Practical Permissions for Race-Free Parallelism

Edwin Westbrook, Jisheng Zhao, Zoran Budimlić, and Vivek Sarkar

Rice University, Houston, TX 77005, USA
{emw4, jisheng.zhao, zoran, vsarkar}@rice.edu

Abstract. Type systems that prevent data races are a powerful tool for parallel programming, eliminating whole classes of bugs that are both hard to find and hard to fix. Unfortunately, it is difficult to apply previous such type systems to "real" programs, as each of them are designed around a specific synchronization primitive or parallel pattern, such as locks or disjoint heaps; real programs often have to combine multiple synchronization primitives and parallel patterns. In this work, we present a new permissions-based type system, which we demonstrate is practical by showing that it supports multiple patterns (e.g., task parallelism, object isolation, array-based parallelism), and by applying it to a suite of non-trivial parallel programs. Our system also has a number of theoretical advances over previous work on permissions-based type systems, including aliased write permissions and a simpler way to store permissions in objects than previous approaches.

1 Introduction

As computer vendors turn towards multi-core processors for continued performance increases, more and more programmers in the computer industry are faced with the prospect of writing, modifying, testing, and debugging parallel programs. However, working with parallel programs can be challenging due to potential *data races*. Data races can cause programs to run in unexpected and counter-intuitive ways, making parallel programs hard to write, debug, and reason about. While the possible effects of data races have been formalized using complex *memory models* [23, 8], just determining whether a race could occur, let alone what the effects of that race could be, is a significant effort. This is true even for parallelism experts working with very small programs.

There has been much research into programming languages and type systems that completely avoid data races [7, 33, 6, 25, 1, 32, 12, 11, 19, 2]. The assumption is that a data race is a bug, which is, in practice, true for most application software. Unfortunately, it is difficult to apply these past approaches to real programs, because past approaches are generally designed around specific synchronization primitives or parallel patterns, such as locks or disjoint heaps. Real programs often have to combine multiple synchronization primitives and parallel patterns, both to get good performance and to match specific algorithms. Even when previous approaches can be used, they often require a high degree of programmer effort, in terms of annotating the code and re-factoring existing parallel

algorithms to fit specific parallel patterns. Further, many programmers are not trained to use the sophisticated type systems required by these approaches.

In this paper, we present a *practical* race-free type system that supports multiple patterns (e.g., task parallelism, object isolation, array-based parallelism), and has been applied to a suite of non-trivial parallel programs. Our system is called Habanero Java with permissions (HJp) and is an extension of the Habanero Java (HJ) language [14], which itself is a task-parallel extension of Java. The core idea in HJp is that each object, at each point in time, is in one of two modes: *shared read*, where any task[1] is permitted to read from it but none is permitted to write to it; or *private read-write*, where only one task can read from or write to the object. We enforce this property by extending *permission types* [21, 35, 13, 4, 5, 12], which capture knowledge about the mode of an object at different points in the program. We introduce two technical advances to permission types. First, our system allows aliased write permissions, while prior work allows write permissions only to non-aliased pointers, which is restrictive in practice.[2] Second, our system introduces a novel approach called *storable permissions* for expressing transitive permissions, e.g., permissions to all elements of a linked list, that requires less technical machinery than previous approaches [13].

To demonstrate the practicality of our approach, we have ported 15 benchmarks from HJ to HJp, totaling almost 14,000 lines of code and covering a range of parallel patterns. This only required modifications to 5% of the lines of code on average. We compared HJp to Deterministic Parallel Java (DPJ) [7, 6], another race-free type system: for the same 5 benchmarks, HJp required modifications to 7.3% of the lines of code on average, as opposed to the 10.5% of the code that required annotations in DPJ. Further, HJp is a *gradual* [28] extension of HJ, meaning that some or all of these annotations can be omitted, and the compiler will insert dynamic type-casts where necessary to ensure race-freedom at run time. A simple and effective algorithm for inserting these type-casts was covered in prior work [34], which introduced a runtime approach for checking permissions on entry to regions of code identified by the programmer. The HJp type system presented here expands on that work by introducing programmer annotations that can reduce the number of type-casts inserted, eventually leading to a statically-verified program with no type-casts.

The remainder of this paper is organized as follows. In Sections 2, 3 and 4 we introduce and discuss the main features of the HJp type system: fractional read/write permissions, storable permissions and gradual typing, using a core calculus, called Core HJp, which is then formalized and proved to be race-free in Section 5. Extensions to Core HJp to support array-based parallelism and objects guarded by critical sections are given in Section 6. Practical experience using HJp is then discussed in Section 7, before discussing related work and concluding in the last two sections.

[1] We use "task" instead of "thread" to distinguish potential parallelism in a program from OS threads that may be used to implement this parallelism.

[2] Bierhoff and Aldrich [5] allow aliased writes for protocol enforcement, but it is unclear if their approach applies to race-free parallel programming.

2 Fractional Read/Write Permissions

Fractional permissions, first introduced by Boyland [12], are a powerful idea for fork/join parallel programming. The basic idea is that a task can start out with some permission p, and can temporarily split p in half, yielding two $\frac{1}{2}p$ permissions, both of which can be used in parallel. When parallel use of these permissions are finished, the two $\frac{1}{2}p$ permissions can be re-combined into p again. The exact fractional $\frac{1}{2}$ is not important, just that the two pieces sum to 1. For example, the following code uses permission p to read field f from x:

$$\textbf{finish}\ (\textbf{async}\ (\ldots = x.f);\ \ \ldots = x.f)$$

This code uses HJ's **async** to fork a child task, which is intuitively passed a $\frac{1}{2}p$ permission from the parent to allow it to read $x.f$. The parent continues to run in parallel, using the remaining $\frac{1}{2}p$ permission to allow it to read $x.f$ as well. The **finish** construct then indicates a join on all child tasks spawned inside the lexical block. After this join completes, the parent has p again.

The same pattern, however, cannot be allowed if p is a read/write permission to $x.f$, since this could potentially allow data races. Specifically, it should not be possible for a task to pass a fractional read/write permission to another task while still retaining any read and/or write permissions to the same object. To address this issue, previous permissions-based type systems only allow writes when a task has an *exclusive* (or "unique") permission, i.e., all of the permissions to an object. Exclusivity is indicated by the fraction "1". This guarantees that no other task can read while a task is writing. Unfortunately, it effectively means that writes are only allowed through unique references, which can be very restrictive to the programmer, especially in an OO setting. For example, with standard fractional permissions, calling even the simple function

$$\mathsf{f}\ (x, y) = x.f := y.f;$$

requires that either x and y are statically known to be distinct, or that they are statically known to be aliased so that the permission for x can be re-used for y. Such proofs require complex machinery in the type system and, further, the number of cases grows exponentially with the number of variables.

HJp, in contrast, does allow fractional write permissions. The key idea is this: it is perfectly fine to form fractional write permissions, as long as they *cannot be passed to other tasks*. Thus, HJp has two sorts of fractional permissions: shared read permissions that can be passed to other tasks; and private write permissions that cannot be passed to other tasks. In this way, a program can write to an object without having to show that the pointer is unique. This is especially useful for gradual typing (see Section 4), as it allows code to be instrumented with dynamic acquires and releases of permissions, without having to worry about potential aliases.

In more detail, HJp defines permissions syntactically as follows:

$$\begin{array}{ll} w ::= 0w \mid 1w \mid \varepsilon & \phi ::= w\mathrm{R} \mid w\mathrm{W} \mid w\mathrm{S} \\ \pi ::= x : \phi \mid \mathrm{F}(\pi) & \Pi ::= \Pi, \pi \mid \cdot \end{array}$$

The *permission words* w define the "fractionality" of a permission. Intuitively, any fractional permission ϕ can be split into two pieces, $0w\phi$ and $1w\phi$, which can be thought of as the left half and the right half of ϕ, respectively. Again, the actual permission word w is not important except to track the splitting and recombining of permissions. The *object permissions* ϕ include read permissions $w\mathrm{R}$, write permissions $w\mathrm{W}$, and sharing permissions $w\mathrm{S}$. We often write Υ to stand for R, W, or S below. Read and write permissions are straightforward, while the sharing permission $w\mathrm{S}$ allows any permissions $w\Upsilon$ with the same permission word w, including the sharing permission itself, to be passed to another task. Note that passing is linear, i.e., permissions cannot be duplicated.

The *permissions* π include: permissions $x : \phi$, that represent permission ϕ to the object pointed to by x; and *future permissions* of the form $\mathrm{F}(\pi)$, which state that the current task will have permission π after the next enclosing **finish** completes. The latter form captures the fact that some child task will hold permission π when it completes. Finally, the *permission sets* Π include zero or more permissions π. We write $\Pi|_l$ to denote the set of all permissions $l : \phi$ in Π.

Permissions are defined this way for theoretical succinctness, but can only occur in practice in permission sets built from one of the following four "building blocks" that represent the programmer view of permissions (see Section 7). Private read permissions $x : w\mathrm{R}$ allow reads $x.f$ of fields of x. Private read-/write permissions $x : w\mathrm{R}, x : w\mathrm{W}$ also allow writes to x. Neither of these can be shared with other tasks. Shared read permissions $x : w\mathrm{R}, x : w\mathrm{S}$ allow reads of x and can be shared with other tasks, but do not allow writes. Finally, exclusive permissions $x : \varepsilon\mathrm{R}, x : \varepsilon\mathrm{W}$ (W and S are interchangeable here, by our permission equality rules) allow reads from and writes to $x.f$, and can further be shared with other tasks. Splitting an exclusive permission, though, will disallow either writes or sharing, depending on how it is split. We sometimes abbreviate exclusive permissions as $l : \mathrm{X}$.

Note that, in HJp, there are no shared mutable variables. Local variables are passed by value to child tasks, and global variables are fields in static class objects, as in Java. Thus we do not include permissions for accessing the variable x itself, and instead all permissions $x : \phi$ refer to the object pointed to by x. This also implies an object granularity of permissions, instead of a field granularity. There is no inherent problem in extending HJp to include field permissions, but we have found in practice that this is not needed. Note that HJp does support array-based parallelism, though; see Section 6.

Permission sets are considered equal up to permutations and the rules

$$\Pi, (x : 0w\Upsilon), (x : 1w\Upsilon) = \Pi, x : w\Upsilon \qquad\qquad \Pi, x : \varepsilon\mathrm{W} = \Pi, x : \varepsilon\mathrm{S}$$

The first rule allows the left half $0w\Upsilon$ and the right half $1w\Upsilon$ of a permission $w\Upsilon$ to be combined. Looking at it the other way, the rule allows $w\Upsilon$ to be split into $0w\Upsilon$ and $1w\Upsilon$. The second rule allows a write permission to be exchanged for a sharing permission, or vice-versa, but only when the permission is not fractional, i.e., when exclusive permissions are held. This means that exclusive permissions can either be split into private write or shared read permissions, but not both. Permission set Π_1 is said to *subsume*, or be *more permissive* than, Π_2, written

$\Pi_1 \geq \Pi_2$, iff adding more permissions to Π_2 can yield Π_1; i.e., this holds iff there is some Π_3 such that $\Pi_2, \Pi_3 = \Pi_1$.

