Parallel Closures

A new twist on an old idea

Nicholas D. Matsakis
Mozilla Research
nmatsakis@mozilla.com

Abstract

This paper presents a lightweight task framework and accompanying type system that statically guarantee deterministic execution. The framework is based on the familiar model of fork-join parallelism, but with two important twists. First, child tasks do not begin execution immediately upon creation, but rather they are both scheduled and joined as one atomic action; this change prevents the parent task from racing with its children. Second, the body of a child task is specified as a *parallel closure*.

Parallel closures are a novel variation on traditional closures in which the data inherited from the environment is read-only. Parallel closures have the important property that they can be executed in parallel with one another without creating data races, even if they share the same environment. We also have a controlled means to grant mutable access to data in the environment where necessary.

We have implemented a prototype of our framework in Java. The prototype includes a typechecker that enforces the constraint that parallel closures cannot modify their environment. The paper describes how the prototype has been used to implement a number of realistic examples and also explains how parallel closures can support the creation of structured parallel programming abstractions.

1. Introduction

As multi-core computers are becoming more widely available, parallel programming is increasing rapidly in importance. One very popular model for writing parallel programs is to use a lightweight task framework. Such frameworks are now available in the standard libraries of both Java 7 and C#, and a number of packages are available for other languages. Unfortunately, while using such a framework makes it easier to address some of the efficiency problems that bedevil threaded programs, they do little to address the possibility of data races.

This paper proposes two small changes to the common model for lightweight task frameworks; these changes make it possible to statically guarantee a deterministic result with only minimal added complexity. Unlike other type systems that offer comparable safety guarantees, we do not require an effect system [23] nor any form of dependent types. The only requirement is the ability to declare transitive read-only pointers, which is itself a commonly requested and useful feature even in sequential code [39]. Our implementation takes the form of an extension to Java, but the ideas are easily ported to other languages and environments.

The first change that we propose is to prevent parent and child tasks from executing concurrently. In all parallel systems of which we are aware[1], forked tasks can potentially run in parallel with their creator. In our system, however, multiple child tasks are accumulated and then forked-and-joined in one atomic action. This prevents the parent task from racing with its children.

The second change that we propose is to specify the body of a child task using a *parallel closure*. As with traditional closures, a parallel closure is a block of code which can access variables from its surrounding environment. In a novel twist on traditional closures, however, parallel closures are only granted read-only access to the data which they inherit from their environment. This change means that two parallel closures can safely execute in parallel with one another, as any data that is shared between them is read-only.

For those cases where read-only access is insufficient, we also provide a means for granting controlled access to mutable data. We show how this can be used to allow large arrays to be divided into disjoint regions, each of which are accessible only one task at a time. Currently, making use of this feature entails a minimal dynamic safety check, though we have plans (discussed in Section 3.3) to remove these checks in the future.

[1] With the exception of the author's previous work [26], which took a rather more complex approach to the type system.

We have implemented these ideas in a Java prototype. The prototype includes a type checker that guarantees that parallel closures do not modify their parent's environment. The type checker is based on the `@ReadOnly` annotation from the Javari [39] project. The runtime is built using a simple wrapper over Java 7's `ForkJoinPool`. We have used the prototype to build a number of examples such as a parallel ray tracer and an equation solver based on Successive Over-relaxation.

The paper begins in Section 2 by explaining how our parallel framework is used as well as the type system extensions that guarantee parallel closures are being used properly. Next, Section 3 describes the prototype implementation in more detail and Section 3.3 explores some possible future directions. Finally, Section 4 summarizes the related work.

2. Using parallel closures

This section covers both the syntax for using our framework and the type system extensions that used to guarantee data-race freedom.

