma \vdash \mathsf{atomic}(E \cup E'') = E' \cup E'''}$$

*Deterministic parallel composition:* Finally, rule COBEGIN has changed. In addition to checking noninterference, as in the basic language, the new rule converts all atomic effects occurring inside the `cobegin` to ordinary effects. This ensures that *no atomic effects are ever propagated outward from inside a* `cobegin`.

$$\textsc{Cobegin}\ \frac{\Gamma \vdash e : T, E \quad \Gamma \vdash e' : T', E' \quad \Gamma \vdash E \# E' \quad \vdash \mathsf{nonatomic}(E \cup E') = E''}{\Gamma \vdash \texttt{cobegin(}e\texttt{,}e'\texttt{)} : T', E''}$$

The judgment $\Gamma \vdash \mathsf{nonatomic}(E) = E'$ says that $E'$ is the result of converting all `atomic` effects to ordinary effects in $E$.

$$\boxed{\Gamma \vdash \mathsf{nonatomic}(E) = E'} \qquad \textsc{Nonatomic-Empty}\ \frac{}{\Gamma \vdash \mathsf{nonatomic}(\emptyset) = \emptyset} \qquad \textsc{Nonatomic-Reads}\ \frac{}{\Gamma \vdash \mathsf{nonatomic}(\texttt{reads}\ R) = \texttt{reads}\ R}$$

NONATOMIC-WRITES

$$\frac{}{\Gamma \vdash \mathsf{nonatomic}(\texttt{writes}\ R) = \texttt{writes}\ R}$$

NONATOMIC-ATOMIC

$$\frac{}{\Gamma \vdash \mathsf{nonatomic}(\texttt{atomic}\ E) = E}$$

NONATOMIC-UNION

$$\frac{\Gamma \vdash \mathsf{nonatomic}(E) = E' \quad \Gamma \vdash \mathsf{nonatomic}(E'') = E'''}{\Gamma \vdash \mathsf{nonatomic}(E \cup E') = E'' \cup E'''}$$

### 6.3.2 Dynamic Semantics

We describe the dynamic semantics of the nondeterministic language in two parts, the first operational and the second non-operational. The first part is just the same semantics as for the deterministic language, with a few minor adjustments to accommodate the new features. The second part describes a *weak isolation constraint* on execution histories generated by the first part. The overall dynamic semantics comprises all execution histories described by the operational semantics that also satisfy weak isolation. In practice, weak isolation would be enforced by a runtime implementation (such as software transactional memory). Such implementations are well understood and have been described in detail elsewhere [63].

**Semantics of Expressions:** The execution environment and definition of program execution are the same as in Section 6.2.2. We add operational semantics rules for `cobegin_nd` and atomic expressions, and we amend the rules for `cobegin` to account for atomic effects.

*Nondeterministic parallel composition:* Execution of `cobegin_nd` is identical to execution of `cobegin`. For completeness, we state the rules in full.

DYN-COBEGIN-ND-EVAL

$$\frac{\mathsf{fresh}(i) \quad \mathsf{fresh}(j)}{((\texttt{cobegin\_nd(}e\texttt{,}e'\texttt{)}, \Sigma, \emptyset), H) \rightarrow (\texttt{cobegin\_nd(}(e, \Sigma, \emptyset)_i\texttt{,}(e', \Sigma, \emptyset)_j\texttt{)}, H)}$$

DYN-COBEGIN-ND-ACCUMULATE

$$\frac{}{(\texttt{cobegin\_nd(}(o, \Sigma, E)_i\texttt{,}(o', \Sigma, E')_j\texttt{)}, H) \rightarrow ((o', \Sigma, E \cup E'), H)}$$

*Atomic expressions:* To execute an expression `atomic` $e$, we execute the expression $e$, then mark all its

effects `atomic`:

$$\frac{\mathsf{fresh}(i)}{((\texttt{atomic}\ e, \Sigma, \emptyset), H) \rightarrow (\texttt{atomic}\ (e, \Sigma, \emptyset)_i, H)}\ \textsc{Dyn-Atomic-Eval} \qquad \frac{\emptyset \vdash \mathsf{atomic}(E) = E'}{(\texttt{atomic}\ (o, \Sigma, E)_i, H) \rightarrow ((o, \Sigma, E'), H)}\ \textsc{Dyn-Atomic-Mark-Effects}$$

Note that the environment used for marking effects is the empty environment, because there are no region parameters in the runtime effects.

*Deterministic parallel composition:* We modify the rule DYN-COBEGIN-ACCUMULATE to mark the effects of the expression non-atomic:

$$\frac{\emptyset \vdash \mathsf{nonatomic}(E \cup E') = E''}{(\texttt{cobegin(}(o, \Sigma, E)_i\texttt{,}(o', \Sigma, E')_j\texttt{)}, H) \rightarrow ((o', \Sigma, E''), H)}\ \textsc{Dyn-Cobegin-Accumulate}$$

**Weak Isolation Constraint:** We state the weak isolation semantics as a constraint on the possible executions given by the operational semantics rules stated above. That is, we assume that executions that are allowed by the rules, but that violate weak isolation, are guaranteed never to happen (because, e.g., they would cause a transactional memory implementation to abort and roll back).

**Definition 6.3.1** (Reduction histories)**.** *Suppose* $((e, \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_i, H')$*. Any sequence of steps in an execution of* $\mathcal{P}$ *witnessing this fact is called a* ***reduction history****. We denote such a history* $\mathbf{H}$*.*

By the definition of a reduction history $\mathbf{H}$, and the definition of a reduction, if $\mathbf{H}$ witnesses $((e, \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_i, H')$, then there must exist a history witnessing $((e_{\mathcal{P}}, \emptyset, \emptyset)_i, \mathsf{null}) \rightarrow^* (e_j, H) \rightarrow^* (e'_j, H')$, with $(e, \Sigma, \emptyset)_i$ a subexpression of $e_j$, and $(o, \Sigma, E)_i$ a subexpression of $e'_j$; and $\mathbf{H}$ is the sequence of steps witnessing $(e_j, H) \rightarrow^* (e'_j, H')$. As before, $(e_j, H)$ and $(e'_j, H')$ are called the initial and final states of the reduction. Note that in general, a reduction history $\mathbf{H}$ is not unique, because (1) parallel tasks may have different interleavings; and (2) the choice of expression identifiers and object references in the history is arbitrary.

In the special case that the initial state of $\mathbf{H}$ is the initial program execution state $((e_{\mathcal{P}}, \emptyset, \emptyset)_i, \mathsf{null})$, we denote the corresponding reduction history (representing a program execution) $\mathbf{H}_{\mathcal{P}}$, and we call it a *program execution history*. If one history $\mathbf{H}_1$ is contained within another one $\mathbf{H}_2$, we say that $\mathbf{H}_1$ is a *subhistory* of

$\mathbf{H}_2$. In particular, all histories occurring in an execution are subhistories of $\mathbf{H}_\mathcal{P}$.

Now we define a conflict relation on `atomic` expressions that allows us to state the weak isolation assumption. First, we need a notion of parallel execution under `cobegin_nd`:

**Definition 6.3.2** (Parallel histories under `cobegin_nd`)**.** *Fix a program execution history $\mathbf{H}_\mathcal{P}$, and consider any pair of subhistories $\mathbf{H}_1$ and $\mathbf{H}_2$ of $\mathbf{H}_\mathcal{P}$. $\mathbf{H}_1$ and $\mathbf{H}_2$ occur* ***in parallel under `cobegin_nd`*** *if $\mathbf{H}_1$ occurs in reducing subexpression $i$, and $\mathbf{H}_2$ occurs in reducing subexpression $j$, of the same expression introduced by rule* DYN-COBEGIN-ND-EVAL.

Then we can state the conflict relation:

**Definition 6.3.3** (Conflict relation on `atomic` expressions)**.** *Fix a program execution history $\mathbf{H}_\mathcal{P}$, and let $I$ be the set of expression identifiers appearing in $\mathbf{H}_\mathcal{P}$ that label atomic expressions (i.e., expressions introduced by rule* DYN-ATOMIC-EVAL*). The* ***conflict relation on atomic expressions in $\mathbf{H}_\mathcal{P}$*** *is the transitive closure of the following relation: the pair $(i, j)$ is in the relation if $i, j \in I$, $i \neq j$, and there are conflicting memory accesses $a_i$ and $a_j$ (i.e., two accesses to the same location, with at least one a write) such that (a) $a_i$ occurs in the reduction of an atomic expression $e_i$; (b) $a_j$ occurs in the reduction of an atomic expression $e_j$; (c) the reductions of $e_i$ and $e_j$ occur in parallel under `cobegin_nd`, and (d) $a_i$ precedes $a_j$ in $\mathbf{H}_\mathcal{P}$.*

Notice that we put the relation *only on parallel expressions under `cobegin_nd` expressions, not `cobegin` expressions*; and we *do not include any conflicts occurring outside of atomic expressions*. That is because the type system will guarantee that there aren't any conflicts between `cobegin` tasks or outside of atomic expressions; this is the soundness result that we prove in the next section.

Now we can define the weak isolation constraint on executions in the language:

**Definition 6.3.4** (Weakly isolated histories)**.** *Let $\mathbf{H}$ be a history. If the conflict relation on `atomic` expressions in $\mathbf{H}$ is a partial order, then we say that $\mathbf{H}$ is* ***weakly isolated****.*

In the rest of this chapter, we assume we have an implementation that guarantees weakly isolated histories. It should be clear that this assumption is equivalent to *weak conflict serializability* (i.e., serializability of tasks with respect to accesses occurring in atomic expressions), which is guaranteed by any standard transactional implementation, even a "weakly atomic" one.

### 6.3.3 Soundness

**Static Environments, Preservation, and Noninterference:** First we reassert Theorem 6.2.1 for the extended language.

**Theorem 6.3.5** (Validity of static expression typing)**.** *Theorem 6.2.1 holds for the deterministic-by-default language defined in Section 6.3.*

*Proof.* By a simple extension of the proof of Theorem 6.2.1, using the typing rules for the new expressions, and the fact that atomic and nonatomic take valid effects to valid effects. □

Lemmas 6.2.2 and 6.2.3 obviously hold for the deterministic-by-default language. We reassert Theorem 6.2.4:

**Theorem 6.3.6** (Preservation)**.** *Theorem 6.2.4 holds for the language defined in Section 6.3.*

*Proof.* Again by structural induction. The arguments for all expressions except the three with new rules (`cobegin_nd`, `atomic`, and `cobegin`) carry over from the proof of Theorem 6.2.4. For those three expressions, the result follows from the induction hypothesis, and by construction. □

Lemma 6.2.5 and Theorem 6.2.6 obviously hold for the extended language. We reassert the theorem:

**Theorem 6.3.7** (Noninterference of effect for `cobegin` branches)**.** *Theorem 6.2.6 holds for the language defined in Section 6.3.*

**Race Freedom:** The following theorem implies that program execution is race-free, assuming an implementation that imposes a synchronization order on `atomic` expressions consistent with the conflict relation in Definition 6.3.3. This property is true, for example, of any transactional memory implementation.

To state the theorem, we extend Definition 6.3.2 in the obvious way to include parallel histories under `cobegin`. We say that two histories that occur in parallel under either `cobegin` or `cobegin_nd` (or both) occur *in parallel*.

**Theorem 6.3.8** (Synchronization of conflicting memory operations)**.** *Suppose $\vdash \mathcal{P}$, and fix a program execution history $\mathbf{H}_{\mathcal{P}}$. Then for any two conflicting memory accesses $a_1$ and $a_2$ occurring in $\mathbf{H}_{\mathcal{P}}$, either (1) $a_1$ and $a_2$ do not occur in parallel; or (2) $a_1$ and $a_2$ are ordered by the conflict relation given in Definition 6.3.3.*

*Proof.* Suppose $a_1$ and $a_2$ are conflicting accesses that occur in parallel. Theorem 6.3.8 says they cannot occur in parallel under a `cobegin`, so they must occur in parallel under a `cobegin_nd` with subexpressions $e_i$ and $e_j$. Now suppose (without loss of generality) $a_1$ does not occur inside any atomic expression contained in $e_i$. Then the effect of branch $i$ of the `cobegin_nd` contains the effect generated by $a_i$, which is not an atomic effect. By Theorem 6.3.7, the static effect of the first branch of the `cobegin_nd` contains a supereffect of that effect, and similarly for the other branch. But that means that $\Gamma \vdash \mathsf{nondet}(E, E')$ is not satisfied as to the static effects of the two branches, so rule COBEGIN_ND does not apply, and the program is not well typed. □

**Corollary 6.3.9** (Race freedom)**.** *If $\vdash \mathcal{P}$, then a history $\mathbf{H}_{\mathcal{P}}$ that is weakly isolated according to Definition 6.3.4, and has synchronization orderings consistent with the conflict relation stated in Definition 6.3.3, contains no data race.*

**Strong Isolation:** The strong isolation result says that certain expressions are guaranteed to be reduced "as if in isolation" (i.e., as if there were no interleavings of the steps of reducing other expressions with the steps of reducing that one). To state the result formally, we use the well-known concept of *serializable histories* [97].

**Definition 6.3.10** (Serial histories)**.** *A history* $\mathbf{H}$ *witnessing* $((e, \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_i, H')$ *is* ***serial with respect to expression*** $i$ *if every step in the history transforms expression $i$ or a subexpression of expression $i$.*

For example:

- The whole program history is serial with respect to the main program expression, because all of it reduces the main expression.
- If the main program expression $e_{\mathcal{P}}$ is `seq(`$e$`,`$e'$`)`, then the histories reducing $e$ and $e'$ are each serial with respect to those expressions.
- In general, the reductions of $e$ and $e'$ embedded in the reductions of `cobegin(`$e$`,`$e'$`)` or `cobegin_nd(`$e$`,`$e'$`)` are not serial.
- In general, if `seq(`$e$`,`$e'$`)` is reduced inside a `cobegin` or `cobegin_nd`, then the reductions of $e$

and $e'$ are not serial, because they may be interleaved with the reductions of the other expression of the `cobegin` or `cobegin_nd`.

**Definition 6.3.11** (Serializable histories)**.** *Fix a program execution history $\mathbf{H}_{\mathcal{P}}$, and let $\mathbf{H}$ be a subhistory of $\mathbf{H}_{\mathcal{P}}$ witnessing $((e, \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_i, H')$. $\mathbf{H}$ is **serializable with respect to expression** $i$ if there exists a program execution history $\mathbf{H}'_{\mathcal{P}}$ such that (1) $\mathbf{H}'_{\mathcal{P}}$ contains a subhistory $\mathbf{H}'$ with the same initial and final states as $\mathbf{H}$; and (2) $\mathbf{H}'$ contains subhistory $\mathbf{H}''$ witnessing $((e, \Sigma, \emptyset)_i, H'') \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_i, H''')$ that is serial with respect to expression $i$, for some heaps $H''$ and $H'''$.*

Intuitively, an expression reduction is serializable if we "could have done it serially," with the same results. For example, in any execution history of $\mathcal{P}$ where $e_{\mathcal{P}}$ is `cobegin(`$e$`,`$e'$`)`, the reductions of $e$ and $e'$ are serializable with respect to those expressions, because there are no conflicts between the steps of reducing $e$ and $e'$ (Theorem 6.2.6).

**Theorem 6.3.12** (Strong isolation)**.** *Suppose $\vdash \mathcal{P}$, let $\mathbf{H}_{\mathcal{P}}$ be a weakly isolated history executing $\mathcal{P}$, and let $\mathbf{H}$ be a subhistory of $\mathbf{H}_{\mathcal{P}}$ witnessing $((e, \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_i, H')$. Then $\mathbf{H}$ is serializable with respect to expression $i$ if (1) $(e, \Sigma, \emptyset)_i$ is not a subexpression of any `cobegin_nd` expression; or (2) no atomic expression appears in $\mathbf{H}$; or (3) $e$ is a `cobegin` or atomic expression.*

*Proof.* (1) If $(e, \Sigma, \emptyset)_i$ is not a subexpression of any `cobegin_nd`, then either $\mathbf{H}$ is a serial reduction, or $(e, \Sigma, \emptyset)_i$ is a subexpression of one or more `cobegin` expressions. In the latter case, for each `cobegin` branch of which the reduction of $(e, \Sigma, \emptyset)_i$ is a subsequence, reduction of the other `cobegin` branch is noninterfering (Theorem 6.3.8). Therefore, there can be no conflicts with the reduction of other expressions, so $\mathbf{H}$ is serializable with respect to expression $i$.

(2) By (1) it suffices to show that the reduction of $(e, \Sigma, \emptyset)_i$ does not interfere with any reduction occurring in parallel under `cobegin_nd`, because noninterference implies serializability. Suppose there exists a `cobegin_nd` expression with branch expressions $e_j$ and $e_k$ such that the reduction of expression $i$ is contained in the reduction of $e_j$, and the reduction of $e_k$ interferes with the reduction of expression $i$. Because the reduction of expression $i$ has to atomic expressions, it cannot produce atomic effects. Therefore the reduction of expression $i$ has a non-atomic dynamic effect that conflicts with some effect in the reduction of $e_k$. By the argument given in the proof of Theorem 6.3.8, this cannot happen for a well-typed program.

(3) *`cobegin(`$e'$`,`$e''$`):`* Same argument as for (2), because by rule DYN-COBEGIN-ACCUMULATE, reducing a `cobegin` expression cannot produce an atomic effect.

*`atomic` $e'$:* Assume expressions $e_i$ and $e_j$ as in the `cobegin` case. By Theorem 6.3.8, any pair of interfering accesses, one in the reduction of $e_i$ and one in the reduction of $e_j$, are both contained in atomic expressions contained in the enclosing `cobegin_nd`. The result follows by the assumption of weak isolation. □

**Determinism by Default:** We now show that the nondeterministic language is *deterministic-by-default*: that is, if an expression has a serializable reduction according to Theorem 6.3.12, and the expression reduction does not contain `cobegin_nd`, then the reduction has deterministic input-output semantics.

First, we state the equivalent of Theorem 6.2.9 for the nondeterministic language:

**Theorem 6.3.13** (Semantic equivalence of `cobegin` and `atomic seq`)**.** *Suppose* $\vdash \mathcal{P}$*. Then*

$$(\texttt{cobegin}(e, e'), \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_i, H')$$

*with initial state* $(e_j, H)$ *if and only if*

$$(\texttt{atomic seq}(e, e'), \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_i, H')$$

*with initial state* $(e_j[(\texttt{cobegin(}e\texttt{,}e'\texttt{)}, \Sigma, \emptyset)_i \leftarrow (\texttt{atomic seq(}e\texttt{,}e'\texttt{)}, \Sigma, \emptyset)_i], H)$*.*

*Proof.* As in the proof of Theorem 6.2.9, the if direction is obvious, because the reduction of `seq` is one legal reduction of `cobegin`. For the only if direction, if the reduction of expression $j$ does not occur inside a `cobegin_nd`, then the result follows from Theorem 6.2.9. Otherwise, by Theorem 6.3.12, there exists a history witnessing

$$(\texttt{cobegin}(e, e'), \Sigma, \emptyset)_i, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_i, H')$$

with initial state $(e_j, H)$, in which the subhistory witnessing the reduction of expression $i$ is serial with respect to expression $i$. By Theorem 6.2.9, that history witnesses the result. □

Finally, we state and prove the property of determinism by default:

**Theorem 6.3.14** (Determinism by default)**.** *Suppose* $\vdash \mathcal{P}$*, and let* $\mathbf{H}$ *be a history witnessing* $((e, \Sigma, \emptyset)_j, H) \rightsquigarrow_{\mathcal{P}} ((o, \Sigma, E)_j, H')$ *that is serializable with respect to expression* $j$*, where no* `cobegin_nd` *expression appears in the reduction of* $e$*. If* $((e, \Sigma, \emptyset)_j, H) \rightsquigarrow_{\mathcal{P}} ((o', \Sigma, E')_j, H'')$ *with the same initial state as in* $\mathbf{H}$*, then* $o \cong o$*,*

*where $\cong$ denotes equivalence up to renaming object references, and $E = E'$. Moreover, if $(e, \Sigma, \emptyset)_j$ is not a subexpression of any* `cobegin` *or* `cobegin_nd` *expression, then $H' \cong H''$.*

*Proof.* If $(e, \Sigma, \emptyset)_j$ is not a subexpression of any `cobegin_nd` expression, then the result follows directly from Theorem 6.2.10. Otherwise, by the definition of a serializable history, $o'$ and $E'$ must be defined by a history that is serial with respect to expression $j$, which (as discussed in the proof of Theorem 6.2.10), is unique up to the choice of object reference identifiers. □

## 6.4 Atomic Regions Language

This section describes the static and dynamic semantics results for the atomic regions language. If the isolation semantics of atomic expressions (Section 6.3.2) is implemented via software transactional memory (STM), then atomic regions allow for more efficient STM implementations, by telling the compiler where memory accesses are guaranteed to be noninterfering inside atomic expressions. No STM synchronization operations (sometimes called "barriers") are necessary for such accesses.

### 6.4.1 Static Semantics

We add the predicate $\mathsf{atomic}(\rho)$ to the environment, to indicate whether $P$ is an atomic region parameter:

$$\Gamma ::= \ldots \mid \mathsf{atomic}(\rho)$$

**Basic Program Elements:** We modify the rules for valid classes and types, to make sure that atomic parameters are instantiated only with atomic regions, and nonatomic parameters with nonatomic regions. This strategy ensures that a particular memory region is always treated consistently (given barriers or not) inside a transaction.

$\boxed{\vdash \mathcal{C}}$

$$\textsc{Class-Atomic}\quad \frac{\{(\texttt{this}, C\texttt{<}\rho\texttt{>}), \mathsf{atomic}(\rho)\} \vdash F^* \quad \{(\texttt{this}, C\texttt{<}\rho\texttt{>}), \mathsf{atomic}(\rho)\} \vdash M^*}{\vdash \texttt{class}\ C\texttt{<atomic}\ P\texttt{>}\ \{\ \mathrm{F}^*\ \mathrm{M}^*\ \}}$$

$\boxed{\Gamma \vdash T}$

$$\textsc{Type}\quad \frac{\texttt{class}\ C\texttt{<}\rho\texttt{>}\ \{\ F^*\ M^*\ \} \in \mathcal{P} \quad \Gamma \vdash R \quad \Gamma \vdash \mathsf{nonatomic}(R)}{\Gamma \vdash C\texttt{<}R\texttt{>}}$$

$$\textsc{Type-Atomic}\quad \frac{\texttt{class}\ C\texttt{<atomic}\ \rho\texttt{>}\ \{\ F^*\ M^*\ \} \in \mathcal{P} \quad \Gamma \vdash R \quad \Gamma \vdash \mathsf{atomic}(R)}{\Gamma \vdash C\texttt{<}R\texttt{>}}$$

The predicates $\mathsf{atomic}(R)$ and $\mathsf{nonatomic}(R)$ say whether a region is atomic:

$\boxed{\Gamma \vdash \mathsf{atomic}(R)}$

$$\textsc{Atomic-Name}\quad \frac{\texttt{atomic region}\ r \in \mathcal{P}}{\Gamma \vdash \mathsf{atomic}(r)} \qquad \textsc{Atomic-Param}\quad \frac{\mathsf{atomic}(\rho) \in \Gamma}{\Gamma \vdash \mathsf{atomic}(\rho)}$$

$\boxed{\Gamma \vdash \mathsf{nonatomic}(R)}$

$$\textsc{Nonatomic-Name}\quad \frac{\texttt{region}\ r \in \mathcal{P}}{\Gamma \vdash \mathsf{nonatomic}(r)} \qquad \textsc{Nonatomic-Param}\quad \frac{\mathsf{atomic}(\rho) \notin \Gamma}{\Gamma \vdash \mathsf{nonatomic}(\rho)}$$

**Typing Expressions:** The new rules for the judgment $\Gamma \vdash \mathsf{atomic}(E) = E'$ say that an atomic expression "makes an effect atomic" only if the effect is on an atomic region. Non-atomic regions never generate atomic effects, even inside a transaction.