3 Storable Permissions

Storable permissions allow permissions to refer to a whole tree of objects instead of just a single object. The idea is that both an object o_2 and exclusive permissions to o_2 can be stored in an object field $o_1.f$. A task with permission ϕ to o_1 can then read o_2 along with ϕ permissions to o_2 out of $o_1.f$, then read ϕ permissions to an object o_3 that is stored in o_2, etc., transitively yielding permission ϕ to a whole group of objects reachable through o_1.

To use storable permissions, some of the fields of class C are designated as *exclusive fields*. (See the **exclusive** keyword in Surface HJp in Section 7.) We write f^{X} for exclusive fields, and write f^{n} for normal, non-exclusive fields. An exclusive field assignment $x.\mathrm{f}^{\mathrm{X}} := y$ implicitly stores an exclusive permission to y in x, meaning that the current task must hold $y : \varepsilon\mathrm{R}, y : \varepsilon\mathrm{W}$ permissions to execute this assignment and these permissions are not held after the assignment.[3] For exclusive field reads $y = x.\mathrm{f}^{\mathrm{X}}$, if the current task holds permission ϕ to x before the read, it then "borrows" this permission from x, yielding a ϕ permission to y after the field read. To specify which object permissions are being borrowed, exclusive field reads are annotated, as $x.\mathrm{f}^{\mathrm{X}}(\vec{\phi})$, where $\vec{\phi}$ indicates a sequence of zero or more object permissions ϕ. The borrowed permissions can then be given back with a remit, written $\mathbf{remit}_y(\vec{\phi} \to x.\mathrm{f}^{\mathrm{X}})$. For example, the code

$$\mathbf{let}\ y = x.\mathrm{f}^{\mathrm{X}}(w\mathrm{R}, w\mathrm{W})\ \mathbf{in}\ y.\mathrm{f}^{\mathrm{n}} := 1; \mathbf{remit}_y(w\mathrm{R}, w\mathrm{W} \to x.\mathrm{f}^{\mathrm{X}})$$

borrows private write permission to $y = x.\mathrm{f}^{\mathrm{X}}$ to write 1 to $y.\mathrm{f}^{\mathrm{n}}$, and then gives back these permissions when complete.

In order to prevent duplication of permissions, the same permissions $\vec{\phi}$ may not be borrowed a second time until either: a remit gives back the borrowed permissions; or another object z is assigned to $x.\mathrm{f}^{\mathrm{X}}$. To track this, we expand the syntax of object permissions as follows:

$$\phi ::= w\mathrm{R} \mid w\mathrm{W} \mid w\mathrm{S} \mid \phi - \mathrm{f}^{\mathrm{X}}@y$$

The new object permission $w\Upsilon - \mathrm{f}^{\mathrm{X}}@y$ indicates that $w\Upsilon$ has been borrowed in variable y for field f^{X}. The construct $-\mathrm{f}^{\mathrm{X}}@y$ is called a *permission subtraction*, and object permissions are considered equal modulo reordering of permission subtractions.[4] The permission subsumption relation is expanded so that $x : \phi \geq x : \phi - \mathrm{f}^{\mathrm{X}}@y$ and so that $x : \phi \geq x : \phi'$ implies $x : \phi - \mathrm{f}^{\mathrm{X}}@y \geq x : \phi' - \mathrm{f}^{\mathrm{X}}@y$. We write $\mathbf{root}(\phi)$ in the below to denote the *root* $w\Upsilon$ of $\phi = w\Upsilon - \vec{B}$.

[3] We could allow shared read permissions to be stored as well, but storing private read or write permissions could allow them to be shared, thereby hurting type soundness.

[4] The permission $x : w\Upsilon - \mathrm{f}^{\mathrm{X}}@y$ can be defined using linear implication as the permission set $x.\neg\mathrm{f}^{\mathrm{X}} : w\Upsilon, (y : w\Upsilon \multimap x.\mathrm{f}^{\mathrm{X}} : w\Upsilon)$ where $x.\mathrm{f}^{\mathrm{X}} : w\Upsilon$ and $x.\neg\mathrm{f}^{\mathrm{X}} : w\Upsilon$ (not defined here) are permissions to just $x.\mathrm{f}^{\mathrm{X}}$ and to all $x.f$ with $f \neq \mathrm{f}^{\mathrm{X}}$, respectively.

```
search :: (this : List, ξ \ this : ξR) → (Bool \ this : ξR)
search (this, ξ) = if P(this.data) then true
                   else if this.next == null then false
                   else let x = this.next(ξR) in let r = search (x, ξ) in
                        remit_x(ξR → this.next); r
```

Fig. 1. List Searching with Storable Permissions

As an example, Figure 1 shows how to write a list searching algorithm, search, with storable permissions. We assume a class List that has fields *data* and *next*, containing the data at the head of the list and the next list element, respectively. The *next* field is exclusive, so holding a permission to the head of a list is equivalent to holding the permission for the whole list. The type of search specifies both types and permissions, separated with a backslash, for input and output. The type portion states that search is a function from a List to a Bool. The permissions portion states that, on entry to search, private read permission ξR to the argument *this* is held, and that it will still be held on exit, where ξ is a permission word variable representing an arbitrary permission word.

The search function first tests if some predicate P holds of *this.data*, returning true if so. If not, then it checks *this.next*, returning false if this is **null** and recursing on *this.next* otherwise. In order to recurse, search first performs an exclusive read of *this.next*, borrowing permission ξR and binding the result to x. The recursive call itself has the form search (x, ξ), which passes x for the argument and ξ for the permission word argument of search. The result of the recursive call is bound to variable r, and the permission ξR on x is given back to *this.next* before the function returns r.

4 Gradual Permission Types in HJp

Gradual type systems [35, 28] are systems which allow some or all of the type-checking of a program to be performed dynamically, rather than statically. This turns strongly typed programming into a gradual process, where the programmer can iteratively add typing annotations to a program to reduce the number of dynamic checks, as opposed to an all-or-nothing effort of passing the type-checker. HJp uses this idea to allow programmers to omit permission annotations where they might not be known by the programmer, or might be hard to guarantee statically. In fact, HJp programs can be compiled and run with absolutely no permission annotations. This means HJp can compile HJ programs that were written with no knowledge of the permissions and HJp still guarantees race-freedom, though some of the dynamic checks may fail at runtime. On the other end of the spectrum, programs with enough permission annotations to not require any dynamic checks are guaranteed not to have any checks that fail at runtime, and, further, suffer no performance loss from dynamic checks. Gradualness in HJp is thus a powerful tool for programmer productivity, making it easy to port existing HJ programs and to write new programs with little or no

initial concern for permissions, while at the same time giving the programmer the ability to gradually perform more work to get more static guarantees.

In previous work [34], we gave a simple but powerful approach to inserting dynamic permission checks. The checks were coarse enough that the runtime overhead was relatively low for un-annotated HJ programs — the slowdown for most benchmarks was under 2.5×, as compared to the order of magnitude slowdown for some of the best dynamic race detectors — but also avoided false positives, i.e., spurious failures of dynamic checks: in 11 HJ benchmarks totaling over 9,000 lines of code, only one modification was needed to prevent false positives.[5] This latter property is especially important for HJp, as it means that the approach generally captures programmer intent, even in HJ code that was not written with permissions in mind. HJp builds upon this work by providing a type system that allows permissions to be fully statically checked. In this section, we briefly review how dynamic permission checks are inserted, and discuss how these checks fit into the type system of HJp.

To support gradual permission typing, Core HJp includes *acquires* and *releases*. An acquire tries to dynamically acquire private read, private write, shared read, or exclusive permissions to x. This succeeds if the requested permission does not conflict with any other permissions held for x. Private reads conflict with private writes in another task, shared reads conflict with any writes, and exclusive conflicts with any other permission. A conflict causes the acquire to fail, and a runtime exception is thrown. A release then gives a permission back to the runtime, indicating that the current task is finished using it.

The fact that acquires throw exceptions on failure mean that HJp implements a fail-stop semantics for data races. This is similar to other work, such as DRFx [29, 24], and means that we view data races as bugs, which is in fact the case most of the time. While an alternate approach would be for acquires to block (awaiting the availability of a permission) instead of throwing exceptions, this would add the significant possibility of deadlock, especially since the compiler inserts acquires and releases, as discussed below, that may be unknown to the programmer. Instead, acquires and releases are meant to act as a form of dynamic type cast, and so should change the semantics as little as possible.

As a side note, although HJp prevents data races, it does not guarantee *determinism*. Different acquires could fail for different executions of the same program, depending on the dynamic schedule, or some execution could report no failures at all. If an execution contains no failures, however, then it is guaranteed to contain no races, though it could still contain *potential* races.

Exclusive acquires are written $\mathbf{acquire}_{\mathrm{X}}(x)$. This construct attempts to acquire exclusive permissions for x, returning $*$ on success and raising an exception on failure. Non-exclusive acquires are written $\mathbf{let}\ \xi = \mathbf{acquire}_{\vec{Y}}(x)\ \mathbf{in}\ M$, where the sequence $\vec{Y}$ can be either R for private read, R, W for private read-write, or R, S for shared read permissions. If successful, M is executed with, respectively, permissions $x : w\mathrm{R}$, permissions $x : w\mathrm{R}, x : w\mathrm{W}$, or permissions $x : w\mathrm{R}, x : w\mathrm{S}$. We abbreviate these permissions as $x : w\vec{Y}$ below. Further, w is also substituted

[5] Array-based parallel loops also had to be modified to use new syntax; see Section 6.

for the permission word variable ξ in M. Note that programs are not allowed to acquire specific permission words w, as this would greatly complicate the implementation of these checks.

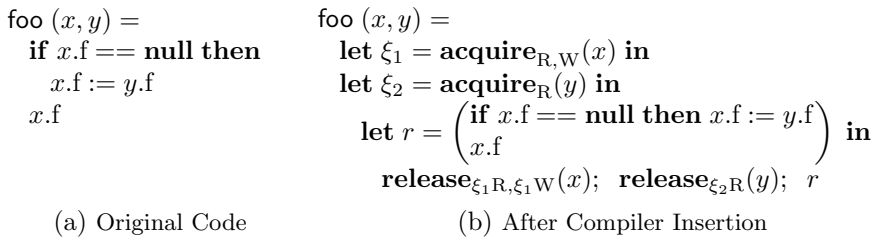
Releases are written $\mathbf{release}_{\vec{\phi}}(x)$. One caveat about releases is that we wish to ensure that a permission can only be released if it was previously acquired. Otherwise, the values of the permission words would matter, complicating the implementation. To do this, we introduce a new permission form $\mathrm{dyn}_x(w\vec{\Upsilon})$, called a *dynamic permission*, which indicates that the sequence $\vec{\Upsilon}$ of object permissions were acquired dynamically with permission word w. A release $\mathbf{release}_{w\vec{\Upsilon}}(x)$ then consumes both the permissions $x : w\vec{\Upsilon}$ and the permission $\mathrm{dyn}_x(w\vec{\Upsilon})$. Exclusive permissions do not involve a permission word w, and so are a special case that do not require a dynamic permission to be released.