2.1 Forking and joining child tasks

To create and join tasks, we borrow the **`async`** and **`finish`** keywords from the X10 [8] language. The **`async`** keyword corresponds to a function whose signature is as follows:

```
<T> Task<T> async(ParClosure<Void, T> fn)
```

Here, the class `ParClosure<A,R>` class is an abstract base class representing instances of parallel closures. The type parameter `A` represents the type of argument expected by the closure and `R` is the type that the closure returns.

The function **`async()`**, therefore, expects a zero argument closure (represented by an argument type of `Void`) which returns a value of type `T`. The result of **`async()`** is a child task object. Unlike in X10, child tasks do not begin execution immediately. Instead, their execution is deferred until the exit from the enclosing **`finish`** block. At that point, all child tasks created within the block are both scheduled and joined. Once all child tasks have completed, the parent will resume execution. There is an implicit **`finish`** block around every task body, so every forked child will eventually be scheduled.

As an example, a simple map-reduce like pattern could be implemented as follows:

```
Task<T> task1, task2;
finish {
  task1 = async({|| /* code for child 1 */ });
  task2 = async({|| /* code for child 2 */ });
}
reduce(task1.get(), task2.get());
```

The code first specifies a finish block and then creates two child tasks within using the **`async()`** function. The two tasks will both begin execution once control flow exits the finish block and they will complete before the finish block itself completes. This flow is depicted graphically in Figure 1.

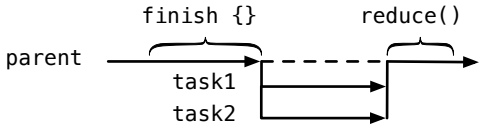

Figure 1. Graphical depiction of the parallel control flow for the example in Section 2.1.

The arguments to the **`async()`** function are parallel closures. Parallel closures are denoted using the Smalltalk- or Ruby-like syntax of a code block that begins with vertical bars (`||`) enclosing a list of parameters.[2] Here, the closures take no arguments, so there are simply two adjacent vertical bars. Later we will see examples of parallel closures which expect an argument as well.

The type of a task in our framework is `Task<T>`, where `T` is the type of the value returned from the closure. `Task` offers two methods, `get()` and `join()`. Both methods return the result of the closure, but they are intended for different purposes. `get()` is used by the parent of the task. It can only be invoked after the **`finish`** block has terminated; attempting to call it earlier results in an exception.

The `join()` method is intended to be used from sibling tasks. It causes the sibling task to block until the joined task is finished. The result type of a `join()` is not `T` but `@ReadOnly T`; as will be explained in the next section, this type guarantees that the data returned from `join()` cannot be modified by the caller. Dynamic monitoring can be used to throw an exception in the case of cyclic joins.

2.2 Preventing data races

Part of the framework is a type system that guarantees data-race freedom. This goal is achieved by enforcing two constraints: first, no parallel closure is permitted to modify data found in its surrounding environment. Second, results of other tasks obtained by a call to `join()` also cannot be mutated. This does not mean that parallel closures cannot use mutable data: they are free to create and modify any number of objects as they please. But objects which they inherit from the surrounding lexical scope or obtain from other tasks must generally be read-only (however, we will see in the next section that there is a mechanism for giving controlled access to mutable data from the environment when appropriate).

These simple rules are sufficient to guarantee data-race freedom because at all times any data that is shared between multiple concurrent tasks is read-only to all of them. The key observation is that the only way for two tasks to share access to the same data is if they either (1) inherit it from their environment or (2) obtain the data through a call to `join()`. In both cases, the only task with write access to

[2] This is an idealized syntax. The prototype implementation uses Java's anonymous classes.

```
1  void parentMethod(List<Data> parentList) {
2    Task<Data> childTask1, childTask2;
3    finish {
4      childTask1 = async({||...});
5      childTask2 = async({||
6        // ERROR: cannot call set() on read-only list
7        parentList.set(0, null);
8
9        // The data contained in parentList is read-only:
10       @ReadOnly Data parentData = parentList.get(0);
11
12       // Data obtained via join() is read-only:
13       @ReadOnly Data siblingData = childTask1.join();
14
15       // Data created within the child can be mutable:
16       Data childData = new Data();
17       childData.setVariousFields();
18       return childData;
19     });
20   }
21
22   // Parent can mutate data created by child:
23   Data childData = childTask.get();
24   childData.update();
25 }
```

Figure 2. Example for explaining the type system

the data is either dormant (parent task) or terminated (joined sibling task).

