Tasks with Effects

A Model for Disciplined Concurrent Programming

Stephen Heumann Vikram Adve
University of Illinois at Urbana-Champaign
{heumann1,vadve}@illinois.edu

Abstract

Today's widely-used concurrent programming models provide few safety guarantees, making it easy to write code with subtle errors. Some parallel programming models offering stronger guarantees have been proposed, but these are often too restrictive to express the full range of uses for concurrency in large applications.

In this paper, we argue for a new programming model based on *tasks with effects*. In this model, the core unit of work is a dynamically-created *task*. The key feature of our model is that each task has programmer-specified *effects*, and a runtime scheduler is used to ensure that two tasks are run concurrently only if they have non-interfering effects. This allows us to guarantee that programs written in this model are data-race free. We describe this programming model and its properties, briefly discuss our prototype implementation of it, and propose several research questions related to this model.

1. Motivation

Concurrent programming has become ubiquitous. To exploit the full capabilities of the multicore processors in today's desktops and even handheld devices, they must be programmed in parallel. This will only become more true in the future, as processor performance gains continue to come largely from increased parallelism. Concurrency is also used for reasons other than providing parallel performance gains. In an interactive application, long-running operations should be run concurrently with the user interface event processing in order to preserve responsiveness, whether or not the time-consuming operation is itself parallel. Some operations are also simply most naturally expressed in terms of concurrent actors [4] communicating asynchronously.

Today's complex applications often combine these forms of concurrency. In nearly any interactive application that does time-consuming computations, it will be desirable both to parallelize them if possible and to run them concurrently with user interface operations. A game, for example, may have high-performance graphics and AI algorithms that are internally parallel, and may also run these operations and others such as network communications concurrently with each other and with user interface operations. Servers responding to client requests also use concurrency in similar ways.

Unfortunately, concurrent programming introduces a great deal of complexity and opportunities for errors. Correctness challenges like data races, deadlocks, atomicity violations, and memory models [2] can be a huge drag on productivity. A particularly insidious issue is that concurrent programs can have nondeterministic behavior that varies from run to run depending on the interleaving of operations, producing bugs that only occasionally manifest themselves.

Today's mainstream programming models do little to address these problems. The most common approach today is to use explicit threads and locks, which are low-level, error prone, and difficult to reason about. They provide no well-defined structure for the parallel control flow and no guarantees about parallel correctness properties. Systems based on the more abstract notion of *tasks*, such as Intel's Cilk [6] and Threading Building Blocks (TBB) [13], Apple's Grand Central Dispatch and operation queues [5], Microsoft's Task Parallel Library in .NET [19], and the `ForkJoinTask` framework in Java 7 [17, 21] provide more structured parallel control and synchronization constructs. However, none of these systems offer checked guarantees of strong safety properties either, not even data-race freedom.

A few languages such as Erlang are based on the actor model of concurrency [4], which does not support shared memory, and so eliminates errors such as data races and atomicity violations. These languages do not eliminate other concurrency errors like deadlock and unintentional nondeterminism. Moreover, a model without shared mutable memory is not well suited for many high performance algorithms that require fine-grained updates to shared global state.

Another limitation of many proposed models for safe parallel programming is performance. In particular, a number of models include mechanisms to execute code speculatively and roll back execution if two pieces of code executing concurrently perform conflicting accesses. A wide variety of systems use such mechanisms, including software transactional memory systems [12], Galois [16], and Aida [18]. These systems can offer appealing programming models, but there is generally a cost in performance, both from the need to log and check information about memory accesses, and from the possibility of discarded work due to rollbacks.

A fundamental question that arises is what guarantees a programming model should provide. The Deterministic Parallel Java (DPJ) language [8] provided a strong set of parallel safety guarantees that greatly simplify reasoning about parallel programs written in it, which can combine deterministic and nondeterministic algorithms. DPJ guaranteed five properties: (a) data race freedom; (b) strong atomicity [1]; (c) deadlock freedom; (d) deterministic semantics for parallel computations, unless explicitly written using nondeterministic parallel constructs; and (e) sequential equivalence for deterministic constructs. These guarantees are stronger than any previous or current parallel programming model we know of that supports both deterministic and nondeterministic parallelism. *We believe that these properties are appropriate for future applications that combine interactive and compute-intensive components.*