In order to support gradual typing, the HJp compiler automatically inserts acquires and releases where necessary. The basic idea, as presented in previous work [34], is to add acquires and releases around variable scopes for the least permission — private write, private read, or none — needed for the given variable. This essentially creates regions of code that are as large as possible during which the current task holds permissions to the given object. As we prove in Section 5, any insertion algorithm will prevent low-level data races (as defined by the Java Memory Model [23]). Making the regions as large as possible, however, also helps to prevent high-level races, where an object is modified concurrently (without a low-level race) while the programmer intends for it to remain constant. Of course, it is impossible in general to infer programmer intent, but the HJp insertion algorithm seems to seems to capture a "sweet spot" with the intuition that programmers do not generally intend objects to be modified while they are in scope as variables. In addition, our experiments on HJ code, which were written without permissions in mind, also showed that in only one instance for all our benchmarks were these regions too big.

Note that, as a special case, object constructors are implicitly considered to be exclusive acquires, meaning that *new* returns exclusive permissions to an object. Thus the HJp insertion algorithm inserts corresponding releases at the end of an allocated object's scope.

Figure 2 illustrates the HJp insertion algorithm, also demonstrating the use of acquires and releases. The figure shows a simple function, $\mathsf{foo}(x, y)$, which first checks if x.f is **null**, assigning y.f to it if so, and then returns the possibly modified value of x.f. Figure 2(b) demonstrates how acquires and releases are added to this code: a permission region is inserted around the body of the entire function, since this is the scope of the variables x and y. The resulting code starts by acquiring write permission to x and (private) read permission to y. It then performs the original computation, binding the result to a new variable r. Finally, the code releases permissions to x and y and returns r.

Note that a more "conservative" approach would only acquire private read permission to x outside the if-expression, acquiring write permission to x and private read permission to y only when x.f equals **null**. The HJp insertion algorithm, however, captures the fact that, *logically*, calling $\mathsf{foo}(x, y)$ requires read



(a) Original Code (b) After Compiler Insertion

Fig. 2. Compiler Insertion of Acquires and Releases

permission to y and write permission to x. A violation of this logic, such as a concurrent write to (the object referenced by) y, is therefore considered to be a data race in HJp, even if the read from y does not actually take place. The user can override this behavior by manually inserting acquires and releases. The user can also completely remove any acquires and releases by changing the type of foo, to indicate that write permission to x and read permission to y are required when foo is called. This is denoted in Surface HJp by adding keywords **writing** and **reading** to x and y, respectively (see Section 7). We explain how this type is written in Core HJp in Section 5. Modifying the type of foo in this way represents a gradual refinement, moving the function foo from dynamic to static type-checking. Since the resulting function has no dynamic permission checks, it is statically guaranteed not to have any data races, though races in callers of foo could lead to exceptions being thrown.

As a final point here, the code inserted by the HJp compiler for acquires and releases is actually slightly more complex than that shown in Figure 2, for two reasons. First, the code must handle the case where an exception is thrown between an acquire and a release. This is done with a try-finally block to ensure that releases are always performed at the end of permission regions. The second reason that the inserted code is more complex is to handle mutable variables in Java. A permission region for a mutable variable x causes the specified permission to be re-acquired for the new value of x each time x is modified. All of the permissions acquired for values of x are not released until the end of the permission region, when all of the permissions are released. Both of these points are discussed more in our previous work [34], and can be modeled in a straightforward way in Core HJp.

5 Syntax and Semantics of Core HJp

In this section, we formalize Core HJp and prove that all executions are race-free. Core HJp is defined using A-normal forms, similar to 3-address codes, where a term consists of a sequence of steps, each of which returns a value that can be **let**-bound in the remainder of the computation. This closely matches the Jimple representation of the Soot optimization framework [31], which is used as a back-end for our HJp implementation. The typing judgment associates input, output, and exception permissions with each term M, in a manner similar to Hoare Type

Theory [26], where the typing judgment associates pre- and post-conditions with terms. We omit a number of high-level features present in Java, HJ and Surface HJp, such as subtyping, constructors, final fields, methods, and conditionals; objects in Core HJp are essentially nominally typed, mutable records. The other features are straightforward to add, but do not significantly affect the results.

The remainder of this section is organized as follows. Section 5.1 gives the syntax and type system for HJp. Section 5.2 then gives an operational semantics for HJp, and proves the main result, that any execution of HJp is race-free. Note that, although our operational semantics is a sequentially consistent semantics, proving that all executions are race-free means that all HJp programs are *correctly synchronized*. This in turn ensures that any execution under a weak, DRF0-based memory model (such as the Java Memory Model) is equivalent to a sequentially consistent execution as given here [23].

5.1 Static Semantics

To define the syntax of Core HJp, we first fix sets of class names and fields, written C and f, respectively, as well as countably infinite sets of term variables, written x, y, and z, and word variables, written ξ. We also syntactically distinguish the normal fields f^{n} from the fields f^{X} marked as exclusive. To denote sequences of zero or more syntactic elements, we use an arrow over a syntactic category. For example, $\vec{x}$ indicates a sequence of zero or more variables, referred to as x_1 through x_n. Further, we often use this notation as a shorthand in compound notation; e.g., $\vec{\mathrm{f}^{\mathrm{n}}} : \vec{\tau}$ denotes a sequence $\mathrm{f}^{\mathrm{n}}_1 : \tau_1, \ldots, \mathrm{f}^{\mathrm{n}}_n : \tau_n$, while $x : \vec{\phi}$ denotes a sequence of permissions $x : \phi_1, \ldots, x : \phi_n$.

The syntax of Core HJp is given in Figure 3. This includes the permissions and permission sets as defined in Section 2 along with the permission subtractions $-\mathrm{f}^{\mathrm{X}}@y$ defined in Section 3 and the permission word variables ξ and dynamic permissions $\mathrm{dyn}_x(\vec{\phi})$ described in Section 4. The Core HJp types τ include the class names C as well as the types $(\Gamma \setminus \Pi_{\mathrm{i}}) \to (y : \tau \setminus \Pi_{\mathrm{o}} \setminus \Pi_{\mathrm{e}})$ of functions that take, as input, values and permission words whose types are specified by Γ. In practice, Γ always has the form $x : \tau_x, \vec{\xi}$, for functions that take one input but quantify over zero or more permission words. The output type is given by τ. The permission set Π_{i} specifies the minimal (under subsumption) permissions that must be held to call the function, while Π_{o} and Π_{e} specify the permissions returned under normal and exceptional exit from the function. The variables x and $\vec{\xi}$ are considered bound in the remaining constructs, while the variable y, which refers to the value of the output, is considered bound in Π_{o} and Π_{e}. The latter allows Π_{o} and Π_{e} to refer to the output of the function.

Next in Figure 3 are the signatures, written Σ, and the type contexts, written Γ or Δ. A signature associates each class name C in a program with a list of fields and their associated types. We assume a fixed signature for the remainder of this section, which we write as Σ below. A typing context Γ associates types with term variables x, and lists word variables ξ that are considered to be in scope; any variables listed in context Γ are considered bound in later types. We implicitly assume that all signatures and typing contexts are well-formed,

$$
\begin{array}{ll}
w ::= 0w \mid 1w \mid \varepsilon \mid \xi & \Upsilon ::= \mathrm{R} \mid \mathrm{W} \mid \mathrm{S} \\
\phi ::= w\Upsilon \mid \phi - \mathrm{f}^{\mathrm{X}}@y & \pi ::= x : \phi \mid \mathrm{F}(\pi) \mid \mathrm{dyn}_x(\phi) \\
\Pi ::= \Pi, \pi \mid \cdot & \tau ::= C \mid (\Gamma \setminus \Pi_{\mathrm{i}}) \to (y : \tau' \setminus \Pi_{\mathrm{o}} \setminus \Pi_{\mathrm{e}}) \\
\Sigma ::= \cdot \mid \Sigma, C \mapsto \{\vec{\mathrm{f}}^{\mathrm{n}} : \vec{\tau}, \vec{\mathrm{f}}^{\mathrm{X}} : \vec{C}\} & \Gamma ::= \cdot \mid \Gamma, x : \tau \mid \Gamma, \xi
\end{array}
$$

$$
\begin{array}{l}
M ::= x \mid \mathbf{let}\ x = M\ \mathbf{in}\ M \mid x\ (y, \vec{\xi}) \mid \lambda(x, \vec{\xi}).M \mid \mathbf{exn} \mid \mathbf{null} \mid x.\mathrm{f}^{\mathrm{n}} \mid x.\mathrm{f}^{\mathrm{n}} := y \\
\quad \mid x.\mathrm{f}^{\mathrm{X}}(\vec{\phi}) \mid x.\mathrm{f}^{\mathrm{X}} := y \mid \mathbf{remit}_x(\phi \to y.\mathrm{f}^{\mathrm{X}}) \mid \mathbf{new}\ C\ \langle \vec{\mathrm{f}} \mapsto \vec{x} \rangle \mid \mathbf{acquire}_{\mathrm{X}}(x) \\
\quad \mid \mathbf{let}\ \xi = \mathbf{acquire}_{\vec{\Upsilon}}(x)\ \mathbf{in}\ M \mid \mathbf{release}_{\vec{\phi}}(x) \mid \mathbf{async}_{\Pi}\ M \mid \mathbf{finish}\ M
\end{array}
$$

Fig. 3. Syntax

meaning they contain only bound occurrences of variables x and ξ. In the below, we use the notations $\Gamma(x)$, $\Sigma(C)$, and $\Sigma(C)(\mathrm{f})$ to denote, respectively, the type associated with x in Γ, the sequence of field-type pairs associated with C in Σ, and the type in this sequence associated with f.

The terms of Core HJp are in A-normal form, in order to allow permission sets to refer to each intermediate value as a variable. Thus field reads and writes, acquires, and releases all refer only to variables as their arguments. Compound expressions can be built using **let**-expressions. Functions have the form $\lambda(x, \vec{\xi}).M$, meaning that they quantify over a value x and zero or more permission words $\vec{\xi}$. Similarly, function applications have the form $x\ (y, \vec{w})$, applying variable x to argument y and permission words $\vec{w}$.

The typing judgment $\Gamma \setminus \Pi_{\mathrm{i}} \vdash M : y : \tau \setminus \Pi_{\mathrm{o}} \setminus \Pi_{\mathrm{e}}$ for Core HJp is defined in Figure 4. It states that M is well-typed under typing context Γ and assuming input permissions Π_{i}, and M returns a value of type τ and permissions given either by Π_{o} or Π_{e}, depending on whether it had a normal or exceptional exit, respectively. We sometimes omit the exceptional permission set Π_{e} when it is allowed to have any value, which implies that the given expression cannot throw an exception. Note that we implicitly assume, for each rule, that the type τ and the permission sets Π_{o} and Π_{e} are well-formed, meaning they contain only variables listed in Γ and, in the case of Π_{o} and Π_{e}, possibly y.

The first rule, T-Sub, allows permissions to be added to the input permissions Π_{i} or to be removed from the output permissions Π_{o} and Π_{e}. It also allows the same permissions to be added to both sides, in a manner similar to the frame rule of separation logic. T-Var types variables x, allowing the output permissions to refer to the output itself as a variable, y, instead of to the variable x; output permissions that still refer to x can be added with T-Sub. Since variables cannot throw exceptions, any Π_{e} may be used. T-Let types **let** $x = M_1$ **in** M_2 by binding x in the type context for M_2, and also states that the output permissions for M_1 must serve as input permissions to M_2. Both must have the same exception permission Π_{e}, since an exception from the **let**-expression could be thrown from either M_1 or M_2.