To prevent parallel closures from mutating their environment, we make use of a simple rule: within a parallel closure, the types of all variables in the environment are qualified with a read-only qualifier. The technique of modifying the type of a variable when accessed from within a closure is unique to our system, to our knowledge.

To implement the read-only qualifier, the prototype uses a variant of the Javari [39] system (we had to make some small changes, as discussed in the Section 3). At a high level, Javari defines a `@ReadOnly` annotation which can be attached to types (e.g., `@ReadOnly List<T>` or `@ReadOnly Date`). `@ReadOnly` is similar to the **`const`** keyword found in C and C++, but stronger: its effects are transitive, so reading a field of a read-only object itself yields a read-only value.

To better explain our technique, consider the example in Figure 2. Here there is a method, `parentMethod()`, which takes as its parameter a list `parentList`. As the name suggests, this list is owned by the parent task.

Within the method `parentMethod()`, a new child task is created. This child can access the `parentList` from its environment. However, within the environment, the type of `parentList` is modified to be `@ReadOnly List<Data>`. As a result, attempts to set the contents of the list result in an error, as shown on line 7.

Furthermore, because the `@ReadOnly` qualifier is transitive, the data read from the list is itself read-only. Therefore, any data items retrieved from the parent list will also be `@ReadOnly`, as shown on line 10. Similarly, data obtained from joining siblings will be `@ReadOnly`, as shown on line 13. This is because the return type of `join()` is specified with the `@ReadOnly` qualifier.

However, there are no restrictions on the types used for data created within the child itself. So, for example, data items and lists created within the child can have a non-readonly type and can be mutated as normal (lines 16–17).

Finally, when the child completes, it may return mutable data to its parent. The `get()` method, unlike `join()`, does not convert its data to read-only (lines 23–24). This is safe because invoking `get()` before the **`finish`** has completed yields an exception.

Importantly, all of these constraints followed from adding the `@ReadOnly` qualifier to the type of `parentList` when viewed from inside the parallel closure. The rest of the Javari type system functions precisely as it did before.

2.3 Controlled mutation and higher-level patterns

One of the goals of this project is to provide a foundation for building parallel control-flow abstractions [11, 28], much as traditional closures are used in functional languages like Scheme and Haskell as well as object-oriented languages like Smalltalk and Ruby. Traditional closures are not well-suited for this role because of the frame problem [27]: that is, the library cannot prevent them from racing based on the data in the environment.

For many parallel patterns, however, the restriction that child tasks may only modify data which the child task itself creates is too severe. Our solution to this problem is to make it possible for library to pass in mutable pointers to a parallel closure through closure parameters. That is, while all variables in the parallel closure's environment have `@ReadOnly` type, the parameters to a parallel closure may have any type.

This puts the library in complete control over what mutable data the closure can access. For example, if the library has control over all aliases to the object in question (e.g., when the object is newly created), then it can choose to provide the object directly to the parallel closure.

If the library cannot guarantee the absence of aliases, then it must dynamically ensure that the object in question is not accessed from multiple tasks. An example of this technique in our framework is the method `divide()` which is offered on arrays. `divide()` creates a number of tasks and provides each task with a different view onto the original array. Each view only permits access to a disjoint range of indices. Meanwhile, a flag is set on the original array which was divided so that any direct access to the array rather than a view yields an exception. This way, we can be sure that each task only accesses the region of memory it was assigned.