$\boxed{\Gamma \vdash \mathsf{atomic}(E) = E'}$

$$\textsc{Atomic-Reads}\quad \frac{\Gamma \vdash \mathsf{atomic}(R)}{\Gamma \vdash \mathsf{atomic}(\texttt{reads}\ R) = \texttt{atomic reads}\ R} \qquad \textsc{Atomic-Reads-1}\quad \frac{\Gamma \vdash \mathsf{nonatomic}(R)}{\Gamma \vdash \mathsf{atomic}(\texttt{reads}\ R) = \texttt{reads}\ R}$$

$$\textsc{Atomic-Writes}\quad \frac{\Gamma \vdash \mathsf{atomic}(R)}{\Gamma \vdash \mathsf{atomic}(\texttt{writes}\ R) = \texttt{atomic writes}\ R} \qquad \textsc{Atomic-Writes-1}\quad \frac{\Gamma \vdash \mathsf{nonatomic}(R)}{\Gamma \vdash \mathsf{atomic}(\texttt{writes}\ R) = \texttt{writes}\ R}$$

### 6.4.2 Dynamic Semantics

The dynamic semantics of this language variant is exactly as given in Section 6.3.2, with the following changes:

1. Marking effects in rule DYN-ATOMIC-MARK-EFFECTS now happens according to the definition of $\emptyset \vdash \mathsf{atomic}(E) = E'$ given above, i.e., effects are marked atomic only if they operate on atomic regions.

2. We redefine the conflict relation on atomic expressions, so that only conflicts involving accesses to atomic regions are synchronized by the implementation. This new conflict relation replaces Definition 6.3.3 and is stated below.

**Definition 6.4.1** (Conflict relation on `atomic` expressions)**.** *Fix a program execution history $\mathbf{H}_{\mathcal{P}}$, and let $I$ be the set of expression identifiers appearing in $\mathbf{H}_{\mathcal{P}}$ that label atomic expressions (i.e., expressions introduced by rule* DYN-ATOMIC-EVAL*). The **conflict relation on atomic expressions in** $\mathbf{H}_{\mathcal{P}}$ is the transitive closure of the following relation: the pair $(i, j)$ is in the relation if $i, j \in I$, $i \neq j$, and there are conflicting memory accesses $a_i$ and $a_j$ (i.e., two accesses to the same location, with at least one a write) such that (a) $a_i$ occurs in the reduction of an atomic expression $e_i$; (b) $a_j$ occurs in the reduction of an atomic expression $e_j$; (c) both $a_i$ and $a_j$ access fields $T\ f$ `in` $R$ whose declared region $R$ is an atomic region or an an atomic region parameter; (d) the reductions of $e_i$ and $e_j$ occur in parallel under `cobegin_nd`; and (e) $a_i$ precedes $a_j$ in $\mathbf{H}_{\mathcal{P}}$.*

Notice that we state the conflict relation in terms of the *static* class declaration. For example, given a class declaration

```
class C<atomic P> {
    int x in P;
}
```

any effect generated by access to `this.x` would be included in the relation. However, if `P` were not declared `atomic`, then the access would not be included. We do this to model a compiler implementation that would insert barriers, or not, according to the information available in the program text at the point of the access. The actual region bound to a region parameter is not available to the compiler code generator at that point.

### 6.4.3 Soundness

**Theorem 6.4.2.** *Theorems 6.3.5–6.3.7 hold for the language defined in Section 3.*

*Proof.* The only new thing to show is that atomic effects are still managed consistently. But this is clear from the fact that the static and dynamic semantics use the same rules for marking atomic effects. □

**Theorem 6.4.3** (Synchronization of conflicting memory operations)**.** *Suppose* $\vdash \mathcal{P}$*, and fix a program execution history* $\mathbf{H}_{\mathcal{P}}$*. Then for any two conflicting memory accesses* $a_1$ *and* $a_2$ *occurring in* $\mathbf{H}_{\mathcal{P}}$*, either (1)* $a_1$ *and* $a_2$ *do not occur in parallel; or (2)* $a_1$ *and* $a_2$ *are ordered by the conflict relation given in Definition 3.1.*

*Proof.* By the same argument as for proving Theorem 6.3.8: any other pair of conflicting accesses cannot generate atomic effects, and so aren't allowed to interfere by the static type system. The only wrinkle is that the conflict relation is based on static region parameters, and the runtime effects are based on the regions bound to the parameters. However, the rules ensure that in computing the actual effect of a field access, the region in the dynamic effect is atomic if and only if the parameter in the field definition is atomic. □

**Theorem 6.4.4** (Strong isolation)**.** *Theorem 6.3.12 holds for the language defined in Section 6.4.*

*Proof.* The argument is the same as in proving Theorem 6.3.12, except in the case `atomic` $e$. In that case we modify the argument to observe that by the static semantics rules, any pair of interfering accesses, one in the reduction of $e_i$ and one in the reduction of $e_j$ must both be in atomic expressions, and must both be accesses that are statically called out as operating on atomic regions. □

**Theorem 6.4.5** (Semantic equivalence of `cobegin` and `atomic seq`)**.** *Theorem 6.3.13 holds for the language defined in Section 3.*

*Proof.* Same argument as Theorem 6.3.13, using Theorem 6.4.4. □

**Theorem 6.4.6** (Determinism by default)**.** *Theorem 6.3.14 holds for the language defined in Section 3.*

*Proof.* The same argument given in Theorem 6.3.14 goes through. □

# Chapter 7

# Specifying and Checking Effects for Framework APIs

This chapter extends DPJ as described in the previous chapters, so it can check that the uses of object-oriented parallel frameworks conform to their effect specifications. Section 7.1 discusses some limitations of effect systems, including DPJ, that we must overcome to support frameworks. Section 7.2 presents new programming techniques and effect system features for specifying and checking effects generated by the uses of frameworks. Section 7.3 discusses an evaluation of the techniques described, in which we wrote three separate frameworks and used each one to implement a realistic parallel algorithm. Section 7.4 discusses related work.

## 7.1 Limitations of Region-Based Systems

As the previous chapters illustrate, DPJ's effect system is quite expressive, and it is a natural choice for checking the effects of framework uses. However, all region-based effect systems, including DPJ, impose some limitations that we must address in our framework design. As we will see, by shifting some of the burden of guaranteeing noninterference from the type system to the framework, we can overcome some of these limitations.

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

Figure 7.1: `Node` class

To illustrate the limitations, consider the code in Figures 7.1 and 7.2. Figure 7.1 defines a simple list

```
1  class NodePair {
2      region First, Second;
3      Node<First> first in First;
4      Node<Second> second in Second;
5      NodePair(Node<First> first, Node<Second> second) pure {
6          this.first = first;
7          this.second = second;
8      }
9      void updateNodes(int firstData, int secondData) {
10         cobegin {
11             /* writes First */
12             first.data = firstData;
13             /* writes Second */
14             second.data = secondData;
15         }
16     }
17 }
```

Figure 7.2: Using region parameters to distinguish object instances

node class that we will also use in subsequent sections. The class has one region parameter `R`. The fields `data` and `next` in lines 2–3 are both located in region `R`. Figure 7.2 shows a simple container class, `NodePair`, that stores a pair of list nodes.

One limitation is that to guarantee soundness we have to prohibit swapping of `first` and `second` in the example:

```
void swap() {
    Node<First> tmp = first;
    /* illegal, can't assign Node<Second> to Node<First> */
    first = second;
    /* illegal, can't assign Node<First> to Node<Second> */
    second = tmp;
}
```

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

For this reason, DPJ and other region-based systems [80] use wildcard types that allow freer assignment. In DPJ, the wildcard type is a partially specified RPL (i.e., an RPL containing `*`), as described in Chapters 3 and 4. For example, in lines 3–4 of Figure 7.2, we could have written both types `Node<*>`, where `*` stands in for any region. Now the swapping shown above is fine, because the types of the variables don't constrain what regions can appear in the dynamic types of the references assigned to them. However, we have lost the ability to distinguish writes to `first.data` and `second.data` using the type system, because now

all we know is that the writes in lines 12 and 14 are to `*`. This is true even though by inspecting Figure 7.2, we (as opposed to the type system) can see that (1) regions `First` and `Second` are distinct coming into the constructor (line 5); and (2) the `swap` operation preserves the distinctness of `First` and `Second` in the dynamic types of `first` and `second`. So the state of the art in region-based type systems forces us to choose: either we can prove that two references don't alias, or we can swap the two references, but not both.

In fact, the situation is worse than this. As shown in Figure 7.3, a `NodePair` holding list nodes can have cross links. The effect system must ensure that (1) the objects associated with fields `first` and `second` are distinct; and (2) when following the references to access the objects in parallel, the cross links are never followed to update the same object. Further, we probably don't want to encode the write to `data` into the framework implementation, as shown in lines 12 and 14. Instead, as discussed in the introduction, we would like to express the operation abstractly, and let the user supply the specific operation. We therefore must constrain the effects of the user-supplied method so that we can argue that for any user-supplied method, this kind of interference cannot happen. Finally, we don't really want a `NodePair` class; instead, we want a `Pair<T>`, where `T` is a generic type.

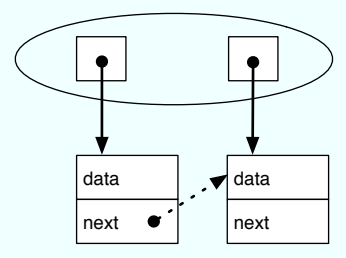


Figure 7.3: A potential race caused by cross links. The references stored in the `NodePair` are distinct; but a race can still occur if the task operating on the left-hand reference follows the cross link represented by the dashed arrow and performs an update.

## 7.2 Safe, Reusable Parallel Frameworks

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

Finally, we address the problem of writing a correct framework implementation.

Although most of this section focuses on the disjoint container as an example, the work is not specific to container frameworks. Section 7.3 shows how to use the same techniques to write a framework for expressing pipeline parallelism, which is a parallel control idiom not available in basic DPJ. Moreover, the next section formalizes these techniques and their soundness properties in general terms, without specifically considering disjoint containers.

### 7.2.1 Abstract Disjoint Containers

We define an abstract data type called an *abstract disjoint container*, to use as an extended example. This type generalizes the trivial `NodePair` container introduced in the previous section. An abstract disjoint container is an abstract data type with the following properties:

1. *Stored references.* It contains references to other objects. The number of stored references can be fixed up front (as with an array) or changed dynamically (as with a resizable array or set).

2. *Slots.* The elements are conceptually stored in slots. An iteration over the elements in the container iterates over the slots. For example, for an array, the slots are the array cells; for a set the slots are the set elements; and for a tree the slots are the tree nodes.

3. *Disjointness of slot regions.* At runtime, every slot $s_i$ either is `null` or points to an object with a region $R_i$ in its type. For any $i \neq j$, if $s_i$ and $s_j$ are both non-`null`, then $R_i$ and $R_j$ are *disjoint* (i.e., $R_i$ and $R_j$ refer to nonintersecting sets of regions).

Property 1 is standard for a container ADT, e.g., any of the containers in `java.util`. We introduce property 2 just so we have a way to talk about the iteration space of a container that is independent of the internal storage pattern (array, tree, etc.). Property 3 is the key to ensuring soundness when the user calls an API method to iterate over the container and update its contents in parallel. As discussed more fully below, the API can constrain a user-supplied method to have effect `writes` $R_i$; that is, it may operate only on the region $R_i$ of slot $s_i$, and not on any $R_j$ of a different slot or any other region (such as a region containing a global variable).

In the interest of concision, we refer to the slots of the container, or the container itself, as "disjoint," when in fact we mean that the associated regions of the slots are disjoint. Note that both versions of

`NodePair` from Section 7.1 are instances of the abstract disjoint container type, where the slots are the fields `first` and `second`, and the associated regions are `First` and `Second`. `First` and `Second` are disjoint, because they are distinct names.

### 7.2.2 A List Node Container

We now show how to use the DPJ type system as described in previous chapters (i.e., without extending the effect system yet) to write an abstract disjoint container API that stores `Node` objects and allows safe parallel updates to the stored objects. The disjoint container *implementation* is not specified; it could be any container (set, list, tree, etc.). The point is that we will be able to write a container API that (1) stores as elements list nodes, which may have cross links between them, as shown in Figure 7.3; and (2) allows update operations on the elements to be done *safely in parallel*, despite the presence of the cross links. While the list node container is somewhat artificial, we will extend the example to a more generic (and more useful) container in later sections.

Writing the list node container API presents two problems: maintaining disjointness, and reasoning about effects. Our key insight is that through careful API design, together with judicious use of method region parameters, we can enforce restrictions like "a factory method must return a new object" or "an apply method must write only to the region of the object it is given." Further, we can impose these restrictions without exposing region names (such as `First` and `Second` in Figure 7.2), that would otherwise prevent swapping and other disjointness-preserving operations inside the framework.

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

An example of (a) is a tree rebalancing or array shuffling that operates only on the internal structure of the container. Here the problem is entirely reduced to writing a correct framework implementation (Section 7.2.4). In the case of (b), however, the framework must restrict what the user can do so that the framework author can reason soundly about uses of the container without knowing exactly what that use

will look like. A core example is putting things into a container. For the container to be useful, the user has to retain control over what is inserted in the container, and how and where those inserted things are created. The trick is to allow some control while still being able to reason about disjointness. We have explored the following two strategies: building one disjoint container from another, and controlled creation of contained objects.

Figure 7.4 shows the simple list node container API that we use to illustrate these strategies. There are two region parameters, RN and RC, because we want to refer separately to the nodes stored in the container, and the container itself. In line 1, we use a *region parameter constraint* (described in Chapter 3) to require that for any instantiation of NodeContainer that binds $R_1$ to RN and $R_2$ to RC, $R_1$:* and $R_2$ are disjoint. This ensures that reading the container to traverse the slots does not interfere with updating the contained objects.

*Building one disjoint container from another:* If we start with a disjoint container $C_1$, and we create a new disjoint container $C_2$, we can populate $C_2$ by copying the reference elements from the slots of $C_1$ to the slots of $C_2$, and $C_2$ will also be disjoint. An example is creating a tree out of the elements of an array or set.

```
1  public interface NodeContainer<region RN,RC | RN:* # RC> {
2
3      /* One linear container from another  */
4      public NodeContainer(NodeContainer<RN,RC> c) writes RC;
5
6      /* Controlled creation of contents */
7      public NodeContainer(NodeFactory fact, int size) writes RC;
8      public interface NodeFactory {
9          public <region R>Node<R> create(int i) pure;
10     }
11
12     /* Data parallel operation on all elements */
13     public void performOnAll(Operation<RN> op) reads RC writes RN:*;
14     public interface Operation {
15         public <region R>void operateOn(Node<R> elt) writes R;
16     }
17
18 }
```

Figure 7.4: Framework API for an abstract disjoint list node container

Line 4 of Figure 7.4 illustrates how we might implement this strategy in DPJ. It says that given one object of type NodeContainer<RN,RC> we can create another one. An important special case in DPJ is creating a disjoint container from an *index-parameterized array*. As described in Chapter 3, the index-

parameterized array type is an array `A` such that cell `A[`$i$`]` has a type like `ListNode<[`$i$`]>` that is parameterized by the integer value $i$. This guarantees disjointness for the array, because the region `[`$i$`]` is distinct in the type of each array cell. However, because the parameterized types are exposed to the rest of the program, it also means that we cannot shuffle the array elements without compromising soundness. (This is exactly the same problem discussed in Section 7.1, just with array cells rather than fields.) If we construct a disjoint container by copying in elements from the cells of an index-parameterized array, then we obtain a container that is disjoint, but on which we can also perform disjointness-preserving operations, such as shuffling, that were prohibited for the original array by doing them *internally* within the framework.

*Controlled creation of contained objects:* Lines 7–10 of Figure 7.4 illustrate this strategy, for an interface to `NodeContainer` that could be implemented in different ways (array, tree, etc). The container implementation does the actual object creation, but the user specifies the number of objects to create and provides a factory method specifying how to create the $i$th object. For example, a use could look like this, assuming a class `NodeArray` that implements `NodeContainer`:

```
/* Implement concrete create method */
public class MyFactory implements NodeContainer.NodeFactory {
    public <region R>Node<R> create(int i) {
        return new Node<R>(i, null);
    }
}
/* Declare new region names NodeRegion and ContainerRegion */
region NodeRegion, ContainerRegion;
/* Bind the declared names to the parameters in the type */
NodeContainer<NodeRegion,ContainerRegion> c =
    new NodeArray<NodeRegion,ContainerRegion>(new MyFactory(), 10);
```

This code creates a new `NodeArray` with 10 list nodes, such that the $i$th one has its `data` field set to $i$. `NodeRegion` and `ContainerRegion` are region names declared by the user and bound to the region arguments in the instantiated types.

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

This strategy gives the framework control over disjointness by hiding the actual regions in the types of the created objects; the user only ever deals with them through the method region parameter in the factory method. For example, an array framework instantiated with `RN = N` could give the `Node` object stored in slot $i$ the type `Node<N:[`$i$`]>`, where `N:[`$i$`]` is the index-parameterized RPL discussed above. Unlike the case of the index-parameterized array, however, that type would never be seen by the user, unless the framework allowed it. The framework might simply not provide any way to ask for a reference to the element in slot $i$. Or, it might give out such a reference with type `Node<N:*>`, saying that the exact region in the type is statically unknown. This is sufficient because, in most cases, the user code does not need to distinguish these types since the parallelism is encapsulated inside the framework. The framework could give out a reference with type `Node<N:[`$i$`]>` if it could soundly match references to their original indices, e.g., if no shuffling of references happened inside the framework.

**Reasoning about Effects:** Lines 17–22 of Figure 7.4 show the part of the API that allows the user to define a method and then pass that method into the container to be applied in parallel to all contained objects. For example, given reference `c` of type `NodeContainer<N,C>`, the user could do this:

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

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

*Effect of `operateOn`:* In the definition of the abstract `operateOn` method in the `Operation` interface (lines 20–21 of Figure 7.4), we again use a method region parameter `R`. We write the type of the formal parameter `elt` as `Node<R>`, and we specify the effect as `writes R`. This causes two things to happen. First, the DPJ type system requires that any user-supplied method implementing `operateOn` must have a declared effect that is a *subeffect* of `writes R`. For example, `reads R` is allowed, but reading or writing some other region is not. (The relevant rules for subeffects are given formally in the next chapter; see also Chapter 4.) Second, because the regions in the objects of the slots are disjoint, the actual regions bound to `R` at runtime will be disjoint as the framework traverses the slots and applies the user-supplied method. Together, these two facts guarantee that the effects of the iterations in the parallel traversal will be noninterfering.

As an example, consider the user code shown above for updating a `Node`. That code is legal, because `data` is declared `in R` inside `Node`, which becomes `in R` (because of the type of `elt`) in the scope of `operateOn`. However, following the `next` field to update `data` of a different object is *not* legal: because the `next` field has type `ListNode<*>`, the effect of that update is `writes *`, which is not a subeffect of `writes R` and so is not allowed. So the API prevents the problem noted in Figure 7.3 of causing a race by following cross links. The cross links themselves are allowed, but problematic traversals of them are not.

*Effect of `performOnAll`:* In Figure 7.4, we have written the effect of `performOnAll` as

```
reads RC writes RN:*.
```

This is correct if, for a particular implementation of the interface, (1) the slots have type `Node<RN:*>`; and (2) the implementation of `performOnAll` reads the container and applies the user's `operateOn` method to the references in the slots. The framework writer is responsible for ensuring that both facts are true. In fact, if the framework itself is written in DPJ, then both facts are checked by the DPJ compiler. We will have more to say in Section 7.2.4 about implementing the framework. For now, note that the effect of `operateOn` is the *only* effect on the regions of the nodes themselves; and because that effect is partially specified (`RN:*`), the framework has freedom to implement the slot regions in different ways.

### 7.2.3 Getting More Flexibility

As noted above, the list node container is a somewhat artificial example; it is too specialized to be really useful. We now show how to extend the example to make it more generic. Doing this will require some extensions to the DPJ effect system, as discussed below.

**Making the Effects Generic:** The first thing that is too restrictive is the bound on the effects of the user-defined `operateOn`. For instance, what if the user wants to specify an `operateOn` method that reads some other region that is disjoint from $R$`:*`, where $R$ is the region bound to `RN` in the instantiation of the framework interface? That is safe and should be allowed, because it cannot interfere with the effect `writes RN:*` of `performOnAll`. Yet it is disallowed by the effect specification `writes R` in the API.

To address this problem, we use effect polymorphism [82]. We give the `Operation` interface an effect parameter `E` (similar to a region parameter, but it specifies an effect) that becomes bound to an actual effect

when the interface is instantiated into a type. To make this strategy work, we need to solve two problems: (1) constraining the effect arguments so that the effects of invoking the user-supplied method on different objects are noninterfering; and (2) ensuring soundness of subtyping when we add effect parameters.

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

Figure 7.5: Making the effects of the `Operation` interface generic

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

Figure 7.5 shows how to write the effect variables and constraints. We define the `Operation` interface (line 1) with an effect variable `E`. We also give the `performOnAll` method (lines 6–8) a *constrained method effect parameter* `E`. After the parameter declaration is a constraint specifying that the effect bound to `E` must be noninterfering with `reads RC writes RN:* effect E`. This constraint ensures that the supplied effect will not interfere with (1) the effect `reads RC` of reading fields of the container; (2) the effect `writes RN:*` of updating the nodes; and (3) itself. The latter means that `E` must either be a read-only effect, or it must be an effect such as a set insert that is declared to commute with itself (see Chapter 3).