T-App types applications $x\ (y, \vec{w})$, turning the function type for x into a typing assertion by substituting the arguments y and $\vec{w}$ into the type for x. Functions $\lambda(x, \vec{\xi}).M$ do the opposite. Note that functions do not consume or produce any permissions, since their bodies do not run until they are called.

$$\frac{\Gamma \setminus \Pi_\mathrm{i} \vdash M : y : \tau \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{e} \qquad \Pi_\mathrm{o} \geq \Pi'_\mathrm{o} \qquad \begin{matrix}\Pi'_\mathrm{j} \geq \Pi_\mathrm{i} \\ \Pi_\mathrm{e} \geq \Pi'_\mathrm{e}\end{matrix}}{\Gamma \setminus \Pi'_\mathrm{i}, \Pi' \vdash M : y : \tau \setminus \Pi'_\mathrm{o}, \Pi' \setminus \Pi'_\mathrm{e}, \Pi'} \text{ T-Sub} \qquad \frac{x : \tau \in \Gamma}{\Gamma \setminus \Pi \vdash x : y : \tau \setminus [y/x]\Pi} \text{ T-Var}$$

$$\frac{\Gamma \setminus \Pi_\mathrm{i} \vdash M_1 : x : \tau' \setminus \Pi' \setminus \Pi_\mathrm{e} \quad \Gamma, x : \tau' \setminus \Pi' \vdash M_2 : y : \tau \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{e}}{\Gamma \setminus \Pi_\mathrm{i} \vdash \mathbf{let}\ x = M_1\ \mathbf{in}\ M_2 : y : \tau \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{e}} \text{ T-Let}$$

$$\frac{x : ((y' : \tau, \vec{\xi}) \setminus \Pi_\mathrm{i}) \to (z : \tau' \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{e}) \in \Gamma \quad \sigma = [y/y', \vec{w}/\vec{\xi}]}{\Gamma \setminus \sigma\Pi_\mathrm{i} \vdash x\ (y, \vec{w}) : z : \tau' \setminus \sigma\Pi_\mathrm{o} \setminus \sigma\Pi_\mathrm{e}} \text{ T-App}$$

$$\frac{\Gamma, x : \tau, \vec{\xi} \setminus \Pi_\mathrm{i} \vdash M : y : \tau' \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{e}}{\Gamma \setminus \cdot \vdash \lambda(x, \vec{\xi}).M : (x : \tau, \vec{\xi} \setminus \Pi_\mathrm{i}) \to (y : \tau' \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{e}) \setminus \cdot} \text{ T-Lam} \qquad \frac{}{\Gamma \setminus \cdot \vdash \mathbf{exn} : y : \tau \setminus \Pi_\mathrm{o} \setminus \cdot} \text{ T-Exn}$$

$$\frac{}{\Gamma \setminus \cdot \vdash \mathbf{null} : y : C \setminus y : \vec{\phi}} \text{ T-Null} \qquad \frac{x : C \in \Gamma \quad \mathbf{root}(\phi) = w\mathrm{R}}{\Gamma \setminus x : \phi \vdash x.\mathrm{f^n} : \Sigma(C)(\mathrm{f^n}) \setminus x : \phi \setminus x : \phi} \text{ T-Read}$$

$$\frac{x : C, y : \Sigma(C)(\mathrm{f^n}) \in \Gamma \quad \mathbf{root}(\vec{\phi}) = w\mathrm{R}, w\mathrm{W}}{\Gamma \setminus x : \vec{\phi} \vdash (x.\mathrm{f^n} := y) : \mathbf{U} \setminus x : \vec{\phi} \setminus x : \vec{\phi}} \text{ T-Write}$$

$$\frac{\vec{\phi} = w\vec{\Upsilon} - \vec{B} \quad \mathrm{R} \in \vec{\Upsilon} \quad x : C \in \Gamma \quad \nexists z.\mathrm{f^X}@z \in \vec{B}}{\Gamma \setminus x : \vec{\phi} \vdash x.\mathrm{f^X}(w\vec{\Upsilon}) : y : \Sigma(C)(\mathrm{f^X}) \setminus x : \vec{\phi} - \mathrm{f^X}@y, y : w\vec{\Upsilon} \setminus x : \vec{\phi}} \text{ T-XRead}$$

$$\frac{x : C \in \Gamma \quad y : \Sigma(C)(\mathrm{f^X}) \in \Gamma \quad \mathbf{root}(\vec{\phi}) = w\mathrm{R}, w\mathrm{W}}{\Gamma \setminus x : \vec{\phi}, y : \mathrm{X} \vdash x.\mathrm{f^X} := y : \mathbf{U} \setminus x : \vec{\phi} + \mathrm{f^X} \setminus x : \vec{\phi}, y : \mathrm{X}} \text{ T-XWrite}$$

$$\frac{x : C \in \Gamma \quad y : C' \in \Gamma \quad \mathbf{root}(\phi) = w\Upsilon}{\Gamma \setminus x : \phi, y : w\Upsilon \vdash \mathbf{remit}_y(w\Upsilon \to x.\mathrm{f^X}) : \cdot \setminus x : \phi + \mathrm{f^X}@y} \text{ T-Remit}$$

$$\frac{\Sigma(C) = \{\mathrm{f^{\vec{n}}} : \Gamma(\vec{x}), \mathrm{f^{\vec{X}}} : \Gamma(\vec{y})\}}{\Gamma \setminus \vec{y} : \mathrm{X} \vdash \mathbf{new}\ C\ \langle \mathrm{f^{\vec{n}}} \mapsto \vec{x}, \mathrm{f^{\vec{X}}} \mapsto \vec{y} \rangle : x : C \setminus x : \mathrm{X}} \text{ T-New}$$

$$\frac{x : C \in \Gamma \quad \Gamma, \xi \setminus \Pi_\mathrm{i}, x : \xi\vec{\Upsilon}, \mathrm{dyn}_x(\xi\vec{\Upsilon}) \vdash M : y : \tau \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{e} \quad \Pi_\mathrm{i} \geq \Pi_\mathrm{e}}{\Gamma \setminus \Pi_\mathrm{i} \vdash \mathbf{let}\ \xi = \mathbf{acquire}_{\vec{\Upsilon}}(x)\ \mathbf{in}\ M : y : \tau \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{e}} \text{ T-Acq}$$

$$\frac{x : C \in \Gamma}{\Gamma \setminus \cdot \vdash \mathbf{acquire}_\mathrm{X}(x) : \mathbf{U} \setminus x : \mathrm{X} \setminus \cdot} \text{ T-AcqX} \qquad \frac{x : C \in \Gamma \quad \forall i.\phi_i = w_i\Upsilon_i}{\Gamma \setminus x : \vec{\phi}, \mathrm{dyn}_x(\vec{\phi}) \vdash \mathbf{release}_{\vec{\phi}}(x) : \mathbf{U} \setminus \cdot} \text{ T-Rel}$$

$$\frac{x : C \in \Gamma}{\Gamma \setminus x : \mathrm{X} \vdash \mathbf{release}_\mathrm{X}(x) : \mathbf{U} \setminus \cdot} \text{ T-RelX} \qquad \frac{\Gamma \setminus \Pi_\mathrm{i} \vdash M : \tau \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{o} \quad \mathbf{sharable}(\Pi_\mathrm{i})}{\Gamma \setminus \Pi_\mathrm{i} \vdash \mathbf{async}_{\Pi_\mathrm{i}}\ M : \mathbf{U} \setminus \mathrm{F}(\Pi_\mathrm{o})} \text{ T-Async}$$

$$\frac{\Gamma \setminus \Pi_\mathrm{i} \vdash M : y : \tau \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{e} \quad \nexists \mathrm{F}(\pi) \in \Pi_\mathrm{i}}{\Gamma \setminus \Pi_\mathrm{i} \vdash \mathbf{finish}\ M : y : \tau \setminus \Pi_\mathrm{o} - \mathbf{F} \setminus \Pi_\mathrm{e} - \mathbf{F}} \text{ T-Finish}$$

Fig. 4. Static Semantics

Exceptions **exn** can have any type and output permissions, as they will not return, but they have the same exception permission set as input permission set. T-Exn gives these both as empty permission sets $\cdot$, but permissions can be added to both sides using T-Sub. Similarly, **null** can have any type and any output permissions to the **null** value, since, intuitively, permissions to **null** do not matter; this is made more formal in the operational semantics, below.

Normal field reads and writes require read and write permission, respectively, for the object being accessed. Writes return the sole element $*$ of the singleton

type **U**. Exclusive field reads $x.\mathrm{f}^{\mathrm{X}}(w\vec{\Upsilon})$ require at least read permission to x but without the borrowing permission subtraction $-\mathrm{f}^{\mathrm{X}}@y$, and append this subtraction to the permissions on x. Note that we use $x : w\vec{\Upsilon}$ to denote the permission set $x : w\Upsilon_1, \ldots, x : w\Upsilon_n$. Exclusive writes $x.\mathrm{f}^{\mathrm{X}} := y$ require exclusive permission $y : \mathrm{X}$ (recall that this is shorthand for $y : \varepsilon\mathrm{R}, y : \varepsilon\mathrm{W}$) and erase any subtractions $\mathrm{f}^{\mathrm{X}}@z$ on permissions for x, where $x : \vec{\phi} + \mathrm{f}^{\mathrm{X}}$ indicates the removal of any $\mathrm{f}^{\mathrm{X}}@z$ in each ϕ_i. These ϕ_i must comprise at least a write permission to x. Note that all of these return the input permission set as the exceptional permission set, in the case the the object being read or written is **null**.

The $\mathbf{remit}_y(w\Upsilon \to x.\mathrm{f}^{\mathrm{X}})$ construct erases a subtraction $-\mathrm{f}^{\mathrm{X}}@y$, using the notation $\phi + \mathrm{f}^{\mathrm{X}}@y$, as well as erasing a $w\Upsilon$ permission to y. Object allocation requires X permissions to all objects assigned to exclusive fields and returns an X permission to the newly allocated object.

An exclusive acquire returns an X permission to x as output permission but returns an empty exception permission. Again, using T-Sub allows the input permission set to be anything and requires the exception permission set to be the same (in case the acquire throws an exception) but adds $x : \mathrm{X}$ to the output permission set. A non-exclusive acquire $\mathbf{let}\ \xi = \mathbf{acquire}_{\vec{\Upsilon}}(x)\ \mathbf{in}\ M$ provides permissions $\xi\vec{\Upsilon}$ to x in the body M, also indicating that these permissions are dynamic. A $\mathbf{release}_{\vec{\phi}}(x)$ releases the permissions $\vec{\phi}$ for x, requiring that either $\vec{\phi}$ equals exclusive permissions or that there is a dynamic permission $\mathrm{dyn}_x(\vec{\phi})$ to indicate that $\vec{\phi}$ can be dynamically released. An $\mathbf{async}\ \Pi_{\mathrm{i}} M$ passes to M the permissions Π_{i}, which must satisfy $\mathbf{sharable}(\Pi_{\mathrm{i}})$. The latter indicates that Π_{i} contains only pairs $x : w\mathrm{R}, x : w\mathrm{S}$ and permissions $\mathrm{dyn}_x(\vec{\phi})$. Any permissions Π_{o} returned by M are returned as future permissions to the current task, where $\mathrm{F}(\Pi)$ maps each π in Π to $\mathrm{F}(\pi)$. Note that an **async** catches any exceptions but does not throw any, so it can have any exceptional permission set, while it requires the output and exception permissions of M to be the same. The latter can always be achieved by T-Sub. Future permissions can be reclaimed with $\mathbf{finish}\ M$, which returns the same permissions as M with any future permissions $\mathrm{F}(\pi)$ turned into π. This is written as $\Pi - \mathbf{F}$, which also converts $\mathrm{F}(\mathrm{F}(\pi))$, etc., to π, to handle nested **async**s. Note that no future permissions from outside the **finish** are passed to M, as these are only available to the next enclosing **finish**.