In general, the runtime checks needed to divide arrays are cheap. They could easily be combined with boundary checks and be made cheaper still. Nonetheless, it would be nice if they were not necessary at all. Unfortunately, Java's permissive attitude towards aliasing makes that virtually impossible without drastic changes to the type system or a whole program analysis. Section 3.3 discusses how the Rust language,

```
void render (Scene scene, int width,
             int[] outputBuffer) {
  finish {
    outputBuffer.divide(1, {|int[] view|
      Range range = view.range;
      for (int i = range.min; i < range.max; i++) {
        int y = i / width;
        int x = i - y * width;
        int color = computeColor(scene, x, y);
        view[i] = color;
      }});
  }}
```

Figure 3. Example of using `divide()`: a parallel raytracer

currently under development at Mozilla, could be used to eliminate these dynamic checks altogether.

The `divide()` method for an array type `T[]` has the following signature:

```
void divide(int chunk, ParClosure<T[], Void> fn)
```

The first parameter, `chunk`, specifies how many items the closure expects to process at a time; the size of each subview will be some multiple of the chunk size. The second parameter, `fn`, is a parallel closure which expects a `T[]` as argument: this will be the view array, whose underlying storage is in fact shared with the parent array, but whose active range is a subset of the parent array. The return type of the parallel closure is simply `Void`.

2.4 Extended Examples

This section gives two more examples highlighting various features of our framework. The first shows how array division can be used. The second illustrates how we can easily accommodate data structures that transition between being mutable and read-only depending on the phase of the program.

2.4.1 Array division: parallel ray tracing

Figure 3 shows how the `divide()` method is used in our parallel raytracer to parallelize over the output buffer. The method `render()` is supplied with a scene and an output buffer, represented as an `int[]`.

The method invokes the `divide()` method to process each pixel in parallel. The closure expects one argument, `view`, specifying the range of pixels that it is responsible for. The view is itself an array instance. Arrays in our system all feature a field `range` which specifies the set of indices that have been assigned to this particular child task (for a non-divided array, the range is simply `0..l-1`, where `l` is the length of the array). In the prototype, these ranges are always continuous subregions. In the future, additional methods can be added which permit different kinds of division, as well as division on more kinds of data structures or on multiple data structures at once.

To process its assigned portion of the output buffer, the child task simply loops over the range found in the view, computes the color for the corresponding pixel, and then stores that color back into the output buffer. Note that the store is done via the `view` array, not the original `outputBuffer` array: use of the `outputBuffer` is prohibited until the children end and would result in an exception.

```
void compute(float[] black, float[] red,
             int cols, int iterations) {
  for (int i = 0; i < iterations; i++) {
    finish {
      black.divide(cols, {|float[] view|
        Range r = view.range;
        for (int i = r.min; i < r.max; i += cols) {
          int r = i / cols;
          blackFromRed(view, red, r);
        }});
    }

    finish {
      red.divide(cols, {|float[] view|
        Range r = view.range;
        for (int i = r.min; i < r.max; i += cols) {
          int r = i / cols;
          redFromBlack(view, black, r);
        }});
    }}}
```

Figure 4. Successive over-relaxation

2.4.2 Support for phases: Successive over-relaxation

The successor over-relaxation (SOR) method is a technique for solving linear equations. The algorithm iterates repeatedly over the matrices representing the linear equations, refining the solution in each iteration. Within an iteration, the processing of rows can be parallelized. The implementation of SOR which we started from makes use of two arrays, labeled as red and black: in each iteration it reads from one array while writing to the other, and then vice versa. It is quite simple to port this over to our system, as shown in Figure 4. We represent the two-dimensional matrix using a one-dimensional array; because the algorithm processes items one row at a time, the `divide()` method is instructed to divide the matrix into row-sized pieces by specifying that the views created by divide should have a multiple of `cols` items.

This example highlights how our type system naturally accommodates data which transitions from mutable to read-only and back again. This kind of time-based, flow-sensitive typing is quite challenging to describe in other type systems, if it is possible at all.