Although DPJ provides strong guarantees, it is too restrictive to express the various forms of concurrency used in complex interactive applications. Most critically, DPJ is restricted to a fork-join parallelism structure, which is not suited to the general, event-driven form of concurrency required by the interactive aspects of these applications. DPJ may be able to express some individual parallel computations within these programs, but it faces two restrictions here as well. First, it can only express fork-join parallelism, and excludes cases like pipelined computations or algorithms with more general task graphs [3]. Second, DPJ relies on a purely static type system to enforce its correctness requirements, and many algorithms cannot be checked with such a system, e.g. graph-based algorithms. Also, DPJ's support for nondeterministic computations relies on a software transactional memory system, which performs poorly due to the cost of logging and rollbacks.

We seek to define a new programming model that can provide most or all of the guarantees provided by DPJ, but with the flexibility and expressivity to support a wide variety of concurrent programs including interactive applications, and without the performance problems of speculation and rollback. To satisfy these goals, we propose a new programming model based on *tasks with effects*. This model uses tasks similar to those of other task-based systems, but requires the programmer to declare the *effects* of each task. The run-time system then schedules tasks so as to ensure that only tasks with non-interfering effects can run concurrently. In combination with a static phase that ensures the declared effects of a task are sound, this can provide a guarantee that the program is data-race free. We also ensure that our effect system does not give rise to deadlocks. Finally, our model supports explicitly declared deterministic algorithms.

2. Basic Programming Model

In our programming model of tasks with effects, a program execution consists entirely of a set of tasks. A program is launched by creating an initial main task, and further tasks may be created by the program as it executes. Each task can optionally take arguments and return a value at its completion, like a future. In our prototype implementation, we use the `executeLater` operation to create a task, and the `getValue` operation to await the completion of a task and get the value it returned, if any.

Effects are used to control the scheduling of tasks. Each task has an effect specification, which is checked at compile time to ensure that it accurately (conservatively) reflects the task's memory accesses. These effect specifications of tasks are in turn used at run time by the task scheduler, which will ensure that no two tasks with interfering effects can run concurrently. This combination of static effect checking and an effect-aware run-time scheduler guarantees that the program will be data race free.

2.1 Effects and Regions

In order to perform effect-based scheduling of tasks, it must be possible to characterize the effects of each task, and to check whether the effects of two different tasks interfere with each other. In our prototype system we focus on the effects of memory accesses, although effects could also be used to control access to other types of resources. Thus, each effect can be thought of as permitting either read or write access to a certain set of memory locations.

Intuitively, two effects interfere if they could cover accesses to the same memory location and at least one of those accesses could be a write. Two sets of effects interfere if there is pairwise interference between any two individual effects in the sets. Two tasks can only be run concurrently if their effects do not interfere, which is the core property enforced by our scheduler.

In our prototype implementation, we use the effect system originally developed for the Deterministic Parallel Java (DPJ) language [7]. DPJ is an extended version of Java that uses type and effect annotations. Based on those annotations, the DPJ compiler can guarantee that a restricted class of parallel programs behave deterministically. In this work, however, we adopt the type and effect system from DPJ for use in combination with our effect-based task scheduling model, which can support a much broader range of concurrent programs.

The DPJ type and effect system is based on a partitioning of memory into *regions*. The programmer can declare the names of regions, and each object field and array cell is specified to be in a particular region. DPJ supports nested hierarchies of regions using a mechanism called *region path lists* (*RPLs*), and a wildcard $*$ can be used in RPLs to specify effects covering a set of regions. The DPJ region system also includes features such as region-parameterized types and a mechanism to place each element of an array in its own region.

Using this partitioning of memory into regions, DPJ allows the effects of any operation in the program to be specified in terms of read and write effects on memory regions. In

a DPJ program, the programmer declares the effects of each method as part of its method signature. Using purely static, intraprocedural checks, the DPJ compiler can then verify that the declared effects of each method actually cover the effects of every operation in the method. The DPJ type and effect system also defines formally under what circurstances two effects can be proven to be non-interfering (intuitively, they must be on entirely disjoint regions of memory, unless both are only read effects).