5.2 Operational Semantics

The additional syntax needed to define the operational semantics is given in Figure 5. Two important changes are that values are now allowed to occur in place of variables in both permissions and terms, and that private read permissions are now annotated with task ids t, where we assume a countably infinite set of task ids. The latter change was unnecessary in the static semantics, since an expression can only refer to permissions for the current task, but will be necessary to model the permissions held globally for an object. There are now distinct private read as well as private read-write permission sets for each task id t. We write Π^t for the result of adding task id t to the (un-annotated) private read

$$
\begin{array}{lll}
\Upsilon & ::= \mathrm{R}^t \mid \mathrm{W} \mid \mathrm{S} & M ::= \ldots \mid l^C \mid [\vec{v}/\vec{x}]M \mid \mathbf{finish}(M \parallel \vec{T}) \\
\pi & ::= \ldots \mid l : \phi \mid \mathrm{dyn}_l(\phi) & H ::= \cdot \mid H, l \mapsto \langle \mathbf{p} \mapsto \Pi, \vec{f} \mapsto \vec{v} \rangle \\
T & ::= (\Pi \setminus M)^t & E^* ::= \Box \mid \mathbf{let}\ x = E^*\ \mathbf{in}\ M \\
v & ::= \mathbf{null} \mid l^C \mid \lambda(x, \vec{\xi}).M & E ::= \Box \mid E^*[E] \mid \mathbf{finish}(E \parallel \vec{T}) \\
res & ::= v \mid \mathbf{exn} &
\end{array}
$$

Fig. 5. Operational Syntax

permissions in Π. These two changes yield the additional equalities:

$$x : w\mathrm{R}^{t_1}, x : w\mathrm{S} = x : w\mathrm{R}^{t_2}, x : w\mathrm{S} \qquad \mathbf{null} : \phi = \cdot$$

The first captures the fact that shared read and exclusive permissions are insensitive to which task holds them, while the second captures the fact that permissions to **null** can be added or removed at will.

The values v include functions, **null**, and heap locations l, where the latter is annotated with its class as l^C. The new term construct $\mathbf{finish}(M \parallel \vec{T})$, which is considered equal modulo permutation of $\vec{T}$, represents a **finish** waiting for parallel tasks $\vec{T}$ to complete. Tasks are written $(\Pi \setminus M)^t$, indicating a task that is executing term M, holds permissions Π, and has task id t. Typing is extended to tasks with the judgement $\vdash T : \Pi_\mathrm{o}$, which holds for task $(\Pi \setminus M)^t$ if and only if $\cdot \setminus \Pi \vdash M : \mathbf{U} \setminus \Pi_\mathrm{o} \setminus \Pi_\mathrm{o}$. Typing is then extended to $\mathbf{finish}(M \parallel \vec{T})$ by requiring $\vdash T_i : \Pi_i$ for each T_i, and adding $\vec{\Pi}$ to the output and exceptional permission sets returned by the construct.

The results res include exceptions and values, while the heaps H map locations l to heap forms $\langle \mathbf{p} \mapsto \Pi, \vec{f} \mapsto \vec{v} \rangle$. The latter are themselves mappings from fields to values and from the special marker $\mathbf{p}$ to the set of permissions Π of permissions to l still available for dynamic acquires, where Π must contain only permissions $l : \phi$. (We use permission sets here to take advantage of permission set equality.) Thus, e.g., $H(l)(f)$ returns the value of field f at location l in H, while $H(l)(\mathbf{p})$ returns permissions to l that are available for dynamic acquires. As a convenience, each heap H also contains a mapping $H(\mathbf{null})(\mathbf{p}) = \cdot$ which (by the above) equals $\mathbf{null} : \vec{\phi}$ for any $\vec{\phi}$.

Finally, Figure 5 defines the evaluation contexts E, which intuitively define a term with single a "hole" $\Box$ indicating where evaluation can take place in a term. This includes the term being bound in a **let** and the body of a **finish**. We write $E[M]$ for the (non-capture-avoiding) replacement of $\Box$ by M. We also define evaluation contexts E^* out of which exceptions can be thrown. These exclude **finish**, requiring child tasks to complete before an exception leaves a **finish**.

The operational semantics is defined in Figure 6 as a small-step relation $H \setminus \Pi \setminus M \xrightarrow{a} H' \setminus \Pi' \setminus M'$. This judgment states that term M with heap H, in a task holding permissions Π, evaluates in one step to term M', changing the heap to H' and the permissions held by the current task to Π'. This step is also labeled with an action label a, which can have any of the forms given in Figure 6. Action labels can optionally have the prefix $(t; \Pi) : a$, indicating that the action occurred in task id t and yielded permission set Π in task t. Only the inner-most prefix is used, so $(t_1; \Pi_1) : (t_2; \Pi_2) : a$ is considered equal to $(t_2; \Pi_2) : a$. These labels are used below to define the happens-before ordering and data races.

$$\frac{H \setminus \Pi \setminus M \xrightarrow{a} H' \setminus \Pi' \setminus M'}{H \setminus (\Pi \setminus M)^t \xrightarrow{(t;\Pi'):a} H' \setminus (\Pi' \setminus M')^t}\ \text{E-Task} \qquad \frac{H \setminus \Pi \setminus M \xrightarrow{a} H' \setminus \Pi' \setminus M'}{H \setminus \Pi \setminus E[M] \xrightarrow{a} H' \setminus \Pi' \setminus E[M']}\ \text{E-Ctx}$$

$$\frac{}{H \setminus \Pi \setminus E^*[\mathbf{exn}] \xrightarrow{\cdot} H \setminus \Pi \setminus \mathbf{exn}}\ \text{E-Exn} \qquad \frac{}{H \setminus \Pi \setminus \mathbf{let}\ x = v\ \mathbf{in}\ M \xrightarrow{\cdot} H \setminus \Pi \setminus [v/x]M}\ \text{E-Let}$$

$$\frac{}{H \setminus \Pi \setminus (\lambda(x, \vec{\xi}).M)\ (v, \vec{w}) \xrightarrow{\cdot} H \setminus \Pi \setminus [v/x, \vec{w}/\vec{\xi}]M}\ \text{E-App}$$

$$\frac{}{H \setminus \Pi \setminus l.\mathrm{f}^{\mathrm{n}} \xrightarrow{l.\mathrm{f}^{\mathrm{n}} \rightharpoonup H(l)(\mathrm{f}^{\mathrm{n}})} H \setminus \Pi \setminus H(l)(\mathrm{f}^{\mathrm{n}})}\ \text{E-Read} \qquad \frac{}{H \setminus \Pi \setminus \mathbf{null}.f \xrightarrow{\cdot} H \setminus \Pi \setminus \mathbf{exn}}\ \text{E-NullR}$$

$$\frac{}{H \setminus \Pi \setminus l.\mathrm{f}^{\mathrm{n}} := v \xrightarrow{l.\mathrm{f}^{\mathrm{n}} \leftharpoonup v} H[(l, \mathrm{f}^{\mathrm{n}}) \mapsto v] \setminus \Pi \setminus *}\ \text{E-Write} \qquad \frac{}{H \setminus \Pi \setminus \mathbf{null}.f := v \xrightarrow{\cdot} H \setminus \Pi \setminus \mathbf{exn}}\ \text{E-NullW}$$

$$\frac{\vec{\phi} = w\vec{\Upsilon} - \vec{B} \quad l' = H(l)(\mathrm{f}^{\mathrm{X}})}{H \setminus \Pi, l : \vec{\phi} \setminus l.\mathrm{f}^{\mathrm{X}}(w\vec{\Upsilon}) \xrightarrow{l.\mathrm{f}^{\mathrm{X}} \rightharpoonup l'} H \setminus \Pi, l : \vec{\phi} - \mathrm{f}^{\mathrm{X}}@l', l' : w\vec{\Upsilon} \setminus l'}\ \text{E-XRead}$$

$$\frac{\nexists (l : \phi) \in \Pi}{H \setminus \Pi, l : \vec{\phi}, l' : \mathrm{X} \setminus l.\mathrm{f}^{\mathrm{X}} := l' \xrightarrow{l.\mathrm{f}^{\mathrm{X}} \leftharpoonup l'} H[(l, \mathrm{f}^{\mathrm{X}}) \mapsto l'] \setminus \Pi, l : \vec{\phi} + \mathrm{f}^{\mathrm{X}} \setminus *}\ \text{E-XWrite}$$

$$\frac{\phi = w\Upsilon - \vec{B}}{H \setminus \Pi, v : \phi, v' : w\Upsilon \setminus \mathbf{remit}_{v'}(w\Upsilon \to v.\mathrm{f}^{\mathrm{X}}) \xrightarrow{\cdot} H \setminus \Pi, v : \phi + \mathrm{f}^{\mathrm{X}}@v' \setminus \cdot}\ \text{E-Remit}$$

$$\frac{l \text{ fresh}}{H \setminus \Pi, \vec{v} : \mathrm{X} \setminus \mathbf{new}\ C\ \langle \mathrm{f}^{\vec{\mathrm{X}}} \mapsto \vec{v}, \mathrm{f}^{\vec{\mathrm{n}}} \mapsto \vec{v'} \rangle \xrightarrow{\mathbf{new}(l; \vec{f} \mapsto \vec{v})} H, l \mapsto \langle \mathbf{p} \mapsto \cdot, \mathrm{f}^{\vec{\mathrm{X}}} \mapsto \vec{v}, \mathrm{f}^{\vec{\mathrm{n}}} \mapsto \vec{v'} \rangle \setminus \Pi, l : \mathrm{X} \setminus l}\ \text{E-New}$$

$$\frac{H(l)(\mathbf{p}) \neq l : \mathrm{X}}{H \setminus \Pi \setminus \mathbf{acquire}_{\mathrm{X}}(l) \xrightarrow{\cdot} H \setminus \Pi \setminus \mathbf{exn}}\ \text{E-AcqXFail}$$

$$\frac{\nexists (\Pi_l, w).\Pi, H(l)(\mathbf{p}) = \Pi, \Pi_l, l : w\vec{\Upsilon}}{H \setminus \Pi \setminus \mathbf{let}\ \xi = \mathbf{acquire}_{\vec{\Upsilon}}(l)\ \mathbf{in}\ M \xrightarrow{\cdot} H \setminus \Pi \setminus \mathbf{exn}}\ \text{E-AcqFail}$$

$$\frac{H(v)(\mathbf{p}) = v : \mathrm{X}}{H \setminus \Pi \setminus \mathbf{acquire}_{\mathrm{X}}(v) \xrightarrow{\mathbf{acq}(v:\mathrm{X})} H[(v, \mathbf{p}) \mapsto \cdot] \setminus \Pi, v : \mathrm{X} \setminus *}\ \text{E-AcqX}$$