3. Implementation and future directions

The current prototype can be downloaded from GitHub, including the runtime, type checker, and all of the examples discussed in this paper [25].

The prototype implementation of the type system is built on the JSR308 annotation framework. Subclasses of the `ParClosure` class are treated specially by our type checker:

they are forbidden from having fields (just as closures do not have fields), and any free variables are assigned a `@ReadOnly` type.

3.1 ReadOnly vs const pointers in C++

The Javari paper [39] itself provides a better summary of the everyday experience of using `@ReadOnly` types than is possible here. However, as many readers are familiar with **`const`** pointers in C++ (and some of their hazards), it is worth briefly summarizing some of the most salient differences between our variant of Javari and C++ (in addition to the fact that **`const`** is shallow and `@ReadOnly` is deep).

First, readers familiar with **`const`** in C++ may recall having occasionally had to duplicate methods in order to have one copy which accepts (and returns) a **`const`** parameter and one copy which does not. This is not necessary in Javari due to the availability of polymorphic qualifiers, which allows a method to specify a return type which is read-only if and only if the parameter type is read-only (the method must assume that the parameter may be read-only).

Second, in our variant of the Javari system, we do not allow fields to be declared mutable, as is possible both in C++ and the original Javari system. The intension of this declaration is to allow for writes that perform caching of intermediate results and otherwise "appear" read-only from the outside. Unfortunately, in a parallel setting, even innocent-looking writes like these can create data races.

3.2 Limitations of the approach and implementation

There are two limitations to the current Java-based checker which cannot be easily rectified without moving to a new language. First, the current checker cannot prevent children from communicating (and potentially racing) via static fields. Second, array division works best for arrays of primitive types; arrays of reference types (i.e., objects) can be divided, but the objects themselves will be read-only. This is because the objects stored in the arrays may be aliased.

There are also numerous areas where the prototype could be extended without undue difficulty. For example, additional patterns for dividing arrays are needed. For example, a common technique for implementing the SOR algorithm described in Section 2.4.2 is to combine the black and red arrays into one array and use a checkerboard pattern. It would be particularly useful to make it easy for users to define custom data access patterns.

It would also be useful to offer an efficient means of handling tasks with structured, regular dependencies (sometimes called wavefront computations). Such computations can be constructed using the join method on tasks, but the result is not especially efficient.

3.3 Future directions

Improved static checking in Rust. To address the shortcomings of the Java-based type checker, the author is planning an experimental integration of parallel closures into the language Rust,[3] currently under development at Mozilla. Rust offers a stricter control of aliasing via unique pointers [29] and flat arrays (i.e., arrays of structures, not arrays of pointers to structures), making it possible to remove all dynamic checks from `divide()` operations.

Dynamic monitoring in JavaScript. Whereas Rust improves on the current prototype by offering stronger static checking, another possibility is to monitor for illegal writes dynamically. This is the approach taken by the PJs project, [24] which is implementing parallel closures in JavaScript. The PJs project is underway but currently in its early stages.

4. Related work

Prominent projects that are based on fork-join processing include Cilk [37], OpenMP [32], and of course X10 [8], from which the **`finish`** and **`async`** keywords were borrowed. Our execution model differs because parents and children do not execute concurrently (and of course we offer static protection from data races in the bargain).

The `join()` method on tasks permits a safe form of future. Futures date back at least as far as MultiLISP[19], but have since spread to numerous other settings [30, 31, 40].

One of the inspirations for this work was McCool's paper on structured parallel programming [28], published at a prior HotPar (as well as similar works, e.g., [11]). Using parallel closures, it is possible for parallel combinators to be sure that the only access which tasks have to shared mutable data is through the parameters provided to those tasks.