In our system, we adopt DPJ's region-based type and effect system and require the programmer to declare the effects of each task and each method, which are statically checked as in DPJ. Unlike in DPJ, however, the compiler generates code to keep track of the effects of each task at run time. This information is then used by the run-time scheduler to guarantee noninterference of effect between concurrent tasks.

2.2 Effect-Based Task Scheduling

The key property that must be enforced by the run-time task scheduler in our model is that two tasks with interfering effects will not be run concurrently. To do this, the scheduler will have to delay the execution of tasks that are created while another task with interfering effects is already executing. In an efficient scheduler, it will probably also be desirable to delay tasks based on the availability of execution resources, e.g. by using a thread pool sized to the number of available cores.

Considerable variation is possible in the design of an effect-aware task scheduler. In our initial prototype implementation, we use a fairly simple approach based on a single queue of tasks. In a higher-performance implementation, the effect checking could be structured around regions, so that tasks accessing entirely disjoint regions do not need to be explicitly checked against each other. It may also be helpful to have the scheduler enforce additional properties related to fairness or task ordering, in addition to the basic property of noninterference. We believe that designing an effect-based task scheduler that guarantees useful properties and offers good performance for a wide range of parallel algorithms will be an important area of future work.

3. Effect Transfer

The model we have described so far envisions the effects of each task remaining unchanged throughout the lifetime of that task. However, it can be valuable to change the effects of tasks during their lifetimes, and in particular to transfer effects from one task to another. Effect transfer can be used to support a guarantee that certain computations are deterministic (see section 4), to eliminate a class of deadlocks, and to support a construct similar to a synchronized block.

An implementation of effect transfer should preserve the property that no two tasks have interfering effects while they are executing concurrently. To do this, the system must ensure that a task does not perform operations covered by a given effect after it transfers that effect away (unless it is later transferred back).

Effect transfer allows the effects covering a task or a method to change while it is running. This introduces some additional complexity in our approach of statically checking that each memory access in the program code is covered by the declared effects of the task or method where it appears. We now need to individually track the *covering effects* applicable to each operation in the program, taking into account changes in the applicable set of effects due to effect transfer operations. We prefer to maintain a purely static approach for checking the effects of individual memory accesses, both for efficiency reasons and to detect errors at compile time. Therefore, we compute and check covering effects conservatively using a new data flow analysis pass in the compiler.

3.1 Effect Transfer on Task Creation and Completion

In our prototype implementation, we support two major types of effect transfer operations. The first is a system to transfer some of the effects of a parent task to a newly-created child task, and later transfer those effects back to the parent task when the child task completes and unblocks the parent. This is particularly useful for fork-join styles of parallelism, and it can be used to ensure that some algorithms are deterministic, as described in section 4.

A parent task can create multiple child tasks, each with effects covering the data it will work on (e.g. an element or range of elements in an array). If they are created with the special operation `spawn`, then the system will ensure that the parent task's effects cover each child task's effects, and will automatically transfer each child task's effects from the parent to that child when it is created. This enables the child task to run immediately, since "ownership" of the effects is transferred directly from the parent to the child task, and thus no other tasks with conflicting effects may be running simultaneously.

If a task is created with `spawn`, its parent task may await its completion with the `join` operation. All spawned tasks are automatically joined at the end of the method that spawned them, if they are not explicitly joined earlier. The join operation transfers the effects of the completed child task back to its parent task.

3.2 Effect Transfer when Waiting for a Task

The second kind of effect transfer we use in our system is from a blocked task to a task it is waiting on, primarily to prevent deadlock. When one task is waiting for another to complete using a `getValue` or `join` operation, we allow the waiting task's effects to be transferred to the task it is waiting on, if necessary in order for that task to execute. This effect transfer is also applied recursively through a chain of waiting tasks. The effects are automatically transferred back to the original task before it resumes execution following the `getValue` or `join` operation.

This mechanism prevents a situation that could otherwise give rise to deadlock, where one task is directly or indirectly waiting on another, but that other task cannot start because its effects conflict with those of the first task. This does not eliminate all deadlocks, but in practice we found that it addressed several cases that would otherwise deadlock in a program written with our prototype system.