$$\frac{\Pi, H(v)(\mathbf{p}) = \Pi, \Pi_v, v : w\vec{\Upsilon} \quad \exists w.\Pi_v \geq w\mathrm{R}^t}{H \setminus \Pi \setminus \mathbf{let}\ \xi = \mathbf{acquire}_{\vec{\Upsilon}}(v)\ \mathbf{in}\ M \xrightarrow{\mathbf{acq}(v:w\vec{\Upsilon})} H[(v, \mathbf{p}) \mapsto \Pi_v] \setminus \Pi, v : w\vec{\Upsilon}, \mathrm{dyn}_v(w\vec{\Upsilon}) \setminus [w/\xi]M}\ \text{E-Acq}$$

$$\frac{\text{if } \vec{\phi} = \varepsilon\mathrm{R}^t, \varepsilon\mathrm{W} \text{ then } \Pi_1 = \cdot \text{ else } \Pi_1 = \mathrm{dyn}_v(\vec{\phi})}{H \setminus \Pi, v : \vec{\phi}, \Pi_1 \setminus \mathbf{release}_{\vec{\phi}}(v) \xrightarrow{\mathbf{rel}(v:\vec{\phi})} H[(v, \mathbf{p}) \mapsto H(v)(\mathbf{p}), v : \vec{\phi}] \setminus \Pi \setminus *}\ \text{E-Rel}$$

$$\frac{t \text{ fresh}}{H \setminus \Pi, \Pi' \setminus \mathbf{finish}(E^*[\mathbf{async}_{\Pi}\ M] \parallel \vec{T}) \xrightarrow{\mathbf{async}(t)} H \setminus \Pi' \setminus \mathbf{finish}(E^*[*] \parallel \vec{T}, (\Pi \setminus M)^t)}\ \text{E-Async}$$

$$\frac{t' \text{ fresh}}{H \setminus \Pi'' \setminus \mathbf{finish}(M'' \parallel \vec{T}, (\Pi, \Pi' \setminus E^*[\mathbf{async}_{\Pi}\ M])^t) \xrightarrow{(t;\Pi'):\mathbf{async}(t')} H \setminus \Pi'' \setminus \mathbf{finish}(M'' \parallel \vec{T}, (\Pi' \setminus E^*[*])^t, (\Pi \setminus M)^{t'})}\ \text{E-AsyncPar}$$

$$\frac{H \setminus T \xrightarrow{a} H' \setminus T'}{H \setminus \Pi \setminus \mathbf{finish}(M \parallel T, \vec{T}) \xrightarrow{a} H' \setminus \Pi \setminus \mathbf{finish}(M \parallel T', \vec{T})}\ \text{E-Par}$$

$$\frac{}{H \setminus \Pi \setminus \mathbf{finish}(res \parallel (\vec{\Pi} \setminus r\vec{e}s)^{\vec{t}}) \xrightarrow{\mathbf{finish}(\vec{t})} H \setminus \Pi, \vec{\Pi} \setminus res}\ \text{E-Finish}$$

Fig. 6. Operational Semantics

The E-Task rule defines evaluation on tasks, which evaluates the term and permission set in the task and then adds the task id and resulting permission set to a. E-Ctx allows evaluation inside evaluation contexts E, while E-Exn allows exceptions to be thrown out of exception evaluation contexts E^*. E-Let and E-App evaluate **let**-expressions and function applications using substitution.

Normal field reads of $l.\mathrm{f}^{\mathrm{n}}$ return the value $H(l)(\mathrm{f}^{\mathrm{n}})$ associated with f^{n} for l in H. The action label for a field read is written $l.\mathrm{f}^{\mathrm{n}} \rightharpoonup H(l)(\mathrm{f}^{\mathrm{n}})$. Normal field writes $l.\mathrm{f}^{\mathrm{n}} := v$ update $H(l)(\mathrm{f}^{\mathrm{n}})$ to point to v, written $H[(l, \mathrm{f}^{\mathrm{n}}) \mapsto v]$. The action label for a field write is $l.\mathrm{f}^{\mathrm{n}} \leftharpoonup v$. Reads from or writes to **null** raise an exception. Exclusive field reads and writes are similar, except that the permissions held by the current task are updated to indicate a borrow and to remove any pending borrows, respectively, as described in the typing rules above. Remits also remove a pending borrow as described above, and have an empty action label, while **new** consumes exclusive permissions to the locations assigned to the exclusive fields of the new object, as discussed above, and is labeled with $\mathbf{new}(l; \mathrm{f}^{\vec{\mathrm{X}}} \mapsto \vec{v}, \mathrm{f}^{\vec{\mathrm{n}}} \mapsto \vec{v'})$, where the v_i and v'_i are the values assigned to $\mathrm{f}_i^{\mathrm{X}}$ and $\mathrm{f}_i^{\mathrm{n}}$, respectively.

Each form of acquire has two rules, one for failure and one for success. An exclusive acquire on l succeeds if $H(l)(\mathbf{p}) \neq l : \mathrm{X}$, meaning that l has an exclusive permission available for dynamic acquisition. Similarly, a non-exclusive acquire of $\vec{\Upsilon}$ on l succeeds if permission set $\Pi, H(l)(\mathbf{p})$ can be separated into $\Pi, l : w\vec{\Upsilon}, \Pi_v$, which combines the permissions Π held by the current task, the requested permissions, and some leftover permissions Π_v remaining in $H(l)(\mathbf{p})$. The current permission set Π is included here to allow read-write permissions to be acquired when the current task holds all the private read permissions to an object, while the side condition on Π_v ensures that not all of the remaining permissions are acquired. If these conditions cannot be satisfied, then either acquire throws an exception. Successful acquires of $\vec{\phi}$ for l are labeled with $\mathbf{acq}(l : \vec{\phi})$. Note that acquires on **null** automatically succeed, since $H(\mathbf{null})(\mathbf{p})$ equals any permission set that satisfies the above.

A release of $l : \vec{\phi}$ gives these permissions back to $H(l)(\mathbf{p})$, and is labeled with $\mathbf{rel}(l : \vec{\phi})$. A release on **null** effectively does nothing. An **async** creates a new task with some id t' inside the most closely containing **finish**, and is labeled with $\mathbf{async}(t')$. Rules E-Async and E-AsyncPar handle an **async** in the main body or in a parallel task, respectively, of a **finish**. Parallelism is modeled by allowing steps in the parallel tasks of a **finish**, as captured by E-Par. Finally, a **finish** completes when the main body and all parallel tasks are results, and all permissions returned by the parallel tasks are collected in the parent task. This is labeled with $\mathbf{finish}(\vec{t})$ where $\vec{t}$ are the ids of the completed tasks.

At the top level, small-step evaluation is applied to *machine states* $H \setminus T$, where T is called the top-level task. If $T = (\Pi \setminus res)^t$, then $H \setminus T$ is called a *final state*. We write *Mach* for machine states, and define the set $\mathbf{perms}_l(\mathit{Mach})$ of permissions for l held by *Mach* as the combination (using ",") of

$$\{\, \Pi|_l \mid \exists (M, t).(\Pi \setminus M)^t \sqsubseteq \mathit{Mach} \,\} \cup H(l)(\mathbf{p}) \cup$$
$$\{\, l : w\Upsilon \mid l' : w\Upsilon - \vec{B} \in \mathbf{perms}_{l'}(\mathit{Mach}) \wedge H(l')(\mathrm{f}^{\mathrm{X}}) = l \wedge \nexists (i, z).B_i = \mathrm{f}^{\mathrm{X}}@z \,\}$$

where $\sqsubseteq$ is the subterm relation. This set is well-defined iff reachability under f^{X} fields is acyclic, which is ensured by machine well-formedness for $\mathit{Mach} = H \setminus T$:

1. $\vdash T : \Pi_{\mathrm{o}}$ for some Π_{o} and $\cdot \vdash H(l^C)(f) : \Sigma(C)(f)$ for all $l \in \mathrm{Dom}(H)$;
2. $l : \mathrm{X} \geq \mathbf{perms}_l(\mathit{Mach})$;
3. Any **async** occurring in Mach is a subterm of a **finish**.

This judgment is written $\vdash \mathit{Mach}$. The first condition ensures that the top-level task and all field values are well-typed. The second ensures that the total permissions in the program to any l are at most X; i.e., there are no duplicated permissions. The final condition ensures that tasks are always spawned inside of a **finish** scope; note that the HJ runtime does this implicitly. Using this definition, we can prove Type Soundness using Preservation and Progress:

Lemma 1 (Preservation). *If* $\vdash \mathit{Mach}$ *and* $\mathit{Mach} \to \mathit{Mach}'$ *then* $\vdash \mathit{Mach}'$.

Lemma 2 (Progress). *If* $\vdash \mathit{Mach}$ *then either Mach is a final state or* $\mathit{Mach} \to \mathit{Mach}'$ *for some* Mach'.

We write $s : \mathit{Mach}_1 \xrightarrow{a} \mathit{Mach}_2$ to denote that s is a step, or derivation of $\mathit{Mach}_1 \xrightarrow{a} \mathit{Mach}_2$. A collection of such steps $\mathit{Mach}_1 \to \ldots \to \mathit{Mach}_n$ is called an *execution*; we write $\mathcal{E} : \mathit{Mach}_1 \dot{\to}^* \mathit{Mach}_n$ to denote that $\mathcal{E}$ is such an execution. We also write $\leq_{\mathcal{E}}$ for the sequence order of the steps in $\mathcal{E}$. In order to prove that any execution has no data races, we shall prove that any steps which conflict must be ordered by the happens-before order. We define these concepts below. To define this notion, we first define when permissions conflict, which intuitively means that they cannot be held at the same time.

Definition 1 (Conflicting Permissions). *We say that* Π_1 *and* Π_2 conflict, *written* $\Pi \bowtie \Pi'$, *iff* $l : \mathrm{X} \geq \Pi_1|_l, \Pi_2|_l$ *does* not *hold for some* l. *We say that steps* $s_1 : \mathit{Mach}_1 \xrightarrow{(t_1;\Pi_1):a_1} \mathit{Mach}_1'$ *and* $s_2 : \mathit{Mach}_2 \xrightarrow{(t_2;\Pi_2):a_2} \mathit{Mach}_2'$ *conflict, written* $s_1 \bowtie s_2$, *iff* $\Pi_1 \bowtie \Pi_2$.

Definition 2 (Happens-Before). *Step* $s_1 : \mathit{Mach}_1 \xrightarrow{(t_1;\Pi_1):a_1} \mathit{Mach}_1'$ happens-before *step* $s_2 : \mathit{Mach}_2 \xrightarrow{(t_2;\Pi_2):a_2} \mathit{Mach}_2'$, *written* $s_1 \preceq s_2$, *iff* $s_1 \leq_{\mathcal{E}} s_2$ *and:*

- $t_1 = t_2$;
- $a_1 = \mathbf{async}(t_2)$;
- $a_2 = \mathbf{finish}(\vec{t'}, t_2, \vec{t''})$;
- $a_1 = \mathbf{rel}(l : \vec{\phi})$ *and* $a_2 = \mathbf{acq}(l : \vec{\phi'})$ *for* $l : \vec{\phi} \bowtie l : \vec{\phi'}$; *OR*
- $s_1 \preceq s' \preceq s_2$ *for some* s'.

Theorem 1 (Race-Freedom). *If* $\mathcal{E} : \mathit{Mach} \dot{\to}^* \mathit{Mach}'$ *for* $\vdash \mathit{Mach}$, *and if* $s_1 \bowtie s_2$ *for* $s_1 \leq_{\mathcal{E}} s_2$, *then* $s_1 \preceq s_2$.