The goal of eliminating data races has a long history in the literature. Even simply focusing on type systems yields a wide variety of approaches [1, 3–6, 16–18, 21, 42]. The distinguishing feature of our work lies in its ability to achieve the strong guarantee of deterministic results while adding only minimal complexity to the type system itself. In contrast, prior systems which achieve comparable results, such Determinstic Parallel Java [4, 5], feature elaborate effect systems for tracking which memory is read and written.

In terms of static race detection, perhaps the closest work to our own is the work on permissions-based type systems [7, 22, 38], which associate permissions with each memory location. A full permission is required for writes, but permissions may be split to allow multiple readers. Our work can be viewed as a permissions-based system in which the permission to an object is implicitly split upon forks and regained upon joins.

As mentioned in Section 3.3, we also aim to explore the use of our model for dynamic race detection, which has been an active research area for some time [2, 9, 10, 12–15, 33, 35, 36, 43] The most closely related work to our own, however, lies in those efforts which have taken advantage of a more limited model than the typical permissive thread

[3] Note though that there are currently no plans to merge these changes into the mainstream Rust.

model (e.g., [13, 20, 34, 41]). Such work has shown that it is possible to radically lower the overheads of dynamic checks in such cases, generally because it is possible to exclude data which is known not to be involved in a race or by avoiding the need to track a fine-grained happens-before relation. Our model would seem to offer a particularly lightweight detection strategy, in that the problem of detecting races is instead reduced to detecting disallowed writes, and the parallel structure of the program can be represented as a fork-join tree.

4.1 Conclusions

This paper presents a lightweight task framework and accompanying type system that statically guarantee data-race freedom. The framework is based on the familiar model of fork-join parallelism, but with two important twists. First, child tasks do not begin execution immediately upon creation, but rather they are both scheduled and joined as one atomic action; this change prevents the parent task from racing with its children. Second, the body of a child task is specified as a parallel closure, which has read-only access to its environment. We also show how parallel closures can be type-checked using read-only pointers.

One of the larger goals of this project is to create abstractions that can be used to build larger parallel libraries. Using closures to build structured control flow has been shown to provide a flexible and expressive alternative to the C approach of using special forms. Parallel closures open the door to using a similar approach to build structured *parallel* control flow without opening the door to data races: library routines can safely schedule closures in parallel in any combination, safe in the knowledge that the closures will not race with one another.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for java. *ACM Trans. Program. Lang. Syst.*, 28:207–255, March 2006.

[2] R. Agarwal and S. D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, PADTAD '06, pages 51–60, New York, NY, USA, 2006. ACM.

[3] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: checking data sharing strategies for multithreaded C. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 149–158, New York, NY, USA, 2008. ACM.

[4] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM.

[5] R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. *SIGPLAN Not.*, 46(1):535–548, Jan. 2011.

[6] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM.

[7] J. Boyland. Checking interference with fractional permissions. In *Proceedings of the 10th international conference on Static analysis*, SAS'03, pages 55–72, Berlin, Heidelberg, 2003. Springer-Verlag.

[8] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[9] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, PLDI '02, pages 258–269, New York, NY, USA, 2002. ACM.

[10] M. Christiaens and K. D. Bosschere. Trade: Data race detection for java. In *Proceedings of the International Conference on Computational Science-Part II*, ICCS '01, pages 761–770, London, UK, UK, 2001. Springer-Verlag.

[11] J. Darlington, A. J. Field, P. G. Harrison, P. H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel programming using skeleton functions. In *Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, PARLE '93, pages 146–160, London, UK, 1993. Springer-Verlag.

[12] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 245–255, New York, NY, USA, 2007. ACM.

[13] M. Feng and C. E. Leiserson. Efficient detection of determinacy races in cilk programs. In *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, SPAA '97, pages 1–11, New York, NY, USA, 1997. ACM.

[14] C. Flanagan and S. N. Freund. The roadrunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '10, pages 1–8, New York, NY, USA, 2010. ACM.

[15] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 293–303, New York, NY, USA, 2008. ACM.