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

```
public class MyOperation implements
  NodeContainer.Operation<reads GlobalRegion> {
    public <region R>void operateOn(Node<R> elt)
      reads GlobalRegion writes R {
        /* Assume global is in region GlobalRegion */
        elt.data = global;
    }
}
c.<reads GlobalRegion>performOnAll(new MyOperation());
```

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

As a matter of notation, notice that in lines 6–8 of Figure 7.5, the effect appearing in the constraint on the method effect parameter `E` (line 6) is identical to the effect of the method for which the parameter is declared (line 8). This is a common case. In this case, as a shorthand, we allow the user to omit the constraint and just declare the parameter `E#`. Using this shorthand, lines 6–8 of Figure 7.5 would look like this:

```
public <effect E#>void performOnAll(Operation<effect E> op)
    reads RC writes RN:* effect E;
```

*Soundness of subtyping:* Once we add class types like `C<`$E$`>`, where $E$ is an effect argument, we need a rule for deciding if `C<`$E_1$`>` is a subtype of `C<`$E_2$`>`. We could require that $E_1$ and $E_2$ be identical effects, but this would be unnecessarily restrictive. Instead, we let $E_1$ be a *subeffect* of $E_2$. With this approach, the key to showing the soundness of effect is to show *type preservation*, i.e., that the dynamic types of object references always agree with the static types of variables that hold them.

However, enforcing type preservation in the presence of effect variables is tricky. For example, consider the following snippet:

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

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

As noted in Chapters 3 and 4, a similar problem occurs with Java generic wildcards and in basic DPJ with partially specified RPLs. The solution is to make the static type of `x.f` `C<effect` $E'$`>`, where $E'$ is a fresh effect parameter (called a *capture parameter*). The tricky thing here is that *all nonempty effects must be captured when substituted for an effect parameter in a type*. This is because all nonempty effects are

essentially wildcards: the runtime effect could be equal to the static effect, or it could be empty (or possibly something else, e.g., `reads` $R$ instead of `writes` $R$, or `reads` $R_1$ instead of `reads` $R_1, R_2$).

**Making the Type Generic:** The second thing that is too restrictive is that we made the class specialized to list nodes. Instead, we would like to write a generic class

```
DisjointContainer<type T, region Cont>.
```

Notice, however, that there are two places where we used the region argument to the `Node` type to write the API. First, in writing the `NodeFactory` interface (line 10 of Figure 7.4), we used a method-local parameter `R` in the return type of `create`. Second, in writing the effect of `performOnAll` (lines 6–8 of Figure 7.5), we used the region `RN` to write both the effect constraint and the effect of updating the contained objects. If we just replaced these types with an ordinary type variable `T`, then we would not be able to write the node factory pattern at all, we would not be able to constrain the effect `E` properly, and we would be forced to use a more conservative effect (such as `writes *`) for the effect of `operateOn`.

To solve this problem, we can use a type constructor [92, 12] that takes a region argument. In our language, type constructors work as follows:

1. A type variable `T` can be declared `type T<region R>`, where `R` declares a fresh parameter. We call `R` a *type region parameter*, by analogy with a class region parameter, which declares a region parameter in a class definition. When a type $T$ becomes bound to `T`, $T$ must have at least one region argument, and `R` represents the first region argument. For instance, if $T = C$`<`$r$`>`, then `R` represents the region $r$.

2. We write uses of the variable `T` as `T<`$r$`>`, where $r$ is a valid region in scope. `R` itself is valid because it was declared in the type variable. `T<R>` represents the unmodified type provided as an argument to the variable, while `T<`$r$`>` represents the same type with the region in its first argument position replaced by $r$.

For convenience, a bare use of `T` is allowed within the class body, and this is equivalent to `T<R>` (in other words, the type constructor `T` also functions as a type, with implicit argument `R`). We can also write $n$ parameters (`T<region` $R_1$`,...,`$R_n$`>`) and arguments (`T<`$r_1$`,...,`$r_n$`>`), for $n \geq 1$. In this case the argument must have at least $n$ parameters, and the first $n$ region arguments are captured, starting from the

left.

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

Figure 7.6: API for an abstract disjoint container with generic types and effects

**Final Container API:** Figure 7.6 shows the final disjoint container API. Line 1 declares an interface `DisjointContainer` with one type parameter `T` and one region parameter `Cont`. The type parameter has one region parameter `Elt` that names the first region argument of the type bound to `T`. In line 11, we write `T<R>` to require that the return type of `create` have the method region parameter `R` as its first region argument. In line 16, the region `Elt` is available to write the effects of `performOnAll`. We do the same thing for the type parameter of the `Operation` interface, in line 17.

Here is an example implementation of `operateOn`, where `c` has type

`DisjointContainer<Node<N>,C>`.

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

This code is identical to the example given in Section 4.2.2, except that it instantiates the generic container instead of the specialized one. The effect argument is `pure`, because no effect is needed for this

implementation of `operateOn`, except for `writes R`, which is already given by the interface (line 19 of Figure 7.6). The effect of the call to `performOnAll` is `reads C writes N:*`.

### 7.2.4 Writing the Framework Implementation

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

**Type preservation:** The soundness results presented in Chapter 8 show that type preservation holds for DPJ as extended in this chapter. Therefore, if the framework is written in DPJ, then this property will be checked "for free," unless the framework does an assignment (using a cast) that violates the typing rules. The DPJ subtyping rules are quite flexible, so we anticipate that unsound assignments will rarely be needed in practice to work around expressivity constraints of DPJ.

A more likely case is that casts are used to interface with code that is not implemented in DPJ, such as an off-the-shelf Java container implementation. In this case, the framework author must reason about type preservation using the specification of the non-DPJ code. For example, pre-Java 5 code implementing a container might represent the container slots as references to `Object`. If references to be stored into the slots always have type `T<R>`, then it would be sound to cast these references to `Object` when putting them in the container, and back to `T<R>` when taking them out. For Java code written with generics, such casts should be rare.

**Effect preservation:** Effect preservation means that the static effect of every statement is a supereffect of the actual runtime effect of every execution of that statement. Again, the extended language guarantees this property, so long as (1) type preservation holds; and (2) every method summary covers the effects of the method body. In DPJ, one can always write a correct method summary (in the extreme case `writes *` is always correct). So property (2) will hold if property (1) does. Here, if the framework calls into non-DPJ

code, then the framework writer will have to reason about effects manually (i.e., the reasoning cannot be checked by DPJ).

**Noninterference of effect:** Noninterference of effect means that parallel tasks have no conflicting memory accesses. While DPJ can establish noninterference in many cases, in some cases it may not be able to. For example, even if a pair of references of type `C<*>` always points to objects with distinct regions at runtime, the type system can't prove that, as discussed in Section 7.1.

```
1  public class DisjointArray<type T<region Elt>, region Cont |
2    Elt:* # Cont> implements DisjointContainer<T,Cont> {
3
4      /* Internal array representation */
5      private DPJArrayList<T<Elt:*>,Cont> elts in Cont;
6
7      /* Implementation of performOnAll */
8      public <effect E#>void performOnAll(Operation<T,effect E> op)
9        reads Cont writes Elt:* effect E {
10         foreach (int i in 0, elts.size()) {
11             op.operateOn(elts.get(i));
12         }
13     }
14
15     /* Swap elements at idx1 and idx2 */
16     public void swap(int idx1, int idx2) writes Cont {
17         T<Elt:*> tmp = elts.get(idx1);
18         elts.add(idx1, elts.get(idx2));
19         elts.add(idx2, tmp);
20     }
21
22 }
```

Figure 7.7: Array implementation of a disjoint container (partial). `DPJArrayList` (line 5) is an ordinary Java `ArrayList`, annotated with region information. The effect of `elts.get(i)` (line 11) is `reads Cont`.

In such cases, the framework author has the freedom to "go outside" the type system, and use a different technique to make the noninterference argument. Figure 7.7 shows an example. This an array implementation of `DisjointContainer`. We have chosen to represent the array internally as a `DPJArrayList`, as shown in line 5. The type argument to `DPJArrayList` is `Elt:*`, reflecting the fact that the dynamic type of element $j$ is `Elt:[`$j$`]`, as discussed in Section 7.2.1. The `performOnAll` method uses the DPJ `foreach` construct (line 10) to iterate in parallel over the slots of the `DPJArrayList` and apply the user-supplied operation to each of its elements. We also add a `swap` method, similar to the method discussed in Section 7.1, for swapping two elements of the array.

To show noninterference, it suffices to establish two things: (1) for distinct values $i$, the region in the dynamic type of `elts.get(`$i$`)` at line 11 is distinct; and (2) `i` attains distinct values $i$ on distinct iterations of the `foreach` in line 10. The first statement follows from the inductive argument we made in Section 7.2.1 about maintaining disjointness: to change the shape of the array, we either have to use an inherited creation method, which preserves disjointness as discussed in Section 7.2.2, or do a swap, which also preserves disjointness, as can be seen from the implementation in lines 17–19. The second statement follows from the semantics of `foreach` in DPJ (Chapter 3). More generally, one would follow the same two-pronged strategy to show noninterference for an a traversal over an arbitrary disjoint container: first show disjointness of slot regions $R_i$, and then argue that the traversal operates in parallel on the slots.

Notice that once the framework implementer checks noninterference in this way, the user never has to see or even know about how the checking occurred. From the user's point of view, if the program type checks, then the noninterference property holds. Further, the framework writer is free to use static or dynamic verification techniques such as program logic, model checking, or testing to check the framework implementation. We can thus think of the techniques presented here as making DPJ into an *extensible language*. By writing a suitable API, and doing appropriate checks, the framework writer can add new capabilities for parallel operations that provide the same guarantees as if those capabilities had been built in as first-class parts of the language. A good example of this extensibility is the pipeline framework described in Section 7.3, which supports a parallel control structure that cannot be expressed in the DPJ language at all. This extensibility makes DPJ much more powerful than if the only checking mechanism were the type system itself.

## 7.3 Evaluation

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

1. Can we use the techniques to write realistic frameworks and user programs? Do any additional issues arise in real frameworks or user code?

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

To perform our evaluation, we first extended the DPJ compiler to support effect variables, effect constraints, and type region parameters as discussed in Section 7.2.3 and Chapter 8. Then we studied how to (1) use our techniques to write generic array, tree, and pipeline frameworks; and (2) use the frameworks to write three parallel codes: a Monte Carlo simulation algorithm, a Barnes-Hut n-body computation using a spatial octtree, and RadixSort expressed as a pipeline. We chose these three algorithms because they exemplify different styles of parallelism: Monte Carlo uses direct loop-style parallelism over arrays; Barnes-Hut uses recursive, divide-and-conquer parallelism over trees; and RadixSort uses concurrent pipelined computations over a stream of inputs.

### 7.3.1 DPJ Frameworks

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

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

1. A `create` method that creates an array with a user-supplied factory method, as discussed in Section 7.2.1.

2. A `withMapping` method that maps one array to another, element by element, with a user-supplied mapping function. Like ParallelArray, we provide two forms of the mapping: the first takes an index variable, and the second does not. As in the factory method pattern, we use a method region parameter `R` to ensure that the mapping function creates a new output object for each element, and the mapping function is allowed to write `R`.

3. A `reduce` method that reduces the array to an object, given a starting element and a user-specified `Reducer` that combines two elements into one. Following the pattern discussed in Section 7.2, the two elements coming into the `Reducer` method are parameterized by method region parameters `R1` and `R2`, and the user-supplied method is allowed to write the regions bound to these parameters. Using distinct parameters ensures that the `Reducer` cannot violate disjointness, e.g., by storing one object into a field of the other.

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

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

1. A `buildTree` method that takes a `DPJDisjointContainer elts` of objects of type `T` and a positive `arity` and inserts the bodies into the leaves of the tree. The user provides an `index` function that takes a `T` to insert, a `T` at the current (inner or leaf) node, and a `T` at the parent node of the current node, and computes which of the children of the current node to follow next when inserting the object in the subtree rooted at the current node. The framework creates the inner nodes as necessary and populates each one with a fresh object of type `T`, using a user-specified factory method.

2. A `visitPO` method that recursively does a parallel postorder tree traversal. As shown in Figure 7.8, this method takes a user-supplied `visit` method that, given a `T` object at the current node and an ArrayList of `V` (result) objects produced from visiting the children (or `null` if the current node is a leaf), produces a `V` object for this node. Again we use two region parameters, `R1` and `R2`, to ensure that disjointness of the `T` objects is preserved by the traversal.

```
1  public class DisjointTree<type T<region Elt>, region Cont>
2    implements DisjointContainer<T,Cont> {
3
4      public <effect E#>double visitPO(POVisitor<T, effect E> visitor)
5           reads Cont writes Elt:* effect E { ... }
6
7       public interface POVisitor<type T<region Elt>,
8                                  type V<region VR>, effect E> {
9           public <region R1, R2> V<R2>
10            visit(T<R1> data, ArrayList<V<R2>, Cont> childResults)
11              reads Cont writes R1, R2 effect E;
12      }
13 }
```

Figure 7.8: The postorder visitor from the region-based spatial tree.

**Parallel pipeline framework:** We implemented a framework called DPJPipeline that supports applications structured as data flowing through a series of pipeline stages, each of which operates on the data. Following

Intel's Threading Building Blocks (TBB) [101] and the StreamIt language [118], we call the operation applied by each stage a *filter*. Each data element flows sequentially through the stages, but different stages can apply their filters to different elements at the same time, creating pipeline parallelism. This parallel control structure cannot be expressed directly in DPJ as described in Chapters 3 through 6.

The DPJPipeline API is parameterized by a type `T<TR>` for the type of an element, a region `PR` for the pipeline internals, and an effect `E` that bounds the user-specified effects of the filters. The effect `E` is constrained not to interfere with writing under `TR` or `PR`, or with itself, ensuring that filters may safely update the data elements and the pipeline state. The API provides two interfaces for the user to implement: a filter and a factory method for creating a filter. The API also provides the following methods for the user to invoke directly:

1. A method `appendStageWithFilter` that accepts a user-defined filter factory, uses it to create a fresh filter, and inserts a stage with that filter at the tail of the pipeline.

2. A method `launch` that launches one task for each pipeline stage.

Internally, each stage is represented by an object of type `Stage` (a private class, not visible to the user) that stores the user-specified `Filter` for that stage and maintains an output buffer for the data items produced by that stage. The output buffer of a stage is the input buffer for the next stage. Extending our framework to a recursive fork-join graph, as supported in StreamIt, or a general DAG would not be difficult.

Effect management for this framework works as follows. Method region parameters on the user-defined factory methods as discussed previously ensure that each filter and each element is a freshly-created object, each in its own region. The `Filter` interface looks like this:

```
public interface Filter<type T<region TR>, region FR, effect E> {
    public <region R>T<R> op(T<R> item) writes R, FR effect E;
}
```

As in the previous examples, this method is invoked only by the framework, in the stage implementation. At a particular invocation of `op`, `R` is bound to the region of the data element being operated on, which is under the region bound to `TR` in the `DPJPipeline` class, and `FR` is bound to the region associated with the current stage, which is under the region bound to `PR` in the `DPJPipeline` class. The actual effect bound to `E` is supplied in the instantiation of the framework and is constrained as discussed above. Thus the user-defined filter operation is limited to updating the regions of the data object and the filter state, and

doing any other noninterfering effects. In particular, it cannot update a data element being operated on by a concurrent filter, or a different filter.

The framework implementation passes the object returned by the filter operation from one stage to the next. The returned object need not be the same as the object passed in. However, the region parameter `R` ensures that the object returned has the same region bound to its type as the input object. In particular, the return object cannot be a data element processed concurrently by a different stage, or even a data element reachable from such a data element, except through a partially-specified RPL.

### 7.3.2 Application Code

**Monte Carlo simulation:** We studied the Monte Carlo simulation benchmark from the Java Grande suite [110]. The computation contains three parallelizable loops: the first one creates `Task` objects; the second one iterates over the objects to compute a return rate for each one; and the third one reduces the return rates into a cumulative average.

We parallelized all three loops using `DPJDisjointArray`. For the first loop, we used the indexed form of `withMapping`. Apart from writing to the `Task` object itself (which does not have to be reported), the effect of the `Task` constructor is read-only, so it can validly be used for aggregate array creation, as shown in line 8 of Figure 7.6.

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

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

**Barnes-Hut center of mass computation:** Next we studied the Barnes-Hut n-body simulation [109], which uses an octree (eight-ary tree) to represent three-dimensional space hierarchically, storing the bodies in the leaves. We focused on the center-of-mass computation, which traverses the tree recursively in parallel and computes, for each node, the center of mass of the subtree rooted at that node. The computation writes into each node as it traverses it, so the noninterference argument requires that the traversal is over a tree. Because

of this fact, the center of mass computation is hard to do efficiently in baseline DPJ; we discuss this point further in Section 7.3.3 below. It would be straightforward to parallelize the force computation using the same array-based techniques that we used for Monte Carlo.

We wrote a program that builds a tree and performs a center of mass computation for a binary tree computation in one-dimensional (1-D) space. 1-D space simplifies the computation, without changing the essential patterns of parallelism. We instantiated `DPJDisjointTree` with a `Node` class that has subclasses `Cell` for the inner node data and `Body` for the leaf data, similarly to both the original and Splash-2 versions of Barnes-Hut [109]. To build the tree, we wrote an `index` method that puts each inserted node in the left or right subtree based on its position, and a factory method that constructs fresh `Cell` objects for each inner node in the tree. To compute the center of mass, we wrote a `postOrderVisitor` that computes the average position and total mass for the bodies in the subtree rooted at each inner node and stores them at the node. This visitor returns a pair of `double` values (for type `V` in the API) for the average position and total mass at the current node.

**Pipelined radix sort:** We used the DPJPipeline framework to write a pipelined version of radix sort. This application is directly modeled after the StreamIt RadixSort benchmark [118]. The first stage produces a stream of arrays to sort, and the successive stages each sort the arrays on a different radix, with the radix recorded in the `Filter` object as `final` variable (so reading it produces no effect). Each sort stage also stores two temporary arrays as persistent mutable data in the filter of the region (such that accessing the arrays produces an effect on the filter region).

When an array enters a sort stage, the filter for that stage adds each array element to one of the temporary arrays, depending on whether the element has a 0 or 1 at the bit position corresponding to the radix for that filter. The filter then copies all the 0 elements followed by all the 1 elements back into the original array, and passes it along to the next stage.

### 7.3.3 Discussion of Evaluation Results

**Support for realistic frameworks:** Our experience shows that the framework techniques in this work can be used to write realistic parallel algorithms. For these codes, we did not find any significant challenges over and above the framework API we discussed in Section 7.2. In the future, we could also easily support other operations, such as ParallelArray's filter and apply.

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

Table 7.1: Annotation counts for the framework code

Getting the region and effect annotations correct for the framework APIs, and using the API design to check noninterference, did require some careful thought. However, all the APIs have a similar pattern; once we mastered that pattern, writing the APIs as discussed in Section 7.2 was straightforward.

Table 7.1 summarizes the effect annotation counts for the framework code. The leftmost data column shows the annotated over the total source lines of code (SLOC), counted with `sloccount`. From the left, the other columns show the number of class (including interface) definitions, class region and effect parameters, class region and effect constraints, region and effect arguments to types, method definitions, method effect summaries, method region and effect parameters, method region and effect constraints, and region and effect arguments to methods. For arguments to class types, the denominator is the total number of types appearing in the program; and for arguments to methods, the denominator is the total number of method invocations.

As expected, the annotations are nontrivial; this is simply a cost of the safety guarantee we provide. We believe the numbers are higher than they would be for production frameworks, because effect annotations appear on the API, and production frameworks would have a higher ratio of internal to API code than our simple frameworks do. Thus, production frameworks should better amortize the overhead of writing the annotations.

**Framework client experience:** Table 7.2 shows the annotation counts for the client code, with the same layout as Table 7.1. As expected, the relative annotation burden is less than for the framework code. As with the deterministic effect system discussed in Chapter 3, most of the annotations are method effect summaries and region arguments to types. In the client codes, the arguments to effect variables were simple: either `pure` or one or two read effects. As expected there were no effect constraints in the client code, only in the framework code.

It is also instructive to compare the client experience for the Monte Carlo and Barnes Hut algorithms written using frameworks to the corresponding ones using baseline DPJ, as presented in Section 3.5. For Monte Carlo, we had used an index-parameterized array to guarantee disjointness in the first two loops, by

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

Table 7.2: Annotation counts for the client code

making the `Task` and `Result` types parameterized by the index `i`. For the third loop, we encapsulated the reduction sum in a method implemented with locks and declared that method `commutative`. This is not attractive because it puts the burden of writing low-level, error-prone synchronization code on the application developer.

Similarly, we could use baseline DPJ to parallelize the center of mass computation in Barnes-Hut. However, we would have to give each tree node a distinct type and *recopy the bodies on insertion into the tree*, because we cannot soundly change the type of a reference in DPJ, as discussed in Section 7.1. We could support such "ownership transfer" with runtime reference counting [13], but this would add its own overhead.

The pipeline framework illustrates a different benefit of this work. Pipelining is a new parallel control idiom that is not provided by DPJ and, even if it were, no useful pipelined parallelism would be expressible in the DPJ type system as explained earlier. Implementing it as a framework elegantly extends the capabilities of the language, while preserving the ability to enforce the DPJ safety properties for pipelined application code. It is an example of a higher level "coordination" mechanism [76] that is used to connect and manage multiple deterministic components, potentially operating concurrently with each other while exchanging data.

Overall, the advantages of the framework approach are (1) simplifying the DPJ types exposed to the client, by avoiding index parameterized arrays or recursive types; (2) eliminating low-level code for common patterns such as reductions; (3) avoiding copies where the baseline type system might require them, as in Barnes-Hut; and (4) extending the language with more flexible parallel control idioms. On the other hand, the baseline DPJ code is closer to the original sequential code, because it uses parallel control constructs directly, rather than factoring the code into helper functions and framework API calls. This last point is not specific to our work, but is a general issue with using frameworks.

## 7.4 Related Work

**Effect systems:** Sections 3.6 and 5.6 discuss the related work on effect systems. None of this work teaches how to write a framework API for safe parallelism using disjoint data structures. Nor does it support mechanisms such as effect constraints and type region parameters that are necessary for generic frameworks.

**Linear type systems:** Wadler [125] introduced linear types as a way to allow in-place updates while preserving the semantic guarantees of pure functional programming. A linear type system can enforce strong guarantees of program correctness [41]. However, linear types prohibit reference aliasing, which makes many common patterns of imperative programming awkward or impossible.

Several researchers have looked at ways to make linear types less restrictive while maintaining meaningful guarantees. Fähndrich and DeLine [47] introduced *adoption and focus* to create aliases of a linear reference with a limited lifetime. Clarke and Wrigstad [38] have observed that *external uniqueness* — the property that every object has at most one reference to it located outside its containing data structure — can express important patterns, such as a unique reference to a doubly-linked list. Boyland and others [26, 117] have used *fractional permissions* to enforce linearity of write references, while allowing sharing of read-only references. Finally, several researchers have shown how to combine unique references with effect systems in interesting ways [59, 25].