As a final point, we prove that the implementation of acquires and releases does not have to track the permission words that are returned by acquires, and only needs to count the number and sorts of acquires and releases:

Lemma 3. *If* $\vdash \cdot \setminus T$ *and* $\mathcal{E} : \cdot \setminus T \dot{\to}^* H \setminus T'$ *then there is a one-to-one mapping between releases in* $\mathcal{E}$ *and subsequent acquires of the same permissions, where* **new** *is considered an exclusive acquire.*

6 Extensions

We now show how Core HJp can be extended to support two common parallel patterns, array-based parallel loops and objects guarded by critical sections. These extensions modify Core HJp very little, and it is straightforward to show that they preserve the race-freedom property proved in Section 5.

6.1 Array-Based Parallelism

For the purposes of this paper, *array parallelism* is the technique of dividing an array into disjoint pieces which are then modified by parallel tasks. (Parallel tasks that read from the same array are straightforward to support using shared read permissions.) One technical difficulty in supporting array parallelism in HJp is the potential aliasing inherent to standard Java arrays. Specifically, each dimension of a standard Java array is an array of pointers to the next dimension, and there is no guarantee that these pointers do not alias. Thus there is no easy way to break a standard multi-dimensional Java array into disjoint pieces. To address this problem, HJ includes a construct called an *array view* [14, 27, 22], which intuitively is an array that is indexed by either one- or many-dimensional *points*. Under the hood, array views are implemented as maps from points to indexes in a one-dimensional Java array. In Core HJp, array views are modeled as maps from points to store locations. Holding a permission ϕ for an array view A is then just a shorthand for holding permission ϕ for all the store locations in A.

To support array parallelism, HJp allows an exclusive permission to an array to be split into exclusive permissions to disjoint pieces of the array, which can then be passed to child tasks for parallel modification. When all the child tasks are done, these exclusive permissions are then combined back into an exclusive permission for the entire array. This is written in Core HJp as follows:

$$\textbf{foreach}\ (x \in r;\ y_1 \subseteq a_1;\ \ldots;\ y_n \subseteq a_n)\ \mathit{body}$$

This expression forks child tasks, as per **async**, to modify disjoint portions of the array views a_1 through a_n, and then waits for the child tasks to complete, as per **finish**. One task is created for each point specified by r, a region expression, and this task executes *body* with the variable x bound to the selected point in r and with each variable y_i is bound to a sub-view of a_i. These sub-views are formed by logically dividing the regions of each array view a_i into rectangular pieces, one for each point in r, and then binding y_i to the sub-view of a_i for the rectangular piece given by the current value of x. For example, the code

$$\textbf{foreach}\ (x \in [1:M, 1:N];\ \mathit{sub} \subseteq a)\ \textbf{for}\ (p \in \mathit{sub}.\mathsf{rgn})\ \mathit{sub}[p] := F(\mathit{sub}[p])$$

modifies 2-dimensional array view a with $M * N$ parallel tasks, each of which applies function F to the elements of a portion of a. The expression $\mathit{sub}.\mathsf{rgn}$ returns the region of the array view *sub*, while $[1:M, 1:N]$ is the rectangular region of points whose dimensions are from 1 to M and from 1 to N, inclusive.

When a **foreach** begins, the parent task must hold exclusive permissions to each array view a_i. Each child task then receives exclusive permissions to the sub-views passed to it in the variables y_i, permissions that it must still hold upon completion. Once all children have completed, the **foreach** then returns the original exclusive permissions for the array views a_i to the parent task.

Permissions can also be stored in and borrowed from the cells of array views just as with normal objects. The one caveat is that only one permission may be borrowed from a given array view at a time. Allowing multiple borrows from the same array view would require complex typing features, such as dependent types, to prove statically that these borrows use different points. This restriction has not been a problem in practice.

6.2 Objects Guarded by Critical Sections

Another common parallel pattern that is supported by HJp is objects that can only be accessed inside critical sections. Critical sections in HJ and HJp are written with the construct **isolated**$(\vec{x})\ M$, which is called an isolated region. This indicates that program term M should be run in a way that is isolated from, meaning not at the same time as, any other conflicting isolated region. Two isolated regions are said to conflict if, at runtime, the values of their variables $\vec{x}$ overlap. Isolated regions are therefore similar to locks and to Java **synchronized** statements. A key difference is that the **isolated** construct prevents deadlock: if an isolated region occurs inside another isolated region, then the variables $\vec{x}$ of the inner isolated region must refer, at runtime, to a subset of a the objects referred to by the outer isolated region. Otherwise, an exception is thrown. This is similar to requiring that a task cannot grow the set of locks it holds while already holding locks. Note that throwing an exception in a potential deadlock situation is a conscious choice in the design of HJp. Although there are type systems to statically prevent deadlocks without exceptions [11], we have found that our approach is intuitive to use and does not cause problems in practice.

Isolated regions act as guards in HJp, allowing unrestricted access to the objects referred to by the variables $\vec{x}$, while preventing any access to these objects outside isolated regions that refer to them. To support this in HJp, any class C can be designated as an *isolated class*, meaning that all objects o of class C can only be accessed inside critical sections for o. Isolated classes are designated in Surface HJp by making them subclasses of the `IsolatedObject` class. The **isolated**$(\vec{x})\ M$ construct then requires the variables $\vec{x}$ to each have type C for some isolated class C. The body M is executed with write permissions $w_i\mathrm{R}, w_i\mathrm{W}$ for each x_i, for some permission words $\vec{w}$.

This can be modeled in Core HJp with acquires and releases of write permissions to the variables $\vec{x}$ at the beginnings and ends of isolated regions. The only change required to Core HJp is that, for isolated objects, acquires never throw exceptions, they simply wait until the acquire can succeed. This change obviously does not violate the race-freedom guarantee from Section 5, since it only restricts the possible executions that must be considered. As per the discussion in Section 4, however, the HJp compiler does *not* insert acquires and releases

for isolated objects, since this would change the synchronization behavior of a program. Instead, if accesses are made to an isolated object outside of an isolated region, the HJp compiler flags a compile-time error.

7 Practical Experience using HJp

Surface HJp is an extension of HJ [14], which itself is an extension of Java to include the **async**, **finish**, and **isolated** constructs, along with a number of constructs for manipulating array views [14, 27, 22]. Surface HJp adds the keywords **reading**, **writing**, **shared_reading**, and **exclusive**, which can be applied to a method argument indicate that the corresponding permission to the argument must be held on entry to and exit from the method. The same keywords can also be applied to an entire method, indicating that the permission must be held for **this**. A method can be annotated with `exclusive_ret` to indicate that exclusive permissions are held for the return value on exit. Other keywords are possible, but these are the common cases that were needed for our benchmarks. Object fields and array view element types can be annotated with **exclusive** to indicate storable permissions. To indicate that a class is an isolated class, it must inherit from `IsolatedObject`. Acquires are written with the methods `acquireR()`, `acquireW()`, `acquireSR()`, or `acquireX()`; similar methods exist for releases.

By design, Surface HJp allows the programmer to think only about read, write, shared read, and exclusive permissions, without having to worry about the complexities of permission words, word variables ξ, etc. Specifically, exclusive field reads are not marked with the permissions that are being borrowed, there are no remits, and acquires do not involve let-bindings for permission words. To support this, the HJp compiler infers all of these; as discussed in Section 4, it also inserts acquires and releases, to support gradual typing. We implement this inference using standard dataflow analysis techniques, using a backwards dataflow to determine where permissions must be acquired or borrowed and a forwards dataflow to insert remits and releases. Although we leave the question of completeness of this inference algorithm for future work, it has worked well in practice. This is all implemented as a lightweight compiler pass on the Soot intermediate language [31] used in the back-end of the existing HJ compiler.

In previous work [34], we examined the performance impact of dynamic permission acquires and releases, which yielded an average slowdown of 1.5×. (This work used a subset of the benchmarks we use here.) In essence, that work represented the minimum possible programmer effort in using HJp. Here, we quantify the maximum possible programmer effort, where programs have been modified enough to remove all acquires and releases; this is checked with the `-staticperms` HJp compiler flag. More specifically, we have taken a set of HJ programs, written without permissions in mind, and ported them to HJp by adding enough annotations to statically guarantee race-freedom. With a few exceptions described below, the resulting HJp programs compile to the exact same programs as the original HJ programs, so there is guaranteed to be no performance penalty.

Benchmark Name	Code Size (Methods)	LoC for array views		LoC for method keywords		LoC for storable perms		LoC for isolated objects		Total	
NPB.CG	1070 (61)	4	0.37%	25	2.33%	7	0.07%	0	0.00%	36	3.36%
JGF.Series	225 (15)	3	1.33%	6	2.67%	3	1.33%	0	0.00%	12	5.33%
JGF.LUFact	467 (20)	0	0.00%	16	3.40%	11	2.36%	0	0.00%	27	5.78%
JGF.SOR	175 (12)	1	0.57%	6	3.43%	4	2.28%	0	0.00%	11	6.29%
JGF.Moldyn	741 (57)	19	2.56%	9	1.20%	29	3.91%	0	0.00%	57	7.69%
JGF.RayTracer	810 (67)	1	0.12%	57	6.75%	22	2.60%	4	0.47%	84	9.94%
BOTS.NQueens	95 (3)	0	0.00%	3	3.15%	0	0.00%	1	1.00%	3	3.16%
BOTS.Fibonacci	70 (3)	0	0.00%	0	0.00%	0	0.00%	0	0.00%	0	0.00%
BOTS.FFT	4480 (46)	13	0.29%	33	0.74%	0	0.00%	0	0.00%	46	1.04%
PDFS	537 (26)	0	0.00%	10	1.80%	8	1.44%	8	1.44%	26	4.67%
DPJ.BarnesHut	682 (56)	1	0.15%	18	2.64%	10	1.47%	0	0.00%	28	4.11%
DPJ.MonteCarlo	2877 (287)	3	0.10%	151	5.25%	22	0.59%	1	0.03%	177	6.15%
DPJ.IDEA	228 (18)	1	0.43%	9	3.94%	8	3.50%	0	0.00%	18	7.89%
DPJ.CollisonTree	1032 (69)	0	0.00%	108	10.40%	24	2.32%	0	0.00%	132	12.70%
DPJ.K-Means	501 (38)	1	0.20%	25	4.99%	6	1.20%	1	0.20%	33*	6.59%
Total	13990 (778)	47	0.33%	476	3.40%	152	1.09%	15	0.11%	690	4.93%

Table 1. Programmer Effort for Statically Verifying Race-Freedom in HJp

We chose a number of small- to large-scale parallel benchmarks from the NAS Parallel Benchmark suite [3] (NPB), JavaGrande benchmark suite [30] (JGF), the BOTS benchmark suite [17], a Parallel Depth First Search application (PDFS), and the benchmarks used for Deterministic Parallel Java [6, 7] (DPJ), another type system for statically ensuring race-freedom. The DPJ benchmarks were originally written in Java and were ported to HJ, while the others were originally written in HJ. We measured the number of lines of code (LoC) that had to be modified (from the HJ version) to statically ensure race-freedom.