[16] A. Greenhouse and W. L. Scherlis. Assuring and evolving concurrent programs: annotations and policy. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 453–463, New York, NY, USA, 2002. ACM.

[17] D. Grossman. Type-safe multithreading in cyclone. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '03, pages 13–25, New York, NY, USA, 2003. ACM.

[18] P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 354–378, Berlin, Heidelberg, 2010. Springer-Verlag.

[19] R. H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 9–17, New York, NY, USA, 1984. ACM.

[20] K. J. Hoffman, H. Metzger, and P. Eugster. Ribbons: a partially shared memory programming model. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 289–306, New York, NY, USA, 2011. ACM.

[21] A. Kulkarni, Y. D. Liu, and S. F. Smith. Task types for pervasive atomicity. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 671–690, New York, NY, USA, 2010. ACM.

[22] K. R. Leino and P. Müller. A basis for verifying multi-threaded programs. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 378–393, Berlin, Heidelberg, 2009. Springer-Verlag.

[23] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.

[24] N. D. Matsakis. https://github.com/mozilla/pjs.

[25] N. D. Matsakis. https://github.com/nikomatsakis/patpar.

[26] N. D. Matsakis. *Intervals: Data-Race-Free Parallel Programming*. PhD thesis, ETH Zurich, 2011.

[27] J. Mccarthy and P. J. Hayes. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*, pages 463–502. Edinburgh University Press, 1969.

[28] M. D. McCool. Structured parallel programming with deterministic patterns. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, HotPar'10, pages 5–5, Berkeley, CA, USA, 2010. USENIX Association.

[29] N. H. Minsky. Towards alias-free pointers. In *Proceedings of the 10th European Conference on Object-Oriented Programming*, pages 189–209, London, UK, 1996. Springer-Verlag.

[30] A. Navabi and S. Jagannathan. Exceptionally safe futures. In *Proceedings of the 11th International Conference on Coordination Models and Languages*, COORDINATION '09, pages 47–65, Berlin, Heidelberg, 2009. Springer-Verlag.

[31] A. Navabi, X. Zhang, and S. Jagannathan. Quasi-static scheduling for safe futures. In *PPoPP*. ACM, 2008.

[32] OpenMP Architecture Review Board. *OpenMP 3.0 Specification*, May 2008.

[33] E. Pozniansky and A. Schuster. Multirace: efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, Mar. 2007.

[34] R. Raman, J. Zhao, V. Sarkar, M. Vechev, and E. Yahav. Efficient data race detection for async-finish parallelism. In *Proceedings of the First international conference on Runtime verification*, RV'10, pages 368–383, Berlin, Heidelberg, 2010. Springer-Verlag.

[35] C. Sadowski, S. Freund, and C. Flanagan. Singletrack: A dynamic determinism checker for multithreaded programs. In G. Castagna, editor, *Programming Languages and Systems*, volume 5502 of *Lecture Notes in Computer Science*, pages 394–409. Springer Berlin / Heidelberg, 2009.

[36] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15:391–411, November 1997.

[37] Supercomputing Technologies Group, Massachusetts Institute of Technology Laboratory for Computer Science. *Cilk 5.4.6 Reference Manual*, Nov. 2001.

[38] T. Terauchi. Checking race freedom via linear programming. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 1–10, New York, NY, USA, 2008. ACM.

[39] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20, 2005.

[40] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for java. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 439–453, New York, NY, USA, 2005. ACM.

[41] E. Westbrook, J. Zhao, Z. Budimlic, and V. Sarkar. Perimssion regions for race-free parallelism. In *Proceedings of the 2nd International Conference on Runtime Verification (RV '11)*, September 2011.

[42] J. Yi, C. Sadowski, and C. Flanagan. Cooperative reasoning for preemptive execution. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 147–156, New York, NY, USA, 2011. ACM.

[43] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 221–234, New York, NY, USA, 2005. ACM.