Our idea of *disjoint data structures* is related to these mechanisms, but also different from all of them. Our insight is that for parallel traversals over the slots of a data structure, all we care about is whether the slots have different regions in their types. This implies that the slots point to distinct objects, but it does not preclude aliasing with other references in the program. DPJ's indexed parameterized arrays (Chapter 3) provide disjoint regions, but they do so by making the regions explicit in user code, thereby preventing reference swapping as discussed in Section 7.1.

**Enforcing API contracts.** The Eiffel language [119] introduced the idea of *design by contract*, which uses preconditions and postconditions to specify interaction between classes. The Java Modeling Language (JML) [75] provides a powerful way to write design-by-contract specifications for Java, which can be checked with a combination of static verification and online checking.

Design by contract ideas have been applied to concurrent programming. Meyer's Systematic Concurrent Object-Oriented Programming (SCOOP) concurrent programming model [89] is based on Eiffel. The Fortress programming language [116] provides a way to write assertions at interface boundaries that can be

checked at runtime. X10 [35] has a sophisticated dependent type system that can specify and check interface assertions, also supported with runtime checking. None of this work addresses parallel noninterference or safe frameworks for shared memory parallelism.

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

**Type constructors:** Type constructors are well known in functional languages like Haskell. Recently type constructors (also called *kinded types*) have been applied to object-oriented languages [92, 12]. A kinded type is like a "type type parameter," i.e., a type parameter with a type parameter, whereas our type region parameter is a type parameter with a region parameter. Also, in work on kinded types, there is no notion of effects or the sound interaction of type parameters with effect judgments.

Another related concept is the C++ mechanism called *template template parameters* [123]. If we followed that approach, we would have the user provide a class $C$ and a region $R$ as separate arguments to the framework, and the framework would put them together to construct the type $C$<$R$>. We chose not adopt this approach because it obscures the relationship between the type and its region argument in the framework API.

# Chapter 8

# Formal Language for Framework API Checking

This chapter formalizes the ideas discussed in the previous chapter, using a formal language similar to the languages presented in Chapters 4 and 6. The language has the following salient features:

- Like Core DPJ (Chapter 4), this language focuses on the mechanisms for expressing effects and noninterference. We do not model parallel constructs, nor do we include the features for supporting nondeterminism discussed in Chapters 5 and 6.
- Compared to Core DPJ, the language has simplified RPLs, no `let` expressions, and no arrays.
- The language incorporates interfaces, classes that implement interfaces, and method parameters, in order to support the style of writing object-oriented frameworks discussed in the previous chapter.
- The language incorporates the effect system features introduced in the previous chapter, i.e., constrained effect variables and type region parameters.

As usual, we give a syntax, a static semantics, a dynamic semantics, and soundness results with proofs.

## 8.1 Syntax

Figure 8.1 shows the syntax for the formal language illustrating the framework support. A program $\mathcal{P}$ consists of region name declarations, interface definitions, class definitions, and an expression to evaluate. An interface $\mathcal{I}$ consists of an interface name $I$, the interface parameters, and zero or more method signatures. There is one type parameter $\tau$, one region parameter $\rho$, and one constrained effect parameter $\eta \# E$. The type parameter $\tau$ has a region parameter $\rho$ that captures the region argument of the type bound to it. A method signature $S$ specifies a region parameter, a constrained effect parameter, a return type, a method name $m$, a typed formal parameter $x$, and an effect.

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

Figure 8.1: Syntax of the formal language supporting frameworks. $r$, $I$, $\tau$, $\rho$, $\eta$, $C$, $f$, $m$, and $x$ are identifiers.

A class $\mathcal{C}$ consists of a class name $C$, the class parameters, the interface type being implemented, and the fields and methods of the class. There is one type parameter and one region parameter. For simplicity, classes do not take effect parameters; the interface effect parameters suffice to write the patterns discussed in the previous chapter. A field $F$ specifies a type, a field name $f$, and an RPL. A method specifies a signature and an expression to evaluate.

A region path list (RPL) $R$ is a named region $r$, a region parameter $\rho$, or an RPL qualified by appending `:`$r$ or `:*`, where `*` stands in for any chain of names. A type $T$ instantiates a named interface with a type, region, and effect; or it instantiates a named class with a type and region; or it instantiates a type parameter with a region; or it is `Null`. `Null` is the type of a null reference. It also functions as a base-case type for type parameter arguments (every other type has its own argument). An effect $E$ is a possibly empty union of read effects, write effects, and effect parameters.

An expression $e$ is a field access, field assignment, method invocation, variable, object creation, or null reference. A variable $v$ is `this` or a method formal parameter $x$.

## 8.2 Static Semantics

### 8.2.1 Typing Environment

The static typing judgments are defined with respect to an environment $\Gamma$:

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

$(z, T) \in \Gamma$ means that variable $z$ has type $T$. $\tau \in \Gamma$ means that type parameter $\tau$ is in scope in $\Gamma$. $\rho \in \Gamma$ means that region parameter $\rho$ is in scope in $\Gamma$. $\eta \in \Gamma$ means that effect parameter $\eta$ is in scope in $\Gamma$. $\eta \# E \in \Gamma$ means that effect parameter $\eta$ is constrained to be disjoint from effect $E$.

### 8.2.2 Programs

**Valid programs:** The judgment $\vdash \mathcal{P}$ means that program $\mathcal{P}$ is valid. The judgment holds if the interfaces and classes of $\mathcal{P}$ are valid, and the main expression of $\mathcal{P}$ is well typed with type $T$ and effect $E$ in the empty environment.

$$\boxed{\vdash \mathcal{P}} \quad \text{PROGRAM} \quad \frac{\forall \mathcal{I}.(\vdash \mathcal{I}) \quad \forall \mathcal{C}.(\vdash \mathcal{C}) \quad \emptyset \vdash e : T, E}{\vdash \mathcal{R}^* \; \mathcal{I}^* \; \mathcal{C}^* \; e}$$

**Valid interfaces:** The judgment $\vdash \mathcal{I}$ means that interface definition $\mathcal{I}$ is valid. The judgment holds if the effect constraint of $\mathcal{I}$ is a valid effect, and all the method signatures of $\mathcal{I}$ are valid. We check these facts in the environment $\Gamma$ consisting of the declared parameters and effect constraint of $\mathcal{I}$.

$$\boxed{\vdash \mathcal{I}} \quad \text{INTERFACE} \quad \frac{\Gamma = \tau \cup \rho_\tau \cup \rho \cup \eta \cup \eta \# E \quad \Gamma \vdash E \quad \forall S.(\Gamma \vdash S)}{\vdash \texttt{interface}\; I\texttt{<}\tau\texttt{<}\rho_\tau\texttt{>}, \rho, \eta \# E\texttt{>}\; \{\; S^* \;\}}$$

**Valid classes:** The judgment $\vdash \mathcal{C}$ means that class definition $\mathcal{C}$ is valid. The judgment holds if $\mathcal{C}$ implements a valid interface type; its fields are valid; and its methods are valid. We check these facts in the environment consisting of the declared parameters of $\mathcal{C}$. For checking methods, we also record the base type to the left of $\vdash$, so that the method-checking rule (METHOD, below) can check the method against its specification in the interface, if any.

$$\boxed{\vdash \mathcal{C}} \quad \text{CLASS} \quad \frac{\begin{array}{c}\Gamma = \tau \cup \rho_\tau \cup \rho \cup (\texttt{this}, C\texttt{<}\tau\texttt{<}\rho_\tau\texttt{>}, \rho\texttt{>}) \quad \Gamma \vdash I\texttt{<}T, R, E\texttt{>} \\ \forall F.(\Gamma \vdash F) \quad \forall M.(\Gamma, I\texttt{<}T, R, E\texttt{>} \vdash M)\end{array}}{\vdash \texttt{class}\; C\texttt{<}\tau\texttt{<}\rho_\tau\texttt{>}, \rho\texttt{>}\; \texttt{implements}\; I\texttt{<}T, R, E\texttt{>}\; \{\; F^*\; M^* \;\}}$$

**Valid method signatures:** The judgment $\Gamma \vdash S$ means that signature $S$ is valid in environment $\Gamma$. The judgment holds if the formal parameter type, return type, effect constraint, and effect summary of $S$ are

valid. We check these facts in the environment $\Gamma'$ formed by adding the method parameters and constraint to $\Gamma$.

$$\boxed{\Gamma \vdash S} \quad \text{SIGNATURE}\ \frac{\Gamma' = \Gamma \cup \rho \cup \eta \cup \eta \# E \quad \Gamma' \vdash T \quad \Gamma' \vdash T' \quad \Gamma' \vdash E \quad \Gamma' \vdash E'}{\Gamma \vdash \texttt{<}\rho, \eta \# E\texttt{>}T\ m\texttt{(}T'\ x\texttt{)}\ E'}$$

**Valid fields:** The judgment $\Gamma \vdash F$ means that field $F$ is valid in environment $\Gamma$. The judgment holds if the type and region of $F$ are valid in $\Gamma$.

$$\boxed{\Gamma \vdash F} \quad \text{FIELD}\ \frac{\Gamma \vdash T \quad \Gamma \vdash R}{\Gamma \vdash T\ f\ \texttt{in}\ R}$$

**Valid methods:** The judgment $\Gamma, T \vdash M$ means that method $M$ is valid in environment $\Gamma$, where $T$ is the interface type implemented by the enclosing class. The judgment holds if the method signature is valid; the method body $e$ is well typed with type $T_e$ and effect $E_e$; $T_e$ is a subtype of the declared return type; and $E_e$ is a subeffect of the declared effect; and if a signature named $m$ appears in the implemented interface type, then the method conforms to the signature given in the interface.

$$\boxed{\Gamma, T \vdash M} \quad \text{METHOD}\ \frac{\begin{array}{c} S = \texttt{<}\rho, \eta \# E\texttt{>}T\ m\texttt{(}T'\ x\texttt{)}\ E' \quad \Gamma \vdash S \\ \Gamma' = \Gamma \cup \rho \cup \eta \cup \eta \# E \cup (x, T') \quad \Gamma' \vdash e : T_e, E_e \quad \Gamma' \vdash T_e \preceq T \quad \Gamma' \vdash E_e \subseteq E \\ m \in \mathsf{Dom}(\mathcal{S}(I)) \Rightarrow \Gamma, I\texttt{<}T'', R, E\texttt{>} \vdash S \preceq \mathcal{S}(I)(m) \end{array}}{\Gamma, I\texttt{<}T'', R, E''\texttt{>} \vdash S\ \{\ e\ \}}$$

We check the type and effect of the method body in the environment $\Gamma'$ formed by adding the method parameters to $\Gamma$. We write $\mathcal{S}(I)(m)$ to mean the signature $S$ with name $m$ defined in interface $I$. If no signature named $m$ appears in the definition of $I$, then we say $m \notin \mathsf{Dom}(\mathcal{S}(I))$.

**Valid method implementations:** The judgment $\Gamma, T \vdash S \preceq S'$ means that signature $S$ in a class definition conforms to the signature $S'$ in the interface type $T$ implemented by the class. The judgment holds if the implementing return type is a subtype of the implemented return type; the implemented formal parameter type is a subtype of the implementing formal parameter type; the implemented effect constraint is a subeffect of the implementing effect constraint; and the implementing effect summary is a subeffect of the implemented effect summary.

$\boxed{\Gamma, T \vdash S \preceq S'}$

IMPLEMENT

$$\frac{\sigma = [\rho_2 \leftarrow \rho_1][\eta_2 \leftarrow \eta_1] \quad \Gamma \vdash \sigma(\phi_T(E_2)) \subseteq E_1 \quad \Gamma \vdash T_1 \preceq \sigma(\phi_T(T_2)) \quad \Gamma \vdash \sigma(\phi_T(T_2')) \preceq T_1' \quad \Gamma \vdash E_1' \subseteq \sigma(\phi_T(E_2'))}{\Gamma, T \vdash \texttt{<}\rho_1, \eta_1 \# E_1\texttt{>}T_1\ m\texttt{(}T_1'\ x\texttt{)}\ E_1' \preceq \texttt{<}\rho_2, \eta_2 \# E_2\texttt{>}T_2\ m\texttt{(}T_2'\ x'\texttt{)}\ E_2'}$$

Notice that we have to instantiate the types and effects appearing in the implemented interface definitions, using the implemented interface type $T$, before we can compare them to the implementing types and effects. To do this we use the translation mapping $\phi_T$, defined in in Section 8.2.8. We also need to substitute for the method region and effect parameters, as shown in the rule.

### 8.2.3 Regions

**Valid regions:** The judgment $\Gamma \vdash R$ means that region $R$ is valid in environment $\Gamma$. The judgment holds if $R$ is a declared region name $r$, or a region parameter $\rho$ in scope, or a valid region with a valid $r$ or `*` appended.

$\boxed{\Gamma \vdash R}$

RGN-NAME $\dfrac{\texttt{region}\ r \in \mathcal{P}}{\Gamma \vdash r}$ RGN-PARAM $\dfrac{\rho \in \Gamma}{\Gamma \vdash \rho}$ RGN-RPL $\dfrac{\Gamma \vdash R \quad \Gamma \vdash r}{\Gamma \vdash R\texttt{:}r}$ RGN-STAR $\dfrac{\Gamma \vdash R}{\Gamma \vdash R\texttt{:*}}$

**Inclusion of regions:** The judgment $\Gamma \vdash R \subseteq R'$ means that the set of dynamic regions represented by $R$ is included in the set of dynamic regions represented by $R$. As before, the judgment is reflexive and transitive. Otherwise, the relation is given by the nesting of RPLs. Here we say that the judgment holds if $R'$ ends in `*` and everything before the `*` is a prefix of $R$. These rules are sufficient for our purpose. In Chapter 4, we gave further rules that allow more expressivity with RPLs, but are not necessary to illustrate the ideas discussed in Chapter 7 and so are omitted here.

$\boxed{\Gamma \vdash R \subseteq R'}$

INCLUDE-REFLEXIVE $\dfrac{}{\Gamma \vdash R \subseteq R}$ INLCUDE-TRANSITIVE $\dfrac{\Gamma \vdash R \subseteq R' \quad \Gamma \vdash R' \subseteq R''}{\Gamma \vdash R \subseteq R''}$

$$\frac{\Gamma \vdash R \subseteq R\texttt{:*}}{\Gamma \vdash R\texttt{:}r \subseteq R\texttt{:*}}\ \textsc{Include-Recursive} \qquad \frac{}{\Gamma \vdash R \subseteq R\texttt{:*}}\ \textsc{Include-Prefix}$$

**Disjointness of regions:** The judgment $\Gamma \vdash R \# R'$ means that the sets of dynamic regions represented by $R$ and $R'$ have empty intersection. The judgment holds if $R$ and $R'$ are distinct names with `*` appended, or if $R$ and $R'$ are included in disjoint RPLs. Again these rules are sufficient for our purposes; further rules can be found in Chapter 4. Notice that the disjointness of $r$ and $r'$ (if $r \neq r'$) follows from DISJOINT-NAMES and INCLUDE-PREFIX. Notice also that in these simplified rules, we model only distinctions from the left (Section 3.2) and omit distinctions from the right. For example, these rules cannot distinguish $r : r_1$ from $r : r_2$. Again, that is to keep things as simple as possible and focus on what is important here.

$$\boxed{\Gamma \vdash R \# R'} \qquad \frac{r \neq r'}{\Gamma \vdash r\texttt{:*} \# r'\texttt{:*}}\ \textsc{Disjoint-Names} \qquad \frac{\Gamma \vdash R \subseteq R' \quad \Gamma \vdash R'' \subseteq R''' \quad \Gamma \vdash R' \# R'''}{\Gamma \vdash R \# R''}\ \textsc{Disjoint-Include}$$

### 8.2.4 Types

**Valid types:** The judgment $\Gamma \vdash T$ means that $T$ is valid in environment $\Gamma$. The judgment holds if $T$ is a valid instantiation of an interface, class, or type parameter; or it is `Null`.

$$\boxed{\Gamma \vdash T} \qquad \frac{\mathsf{defined}(I) \quad \Gamma \vdash T \quad \Gamma \vdash R \quad \Gamma \vdash E \quad \Gamma \vdash E \# \phi_{I\texttt{<}T,R,E\texttt{>}}(\mathcal{E}(I))}{\Gamma \vdash I\texttt{<}T, R, E\texttt{>}}\ \textsc{Type-Interface}$$

$$\frac{\mathsf{defined}(C) \quad \Gamma \vdash T \quad \Gamma \vdash R}{\Gamma \vdash C\texttt{<}T, R\texttt{>}}\ \textsc{Type-Class} \qquad \frac{\tau \in \Gamma \quad \Gamma \vdash R}{\Gamma \vdash \tau\texttt{<}R\texttt{>}}\ \textsc{Type-Param} \qquad \frac{}{\Gamma \vdash \texttt{Null}}\ \textsc{Type-Null}$$

$\mathsf{defined}(I)$ and $\mathsf{defined}(C)$ mean that a definition of interface $I$ or class $C$ appears in the program. Notice that rule TYPE-INTERFACE checks that the instantiating effect complies with the effect constraint specified in the interface. $\mathcal{E}(I)$ represents the effect $E$ appearing in the parameter constraint of interface $I$. The translation mapping $\phi_T$ is defined in Section 8.2.8.

**Subtypes:** The judgment $\Gamma \vdash T \preceq T'$ means that $T$ is a subtype of $T'$. The judgment is reflexive and

transitive:

$$\boxed{\Gamma \vdash T \preceq T'} \quad \textsc{Subtype-Reflexive}\ \frac{}{\Gamma \vdash T \preceq T} \qquad \textsc{Subtype-Transitive}\ \frac{\Gamma \vdash T \preceq T' \quad \Gamma \vdash T' \preceq T''}{\Gamma \vdash T \preceq T''}$$

Otherwise, the judgment holds if $T$ is a class type and $T'$ is the interface type that it implements; or $T$ is `Null` (and $T'$ is any type); or $T$ and $T'$ are related by inclusion.

$$\boxed{\Gamma \vdash T \preceq T'} \quad \frac{\textsc{Subtype-Interface-Class}\quad \texttt{class}\ C\texttt{<}\tau\texttt{>}\ \{\ \rho\ \}\rho' I\texttt{<}T, R, E\texttt{>} F^*\ M^* \in \mathcal{P}}{\Gamma \vdash C\texttt{<}T', R'\texttt{>} \preceq \phi_{C\texttt{<}T', R'\texttt{>}}(I\texttt{<}T, R, E\texttt{>})}$$

$$\textsc{Subtype-Null}\ \frac{}{\Gamma \vdash \texttt{Null} \preceq T} \qquad \textsc{Subtype-Include}\ \frac{\Gamma \vdash T \subseteq T'}{\Gamma \vdash T \preceq T'}$$

Note that the inclusion relation $\subseteq$ implies subtyping (rule Subtype-Include), but not vice versa.

**Inclusion of types:** The judgment $\Gamma \vdash T \subseteq T'$ means that $T$ and $T'$ are the same type, except for bindings to region and effect arguments, which are related by inclusion.

$$\boxed{\Gamma \vdash T \subseteq T'} \quad \textsc{Include-Reflexive}\ \frac{}{\Gamma \vdash T \subseteq T}$$

$$\textsc{Include-Transitive}\ \frac{\Gamma \vdash T \subseteq T' \quad \Gamma \vdash T' \subseteq T''}{\Gamma \vdash T \subseteq T''} \qquad \textsc{Include-Interface}\ \frac{\Gamma \vdash T \subseteq T' \quad \Gamma \vdash R \subseteq R' \quad \Gamma \vdash E \subseteq E'}{\Gamma \vdash I\texttt{<}T, R, E\texttt{>} \subseteq I\texttt{<}T', R', E'\texttt{>}}$$

$$\textsc{Include-Class}\ \frac{\Gamma \vdash T \subseteq T' \quad \Gamma \vdash R \subseteq R'}{\Gamma \vdash C\texttt{<}T, R\texttt{>} \subseteq C\texttt{<}T', R'\texttt{>}} \qquad \textsc{Include-Param}\ \frac{\Gamma \vdash R \subseteq R'}{\Gamma \vdash \tau\texttt{<}R\texttt{>} \subseteq \tau\texttt{<}R'\texttt{>}}$$

Note that it would not be sound to put $\Gamma \vdash T \preceq T'$ in the condition of Include-Interface or Include-Class, for the same reason that it is not sound to treat $C\texttt{<}C'\texttt{>}$ as a subtype of $C\texttt{<Object>}$ in ordinary Java [57]. It is sound, however, to make inclusion a condition of subtyping, because we capture regions and effects as discussed in Section 8.2.7.

### 8.2.5 Effects

**Valid effects:** The judgment $\Gamma \vdash E$ means that effect $E$ is valid in environment $\Gamma$. An effect is valid if it is the empty effect, a valid read or write effect, a valid method parameter, or a union of valid effects.

$\boxed{\Gamma \vdash E}$

$$\textsc{Effect-Empty}\ \frac{}{\Gamma \vdash \emptyset} \qquad \textsc{Effect-Reads}\ \frac{\Gamma \vdash R}{\Gamma \vdash \texttt{reads}\ R} \qquad \textsc{Effect-Writes}\ \frac{\Gamma \vdash R}{\Gamma \vdash \texttt{writes}\ R}$$

$$\textsc{Effect-Param}\ \frac{\eta \in \Gamma}{\Gamma \vdash \eta} \qquad \textsc{Effect-Union}\ \frac{\Gamma \vdash E \quad \Gamma \vdash E'}{\Gamma \vdash E \cup E'}$$

**Subeffects:** The judgment $\Gamma \vdash E \subseteq E'$ means that $E$ is a subeffect of $E'$. As in Chapter 4, the subeffect relation allows us to approximate an effect $E$ by a "larger" effect $E'$, while retaining soundness. The relation is based on three criteria: (1) literal inclusion of component effects; (2) covering of an effect on $R$ by the same effect on $R'$, if $R'$ includes $R$ (Section 8.2.3); and (3) covering of reads by writes. These criteria ensure that if $E$ is a subeffect of $E'$, and some other effect $E''$ is noninterfering with $E'$, then $E''$ is noninterfering with $E$ as well.