This also allows for a programming paradigm similar to a synchronized or atomic block in traditional programming models. One task can launch a second task with a superset of its effects, and then use a `getValue` operation to wait for the second task. This transfers the first task's effects to the second task (allowing it to access the same regions as the first task), and leaves the second task to wait until it can acquire access to the regions covered by its other effects, which would typically correspond to a shared resource.

4. Guaranteed Determinism

Programs written using our model are not required to be deterministic. There are significant classes of parallel algorithms that are inherently nondeterministic, and it is also not generally useful to speak of the determinism of an interactive program driven by user input or external requests. Therefore, we think it is important that our system not restrict all programs written it to be deterministic.

However, many parallel algorithms are in fact deterministic. That is, they always produce the same output given the same input state. Since this is an expected property of many algorithms, detecting violations of it can be a useful way of finding bugs. Moreover, knowing that a program or an algorithm within a program is deterministic makes it much easier to reason about: the user of the program or algorithm knows that it will always produce the same output given the same input, so they need not be concerned that different parallel interleavings of operations may produce different results. Determinism also makes a program or algorithm much simpler to debug, since one knows that the same result will be produced every time it is run with a given input.

DPJ [7] can provide a compile-time guarantee of determinism using the combination of its type and effect system and simple parallelism constructs supporting only fork-join patterns of parallelism. We aim to provide a similar static guarantee of determinism for deterministic algorithms or programs in our system.

To do this, we allow the programmer to annotate tasks or methods as `@Deterministic`. In code that has this annotation, the compiler will enforce that the only task-related operations used in the code are the `spawn` and `join` operations described in section 3.1. Also, code annotated as deterministic may call only other deterministic methods and spawn other deterministic tasks.

These restrictions ensure that the code invoked from a deterministic task or method (including through the creation of other tasks) accesses memory only as specified by its declared effects. Moreover, there is a defined order by which control of each region covered by those effects is transferred between tasks, as determined by `spawn` and `join` operations. (Note that effect transfer on `join` operations, as described in section 3.2, will never be needed in a deterministic computation, and thus will not occur.) Therefore, for a given input state of the memory in regions covered by the effects of the deterministic task or method, there is a deterministic output state that will not vary between executions of the deterministic code. This state is the same as the state produced if the code were executed sequentially with each task's code run at the point where the task is spawned. These deterministic computations are also deadlock-free.

```
1  class Image {
2      region Top, Bottom;
3      final int[]<Top> topHalf;        // pixel values
4      final int[]<Bottom> bottomHalf;
5      ...
6      @Task @Deterministic void
7      increaseContrast() writes Top, Bottom {
8          SpawnedTaskFuture<Void, writes Top> f =
9              increasePixelContrast.spawn(topHalf);
10         increasePixelContrast(bottomHalf);
11         f.join();
12     }
13     @Task @Deterministic private <region R> void
14     increasePixelContrast(int[]<R> pixels) writes R {
15         // modify values in pixels array
16     }
17 }
```

Figure 1. Example computation.

Our system can provide a compile-time guarantee of determinism for a large class of programs, including all deterministic programs supported by DPJ. In addition, we can provide a determinism guarantee for individual algorithms within larger programs that are not fully deterministic. We believe this is very valuable for realistic programs such as interactive applications. It is often not possible to guarantee that the entire program runs deterministically based on its initial inputs, but it is still useful to check that individual algorithms behave deterministically.

5. Example

Figure 1 gives an example of how our task system might be used in an image editing program. It shows a class `Image` representing an image, with the pixel values held in two arrays, `topHalf` and `bottomHalf`. The cells of these two arrays are defined as being in the regions `Top` and `Bottom`, respectively. (We show this arrangement for simplicity. In a real program, we could put each array cell in its own region. This would allow for more fine-grained parallelism, e.g. at the level of rows in the image. We could also place the data for different `Image` objects in separate regions, potentially allowing them to be updated concurrently.)

The task `increaseContrast` can be executed to increase the contrast of the image. It would typically be run with `executeLater`, which would place it in a queue to be scheduled. Because its effects are declared as `writes Top, Bottom`, our scheduler will ensure that it is

scheduled only when no other tasks that access the regions `Top` or `Bottom` are running concurrently.

A separate task `increasePixelContrast` actually updates the pixel values in an array. It has a region parameter `R` corresponding to the region containing the cells of the array passed to it. Since its declared effect is `writes R`, `increasePixelContrast(topHalf)` has the effect `writes Top`.