The results of this experiment are summarized in Table 1, which presents, for each benchmark, the name prefixed with the suite it came from, the code size in LoC with the number of methods in parentheses, and then the LoC modified from the original code. The latter are divided into modifications needed for: array parallelism, which mostly included adding the **foreach** construct from Section 6.1; adding the method keywords discussed above; adding the **exclusive** keyword to fields and array view elements types, to use storable permissions; and designating classes as subclasses of `IsolatedObject`. On average, HJp requires about 5% of the LoC to be annotated, with the majority of the annotations, accounting for 3.4% of the LoC, being method keywords.

There is one additional modification that was necessary for the K-Means DPJ benchmark, which is not reflected in Table 1; the total LoC for this benchmark is marked with an asterisk. The issue is that the original code uses an array `lock[]` of locks, where each `lock[i]` guards accesses to the elements `new_centers[i]` and `globalSize[i]` of two other arrays. To support this access pattern in HJp, we had to refactor these into a single array of a new class `ClusterAttr` that inherits from

Benchmark Name	Code Size (Methods)	LoC modified in DPJ		LoC modified in HJp		Execution Time (s) Xeon		Execution Time (s) T2	
						DPJ	HJp	DPJ	HJp
DPJ.BarnesHut	682 (56)	80	11.73%	28	4.11%	4.207	4.041	5.715	5.695
DPJ.MonteCarlo	2877 (287)	220*	7.64%	177	6.15%	3.102	3.047	6.065	5.792
DPJ.IDEA	228 (18)	24	10.52%	18	7.89%	0.725	0.731	0.737	0.705
DPJ.CollisonTree	1032 (69)	233	22.58%	132	12.70%	1.253	1.268	3.282	3.245
DPJ.K-Means	501 (38)	5*	1.00%	33*	6.59%	20.188	19.016	65.084	64.953
Total	5320 (468)	557	10.47%	388	7.29%				

Table 2. Comparison of Programmer Effort and Runtimes between DPJ and HJp

`IsolatedObject`. This class has two fields, `new_centers` and `globalSize`, where the first is marked as an **exclusive** field to allow it to be accessed during an isolated region **isolated**`(x)` for the parent `ClusterAttr` object `x`. The second field, `globalSize`, is a primitive Java integer, and so does not need to be **exclusive**.

We also directly compared HJp with DPJ on the DPJ benchmarks. This comparison is summarized in Table 2. On average, HJp required modification to only 7.3% of the LoC, while DPJ required modification to 10.5% of the LoC. For most benchmarks, HJp requires fewer annotations; for K-Means, however, HJp requires annotations on 33 LoC, as opposed to the 5 LoC for DPJ. Both the K-Means and MonteCarlo benchmarks, however, are not completely verified by DPJ: each of these benchmarks require the user to add a `commutative` annotation to one of the methods. This is an unchecked user assertion in DPJ stating that two parallel executions of the method always commute. Thus, although the HJp version of K-Means requires more annotations, this version is statically verified. Further, the HJp version of the MonteCarlo benchmark is also statically verified, and requires fewer annotations than DPJ.

The execution times of DPJ and HJp were also compared, to ensure that there is nothing about HJp that limits performance. The results are given on the right side of Table 2. This includes numbers for two machines: a 16-core (quad-socket, quad-core per socket) Intel Xeon 2.4GHz system with 30GB of memory, running Red Hat Linux (RHEL 5) and Sun JDK 1.6 (64-bit version); and a 128-thread (dual-socket, 8 cores per socket, 8 threads per core) 1.2 GHz UltraSPARC T2 (Niagara 2) with 32 GB main memory, running Solaris 10 and Sun JDK 1.6 (64-bit version). The size of the thread pool for DPJ was varied, and the table shows the best numbers obtained. In most cases, DPJ and HJp performed comparably. For K-Means, HJp performed better; we believe this is because the DPJ runtime uses JUC locks, while HJp uses the built-in **synchronized** construct of Java, which is significantly faster.

8 Related Work

There has been much recent work on imperative parallel programming languages that prevent data races, mostly based either on *ownership* or *permissions*. In the

former, each object has an *owner*, specified in its type, that mediates all accesses to the object. Originally introduced by Clarke et al. [15] to control aliasing, Boyapati et al. [11] showed how to use ownership to ensure race-freedom by only allowing accesses to an object when the current task either owns an object or holds a lock that owns an object. Static race detection [25, 1, 19, 2] is form of ownership, where each object is either owned by a lock or by the current task. In the work of Vaziri et al. [33, 32], object fields are owned by atomic set objects, which ensure that all sequences of accesses to a group of related values satisfy a strong consistency guarantee called *atomic-set serializability*. Deterministic Parallel Java (DPJ) [7, 6] also fits the ownership model, where each object is owned by a memory region. DPJ ensures determinism by restricting parallelism to disjoint regions.

Though it is a powerful notion, ownership suffers from a number of drawbacks. First, it requires programmer annotations to specify ownership; e.g., the comparison of HJp with DPJ in Section 7 indicates a higher annotation burden in DPJ. Second, ownership-based systems are only designed for a single parallel pattern; e.g., traversal-based ownership in DPJ and lock-based ownership in other approaches. The work of Vaziri et al. [33, 32] partially addresses this concern, since all synchronizations are automatically generated by the compiler after the atomic sets are specified. A final issue is the static nature of ownership. This means that the synchronization behavior of an object cannot change over time, which again limits the algorithms that can be written using ownership.

Permission-based systems, in contrast, view the ability to access an object as a resource, which may change over time. They are closely related to linear type systems, which ensure that resources are not duplicated or deleted when doing so is disallowed. A number of systems for avoiding races have been based on linear types, since only one task can have permission on a linear pointer at a time. Haller and Odersky [20] describe one such system, Scala capabilities. A major breakthrough was Boyland's work on fractional permissions [12], which showed how a linear read/write permission could be split into fractional read permissions. One approach that builds on this work is typestate-oriented programming (TSOP) [35, 4, 5], in which the "state" of an object may be changed only when an exclusive, non-fractional permission is held for it. Beckman et al. [4] use this approach to ensure that state changes do not cause data races. Although gradual typing has been studied for TSOP [35], it is not clear that this could be directly applied to race-freedom as in HJp. In addition, permissions have been studied in the context of program verification using separation logic [16, 10, 9].

The storable permissions of HJp can be seen as a restricted form of Boyland's nested permissions [13]. Although storable permissions are less expressive, they do seem to correspond to many of Boyland's examples in a more concise way. Specifically, storable permissions allow these examples to be expressed without using existentials, object equality, and disjunction at the type level, which seem to be required to express them using nested permissions. Fahndrich and DeLine [18] have also introduced a notion of permission guards, where permission "key" ρ allows access to a linear permission τ. The latter can be temporarily borrowed

using the "focus" construct when permission key ρ is held, in a manner similar to permission borrowing for exclusive fields in HJp.

9 Conclusions

In this paper, we present a new type system for race-free parallel programming, based on Boyland's fractional permissions. Our system, Habanero Java with permissions (HJp), is an extension of the Habanero Java (HJ) task-parallel language. HJp is designed to be *gradual*, meaning that it can compile parts of a program that do not contain any annotations or types related to race-freedom, by inserting dynamic checks. This allows existing programs to be compiled with no modifications. The programmer can then gradually add permission annotations to increase performance and static guarantees, eventually leading to a fully annotated, race-free program. Further, no parallel or concurrent programming expertise is necessary to understand these permission annotations. We demonstrate how a number of different concurrency patterns, such as fork-join, array partitioning, and objects guarded by critical sections, can be accommodated in HJp. We also introduce a number of theoretical advances over previous work on fractional permissions, including aliased write permissions and simpler way to store permissions in objects than previous approaches. Finally, we evaluate the annotation burden required to yield statically-verified race-free benchmarks in HJp starting from existing HJ benchmarks, using a complete implementation of the compiler and runtime of the HJp type system. Our results show that for 15 benchmarks we have been able to statically verify race-freedom with only a modest number (5% of the lines of code on average) of annotations.

Acknowledgments

We would like to acknowledge the generous help of John Boyland for his many comments and suggestions. This work was supported in part by the U.S. National Science Foundation through awards 0926127 and 0964520.

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, 2006.
2. D. F. Bacon, R. E. Strom, and A. Tarafdar. Guava: a dialect of java without data races. In *OOPSLA*, 2000.
3. D. H. Bailey et al. The NAS parallel benchmarks. *Intl. Journal of Supercomputer Applications*, 5(3), 1994.
4. N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. In *OOPSLA'08*, 2008.
5. K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *OOPSLA'07*, 2007.
6. R. L. Bocchino et al. A type and effect system for deterministic parallel java. In *OOPSLA'09*, 2009.

7. R. L. Bocchino et al. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL'11*, 2011.
8. H.-J. Boehm and S. V. Adve. Foundations of the c++ concurrency memory model. In *PLDI*, 2008.
9. R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In *POPL*, 2005.
10. R. Bornat, C. Calcagno, and H. Yang. Variables as resource in separation logic. *Electron. Notes Theor. Comput. Sci.*, 155:247–276, 2006.
11. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA'02*, 2002.
12. J. Boyland. Checking interference with fractional permissions. In *SAS '03*, 2003.
13. J. Boyland. Semantics of fractional permissions with nesting. *ACM Trans. Program. Lang. Syst.*, 32, 2010.
14. V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: the new adventures of old X10. In *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, 2011.
15. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, 1998.
16. M. Dodds, S. Jagannathan, and M. J. Parkinson. Modular reasoning for deterministic parallelism. In *POPL'11*, 2011.
17. A. Duran et al. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In *ICPP*, 2009.
18. M. Fahndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. In *PLDI*, 2002.
19. C. Flanagan and S. N. Freund. Type-based race detection for java. In *PLDI*, 2000.
20. P. Haller and M. Odersky. Capabilities for uniqueness and borrowing. In *ECOOP '10*, 2010.
21. S. Heule et al. Fractional permissions without the fractions. In *Proceedings of the 13th Workshop on Formal Techniues for Java-Like Programs (FTfJP)*, 2011.
22. M. Joyner. *Array Optimizations for High Productivity Programming Languages*. PhD thesis, Rice University, 2008.
23. J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *POPL '05*, 2005.
24. D. Marino et al. DRFx: a simple and efficient memory model for concurrent programming languages. In *Proceedings of PLDI '10*, 2010.
25. M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI*, 2006.
26. A. Nanevski, G. Morrisett, and L. Birkedal. Hoare type theory, polymorphism and separation. *J. Funct. Program.*, 18:865–911, 2008.
27. J. Shirako, H. Kasahara, and V. Sarkar. Language extensions in support of compiler parallelization. In *LCPC*, 2007.
28. J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, 2006.
29. A. Singh et al. Efficient processor support for DRFx, a memory model with exceptions. In *Proceedings of ASPLOS '11*, 2011.
30. L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing*, 2001.
31. R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a java optimization framework. In *Proceedings of CASCON'99*, 1999.
32. M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*, 2006.
33. M. Vaziri, F. Tip, J. Dolby, C. Hammer, and J. Vitek. A type system for data-centric synchronization. In *ECOOP'10*, 2010.
34. E. Westbrook, J. Zhao, Z. Budimlić, and V. Sarkar. Permission regions for race-free parallelism. In *RV*, 2011.
35. R. Wolff, R. Garcia, E. Tanter, and J. Aldrich. Gradual typestate. In *ECOOP*, 2011.