$\boxed{\Gamma \vdash E \subseteq E'}$

$$\textsc{SE-Reflexive}\ \frac{}{\Gamma \vdash E \subseteq E} \qquad \textsc{SE-Transitive}\ \frac{\Gamma \vdash E \subseteq E' \quad \Gamma \vdash E' \subseteq E''}{\Gamma \vdash E \subseteq E''}$$

$$\textsc{SE-Empty}\ \frac{}{\Gamma \vdash \emptyset \subseteq E} \qquad \textsc{SE-Reads}\ \frac{\Gamma \vdash R \subseteq R'}{\Gamma \vdash \texttt{reads}\ R \subseteq \texttt{reads}\ R'} \qquad \textsc{SE-Writes}\ \frac{\Gamma \vdash R \subseteq R'}{\Gamma \vdash \texttt{writes}\ R \subseteq \texttt{writes}\ R'}$$

$$\textsc{SE-Reads-Writes}\ \frac{\Gamma \vdash R \subseteq R'}{\Gamma \vdash \texttt{reads}\ R \subseteq \texttt{writes}\ R'} \qquad \textsc{SE-Union-1}\ \frac{\Gamma \vdash E \subseteq E'}{\Gamma \vdash E \subseteq E' \cup E''} \qquad \textsc{SE-Union-2}\ \frac{\Gamma \vdash E' \subseteq E \quad \Gamma \vdash E'' \subseteq E}{\Gamma \vdash E' \cup E'' \subseteq E}$$

**Noninterfering effects:** The judgment $\Gamma \vdash E \# E'$ means that effects $E$ and $E'$ are noninterfering. The noninterference relation is symmetric (obvious rule omitted). Noninterference is based on four criteria: reads are always noninterfering; disjoint writes are noninterfering; an effect parameter is noninterfering

with the effect in its noninterference constraint; and two effects are noninterfering if they are included in noninterfering effects. These criteria ensure that at runtime, any pair of effects covered by two noninterfering effects cannot perform a conflicting access to the same memory location.

$\boxed{\Gamma \vdash E \# E'}$

NI-EMPTY
$$\frac{}{\Gamma \vdash \emptyset \# E}$$

NI-READS
$$\frac{}{\Gamma \vdash \texttt{reads}\ R \# \texttt{reads}\ R'}$$

NI-WRITES
$$\frac{\Gamma \vdash R \# R'}{\Gamma \vdash \texttt{writes}\ R \# \texttt{writes}\ R'}$$

NI-PARAM
$$\frac{\eta \# E \in \Gamma}{\Gamma \vdash \eta \# E}$$

NI-INCLUDE
$$\frac{\Gamma \vdash E \# E' \quad \Gamma \vdash E'' \subseteq E \quad \Gamma \vdash E''' \subseteq E'}{\Gamma \vdash E'' \# E'''}$$

NI-UNION
$$\frac{\Gamma \vdash E \# E'' \quad \Gamma \vdash E' \# E''}{\Gamma \vdash E \cup E' \# E''}$$

### 8.2.6 Expressions

As in the Core DPJ (Chapter 4), every valid expression has a type and an effect. The judgment $\Gamma \vdash e : T, E$ means that expression $e$ is well typed with type $T$ and effect $E$ in environment $\Gamma$.

**Field access:** To type a field access expression $\texttt{this}.f$, we look in the environment to get the class $C$ bound to `this`, then we look in the definition of $C$ to get the type and region of $f$. $\mathcal{F}(C)(f)$ means the field with name $f$ declared in class $C$. The effect is a read of the region of $f$.

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

ACCESS
$$\frac{(\texttt{this}, C\texttt{<}\tau\texttt{<}\rho_\tau\texttt{>}, \rho\texttt{>}) \in \Gamma \quad \mathcal{F}(C)(f) = T\ f\ \texttt{in}\ R}{\Gamma \vdash \texttt{this}.f : T, \texttt{reads}\ R}$$

**Field assignment:** To type a field assignment $\texttt{this}.f\texttt{=}e$, we get the type and region of $f$ as discussed for field access. We also type $e$ and check that its type is a subtype of the type of $f$. The overall effect is the union of the effect $E$ of $e$ and the write to the region of $f$.

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

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

**Variable access:** To type a variable access $v$, we just get the type out of the environment. There is no effect,

because effects only track heap accesses.

$$\boxed{\Gamma \vdash e : T, E} \qquad \text{VARIABLE}\ \frac{(v, T) \in \Gamma}{\Gamma \vdash v : T, \emptyset}$$

**Method invocation:** To type a method invocation $e_1\texttt{.<}R, E\texttt{>}m\texttt{(}e_2\texttt{)}$, we do the following. (1) Compute the type and effect of $e_1$ and $e_2$. (2) Use the type of $e_1$ to find the signature of the method being invoked. $\mathcal{S}(T)(m)$ denotes the signature of the method named $m$ in the class or interface corresponding to $T$, which must be a class or interface type. (3) Check the method effect argument for compliance with the disjointness constraint in the signature, after translating the constraint by the type of $e_1$ (Section 8.2.8) and substituting for the method region and effect arguments. (4) Capture the type of $e_1$ (Section 8.2.7). (5) Check that the type of $e_2$ is a subtype of the formal parameter type in the signature, after translating the formal parameter type by the captured type of $e_1$ and substituting for the method region and effect arguments.

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

The type of the invocation expression is the return type in the signature, after translation. The effect is the union of the effects of $e_1$ and $e_2$, and the declared effect in the signature, after translation.

**Object creation:** To type an object creation expression $\texttt{new}\ C\texttt{<}T, R\texttt{>}$, we check that the type $C\texttt{<}T, R\texttt{>}$ is valid. There is no effect.

$$\boxed{\Gamma \vdash e : T, E} \qquad \text{NEW}\ \frac{\Gamma \vdash C\texttt{<}T, R\texttt{>}}{\Gamma \vdash \texttt{new}\ C\texttt{<}T, R\texttt{>} : C\texttt{<}T, R\texttt{>}, \emptyset}$$

**Null references:** A null reference is always valid with type `Null` and empty effect.

$$\boxed{\Gamma \vdash e : T, E} \qquad \text{NULL}\ \frac{}{\Gamma \vdash \texttt{null} : \texttt{Null}, \emptyset}$$

### 8.2.7 Capturing Types, Regions, and Effects

The capture of a type is an essential concept in standard Java [57] as well as in Core DPJ (Chapter 4) and this language. As discussed in Chapter 7, capturing types prevents errors such as the following:

```
class A<effect E> {
    B<E> f;
}
A<writes r> x = new A<pure>()
x.f = new B<writes r>(); // This should not be allowed!
```

The last assignment violates type preservation, because the dynamic type of `x.f` is `B<pure>`, and `writes r` is not a subtype of `pure`. So that last assignment should not be allowed. However, the assignment *would* be allowed if the static type of `x.f` were `B<writes r>`. And that is the type you would get by just substituting the effect argument `writes r` in the static type of `x` for $E$ in the definition of `f` in class `A`. So that naive method of computing the type of `x.f` is incorrect.

Instead, we first *capture* the type `A<writes r>` of `x` to generate the type `A<`$E$`>`, where $E$ is a fresh effect parameter (called a *capture parameter*). Then we substitute the region and effect arguments *of the captured type* in computing the type of `x.f`. So the type of `x.f` is `B<`$E$`>`, where $E$ is the capture parameter. The capture parameter represents the unknown effect argument in the dynamic type of the object reference stored in the variable `x`. As in Core DPJ (Chapter 4), all partially specified RPLs in type region arguments must be captured, because the true runtime region is unknown. Further, all nonempty effects in type effect arguments must be captured, because (except for empty effects), the precise effect is never known.

In the full language, the capture parameter carries an inclusion (for regions) or subeffect (for effects) bound. The bound is given by the type being captured. For example, the capture of `A<writes r>` may be legally assigned to `A<`$E$`>`, where $E$ is any supereffect of `writes r`. For simplicity, we omit the bounds in the formal language. The bounds are not needed for the examples discussed in Chapter 7.

**Capturing types:** The judgment $\Gamma \vdash \mathsf{capt}(T) = (T', \Gamma')$ means that capturing type $T$ in environment $\Gamma$

produces type $T'$ and environment $\Gamma'$. To capture a non-null type, we capture its component types, regions, and effects.

$$\boxed{\Gamma \vdash \mathsf{capt}(T) = (T', \Gamma')} \quad \textsc{Capture-Interface-Type}\ \frac{\Gamma \vdash \mathsf{capt}(T) = T', \Gamma' \quad \Gamma' \vdash \mathsf{capt}(R) = R', \Gamma'' \quad \Gamma'' \vdash \mathsf{capt}(E) = E', \Gamma'''}{\Gamma \vdash \mathsf{capt}(I\texttt{<}T, R, E\texttt{>}) = (I\texttt{<}T', R', E'\texttt{>}, \Gamma''')}$$

$$\textsc{Capture-Class-Type}\ \frac{\Gamma \vdash \mathsf{capt}(T) = (T', \Gamma') \quad \Gamma' \vdash \mathsf{capt}(R) = (R', \Gamma'')}{\Gamma \vdash \mathsf{capt}(C\texttt{<}T, R\texttt{>}) = (C\texttt{<}T', R'\texttt{>}, \Gamma'')} \qquad \textsc{Capture-Type-Param}\ \frac{\Gamma \vdash \mathsf{capt}(R) = (R', \Gamma')}{\Gamma \vdash \mathsf{capt}(\tau\texttt{<}R\texttt{>}) = (\tau\texttt{<}R'\texttt{>}, \Gamma')}$$

$$\textsc{Capture-Null}\ \frac{}{\Gamma \vdash \mathsf{capt}(\texttt{Null}) = (\texttt{Null}, \Gamma)}$$

**Capturing regions:** The judgment $\Gamma \vdash \mathsf{capt}(R) = (R', \Gamma')$ means that capturing region $R$ in environment $\Gamma$ produces region $R'$ and environment $\Gamma'$. If $R$ does not contain `*`, then the capture operation leaves the original $R$ and $\Gamma$ unchanged. Otherwise, $R$ is replaced with a fresh parameter $\rho$, and $\Gamma' = \Gamma \cup \rho$.

$$\boxed{\Gamma \vdash \mathsf{capt}(R) = (R', \Gamma')} \quad \textsc{Capture-Name}\ \frac{}{\Gamma \vdash \mathsf{capt}(r) = (r, \Gamma)} \qquad \textsc{Capture-Param}\ \frac{}{\Gamma \vdash \mathsf{capt}(\rho) = (\rho, \Gamma)}$$

$$\textsc{Capture-Recursive-Full}\ \frac{\Gamma \vdash \mathsf{capt}(R) = (R, \Gamma)}{\Gamma \vdash \mathsf{capt}(R\texttt{:}r) = (R\texttt{:}r, \Gamma)} \qquad \textsc{Capture-Recursive-Partial}\ \frac{\Gamma \vdash \mathsf{capt}(R) = (\rho, \Gamma \cup \rho) \quad \rho \notin \Gamma}{\Gamma \vdash \mathsf{capt}(R\texttt{:}r) = (\rho, \Gamma \cup \rho)}$$

$$\textsc{Capture-Star}\ \frac{\rho \notin \Gamma}{\Gamma \vdash \mathsf{capt}(R\texttt{:*}) = (\rho, \Gamma \cup \rho)}$$

**Capturing effects:** The judgment $\Gamma \vdash \mathsf{capt}(E) = (E', \Gamma')$ means that capturing effect $E$ in environment $\Gamma$ produces effect $E'$ and environment $\Gamma'$. For simplicity, we capture all effects with a fresh parameter, though we could avoid capturing empty effects.

$$\boxed{\Gamma \vdash \mathsf{capt}(E) = (E', \Gamma')} \quad \textsc{Capture-Effect}\ \frac{\eta \notin \Gamma}{\Gamma \vdash \mathsf{capt}(E) = (\eta, \Gamma \cup \eta)}$$

### 8.2.8 The Translation Mapping $\phi_T$

The mapping $\phi_T$ translates a type, region, or effect defined in an interface $I$ or class $C$ to its use as a member of type $T$ that instantiates $I$ or $C$ (the *instantiating type*). $T$ must be a class or interface type.

**Types:** To translate an interface or class type, we translate its arguments:

$$\begin{aligned} \phi_T(I\texttt{<}T', R, E\texttt{>}) &= I\texttt{<}\phi_T(T'), \phi_T(R), \phi_T(E)\texttt{>} \\ \phi_T(C\texttt{<}T', R\texttt{>}) &= C\texttt{<}\phi_T(T'), \phi_T(R)\texttt{>} \end{aligned}$$

To translate the parameter type $\tau\texttt{<}R\texttt{>}$, we use the instantiating type $T$, but we replace its region argument with the parameter's region argument $R$, after translating it:

$$\begin{aligned} \phi_{I\texttt{<}T,R,E\texttt{>}}(\tau\texttt{<}R'\texttt{>}) &= I\texttt{<}T, \phi_{I\texttt{<}T,R,E\texttt{>}}(R'), E\texttt{>} \\ \phi_{C\texttt{<}T,R\texttt{>}}(\tau\texttt{<}R'\texttt{>}) &= C\texttt{<}T, \phi_{I\texttt{<}T,R\texttt{>}}(R')\texttt{>} \end{aligned}$$

Finally, $\phi_T(\texttt{Null}) = \texttt{Null}$.

**Regions:** Let $\rho(T)$ and $\rho_\tau(T)$ be the region parameter and type region parameter of the interface or class that $T$ instantiates. $\phi_T$ is the identity on all regions except $\rho(T)$ and $\rho_\tau(T)$. For $\rho(T)$, we use the region argument of the instantiating type:

$$\begin{aligned} \phi_{I\texttt{<}T,R,E\texttt{>}}(\rho(I\texttt{<}T, R, E\texttt{>})) &= R \\ \phi_{C\texttt{<}T,R\texttt{>}}(\rho(C\texttt{<}T, R\texttt{>})) &= R \end{aligned}$$

For $\rho_\tau(T)$, we use the region argument of the type argument of the instantiating type:

$$\begin{aligned} \phi_{I\texttt{<}T,R,E\texttt{>}}(\rho_\tau(I\texttt{<}T, R, E\texttt{>})) &= R(T) \text{ if } T \neq \texttt{Null}, \text{ else } R \\ \phi_{C\texttt{<}T,R\texttt{>}}(\rho_\tau(C\texttt{<}T, R\texttt{>})) &= R(T) \text{ if } T \neq \texttt{Null}, \text{ else } R \end{aligned}$$

$R(T)$ is the region argument of the type. If the type argument is `Null`, then we treat $\rho_\tau$ as an alias for $\rho$. This is an appropriate solution for our simplified formal language, in which every class and interface has a parameter $\tau\texttt{<}\rho\texttt{>}$. As noted in 7, in the full language, we support classes and interfaces with no type region

parameter (or no type parameter at all), and we disallow bindings of types lacking a region argument to a type parameter with a region parameter.

**Effects:** The following rules define $\phi_T(E)$, where $\eta(I)$ is the declared effect parameter of interface $I$, and $\phi_T(E) = E$ in any case not defined below:

$$\phi_T(\emptyset) = \emptyset \qquad \phi_T(\texttt{reads}\ R) = \texttt{reads}\ \phi_T(R) \qquad \phi_T(\texttt{writes}\ R) = \texttt{writes}\ \phi_T(R)$$

$$\phi_{I\texttt{<}T,R,E\texttt{>}}(\eta(I)) = E \qquad \phi_T(E \cup E') = \phi_T(E) \cup \phi_T(E')$$

## 8.3 Dynamic Semantics

### 8.3.1 Execution Environment

We give a large-step semantics for program execution, using the following transition relation:

$$(e, \Sigma, H) \rightarrow (o, H', E).$$

$e$ is a program expression. The dynamic environment $\Sigma$ maps variables $v$ to object references $o$, region parameters $\rho$ to regions $R$, and effect parameters $\eta$ to effects $E$:

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

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

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

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

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

The effect $E$ collects the effect of the evaluation. A program evaluates to reference $o$ with heap $H$ and effect $E$ if its main expression is $e$ and

$$(e, \texttt{null}, \emptyset) \rightarrow (o, H, E)$$

according to the transition rules given in the next section.

### 8.3.2 Transition Rules

**Field access:** To evaluate `this.`$f$, we look in the environment to get the object reference $o$ bound to `this`; look in the heap to get the object $O$ and type $C\texttt{<}T, R\texttt{>}$ bound to $o$; look in the definition of $C$ to get the region of $f$; and read $O(f)$ out of the heap. We record the read effect on the region of $f$, after applying the dynamic translation function (Section 8.3.3).

$$\boxed{(e, \Sigma, H) \rightarrow (o, H', E)} \qquad \frac{(\texttt{this}, o) \in \Sigma \quad H(o) = (O, C\texttt{<}T, R\texttt{>}) \quad \mathcal{F}(C)(f) = T'\ f\ \texttt{in}\ R'}{(\texttt{this.}f, \Sigma, H) \rightarrow (O(f), H, \texttt{reads}\ \phi_{\Sigma,H}(R'))}\ \text{DYN-ACCESS}$$

**Field assignment:** To evaluate `this.f=`$e$, we evaluate $e$, yielding an object reference $o$ and an effect $E$. Then we look up the object and type of `this` as for field access, except that we write $o$ to $f$ instead of reading it. To represent the heap update, we write $g[a \mapsto b]$ (where $g$ is a function) to denote the function identical to $g$ everywhere on its domain, except that it maps $a$ to $b$. We record the write effect on the region of $f$, after applying the dynamic translation function.

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

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

**Method invocation:** To evaluate $e_1\texttt{.<}R, E\texttt{>}m\texttt{(}e_2\texttt{)}$, we evaluate $e_1$ and $e_2$. We look up the object corresponding to $e_1$ and look up the method $m$ for that object. Then we evaluate the method body in the environment formed by binding the value, region, and effect arguments as given in the invocation expres-

sion. We accumulate the effects from all the evaluations.

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

DYN-INVOKE

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

**Variable access:** To evaluate a variable access, we look the variable up in the environment. There is no effect.

$$\boxed{(e, \Sigma, H) \to (o, H', E)} \qquad \text{DYN-VARIABLE}\ \frac{(v, o) \in \Sigma}{(v, \Sigma, H) \to (o, H, \emptyset)}$$

**Object creation:** To evaluate an object creation expression $\texttt{new}\ C\texttt{<}T, R\texttt{>}$, we bind a fresh object reference to a fresh object in the heap, and give it the type $C\texttt{<}T, R\texttt{>}$, after applying the dynamic translation function. $\mathsf{new}(C)$ is the function taking each field of class $C$ to `null`.

$$\boxed{(e, \Sigma, H) \to (o, H', E)} \qquad \text{DYN-NEW}\ \frac{o \notin \mathsf{Dom}(H) \quad H' = H \cup o \mapsto (\mathsf{new}(C), \phi_{\Sigma,H}(C\texttt{<}T, R\texttt{>}))}{(\texttt{new}\ C\texttt{<}T, R\texttt{>}, \Sigma, H) \to (o, H', \emptyset)}$$

### 8.3.3 The Dynamic Translation Function $\phi_{\Sigma,H}$

The dynamic translation function $\phi_{\Sigma,H}$ translates a static type, region, or effect to a dynamic type, region, or effect using the current environment $\Sigma$ and heap $H$. First we substitute for region parameters using the bindings in $\Sigma$; then we substitute for effect parameters in $\Sigma$; then we apply the translation function $\phi_T$ from Section 8.2.8, where $T$ is the type in $H$ of the reference bound to `this` in $\Sigma$.

Formally, the definition of $\phi_{\Sigma,H}$ applied to regions is as follows:

1. If $\Sigma = (\rho, R') \cup \Sigma'$, then $\phi_{\Sigma,H}(R) = \phi_{\Sigma',H}(R[\rho \leftarrow R'])$.
2. Otherwise if $\Sigma = (\eta, E) \cup \Sigma'$, then $\phi_{\Sigma,H}(R) = \phi_{\Sigma',H}(R[\eta \leftarrow E])$.

3. Otherwise if $\Sigma = (\texttt{this}, o) \cup \Sigma'$, then $\phi_{\Sigma,H}(R) = \phi_T(R)$, where $H(o) = (O, T)$, and $\phi_T$ is the translation function defined in Section 8.2.8.

4. Otherwise $\phi_{\Sigma,H}(T) = T$.

$\phi_{\Sigma,H}$ applies to types and effects in the same way.

## 8.4 Soundness

We prove soundness as follows. In Section 8.4.1, we define valid static environments. In Section 8.4.2, we show that typing expressions in a valid static environment yields valid types and effects. In Section 8.4.3, we define valid execution state. In Section 8.4.4, we state and prove *type and effect preservation*, i.e., that the dynamic types and effects agree with their static approximations. In Section 8.4.5, we prove *preservation of noninterference*, i.e., that the static noninterference judgment implies noninterference at runtime.

### 8.4.1 Static Environments

A *static environment* $\Gamma$ consists of bindings $(v, T)$, type parameters $\tau$, region parameters $\rho$, effect parameters $\eta$, and effect constraints $\eta \# E$. A static environment is valid if its elements are valid with respect to itself:

$$\boxed{\vdash \Gamma} \quad \textsc{Env}\ \frac{\Gamma \vdash \Gamma}{\vdash \Gamma} \qquad \boxed{\Gamma \vdash \Gamma'} \quad \textsc{Env-Empty}\ \frac{}{\Gamma \vdash \emptyset} \qquad \textsc{Env-Var}\ \frac{\Gamma \vdash T}{\Gamma \vdash (v, T)} \qquad \textsc{Env-Type-Param}\ \frac{}{\Gamma \vdash \tau}$$

$$\textsc{Env-Rgn-Param}\ \frac{}{\Gamma \vdash \rho} \qquad \textsc{Env-Effect-Param}\ \frac{}{\Gamma \vdash \eta} \qquad \textsc{Env-Const}\ \frac{\Gamma \vdash E}{\Gamma \vdash \eta \# E} \qquad \textsc{Env-Union}\ \frac{\Gamma \vdash \Gamma' \quad \Gamma \vdash \Gamma''}{\Gamma \vdash \Gamma' \cup \Gamma''}$$

### 8.4.2 Validity of Static Typing

Our first soundness result is a claim about the static typing rules for expressions in Section 8.2.6. It says that if we type a valid expression in a valid static environment, we get a valid type and a valid effect. The hard part of the proof is to show that composing the substitutions $\sigma$ (for the method parameters) and $\phi_T$ (for the class parameters) in rule INVOKE produces valid types and effects when applied to the return type and

effect summary of the method signature, assuming a valid signature checked by rule SIGNATURE. Proving this result requires a fair amount of machinery in the form of supporting lemmas.

This section proceeds in four parts. In part 1, we show that $\sigma \circ \phi_T$ preserves validity, inclusion, and disjointness of regions. We also show that applying $\phi_T \circ \phi_{T'}$ to a region is equivalent to applying $\phi_{\phi_T(T')}$ to that region. In part two, we show the same results for effects. In part 3, we show that $\sigma \circ \phi_T$ preserves validity of types. In part 4, we use the results of parts 1–3 to prove the final result about valid static typing. The proof is easy, once the machinery in parts 1–3 is in place.