Within the `increaseContrast` task, we wish to operate in parallel on the two halves of the image. We do this using the `spawn` operation (described in section 3.1) to create an instance of the `increasePixelContrast` task with the argument `topHalf`. This transfers the effect `writes Top` directly from the parent `increaseContrast` task to the new child task, which means the new task can be enabled for execution immediately, since there cannot be any other tasks with conflicting effects executing concurrently. The parent task also continues executing concurrently, with its remaining effect `writes Bottom`. It runs `increasePixelContrast(bottomHalf)` as a method within the same task, which is possible since it still retains the effect `writes Bottom`.

After that computation finishes, the parent task joins the future returned when the child task was spawned. This `join` operation also transfers the child task's effect `writes Top` back to the parent task. After this, both halves of the image will have been updated, so any other task waiting for the `increaseContrast` task to finish (using `getValue`) will know that the full operation is complete by that point.

The computation done by `increaseContrast` is deterministic. The `@Deterministic` annotations on both tasks reflect this, and the compiler will verify that these tasks behave deterministically, as described in section 4.

6. Research Questions

There are several key research questions raised by our model of tasks with effects. These include the following:

How should effects be represented in source code? Important considerations in designing the effect system include ease of use for the programmer, expressiveness to support various patterns of data access and sharing, and the performance overheads of the run-time scheduling mechanisms.

We have adopted the region-based effect system from DPJ in our initial prototype, but other effect systems are possible. Ownership type systems express effects in terms of objects [9, 20], which have somewhat more limited expressivity (at least in systems so far) but might be easier to use. ACCORD [15] expresses effects directly in terms of program variables, which is convenient but raises the problem of aliasing between variable names. It might also be desirable to introduce new types of effects focused on resources such as files, database records, or other shared resources.

A different dimension of simplifying the programming burden is minimizing the effect annotations that must be written. One attractive choice is to design ways to encapsulate code without internal effect annotations, e.g., so that annotations are only needed at module boundaries for separately compiled modules. Another valuable approach could be to relate higher-level and lower-level effect annotations at abstraction boundaries. Automatic effect inference algorithms may be needed for these strategies [22].

How should effects be represented and compared at run time? Our current prototype implementation uses a task queue, where all the effects of each task must be checked against all the effects of the other tasks that are running or are ahead of it in the queue. But for improved performance and scalability, we would like to use an effect-checking scheme based on the actual regions on which each task has effects. Effect checking would then only require checks against other tasks accessing the same regions, and it could proceed in parallel for tasks with effects on completely unrelated regions.

To design a system based on these principles, we will also have to define suitable data structures to represent effects. If we continue to use our current hierarchical region system, perhaps a tree-like structure accessed using hand-over-hand locking could be appropriate. It may also be desirable to implement fast special cases for simple, commonly-occurring patterns of effect usage. These run-time performance considerations may in turn affect how the effects must be represented in the program code, requiring this question to be considered together with the previous one.

How can the tasks-with-effects model be extended for heterogeneous and non-shared-memory systems? Our work so far has focused on shared-memory multicore systems, but we believe our model can also be applicable to other parallel system architectures. Most client systems today include both a CPU and a programmable GPU, and future systems are predicted to have an even more diverse range of hardware, including general purpose cores, a GPU, Digital Signal Processors (DSPs), programmable logic devices like FPGAs, and multiple custom or semi-custom accelerator cores for different functions [11, 14]. It would be desirable to program these using a unified programming model. We believe the tasks-with-effects model can be used for a wide range of different hardware, including heterogeneous systems. One advantage of our model is that explicit, region-based effects provide a natural means for specifying the data movement between cores and accelerators, many of which require explicit data copying between global memory and local, scratchpad memories.

In any architecture where data must be explicitly transferred between different processors, regions and region-based effects might be used as a means of controlling data transfers. For example, the DeNovo project [10] has shown that on a global address-space architecture, it is possible to implement a highly efficient region-based coherence protocol using extremely simple hardware. One key question is how the DeNovo model could be extended for a true

message-passing system, where a global address space is not available. When a task with effects on a region is started on a processor different from the one that owns the region, the system could automatically transfer the necessary data.