We use the following notation:

- $\Gamma_I$ denotes the environment we use to type interface $I$ in rule INTERFACE (Section 8.2.2). $\Gamma_C$ denotes the environment we use to type class $C$ in rule CLASS (Section 8.2.2). $\Gamma_T$ denotes $\Gamma_I$ (if $T$ is an interface type that instantiates interface $I$) or $\Gamma_C$ (if $T$ is a class type that instantiates class $C$).
- As in Section 8.2.8, $\rho(T)$ and $\rho_\tau(T)$ denote the region parameter and type region parameter of the class or interface instantiated by $T$. Similarly for $\tau(T)$ (type parameter) and $\eta(T)$ (effect parameter).
- As in Section 8.2.8, $R(T)$ denotes the region argument of $T$. Similarly for $T(T)$ (type argument) and $E(T)$ (effect argument).

**1. Translation of Regions**

**Valid regions:** We show that $\sigma \circ \phi_T$ takes valid regions to valid regions. For simplicity, we omit the effect parameters, arguments, and constraints, because they are irrelevant to judgments about regions.

**Lemma 8.4.1.** *If* $\Gamma_T \cup \rho \vdash R$ *and* $\sigma = [\rho \leftarrow R']$ *and* $\Gamma \vdash T$ *and* $\Gamma \vdash R'$*, then* $\Gamma \vdash \sigma(\phi_T(R))$.

*Proof.* Use induction on the length of $R$ (i.e., how many colon-separated elements $R$ contains, according to the syntax in Section 8.1). In the base case, there is nothing to show unless $R$ is $\rho(T)$ or $\rho_\tau(T)$ or $\rho$. In the first case, $\phi_T(R) = R(T)$, which is valid by $\Gamma \vdash T$ and rules TYPE-INTERFACE and TYPE-CLASS. In the second case, if $T(T)$ is `Null`, then the situation is identical to the first case. Otherwise $T(T)$ is an interface, class, or type parameter instantiated with region $R'$, and $\phi_T(R) = R'$, which is valid by $\Gamma \vdash T$ and rules TYPE-INTERFACE, TYPE-CLASS, and TYPE-PARAM. In the third case, the result holds because we are replacing $\rho$ with $R'$, which is valid in $\Gamma$. The inductive case follows immediately from the induction hypothesis, because the appended elements $r$ and `*` are unaffected by $\phi_T$. □

**Inclusion of regions:** We show that $\sigma \circ \phi_T$ preserves inclusion of regions. Again we ignore effect parameters.

**Lemma 8.4.2.** *Let* $\Gamma = \Gamma_T \cup \rho$ *and* $\sigma = [\rho \leftarrow R'']$. *If* $\Gamma \vdash R$ *and* $\Gamma \vdash R'$ *and* $\Gamma \vdash R \subseteq R'$ *and* $\Gamma' \vdash T$ *and* $\Gamma' \vdash R''$, *then* $\Gamma' \vdash \sigma(\phi_T(R)) \subseteq \sigma(\phi_T(R'))$.

*Proof.* It suffices to show that applying $\sigma \circ \phi_T$ to every term of a proof of $\Gamma \vdash R \subseteq R'$ yields a proof of $\Gamma' \vdash \sigma(\phi_T(R)) \subseteq \sigma(\phi_T(R'))$. To show this, use induction on the length of the proof of $\Gamma \vdash R \subseteq R'$. In the base case there is one rule application, either reflexivity or INCLUDE-PREFIX. In both cases the result holds because we are substituting the same thing for the same region parameters in both $R$ and $R'$. In the inductive case, if the last rule application is reflexivity or INCLUDE-PREFIX, the result holds by the argument just given. If the last rule application is transitivity or INCLUDE-RECURSIVE, then the result follows from the induction hypothesis. □

**Composition of $\phi_T$ and $\phi_{T'}$:** We show that $\phi_T \circ \phi_{T'}$ is equivalent to $\phi_{\phi_T(T')}$, when applied to regions.

**Lemma 8.4.3.** *Let* $T$ *and* $T'$ *be class or interface types. Then* $\phi_{\phi_T(T')}(R) = \phi_T(\phi_{T'}(R))$.

*Proof.* In the base case there is nothing to show unless $R$ is $\rho(T')$ or $\rho_\tau(T')$. In the first case, if $T'$ is the class type $C\texttt{<}T'', R'\texttt{>}$, then

$$\phi_{\phi_T(T')}(R) = \phi_{C\texttt{<}\phi_T(T''),\phi_T(R')\texttt{>}}(\rho(T')) = \phi_T(R'),$$

while

$$\phi_T(\phi_{T'}(R)) = \phi_T(R').$$

The argument if $T'$ is an interface type is almost identical. In the second case, if $T(T') = \texttt{Null}$, then we are in the same situation as the first case. Otherwise, $T(T') = I\texttt{<}T'', R', E'\texttt{>}$, or $T(T') = C\texttt{<}T'', R'\texttt{>}$, or $T(T') = \tau\texttt{<}R'\texttt{>}$. In the first two cases, $\phi_{\phi_T(T')}(R) = \phi_T(R')$, while $\phi_T(\phi_{T'}(R)) = \phi_T(R')$. In the third case, if $T$ instantiates class $C$ and $T'$ instantiates class $C'$, then

$$\phi_T(T') = C'\texttt{<}C\texttt{<}T'', \phi_T(R')\texttt{>}, R''\texttt{>},$$

and $\phi_{\phi_T(T')}(\rho_\tau(T')) = \phi_T(R')$. On the other hand $\phi_T(\phi_{T'}(\rho_\tau(T'))) = \phi_T(R')$. The argument if $T$ and/or

$T'$ is an interface type is almost identical. The inductive case is obvious. □

**Disjoint regions:** We show that $\sigma \circ \phi_T$ preserves disjointness of regions. Again we ignore effect parameters.

**Lemma 8.4.4.** *Let* $\Gamma = \Gamma_T \cup \rho$ *and* $\sigma = [\rho \leftarrow R'']$. *If* $\Gamma \vdash R$ *and* $\Gamma \vdash R'$ *and* $\Gamma \vdash R \# R'$ *and* $\Gamma' \vdash T$ *and* $\Gamma' \vdash R''$, *then* $\Gamma' \vdash \sigma(\phi_T(R)) \# \sigma(\phi_T(R'))$.

*Proof.* By induction on the length of $R$. The result holds in the base case (DISJOINT-NAMES) because parameter substitution has no effect on the application of that rule. The inductive case is obvious. □

**2. Translation of Effects**

**Valid effects:** We show that $\sigma \circ \phi_T$ takes valid effects to valid effects. We omit the effect constraints, as they are irrelevant to validity of effects.

**Lemma 8.4.5.** *If* $\Gamma_T \cup \rho \cup \eta \vdash E$ *and* $\sigma = [\rho \leftarrow R][\eta \leftarrow E']$ *and* $\Gamma \vdash T$ *and* $\Gamma \vdash R$ *and* $\Gamma \vdash E'$, *then* $\Gamma \vdash \sigma(\phi_T(E))$.

*Proof.* To compute $\phi_T(E)$ we substitute for region and effect parameters. Lemma 8.4.1 gives the result for the region parameters. For effect parameters, we are either substituting $E(T)$ for $\eta(T)$, or we are substituting $E'$ for $\eta$. In the first case, $E(T)$ is valid by $\Gamma \vdash T$ and TYPE-INTERFACE. In the second case, $E'$ is valid by hypothesis. □

**Subeffects:** We show that $\sigma \circ \phi_T$ preserves subeffects. Again we ignore the effect constraints.

**Lemma 8.4.6.** *Let* $\Gamma = \Gamma_T \cup \rho \cup \eta$ *and* $\sigma = [\rho \leftarrow R][\eta \leftarrow E'']$. *If* $\Gamma \vdash E$ *and* $\Gamma \vdash E'$ *and* $\Gamma \vdash E \subseteq E'$ *and* $\Gamma' \vdash T$ *and* $\Gamma' \vdash R$ *and* $\Gamma' \vdash E''$, *then* $\Gamma' \vdash \sigma(\phi_T(E)) \subseteq \sigma(\phi_T(E'))$.

*Proof.* Use the same technique as for the proof of Lemma 8.4.2. The base case is a proof using only SE-EMPTY, which obviously yields a correct proof under transformation by $\sigma \circ \phi_T$. In the inductive case, if the last rule application is SE-READS, SE-WRITES, or SE-READS-WRITES, then the result follows from Lemma 8.4.2. Otherwise, the result follows from the induction hypothesis. □

**Composition of $\phi_T$ and $\phi_{T'}$:** We show that $\phi_T \circ \phi_{T'}$ is equivalent to $\phi_{\phi_T(T')}$, when applied to effects.

**Lemma 8.4.7.** *Let* $T$ *and* $T'$ *be class or interface types. Then* $\phi_{\phi_T(T')}(E) = \phi_T(\phi_{T'}(E))$.

*Proof.* In view of Lemma 8.4.3, it suffices to show $\phi_{\phi_T(T')}(\eta(T')) = \phi_T(\phi_{T'}(\eta(T')))$. Pushing through the rules shows that on both sides we have $\phi_T(E(T'))$. □

**Noninterfering effects:** We show that $\sigma \circ \phi_T$ preserves noninterfering effects. Here we need the effect constraints to establish disjointness.

**Lemma 8.4.8.** *Let* $\Gamma = \Gamma_T \cup \rho \cup \eta \cup \eta \# E_\eta$ *and* $\sigma = [\rho \leftarrow R][\eta \leftarrow E'']$. *If* $\Gamma \vdash E$ *and* $\Gamma \vdash E'$ *and* $\Gamma \vdash E \# E'$ *and* $\Gamma' \vdash T$ *and* $\Gamma' \vdash R$ *and* $\Gamma' \vdash E''$, *and* $\Gamma' \vdash E'' \# \sigma(\phi_T(E_\eta))$, *then* $\Gamma' \vdash \sigma(\phi_T(E)) \# \sigma(\phi_T(E'))$.

*Proof.* We break the proof into two parts. In part 1, we show $\Gamma'' \vdash \phi_T(E) \# \phi_T(E')$, where $\Gamma'' = \Gamma' \cup \rho \cup \eta \cup \eta \# \phi_T(E_\eta)$. In part 2, we use part 1 to show the final result.

**Part 1:** In view of NI-UNION and NI-EMPTY, it suffices to assume that $E$ and $E'$ are each a single read effect, write effect, or effect parameter. If neither effect is a parameter, then either both are reads or the regions are disjoint, so the result follows from Lemma 8.4.4. Otherwise, we may assume without loss of generality that $E = \eta(T)$ or $E = \eta$. In either case, the only way to establish $\Gamma_T \vdash E \# E'$ is via NI-PARAM and NI-INCLUDE, using an effect constraint.

If $E = \eta(T)$, then we must have $\Gamma_T \vdash E' \subseteq E_\eta(T)$, where $E_\eta(T)$ denotes the effect in the parameter constraint of $T$ (also, $T$ is an interface type, since class types don't have effect parameters in this language). By Lemma 8.4.6 (with $R'' = \rho$ and $E' = \eta$), (a) $\Gamma'' \vdash \phi_T(E') \subseteq \phi_T(E_\eta(T))$. By assumption $\Gamma' \vdash T$, which implies $\Gamma'' \vdash T$. By that fact together with TYPE-INTERFACE, (b) $\Gamma'' \vdash \phi_T(\eta(T)) \# \phi_T(E_\eta(T))$. (a) and (b) together with NI-INCLUDE yield $\Gamma'' \vdash \phi_T(\eta(T)) \# \phi_T(E')$.

If $E = \eta$, then we must have $\Gamma_T \vdash E' \subseteq E_\eta$. By Lemma 8.4.6 applied to that fact, (a) $\Gamma'' \vdash \phi_T(E') \subseteq \phi_T(E_\eta)$. On the other hand, because $\eta$ is not a parameter of $T$, we have $\phi_T(\eta) = \eta$. Therefore the definition of $\Gamma''$ gives (b) $\Gamma'' \vdash \phi_T(\eta) \# \phi_T(E_\eta)$. Again by NI-INCLUDE, (a) and (b) yield $\Gamma'' \vdash \phi_T(\eta) \# \phi_T(E')$.

**Part 2:** In view of part 1 and Lemma 8.4.4, it suffices to show (renaming variables) that if $\Gamma = \Gamma' \cup \eta \cup \eta \# E_\eta$ and $\Gamma \vdash E$ and $\Gamma \vdash E'$ and $\Gamma \vdash E \# E'$ and $\Gamma' \vdash E''$ and $\Gamma' \vdash E'' \# \sigma(E_\eta)$, then $\Gamma' \vdash \sigma(E) \# \sigma(E')$, where $\sigma = [\eta \leftarrow E'']$. The result obviously holds unless $E$ or $E'$ contains $\eta$; so assume without loss of generality that $E = \eta$. Then we are trying to show $\Gamma' \vdash E'' \# \sigma(E')$. By NI-INCLUDE and the assumptions it suffices to show $\Gamma' \vdash \sigma(E') \subseteq \sigma(E_\eta)$. By the same argument as in the proof of Lemma 8.4.4, this is true if $\Gamma \vdash E' \subseteq E_\eta$. But as in the proof of part 1, that fact follows from the assumption $\Gamma \vdash \eta \# E'$. □

### 3. Translation of Types

We show that $\sigma \circ \phi_T$ takes valid types to valid types.

**Lemma 8.4.9.** *Let* $\Gamma = \Gamma_T \cup \rho \cup \eta \cup \eta \# E_\eta$ *and* $\sigma = [\rho \leftarrow R][\eta \leftarrow E]$. *If* $\Gamma \vdash T'$ *and* $\Gamma' \vdash T$ *and* $\Gamma' \vdash R$ *and* $\Gamma' \vdash E$ *and* $\Gamma' \vdash E \# \sigma(\phi_T(E_\eta))$, *then* $\Gamma' \vdash \sigma(\phi_T(T'))$.

*Proof.* Again we break the proof into two parts. In part 1, we show $\Gamma'' \vdash \phi_T(T')$, where $\Gamma'' = \Gamma' \cup \rho \cup \eta \cup \eta \# \phi_T(E_\eta)$. In part 2, we use part 1 to show the final result.

**Part 1:** Use induction on the number of applications of $\phi_T$ to a type. The base cases are $T' = \texttt{Null}$ and $T' = \tau(T)\texttt{<}R\texttt{>}$. The first case is obvious. In the second case, if $T$ is a class type $C\texttt{<}T'', R'\texttt{>}$, then $\phi_T(T') = C\texttt{<}T'', \phi_T(R)\texttt{>}$, and all the requirements of TYPE-CLASS are implied by $\Gamma' \vdash T$ except $\Gamma' \vdash \phi_T(R)$, which is given by Lemma 8.4.1. If $T$ is an interface type, the argument is nearly identical.

In the inductive case, if $T'$ is a class type $C\texttt{<}T'', R'\texttt{>}$, then $\phi_T(T') = C\texttt{<}\phi_T(T''), \phi_T(R')\texttt{>}$, and all the requirements of TYPE-CLASS are implied by the induction hypothesis and Lemma 8.4.1. If $T'$ is an interface type $I\texttt{<}T'', R', E'\texttt{>}$, then $\phi_T(T') = I\texttt{<}\phi_T(T''), \phi_T(R'), \phi_T(E')\texttt{>}$, and all the requirements of TYPE-INTERFACE are implied by the induction hypothesis and Lemmas 8.4.1 and 5, except

$$\Gamma'' \vdash \phi_T(E') \# \phi_{\phi_T(T')}(E_\eta(T')).$$

By Lemma 8.4.7, this is equivalent to

$$\Gamma'' \vdash \phi_T(E') \# \phi_T(\phi_{T'}(E_\eta(T'))).$$

By $\Gamma \vdash T'$ and TYPE-INTERFACE, $\Gamma \vdash E' \# \phi_{T'}(E_\eta(T'))$. The result then follows from Lemma 8.4.8.

**Part 2:** In view of part 1, it suffices to show (renaming variables) that if $\Gamma = \Gamma' \cup \rho \cup \eta \cup \eta \# E_\eta$ and $\sigma = [\rho \leftarrow R][\eta \leftarrow E]$ and $\Gamma \vdash T$ and $\Gamma' \vdash R$ and $\Gamma' \vdash E$ and $\Gamma' \vdash E \# \sigma(E_\eta)$, then $\Gamma' \vdash \sigma(T)$. Give $\sigma$ its obvious recursive definition for types, regions, and effects, and use induction on the number of applications of $\sigma$ to a type. The base cases are $T = \texttt{Null}$ and $T = \tau\texttt{<}R'\texttt{>}$. The first case is obvious, and the second one follows from the argument used to prove Lemma 8.4.1.

In the inductive case, if $T$ is a class type $C\texttt{<}T', R'\texttt{>}$, then $\sigma(T) = C\texttt{<}\sigma(T'), \sigma(R')\texttt{>}$, and all the requirements of TYPE-CLASS are implied by the induction hypothesis and the argument used to prove

Lemma 8.4.1. If $T$ is an interface type $I\texttt{<}T', R', E'\texttt{>}$, then $\sigma(T) = I\texttt{<}\sigma(T'), \sigma(R'), \sigma(E')\texttt{>}$, and all the requirements of TYPE-INTERFACE are implied by the induction hypothesis, the argument used to prove Lemma 8.4.1, and the argument used to prove Lemma 5, except

$$\Gamma' \vdash \sigma(E') \# \phi_{\sigma(T)}(E_\eta(T)).$$

By an argument similar to the proof of Lemma 8.4.7, this is equivalent to

$$\Gamma' \vdash \sigma(E') \# \sigma(\phi_T(E_\eta(T))).$$

By $\Gamma \vdash T$ and TYPE-INTERFACE, $\Gamma \vdash E' \# \phi_T(E_\eta(T))$. So the result is obvious unless $E'$ or $\phi_T(E_\eta(T))$ contains $\eta$. Assume without loss of generality (as before) that $E' = \eta$. Then we must have $\Gamma \vdash \phi_T(E_\eta(T)) \subseteq E_\eta$. By hypothesis, $\Gamma' \vdash \sigma(E') \# \sigma(E_\eta)$. So the result follows from DISJOINT-INCLUDE if

$$\Gamma' \vdash \sigma(\phi_T(E_\eta(T))) \subseteq \sigma(E_\eta).$$

But this is true by the argument given in the proof of Lemma 8.4.6. □

**4. Validity of Expression Typing**

**Theorem 8.4.10** (Validity of static expression typing)**.** *If* $\vdash \mathcal{P}$ *and* $\vdash \Gamma$ *and* $\Gamma \vdash e : T, E$*, then* $\Gamma \vdash T$ *and* $\Gamma \vdash E$.

*Proof.* By induction on the structure of $e$.

**Base cases:** The base cases are rules ACCESS, VARIABLE, and NEW. As to ACCESS, in order for the rule to apply, the expression $e$ being typed must be in the body of a method of some class $C$. Rule FIELD guarantees that $\Gamma_C \vdash T$ and $\Gamma_C \vdash R$. Further, by METHOD, we have $\Gamma_C \subseteq \Gamma$, so $\Gamma \vdash T$ and $\Gamma \vdash R$. For VARIABLE, the result follows from the definition of $\vdash \Gamma$. For NEW, the type is checked in applying the rule, and the effect is empty.

**Inductive cases:** The inductive cases are ASSIGN and INVOKE. For ASSIGN, the result follows from the induction hypothesis applied to the subexpression, together with the same argument used for ACCESS for the type and region of the field $f$. For INVOKE, the result follows from the induction hypothesis applied to $E_1$ and $E_2$, Lemma 8.4.9 applied to $\sigma(\phi_{T_1}(T_3))$, and Lemma 8.4.5 applied to $\sigma(\phi_{T_1}(E_4))$. □

### 8.4.3 Execution State

**Heaps:** To describe valid heaps, we need some rules for typing references:

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

Now we can describe the typing of heaps. A heap is valid if its elements are valid:

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

An object-type pair $(O, T)$ is valid if (1) $T$ is a valid type in the empty environment; and (2) for every field $f$ in $\mathcal{F}(C)$, $O(f)$ is defined, and its type is a subtype of the static type of $f$, after translation via $\phi_T$:

$\boxed{H \vdash (O, T)}$

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

Notice that at runtime, we check types in the empty environment $\emptyset$, because all parameters have been substituted away.

**Dynamic environments:** A dynamic environment $\Sigma$ is valid if its elements are valid with respect to a heap $H$:

$$\boxed{H \vdash \Sigma} \qquad \textsc{Dyn-Env-Empty}\ \frac{}{H \vdash \emptyset} \qquad \textsc{Dyn-Env-Var}\ \frac{H \vdash o : T}{H \vdash (v, o)} \qquad \textsc{Dyn-Env-Rgn-Param}\ \frac{\emptyset \vdash R}{H \vdash (\rho, R)}$$

$$\textsc{Dyn-Env-Effect-Param}\ \frac{\emptyset \vdash E}{H \vdash (\eta, E)} \qquad \textsc{Dyn-Env-Union}\ \frac{H \vdash \Sigma \quad H \vdash \Sigma'}{H \vdash \Sigma \cup \Sigma'}$$

**Instantiation of environments:** The judgment $H \vdash \Sigma \preceq \Gamma$ says that $\Sigma$ *instantiates* a static environment $\Gamma$. That means the variables and parameters appearing in both environments match; the types of the variable bindings in both environments match; and the effect bindings in $\Sigma$ obey the disjointness constraints specified by $\Gamma$. Instantiation allows us to use the static typing of expressions to infer that the dynamic execution of those expressions is well-behaved.

The basic rule for instantiation just records the original dynamic environment $\Sigma$ to the left of the $\vdash$. This makes the original dynamic environment is available as we dissect the environment to compare it to the static environment element by element:

$$\boxed{H \vdash \Sigma \preceq \Gamma} \quad \frac{\Sigma, H \vdash \Sigma \preceq \Gamma}{H \vdash \Sigma \preceq \Gamma} \text{ INSTANTIATE}$$

Next we have the element-by-element rules. First we give the usual rules for empty environments and unions; these just say formally that we compare the two environments element by element:

$$\boxed{\Sigma, H \vdash \Sigma' \preceq \Gamma} \quad \frac{}{\Sigma, H \vdash \emptyset \preceq \emptyset} \text{ INST-EMPTY} \qquad \frac{\Sigma, H \vdash \Sigma' \preceq \Gamma \quad \Sigma, H \vdash \Sigma'' \preceq \Gamma'}{\Sigma, H \vdash \Sigma' \cup \Sigma'' \preceq \Gamma \cup \Gamma'} \text{ INST-UNION}$$

The rule for variables says that the dynamic type of the reference $o$ bound to $v$ in $\Sigma$ has to match the static type $T$ of $v$ in $\Gamma$:

$$\boxed{H \vdash \Sigma \preceq \Gamma} \quad \frac{H \vdash o : T \quad \emptyset \vdash T \preceq \phi_{\Sigma,H}(T')}{\Sigma, H \vdash (v, o) \preceq (v, T')} \text{ INST-VAR}$$

For region parameters, we need three rules. The first handles method region parameters, whose bindings appear explicitly in $\Sigma$. The second and third handle the class region parameters, whose bindings are given implicitly by the type of the reference bound to `this` in $\Sigma$:

$$\boxed{H \vdash \Sigma \preceq \Gamma} \quad \frac{}{\Sigma, H \vdash (\rho, R) \preceq \rho} \text{ INST-METHOD-RGN-PARAM}$$

$$\frac{(\texttt{this}, o) \in \Sigma \quad H \vdash o : T}{\Sigma, H \vdash \emptyset \preceq \rho(T)}\ \text{INST-CLASS-RGN-PARAM} \qquad \frac{(\texttt{this}, o) \in \Sigma \quad H \vdash o : T}{\Sigma, H \vdash \emptyset \preceq \rho_\tau(T)}\ \text{INST-CLASS-TYPE-RGN-PARAM}$$

For effect parameters, we just need to handle method effect parameters, because there are no class effect parameters. We must ensure that the effect parameter has a binding, and that the effect constraints are satisfied:

$$\boxed{H \vdash \Sigma \preceq \Gamma} \qquad \frac{}{\Sigma, H \vdash (\eta, E) \preceq \eta}\ \text{INST-EFFECT-PARAM} \qquad \frac{\emptyset \vdash \phi_{\Sigma,H}(\eta) \# \phi_{\Sigma,H}(E)}{\Sigma, H \vdash \emptyset \preceq \eta \# E}\ \text{INST-CONSTRAINT}$$

The rule for type parameters is simple, since these don't appear in $\Sigma$:

$$\boxed{H \vdash \Sigma \preceq \Gamma} \qquad \frac{}{\Sigma, H \vdash \emptyset \preceq \tau}\ \text{INST-TYPE-PARAM}$$

**Execution state:** The judgment $\Gamma \vdash (e, \Sigma, H) : T, E$ means that execution state $(e, \Sigma, H)$ is valid with respect to static environment $\Gamma$ (the environment in which $e$ was typed in the static semantics) with type $T$ and effect $E$. That means $\Gamma$, $\Sigma$, and $H$ are valid; $\Sigma$ instantiates $\Gamma$; and $e$ is well typed in $\Gamma$ with type $T$ and effect $E$.

$$\boxed{\Gamma \vdash (e, \Sigma, H) : T, E} \qquad \frac{\vdash \Gamma \quad \vdash H \quad H \vdash \Sigma \quad H \vdash \Sigma \preceq \Gamma \quad \Gamma \vdash e : T, E}{\Gamma \vdash (e, \Sigma, H) : T, E}\ \text{STATE}$$

### 8.4.4 Preservation of Type and Effect

The second soundness result states that the static types and effects computed according to Section 8.2.6 approximate the dynamic types and effects produced by execution according to Section 8.3.2. More precisely, if we evaluate $e$ to $o$ starting in a valid execution state, then the resulting heap is valid; $o$ is well typed, and its type is a subtype of the static type of $e$; and the resulting effect is valid and a subeffect of the static effect of $e$. In the rest of this section, assume $H$ is a valid heap, $\Sigma$ is a valid dynamic environment, $\Gamma$ is a valid static environment, and $\Sigma$ instantiates $\Gamma$. In symbols, that is $\vdash H$, $H \vdash \Sigma$, $\vdash \Gamma$, and $H \vdash \Sigma \preceq \Gamma$.

**Lemma 8.4.11.** *If $\Gamma \vdash R$, then $\emptyset \vdash \phi_{\Sigma,H}(R)$. The same result holds replacing $R$ with $E$ or $T$.*

*Proof.* **Regions:** By induction on the number of applications of $\phi_{\Sigma,H}$. The first base case is (4) in the definition of $\phi_{\Sigma,H}$ (Section 8.3.3). In this case, $\phi_{\Sigma,H}(R) = R$, so we must show $\emptyset \vdash R$. By $H \vdash \Sigma \preceq \Gamma$, there cannot be any parameters in scope in $\Gamma$, since there are none in $\Sigma$. So by $\Gamma \vdash R$, $R$ is not a parameter, and $\emptyset \vdash R$. The second base case is (3) in the definition of $\phi_{\Sigma,H}$. In that case, by $H \vdash \Sigma \preceq \Gamma$, $\Gamma = \Gamma_T \cup \Gamma'$, and $\Gamma_T \vdash R$. Since $\emptyset \vdash T$ by OBJECT, the result follows from Lemma 8.4.1. The first inductive case is (1) in the definition of $\phi_{\Sigma,H}$, i.e., $(\rho, R') \in \Sigma$. In that case, by $H \vdash \Sigma \preceq \Gamma$, $\Gamma = \rho \cup \Gamma'$. Further, $\Gamma' \vdash R[\rho \leftarrow R']$, because the substitution eliminates $\rho$. The result then follows from the induction hypothesis. The second inductive case is (2) in the definition of $\phi_{\Sigma,H}$. But this case obviously holds, as regions have no effect parameters.

**Types and effects:** The identical argument goes through using Lemma 8.4.5 for effects and Lemma 8.4.9 for types, except that we treat cases (1) and (2) together and use them to establish the preconditions of the lemmas. □

**Lemma 8.4.12.** *If $\Gamma \vdash R$ and $\Sigma \vdash R'$ and $\Gamma \vdash R \subseteq R'$, then $\emptyset \vdash \phi_{\Sigma,H}(R) \subseteq \phi_{\Sigma,H}(R')$. The same result holds replacing $R$ and $R'$ with $E$ and $E'$.*

*Proof.* Same proof as for Lemma 8.4.11, except that the first base case is obvious from the definition of $\phi_{\Sigma,H}$, and the argument for the second base case uses Lemmas 8.4.2 and 8.4.6. □

**Lemma 8.4.13.** *If $\Gamma \vdash R$ and $\Gamma \vdash R'$ and $\Gamma \vdash R \# R'$, then $\emptyset \vdash \phi_{\Sigma,H}(R) \# \phi_{\Sigma,H}(R')$. The same result holds replacing $R$ with $R'$ and $E$ with $E'$.*

*Proof.* Same proof as for Lemma 8.4.12, using Lemmas 8.4.4 and 8.4.8 instead of Lemmas 8.4.2 and 8.4.6.

□

**Lemma 8.4.14.** *If $\Gamma \vdash T \preceq T'$, then $\Sigma \vdash \phi_{\Sigma,H}(T) \preceq \phi_{\Sigma,H}(T')$.*

*Proof.* Consider each of the three possibilities for the last rule applied in the proof of $\Gamma \vdash T \preceq T'$. In the case of SUBTYPE-NULL, the result is obvious. In the case of SUBTYPE-INTERFACE-CLASS, it suffices to show

$$\phi_{\Sigma,H}(\phi_{C\texttt{<}T',R'\texttt{>}}(I\texttt{<}T,R,E\texttt{>})) = \phi_{\phi_{\Sigma,H}(C\texttt{<}T',R'\texttt{>})}(I\texttt{<}T,R,E\texttt{>}).$$

But we can do this easily with an argument similar to the one used to prove Lemmas 8.4.3 and 8.4.7. In the case of SUBTYPE-INCLUDE, the problem is reduced to proving that $\Gamma \vdash T \subseteq T'$ implies $\Sigma \vdash \phi_{\Sigma,H}(T) \subseteq \phi_{\Sigma,H}(T')$.

To prove the last fact, use induction on the height of the proof that $\Sigma \vdash T \subseteq T'$. The base case is reflexivity. Otherwise, the result is given by the induction hypothesis, together with Lemmas 8.4.2 and 8.4.6.

□

**Lemma 8.4.15.** *If $\emptyset \vdash T_1$ and $\emptyset \vdash T_2$ and $\emptyset \vdash T_1 \subseteq T_2$, then $\emptyset \vdash \phi_{T_1}(R) \subseteq \phi_{T_2}(R)$. The same result holds replacing $R$ with $E$ or $T$.*

*Proof.* Via a straightforward induction, using the fact that all the preconditions in all the rules for $\Sigma \vdash T \subseteq T'$ (Section 8.2.4) are written in terms of $\subseteq$. □

**Theorem 8.4.16** (Preservation of type and effect)**.** *If $\vdash \mathcal{P}$ and $\Gamma \vdash (e, \Sigma, H) : T_s, E_s$ and $(e, \Sigma, H) \rightarrow (o, H', E)$, then (a) $\vdash H'$; (b) $H' \vdash o : T$; (c) $\emptyset \vdash T \preceq \phi_{\Sigma,H'}(T_s)$; (d) $\emptyset \vdash E$; and (e) $\emptyset \vdash E \subseteq \phi_{\Sigma,H'}(E_s)$.*

*Proof.* By induction on the structure of $e$.

**Base cases:** DYN-ACCESS: (a) holds because the heap is unchanged. (b) and (c) hold by $\vdash H$, OBJECT, and the definition of $H \vdash o : T$. (d) holds by FIELD and Lemma 10. (e) follows directly from the definitions of ACCESS and DYN-ACCESS.

DYN-VARIABLE: (a), (d), and (e) are trivial. (b) holds by $H \vdash \Sigma$. (c) holds by comparing VARIABLE with DYN-VARIABLE, and by $H \vdash \Sigma \preceq \Gamma$.

DYN-NEW: For (b), it suffices to show $H \vdash \phi_{\Sigma,H}(C\texttt{<}T, R\texttt{>})$ in DYN-NEW. But this follows from NEW and Lemma 8.4.11. (a) follows from (b) and OBJECT. (c) is obvious from DYN-NEW. (d) and (e) are trivial.

**Inductive cases:** DYN-ASSIGN: The induction hypothesis applied to the subexpression $e$ gives $\vdash H'$, so to establish (a) it suffices to show that the type $T_o$ of $o$ is legal to assign to $f$, according to rule OBJECT. Let $T_e$ be the static type of $e$ according to rule ASSIGN, and $T_f$ be the type of field $f$. By Theorem 8.4.10, $\Gamma \vdash T_e$, and by ASSIGN, $\Gamma \vdash T_e \preceq T_f$. Lemma 8.4.14 yields $\Sigma \vdash \phi_{\Sigma,H}(T_e) \preceq \phi_{\Sigma,H}(T_f)$, and the induction hypothesis gives $\Sigma \vdash T_o \preceq \phi_{\Sigma,H}(T_e)$. Together with the transitivity of subtyping, this establishes the result. (b) and (c) are given directly by the induction hypothesis. (d) and (e) hold for the same reasons given for DYN-ACCESS.

DYN-INVOKE: The induction hypothesis yields $H_3 \vdash \Sigma'$ and $\vdash H_4$, and it is obvious that $H_4 \vdash \Sigma'$. Let $\Gamma^m$ be the environment in which we typed the body of method $m$ of class $C$ using METHOD. We break the proof into two parts. First, we show $H_4 \vdash \Sigma' \preceq \Gamma^m$. Second, we use that fact to show the final result.

*$H_4 \vdash \Sigma' \preceq \Gamma^m$:* Let $\Gamma^m = \Gamma_C \cup \rho \cup \eta \cup \eta \# E \cup (x, T_x)$. By the definition of instantiation of a static environment (Section 8.4.3), it suffices to show (a) $\emptyset \vdash T_{o_2} \preceq \phi_{\Sigma',H_4}(T_x)$ and (b) $\emptyset \vdash \phi_{\Sigma',H_4}(\eta) \# \phi_{\Sigma',H_4}(E)$.

(a) First assume that the static type $T_{e_1}$ of $e_1$ is a class type. By INVOKE, $\Gamma \cup \Gamma^c \vdash T_{e_2} \preceq \sigma(\phi_{T^c_{e_1}}(T_x))$, where $T_{e_2}$ is the static type of $e_2$, $T^c_{e_1}$ is the capture of $T_{e_1}$, and $\Gamma^c$ represents the extra parameters added by the capture operation. Use induction on the number of parameters appearing in $\Gamma^c$. In the base case ($\Gamma^c$ is empty, i.e., no capture parameters), we have $\Gamma \vdash T_{e_2} \preceq \sigma(\phi_{T_{e_1}}(T_x))$. By Lemma 8.4.14, $H_4 \vdash \phi_{\Sigma,H_4}(T_{e_2}) \preceq \phi_{\Sigma,H_4}(\sigma(\phi_{T_{e_1}}(T_x)))$. The induction hypothesis yields $H_4 \vdash T_{o_2} \preceq \phi_{\Sigma,H_4}(T_{e_2})$, so by transitivity of subtyping, we have $H_4 \vdash T_{o_2} \preceq \phi_{\Sigma,H_4}(\sigma(\phi_{T_{e_1}}(T_x)))$. By an argument similar to the proof of Lemmas 8.4.3 and 8.4.9, we can show that the right-hand side is equal to $\phi_{\phi_{\Sigma,H_4}(\sigma(T_{e_1}))}(T_x)$, which is the same as $\phi_{\Sigma',H_4}(T_x)$. In the inductive case, suppose we have added a single capture parameter. Construct the environment $\Sigma \cup (\rho, R)$ or $\Sigma \cup (\eta, E)$ by adding that parameter to $\Sigma$, with its actual binding. This operation preserves instantiation of environments, so the same argument goes through using that environment instead of $\Sigma$.

Now assume that $T_{e_1}$ is an interface type. Then INVOKE gives $\Gamma \cup \Gamma^c \vdash T_{e_2} \preceq \sigma(\phi_{T^c_{e_1}}(T'_x))$, where $T'_x$ is the formal parameter type in the interface signature implemented by the method $m$. By the same argument as for the class type case, we obtain $H_4 \vdash T_{o_2} \preceq \phi_{\Sigma,H_4}(\sigma(\phi_{T_{e_1}}(T'_x)))$. Now factor $\phi_{T_{e_1}}$ into $\phi_2 \circ \phi_1$, where $\phi_1$ is the translation from the interface definition to the implementing class definition (written $\sigma \circ \phi_T$ in rule IMPLEMENT), and $\phi_2$ is the translation from the class definition to the class type. Then we have $H_4 \vdash T_{o_2} \preceq \phi_{\Sigma,H_4}(\sigma(\phi_2(\phi_1(T'_x))))$. By IMPLEMENT, $\Gamma^m \vdash \phi_1(T'_x) \preceq T_x$, and it is straightforward to show that $H_4 \vdash \phi_{\Sigma,H_4}(\sigma(\phi_2(\phi_1(T'_x)))) \preceq \phi_{\Sigma,H_4}(\sigma(\phi_2(T_x)))$. By transitivity of subtyping, this gives $H_4 \vdash \phi_{\Sigma,H_4}(\sigma(\phi_2(T_x)))$. The rest of the proof of the class type case then goes through.

(b) By INVOKE, $\Gamma \vdash E_5 \# \sigma(\phi_{T_1}(E_3))$. By Lemma 8.4.13, $\emptyset \vdash \phi_{\Sigma,H_4}(E_5) \# \phi_{\Sigma,H_4}(\sigma(\phi_{T_1}(E_3)))$. By DYN-INVOKE, the left-hand side equals $\phi_{\Sigma',H_4}(\eta)$. By an argument similar to (a), the right-hand side equals $\phi_{\Sigma',H}(E)$.

*Final result:* Now that we have established $H_4 \vdash \Sigma' \preceq \Gamma^m$, (a), (b), and (d) follow directly from the induction hypothesis applied to the execution of $e_3$ in DYN-INVOKE. As to (c), the induction hypothesis

gives $\emptyset \vdash T_{o_1} \preceq \phi_{\Sigma,H_4}(T_{e_1})$ and $\emptyset \vdash T_{o_3} \preceq \phi_{\Sigma',H_4}(T_r)$, where $T_r$ is the return type of $m$. Factor $\phi_{\Sigma',H_4}$ into $\phi_{T_{o_1}} \circ \sigma$ where $\sigma$ substitutes for the method parameters. If $T_{o_1}$ and $T_{e_1}$ are both instantiations of the same class, then by Lemma 14, we have $\Gamma \vdash T_{o_3} \preceq \phi_{\phi_{\Gamma,H_4}(T_{e_1})}(\sigma(T_3))$. An argument similar to the proof of Lemma 8.4.7 then gives $\Gamma \vdash T_{o_3} \preceq \phi_{\Gamma,H_4}(\phi_{T_{e_1}}(\sigma(T_3)))$, which establishes the result. If $T_{o_1}$ and $T_{e_1}$ are not both instantiations of the same class, then $T_{o_1}$ must be a class type, $\phi_{\Gamma,H_4}(T_{e_1})$ must be the interface type it implements, and the same result goes through via the definition of rule SUBTYPE-INTERFACE-CLASS.

As to (e), by SE-UNION-2, it suffices to show the result for each of the three effects that form the union. The first two effects are given directly by the induction hypothesis. For the third effect, we have to show $\emptyset \vdash E \subseteq \phi_{\Sigma,H}(\sigma(\phi_{T_{e_1}}(E_4)))$, where $E$ is the actual effect of executing the method. The argument is the same as for (c). □

### 8.4.5 Soundness of Noninterference

The third soundness result states that the static noninterference judgment for expressions is sound: if two expressions have statically noninterfering effects, then the execution of the two expressions is noninterfering at runtime. Again, we assume all environments and heaps are valid throughout.

First we define $\mathcal{R}_f(o, H)$, the region of field $f$ of object $o \in \mathsf{Dom}(H)$. This definition formalizes the idea that regions $R$ in the field declarations $T\ f\ \texttt{in}\ R$ partition the heap:

**Definition 8.4.17** (Region of a field)**.** *If $H \vdash o : T$ and $\mathcal{F}(T)(f) = T'\ f\ \texttt{in}\ R$, then $\mathcal{R}_f(o, H) = \phi_T(R)$.*

Next we prove a property of the dynamic effects produced by program execution: for a well-typed program, if we evaluate $e$ and $e'$ in sequence, and if the two evaluations have noninterfering effects, then the individual read and write effects of $e$ and $e'$ can be arbitrarily interleaved, with identical results.

**Lemma 8.4.18.** *If $o \in H$ and $H \subseteq H'$, then $\mathcal{R}_f(o, H) = \mathcal{R}_f(o, H')$.*

*Proof.* It suffices to show that $\mathcal{R}_f(o, H)$ is unique and does not change during program execution. But this is true, because $\mathcal{R}_f(o, H)$ is uniquely determined by (1) the type $T$ given to $o \mapsto (O, T)$ when the object is added to the heap via DYN-NEW and (2) the declaration $\mathcal{F}(T)(f) = T'\ f\ \texttt{in}\ R$, and neither of these changes during program execution. □

**Lemma 8.4.19.** *If $e'$ is a subexpression of $e$, and $(e', \Sigma', H'') \to (o', E', H''')$ appears in the proof tree for $(e, \Sigma, H) \to (o, E, H')$, then $\emptyset \vdash E' \subseteq E$.*

*Proof.* Clear by the structure of the rules in Section 8.3.2, since the dynamic effects of every expression include the union of effects of the subexpressions. □

**Proposition 8.4.20.** *If* $(e, \Sigma, H) \to (o, H', E)$ *and* $(e', \Sigma, H') \to (o', H'', E')$ *and* $\emptyset \vdash E \# E'$*, then then there are no conflicting accesses to the same object field in the evaluations of* $e$ *and* $e'$*.*

*Proof.* First, "conflicting accesses to the same object field" is well-defined, because objects $o \mapsto (O, T)$ are added via DYN-NEW and never subtracted, so $H \subseteq H' \subseteq H''$, and the domain of $H(o)$ never changes. So all accesses occur to object fields of $H''$. Now suppose there is a conflicting access. Accesses happen via DYN-ACCESS and DYN-ASSIGN, and each of those rules records the effect on $\mathcal{R}_f(o, H)$, for an access to field $f$ of object $o$. So by Lemmas 8.4.18 and 8.4.19, there must be two conflicting accesses to the same region $R$, one contained in $E$ and the other contained in $E'$. But by the rules in Section 2.5, this means that $\emptyset \vdash E \# E'$ does not hold. □

Finally, by extending this result to static effects, we obtain the main soundness property of the core language.

**Theorem 8.4.21** (Noninterference)**.** *If* $\vdash \mathcal{P}$ *and* $\Gamma \vdash (e, \Sigma, H) : T_s, E_s$ *and* $\Gamma \vdash (e', \Sigma, H') : T'_s, E'_s$ *and* $\Gamma \vdash E_s \# E'_s$ *and* $(e, \Sigma, H) \to (o, H', E)$ *and* $(e', \Sigma, H') \to (o', H'', E')$*, then there are no conflicting accesses to the same object field in the evaluations of* $e$ *and* $e'$*.*

*Proof.* Theorem 8.4.16 gives $\emptyset \vdash E \subseteq \phi_{\Sigma,H'}(E_s)$ and $\emptyset \vdash E' \subseteq \phi_{\Sigma,H''}(E'_s)$. It is easy to see the first statement implies $\emptyset \vdash E \subseteq \phi_{\Sigma,H''}(E_s)$. Lemma 8.4.13 gives $\phi_{\Sigma,H''}(E_s) \# \phi_{\Sigma,H''}(E'_s)$. NI-INCLUDE then gives $\emptyset \vdash E \# E'$, and Proposition 8.4.20 gives the result. □

# Chapter 9

# Conclusion

This thesis has presented Deterministic Parallel Java, a new *deterministic by default* language that uses a novel type and effect system to (1) enforce determinism at compile time with no runtime checking overhead; (2) provide strong compile-time safety guarantees and performance optimizations for nondeterministic code supported by weakly isolated transactional memory; and (3) check that the uses of object-oriented parallel frameworks conform to their effect specifications. We have presented the new language and effect system features both informally and formally, and we have proved soundness for the formally described features. We have also described evaluations showing that the new features are useful and effective.

As discussed in Chapter 2, several open questions remain after this thesis, and should be a fruitful source of continuing research:

1. Inferring region and effect information can reduce the programmer burden of an effect system like DPJ's. Work on this problem is ongoing by Vakilian and others, and has already produced an algorithm for inferring method effect summaries [122].

2. Supplementing the DPJ effect system with runtime checks for properties such as disjointness of reference can make the language more expressive and/or reduce the programmer annotation burden, at the cost of weakening the compile-time guarantees and/or adding runtime overhead. Exploring the tradeoffs of static versus runtime checks is an interesting subject for future research.

3. DPJ's support for object-oriented frameworks leads naturally to further work on new *parallel abstractions*, implemented as frameworks or possibly even first-class language features. Abstractions make programmers more productive by allowing them to think at a higher level, without worrying about implementation details. This thesis has explored two kinds of parallel abstractions (data parallel operations on disjoint containers, and pipelined loops), but many more abstractions remain to be explored, including both general and domain-specific ones.