How can stronger safety properties be enforced in the tasks-with-effects model? Our system ensures noninterference of effect among concurrently-executing tasks, and with our region-based effect system this gives a guarantee of data-race freedom. We can also provide a guarantee of determinism for many deterministic algorithms, as described in section 4. We believe these are important safety properties that should be guaranteed by the system. Moreover, the programmer can work around restrictions on the expressiveness of the system by choosing whether or not to request the strongest guarantees, such as our determinism guarantee.

The major property that is not supported sufficiently in the current model is deadlock freedom. Guaranteed-deterministic algorithms are deadlock-free, and the effect transfer mechanism described in section 3.2 eliminates a large class of deadlocks, but it is still possible to write a program that deadlocks using our system. Since all synchronization in our model relies on our runtime system, it should be possible to detect a deadlock and fail in an orderly way, although doing so may impact performance. It would be valuable, if possible, to also provide a static guarantee of deadlock freedom, at least for some large class of programs.

7. Summary

Future client-side applications will increasingly combine rich interactivity with more demanding compute-intensive algorithms, requiring sophisticated concurrent programming models. In order to preserve programmer productivity while achieving high performance, these programming models must enforce strong correctness guarantees, such as data race freedom, atomicity, deadlock freedom, and determinism.

We have described a new programming model based on tasks with effects. The effects of each task are specified by the programmer and statically checked by the compiler. A run-time task scheduler is then used to ensure that tasks with interfering effects are not run concurrently. This effect-based scheduling approach guarantees the absence of data races. We also define the concept of effect transfer between tasks, and show how this can be used to verify that many algorithms are deterministic.

We believe our programming model is suitable for a wide range of concurrent and parallel programs, including large applications that use concurrency in multiple different ways. More generally, we believe a combination of static checks and a dynamic runtime that enforces certain properties is a good approach to verifying safety properties without unduly compromising expressivity. We feel that our programming model, and more broadly the problem of designing disciplined yet flexible systems for concurrent programming, offers numerous opportunities for valuable ongoing research.

Acknowledgements

This work was funded by the Illinois-Intel Parallelism Center at the University of Illinois at Urbana-Champaign. The Center is sponsored by the Intel Corporation.

References

[1] M. Abadi et al. Semantics of transactional memory and automatic mutual exclusion. In *POPL*, 2008.

[2] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Comp., Special Issue on Shared-Mem. Multiproc.*, pages 66–76, December 1996.

[3] V. S. Adve and M. K. Vernon. Parallel program performance prediction using deterministic task graph analysis. *ACM Trans. on Comp. Systs.*, 22(1):94–136, 2004.

[4] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[5] Apple. Concurrency Programming Guide. http://developer.apple.com/library/mac/documentation/General/Conceptual/ConcurrencyProgrammingGuide/, 2011.

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

[7] R. L. Bocchino et al. A type and effect system for Deterministic Parallel Java. In *OOPSLA*, 2009.

[8] R. L. Bocchino et al. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL*, 2011.

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

[10] B. Choi et al. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In *PACT*, 2011.

[11] R. Hameed et al. Understanding sources of inefficiency in general-purpose chips. In *ISCA*, 2010.

[12] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*. Morgan & Claypool Publishers, 2010.

[13] Intel. Intel Thread Building Blocks Reference Manual. http://software.intel.com/sites/products/documentation/hpc/tbb/referencev2.pdf, Aug. 2011.

[14] R. Iyer et al. Cogniserve: Heterogeneous server architecture for large-scale recognition. *IEEE Micro*, 31:20–31, May 2011.

[15] R. K. Karmani et al. Thread contracts for safe parallelism. In *PPoPP*, 2011.

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

[17] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, 2000.

[18] R. Lublinerman et al. Delegated isolation. In *OOPSLA*, 2011.

[19] Microsoft. Task Parallel Library. http://msdn.microsoft.com/en-us/library/dd460717.aspx.

[20] P. Müller and A. Rudich. Ownership transfer in universe types. In *OOPSLA*, 2007.

[21] Oracle. Java Platform, Standard Edition 7 API specification. http://download.oracle.com/javase/7/docs/api/.

[22] M. Vakilian et al. Inferring Method Effect Summaries for Deterministic Parallel Java. In *ASE*, 2009.