4. There is work to be done on *verifying the implementations* of frameworks and other parallel abstractions, for properties such as type preservation, effect preservation, and noninterference explored in this thesis. Such verification can be done with a combination of the effect system techniques discussed here and other static and dynamic techniques, including more general program logic, testing, and model checking.

In sum, this thesis has contributed to the state of the art in parallel programming languages and effect systems by (1) articulating a coherent approach to the problem of making parallel programming easier; (2) introducing a practical new language supported by novel technical contributions in support of that approach; and (3) identifying several important areas of ongoing and future research.

# References


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

[2] `http://iss.ices.utexas.edu/lonestar/`.

[3] `http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166ydocs/jsr166y/forkjoin/package-summary.html`.

[4] `http://http://sites.google.com/site/deucestm`.

[5] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *ACM SIGACT-SIGPLAN Symp. on Principles of Prog. Langs. (POPL)*, pages 63–74, New York, NY, USA, 2008. ACM Press.

[6] Martín Abadi, Cormac Flanagan, and Stephen N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. on Prog. Langs. and Systs. (TOPLAS)*, 28(2):207–255, 2006.

[7] Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog. (PPOPP)*, pages 185–196, New York, NY, USA, 2009. ACM Press.

[8] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *ACM Conf. on Prog. Lang. Design and Impl. (PLDI)*, pages 26–37, New York, NY, USA, 2006. ACM Press.

[9] Yehuda Afek, Guy Korland, and Arie Zilberstein. Lowering STM overhead with static analysis. In *LCPC*, 2010.

[10] Farhana Aleen and Nathan Clark. Commutativity analysis for software parallelization: Letting program transformations see the big picture. In *Int'l Conf. on Arch. Support for Prog. Langs. and Operating Systs. (ASPLOS)*, pages 241–252, New York, NY, USA, 2009. ACM Press.

[11] Matthew D. Allen, Srinath Sridharan, and Gurindar S. Sohi. Serialization sets: A dynamic dependence-based parallel execution model. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog. (PPOPP)*, pages 85–96, New York, NY, USA, 2009. ACM Press.

[12] Philippe Altherr and Vincent Cremet. Adding type constructor parameterization to Java. In *Formal Techniques for Java-like Programs (FTFJP)*, 2007.

[13] Zachary Anderson, David Gay, Rob Ennals, and Eric Brewer. SharC: Checking data sharing strategies for multithreaded c. In *ACM Conf. on Prog. Lang. Design and Impl. (PLDI)*, pages 149–158, New York, NY, USA, 2008. ACM Press.

[14] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *USENIX Symp. on Operating Syst. Design and Impl. (OSDI)*, 2010.

[15] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. In *ACM SIGPLAN Conf. on Object-Oriented Prog., Systs., Langs., and Apps. (OOPSLA)*, pages 227–244, New York, NY, USA, 2008. ACM.

[16] Nels E. Beckman, Yoon Phil Kim, Sven Stork, and Jonathan Aldrich. Reducing STM overhead with access permissions. In *Int'l Workshop on Aliasing, Confinement and Ownership in Object-Oriented Prog. (IWACO)*, pages 1–10, New York, NY, USA, 2009. ACM Press.

[17] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: A compiler and runtime system for deterministic multithreaded execution. In *Int'l Conf. on Arch. Support for Prog. Langs. and Operating Systs. (ASPLOS)*, pages 53–64, New York, NY, USA, 2010. ACM Press.

[18] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe multithreaded programming for C/C++. In *ACM SIGPLAN Conf. on Object-Oriented Prog., Systs., Langs., and Apps. (OOPSLA)*, pages 81–96, New York, NY, USA, 2009. ACM Press.

[19] Ganesh Bikshandi, Jia Guo, Daniel Hoeflinger, Gheorghe Almasi, Basilio B. Fraguela, María J. Garzarán, David Padua, and Christoph von Praun. Programming for parallelism and locality with hierarchically tiled arrays. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog. (PPOPP)*, pages 48–57, New York, NY, USA, 2006. ACM Press.

[20] Guy E. Blelloch. Programming parallel algorithms. *Commun. of the ACM*, 39(3):85–97, 1996.

[21] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *J. Parallel and Distrib. Comp.*, 21(1):4–14, April 1994.

[22] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog. (PPOPP)*, 1995.

[23] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel Java. In *ACM SIGPLAN Conf. on Object-Oriented Prog., Systs., Langs., and Apps. (OOPSLA)*, pages 97–116, New York, NY, USA, 2009. ACM Press.

[24] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *ACM SIGPLAN Conf. on Object-Oriented Prog., Systs., Langs., and Apps. (OOPSLA)*, pages 211–230, New York, NY, USA, 2002. ACM Press.

[25] John Boyland. The interdependence of effects and uniqueness. In *Workshop on Formal Techs. for Java Progs.*, 2001.

[26] John Boyland. Checking interference with fractional permissions. In *Int'l Symp. on Static Analysis (SAS)*, pages 55–72. Springer, 2003.

[27] Matthew Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David August. Revisiting the sequential programming model for multi-core. In *ACM/IEEE Int'l Symp. on Microarchitecture (MICRO)*, pages 69–84, Washington, DC, USA, 2007. IEEE Computer Society.

[28] Nathan G. Bronson, Christos Kozyrakis, and Kunle Olukotun. Feedback-directed barrier optimization in a strongly isolated STM. In *ACM SIGACT-SIGPLAN Symp. on Principles of Prog. Langs. (POPL)*, pages 213–225, New York, NY, USA, 2009. ACM Press.

[29] Sebastian Burckhardt, Alexandro Baldassin, and Daan Leijen. Concurrent programming with revisions and isolation types. In *ACM SIGPLAN Conf. on Object-Oriented Prog., Systs., Langs., and Apps. (OOPSLA)*, pages 691–707, New York, NY, USA, 2010. ACM Press.

[30] Nicholas R. Cameron, Sophia Drossopoulou, James Noble, and Matthew J. Smith. Multiple ownership. In *ACM SIGPLAN Conf. on Object-Oriented Prog., Systs., Langs., and Apps. (OOPSLA)*, pages 441–460, New York, NY, USA, 2007. ACM Press.

[31] Michael J. Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *ACM SIGPLAN Conf. on Object-Oriented Prog., Systs., Langs., and Apps. (OOPSLA)*, pages 414–426, New York, NY, USA, 1994. ACM Press.

[32] Bradford L. Chamberlain, Sung-Eun Choi, E. Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. ZPL: A machine independent programming language for parallel computers. *IEEE Trans. Softw. Eng.*, 26(3):197–211, 2000.

[33] K. M Chandy and Ian Foster. A deterministic notation for cooperating processes. Technical report, California Institute of Technology, Pasadena, CA, 1993.

[34] K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. Technical Report CaltechCSTR:1992.cs-tr-92-13, California Institute of Technology, 1992.

[35] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *ACM SIGPLAN Conf. on Object-Oriented Prog., Systs., Langs., and Apps. (OOPSLA)*, pages 519–538, New York, NY, USA, 2005. ACM Press.

[36] L. Paul Chew. Guaranteed-quality mesh generation for curved surfaces. In *Symp. on Comp. Geom.*, pages 274–280, New York, NY, USA, 1993. ACM Press.

[37] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *ACM SIGPLAN Conf. on Object-Oriented Prog., Systs., Langs., and Apps. (OOPSLA)*, pages 292–310, New York, NY, USA, 2002. ACM Press.

[38] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *Euro. Conf. on Object-Oriented Prog. (ECOOP)*, 2003.

[39] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *ACM SIGPLAN Conf. on Object-Oriented Prog., Systs., Langs., and Apps. (OOPSLA)*, pages 48–64, New York, NY, USA, 1998. ACM Press.

[40] Thinking Machines Corp. CM Fortran reference manual, version 1.0. Technical report, Cambridge, Massachusetts, February 1991.

[41] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *ACM Conf. on Prog. Lang. Design and Impl. (PLDI)*, pages 59–69, New York, NY, USA, 2001. ACM Press.

[42] Jayant Desouza and Laxmikant V. Kal. MSA: Multiphase specifically shared arrays. In *Int'l Workshop on Langs. and Compilers for Parallel Comp. (LCPC)*, 2004.

[43] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. DMP: Deterministic shared memory multiprocessing. In *Int'l Conf. on Arch. Support for Prog. Langs. and Operating Systs. (ASPLOS)*, pages 85–96, New York, NY, USA, 2009. ACM Press.

[44] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Int'l Symp. on Distrib. Comp. (DISC)*, 2006.

[45] Pedro C. Diniz. Commutativity analysis: A new analysis technique for parallelizing compilers. *ACM Trans. on Prog. Langs. and Systs. (TOPLAS)*, 19(6):942–991, 1997.

[46] Malcolm Dowse and Andrew Butterfield. Modelling deterministic concurrent I/O. In *Int'l Conf. on Funct. Prog. (ICFP)*, pages 148–159, New York, NY, USA, 2006. ACM Press.

[47] Manuel Fähndrich and Robert DeLine. Adoption and focus: Practical linear types for imperative programming. In *ACM Conf. on Prog. Lang. Design and Impl. (PLDI)*, pages 13–24, New York, NY, USA, 2002. ACM Press.

[48] Mingdong Feng and Charles E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *ACM Symp. on Parallelism in Algorithms and Arch. (SPAA)*, pages 1–11, New York, NY, USA, 1997. ACM Press.

[49] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the Sisal language project. *J. Parallel and Distrib. Comp.*, 10(4):349–366, 1990.

[50] C. Flanagan and M. Felleisen. The semantics of future and an application. *J. Funct. Prog.*, 9(1):1–31, 1999.

[51] Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimization. In *ACM SIGACT-SIGPLAN Symp. on Principles of Prog. Langs. (POPL)*, pages 209–220, New York, NY, USA, 1995. ACM Press.

[52] Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for Java. *ACM Trans. on Prog. Langs. and Systs. (TOPLAS)*, 30(4):1–53, 2008.

[53] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. Fortran D language specification. Technical Report CRPC-TR90079, Rice University, Houston, TX, December 1990.

[54] Rakesh Ghiya, Daniel Lavery, and David Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *ACM Conf. on Prog. Lang. Design and Impl. (PLDI)*, pages 47–58, New York, NY, USA, 2001. ACM Press.

[55] D. K. Gifford, P. Jouvelot, J. M. Lucassen, and M. A. Sheldon. FX-87 reference manual. Technical Report MIT/LCS/TR-407, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1987.

[56] David K. Gifford, Pierre Jouvelot, Mark A. Sheldon, and James W. O'Toole. Report on the FX-91 programming language. Technical Report MIT/LCS/TR-531, 1992.

[57] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, 3rd edition, 2005.

[58] Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. Thread-modular shape analysis. In *ACM Conf. on Prog. Lang. Design and Impl. (PLDI)*, pages 266–277, New York, NY, USA, 2007. ACM Press.

[59] Aaron Greenhouse and John Boyland. An object-oriented effects system. In *Euro. Conf. on Object-Oriented Prog. (ECOOP)*, pages 205–229, London, UK, 1999. Springer-Verlag.

[60] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *ACM Conf. on Prog. Lang. Design and Impl. (PLDI)*, pages 282–293, New York, NY, USA, 2002. ACM Press.

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

[62] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *ACM SIGPLAN Conf. on Object-Oriented Prog., Systs., Langs., and Apps. (OOPSLA)*, pages 388–402, New York, NY, USA, 2003. ACM Press.

[63] Tim Harris, Jim Larus, and Ravi Rajwar. *Transactional Memory (Synthesis Lectures on Computer Architecture)*. Morgan & Claypool Publishers, 2nd edition, 2010.

[64] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog. (PPOPP)*, pages 48–60, New York, NY, USA, 2005. ACM Press.

[65] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *ACM Conf. on Prog. Lang. Design and Impl. (PLDI)*, pages 14–25, New York, NY, USA, 2006. ACM Press.

[66] Philip J. Hatcher, Anthony J. Lapadula, Robert R. Jones, Michael J. Quinn, and Ray J. Anderson. A production-quality C* compiler for hypercube multicomputers. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog. (PPOPP)*, pages 73–82, New York, NY, 1991. ACM Press.

[67] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Rice University, Houston, TX, 1993.

[68] Bart Jacobs, Frank Piessens, Jan Smans, K. Rustan M. Leino, and Wolfram Schulte. A programming model for concurrent object-oriented programs. *ACM Trans. on Prog. Langs. and Systs. (TOPLAS)*, 31(1):1–48, 2008.

[69] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comp. Survs.*, 36(1):1–34, 2004.

[70] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *ACM SIGACT-SIGPLAN Symp. on Principles of Prog. Langs. (POPL)*, pages 295–308, St. Petersburg Beach, Florida, 21–24 1996.

[71] Pierre Jouvelot and David Gifford. Algebraic reconstruction of types and effects. In *ACM SIGACT-SIGPLAN Symp. on Principles of Prog. Langs. (POPL)*, pages 303–310, New York, NY, 1991. ACM Press.

[72] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[73] Aditya Kulkarni, Yu David Liu, and Scott F. Smith. Task types for pervasive atomicity. In *ACM SIGPLAN Conf. on Object-Oriented Prog., Systs., Langs., and Apps. (OOPSLA)*, pages 671–690, New York, NY, USA, 2010. ACM Press.

[74] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *ACM Conf. on Prog. Lang. Design and Impl. (PLDI)*, pages 211–222, New York, NY, USA, 2007. ACM Press.

[75] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 2006.

[76] Edward A. Lee. The problem with threads. *Computer*, 39:33–42, 2006.

[77] K. Rustan M. Leino, Arnd Poetzsch-Heffter, and Yunhong Zhou. Using data groups to specify and check side effects. In *ACM Conf. on Prog. Lang. Design and Impl. (PLDI)*, pages 246–257, New York, NY, USA, 2002. ACM Press.

[78] Wei Liu, James Tuck, Luis Ceze, Wonsun Ahn, Karin Strauss, Jose Renau, and Josep Torrellas. POSH: a TLS compiler that exploits program structure. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog. (PPOPP)*, pages 158–167, New York, NY, USA, 2006. ACM Press.

[79] H.-W. Loidl, F. Rubio, N. Scaife, K. Hammond, S. Horiguchi, U. Klusik, R. Loogen, G. J. Michaelson, R. Pena, S. Priebe, Á J. Rebón, and P. W. Trinder. Comparing parallel functional languages: Programming and performance. *Higher Order Symbol. Comput.*, 16(3):203–251, 2003.

[80] Yi Lu and John Potter. Protecting representation with effect encapsulation. In *ACM SIGACT-SIGPLAN Symp. on Principles of Prog. Langs. (POPL)*, pages 359–371, New York, NY, USA, 2006. ACM Press.

[81] Roberto Lublinerman, Swarat Chaudhuri, and Pavol Cerny. Parallel programming with object assemblies. In *ACM SIGPLAN Conf. on Object-Oriented Prog., Systs., Langs., and Apps. (OOPSLA)*, pages 61–80, New York, NY, USA, 2009. ACM Press.

[82] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *ACM SIGACT-SIGPLAN Symp. on Principles of Prog. Langs. (POPL)*, pages 47–57, New York, NY, USA, 1988. ACM Press.

[83] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, F. Bodin, and S. Kesavan. Implementing a parallel C++ runtime system for scalable parallel systems. In *ACM/IEEE Conf. on Supercomp.*, pages 588–597, New York, NY, USA, 1993. ACM Press.

[84] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comp. Arch. Letters*, 5(2):17, 2006.

[85] Nicholas D. Matsakis and Thomas R. Gross. A time-aware type system for data-race protection and guaranteed initialization. In *OOPSLA*, 2010.

[86] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: Synchronization inference for atomic sections. In *ACM SIGACT-SIGPLAN Symp. on Principles of Prog. Langs. (POPL)*, pages 346–358, New York, NY, USA, 2006. ACM Press.

[87] Michael Metcalf. Fortran 95. *SIGPLAN Fortran Forum*, 15(2):19–22, 1996.

[88] Michael Metcalf and John Reid. *Fortran 90 Explained*. Oxford University Press, New York, 1992.

[89] Bertrand Meyer. Systematic concurrent object-oriented programming. *Commun. of the ACM*, 1993.

[90] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IEEE Int'l Symp. on Workload Characterization (IISWC)*, 2008.

[91] Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *ACM SIGACT-SIGPLAN Symp. on Principles of Prog. Langs. (POPL)*, pages 51–62, New York, NY, USA, 2008. ACM Press.

[92] Adriaan Moors, Frank Piessens, and Martin Odersky. Towards equal rights for higher-kinded types. In *Int'l Workshop on Multiparadigm Programming*, 2007.

[93] Luc Moreau. The semantics of Scheme with future. In *Int'l Conf. on Funct. Prog. (ICFP)*, pages 146–156, New York, NY, USA, 1996. ACM Press.

[94] Rishiyur S. Nikhil. ID version 90.0 reference manual. Technical Report Computation Structures Group Memo 284-1, Laboratory for Computer Science, Massachusetts Institute of Technology, July 1990.

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

[96] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Int'l Conf. on Arch. Support for Prog. Langs. and Operating Systs. (ASPLOS)*, pages 97–108, New York, NY, USA, 2009. ACM Press.

[97] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.

[98] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *ACM SIGACT-SIGPLAN Symp. on Principles of Prog. Langs. (POPL)*, pages 75–86, New York, NY, USA, 2008. ACM Press.

[99] Manohar K. Prabhu and Kunle Olukotun. Using thread-level speculation to simplify manual parallelization. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog. (PPOPP)*, pages 1–12, New York, NY, USA, 2003. ACM Press.

[100] Mohammad Raza, Cristiano Calcagno, and Philippa Gardner. Automatic parallelization with separation logic. In *Euro. Symp. on Langs. and Systs. (ESOP)*, pages 348–362, Berlin, Heidelberg, 2009. Springer-Verlag.

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

[102] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, New York, NY, 1999.

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

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

[105] Martin C. Rinard and Monica S. Lam. The design, implementation, and evaluation of Jade. *ACM Trans. on Prog. Langs. and Systs. (TOPLAS)*, 20(3):483–545, 1998.

[106] Caitlin Sadowski, Stephen N. Freund, and Cormac Flanagan. SingleTrack: A Dynamic Determinism Checker for Multithreaded Programs. In *Euro. Symp. on Langs. and Systs. (ESOP)*, pages 394–409, Berlin, Heidelberg, 2009. Springer-Verlag.

[107] Florian T. Schneider, Vijay Menon, Tatiana Shpeisman, and Ali-Reza Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *ACM SIGPLAN Conf. on Object-Oriented Prog., Systs., Langs., and Apps. (OOPSLA)*, pages 181–194, New York, NY, USA, 2008. ACM Press.

[108] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. In *ACM Conf. on Prog. Lang. Design and Impl. (PLDI)*, pages 78–88, New York, NY, USA, 2007. ACM Press.

[109] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford parallel applications for shared-memory. Technical report, Stanford University, 1992.

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

[111] Matthew Smith. Towards an effects system for ownership domains. In *Euro. Conf. on Object-Oriented Prog. (ECOOP)*, 2005.

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

[113] J. Steffan and T Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *Int'l Symp. on High-Performance Comp. Arch.*, page 2, Washington, DC, USA, 1998. IEEE Computer Society.

[114] J. Gregory Steffan, Christopher Colohan, Antonia Zhai, and Todd C. Mowry. The STAMPede approach to thread-level speculation. *ACM Trans. on Comp. Systs.*, 23(3):253–300, 2005.

[115] Sven Stork, Paulo Marques, and Jonathan Aldrich. Concurrency by default: Using permissions to express dataflow in stateful programs. In *ACM SIGPLAN Conf. on Object-Oriented Prog., Systs., Langs., and Apps. (OOPSLA)*, pages 933–940, New York, NY, USA, 2009. ACM Press.

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

[117] Tachio Terauchi and Alex Aiken. A capability calculus for concurrency and determinism. *ACM Trans. on Prog. Langs. and Systs. (TOPLAS)*, 30(5):1–30, 2008.

[118] William Thies, Michal Karczmarek, and Saman Amarasinghe. Streamit: A language for streaming applications. In *Int'l Conf. on Compiler Construction (CC)*, 2002.

[119] Peter Thomas and Ray Weedon. *Object-Oriented Programming in Eiffel: 2nd Ed.* Addison-Wesley Longman, 1998.

[120] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher Order Symbolic Comp.*, 17(3):245–265, 2004.

[121] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Info. Comp.*, 132(2):109–176, 1997.

[122] Mohsen Vakilian, Danny Dig, Robert Bocchino, Jeffrey Overbey, Vikram Adve, and Ralph Johnson. Inferring method effect summaries for nested heap regions. In *Int'l Conf. on Softw. Eng'g (ASE)*, pages 421–432, Washington, DC, USA, 2009. IEEE Computer Society.

[123] David Vandevoorde and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, November 2002.

[124] Christoph von Praun, Luis Ceze, and Calin Caşcaval. Implicit parallelism with ordered transactions. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Prog. (PPOPP)*, pages 79–89, New York, NY, USA, 2007. ACM Press.

[125] P. Wadler. Linear types can change the world! *Working Conf. on Prog. Concepts and Methods*, pages 347–359, 1990.

[126] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. on Comp. Logic*, 4(1):1–32, 2003.

[127] Adam Welc et al. Revocation techniques for Java concurrency. *Concurrency and Computation: Practice and Experience*, 2006.

[128] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for Java. In *ACM SIGPLAN Conf. on Object-Oriented Prog., Systs., Langs., and Apps. (OOPSLA)*, pages 439–453, New York, NY, USA, 2005. ACM Press.

[129] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: Why the going gets tough. In *ACM Symp. on Parallelism in Algorithms and Arch. (SPAA)*, pages 265–274, New York, NY, USA, 2008. ACM Press.

[130] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *ACM Conf. on Prog. Lang. Design and Impl. (PLDI)*, pages 349–361, New York, NY, USA, 2008. ACM Press.

[131] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, and A. Schwald. Vienna Fortran — A language specification. Technical Report Internal Report 21, ICASE, NASA Langley Research Center, March 1992.

# Author's Biography

Robert Bocchino grew up in Haverford, Pennsylvania, outside of Philadelphia. He attended Harvard College from 1990–94 and graduated with a Bachelor of Arts degree, *cum laude* in Mathematics. From 1994–97, he attended Harvard Law School, where he graduated with a J.D., *cum laude*. He practiced intellectual property law at the Boston firm of Foley Hoag LLP for several years before relocating to Urbana, Illinois, to pursue graduate studies in computer science. In the fall of 2010, Mr. Bocchino joined the Special Faculty of Carnegie Mellon University as a Postdoctoral Associate working with Professor Jonathan Aldrich and supported by a Computing Innovation Fellows grant.

Outside of computer science, Mr. Bocchino pursues many varied interests. These include Baroque violin playing, singing, and recorder playing; music theory and composition; baseball (he supports the Boston Red Sox); designing and playing fantasy and science fiction role playing games; and reading literature and philosophy.