A Time-Aware Type System For Data-Race Protection and Guaranteed Initialization

Nicholas D. Matsakis
ETH Zurich, Switzerland
nmatsaki@inf.ethz.ch

Thomas R. Gross
ETH Zurich, Switzerland
trg@inf.ethz.ch

Abstract

We introduce a type system based on *intervals*, objects representing the time in which a block of code will execute. The type system can verify time-based properties such as when a field will be accessed or a method will be invoked.

One concrete application of our type system is data-race protection: For fields which are initialized during one phase of the program and constant thereafter, users can designate the interval during which the field is mutable. Code which *happens after* this initialization interval can safely read the field in parallel. We also support fields guarded by a lock and even the use of dynamic race detectors.

Another use for intervals is to designate different phases in the object's lifetime, such as a constructor phase. The type system then ensures that only appropriate methods are invoked in each phase.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: parallel programming

General Terms Design, Theory, Verification

1. Introduction

The notion of time is central to reasoning about the correctness of programs. Most objects, for example, can only safely be used after a designated construction period, which establishes the class invariants. Similarly, in parallel programming, it is common to restrict the operations permitted on an object during the times when it is accessible to multiple threads; for example, one might require that an object be immutable while it is shared between threads.

Despite the importance of time, most languages offer only implicit means for interacting with it. There are no constructs for naming the period of time in which a block of code executes or in which multiple threads are active. In fact, in a parallel context, even the relative ordering of program statements is not fully exposed in the language itself. In a typical threading API, for example, *happens-before* relations between statements in different threads are not declared, but rather come about as a side-effect of using library primitives like signals and locks.

The intervals model [21] incorporates program time into the language as a first-class construct. An *interval* is an object representing the span of program time in which a certain piece of code — such as a statement, method call, or asynchronous task — executes. Intervals are partially ordered through a *happens-before* relation.

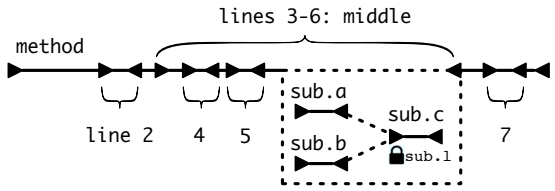
The key contribution of this paper is a static type system that is aware of intervals and the *happens before* relation. We focus on two concrete applications:

- *Flexible Data-Race Protection:* Our system supports both lock-free and locked parallel patterns. Users can choose which data-race protection scheme to use on a field-by-field, object-by-object basis. For example, some fields might be immutable after a specific interval has elapsed, while others are protected by a lock and changed continuously.
- *Controlled Object Initialization:* Users can designate when and under what conditions fields and methods are accessible. This can be used, for example, to ensure that initialization methods are only invoked during a specific construction phase.

Our type system allows each method to be checked modularly, without the need for inter-procedural or whole-program analysis. Although they are rarely needed, we also allow checked "escape hatches" (similar to a checked downcast). These checks allow the user to assert that data is readable or writable if the static rules prove to be insufficient.

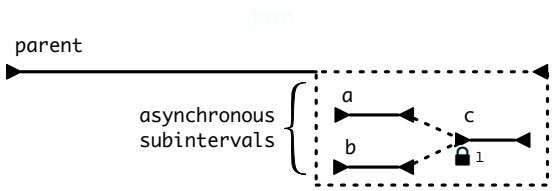
The paper begins with a high-level overview of the intervals model. We then introduce our type system and give a series of examples demonstrating its various features. Next, we present a formal model of our type checker and report on our experiences using our implementation. Finally, we discuss related work and conclude the paper.

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.




```
1 void example(Lock l) {
2   doX();
3   middle: {
4     Subintervals sub = new Subintervals(middle, l);
5     doY();
6   } // (Class "Subintervals" defined in Figure 2)
7   doZ();
8 }
```

Figure 1. Inline subintervals. An interval is depicted by two inward facing triangles. The numbers indicate the line number of the statement(s) represented by each interval.




```
1 class Subintervals(Interval parent, Lock l) {
2   interval a(this.parent) { /* code for a */ }
3   interval b(this.parent) { /* code for b */ }
4   interval c(this.parent) { /* code for c */ }
5   this.a hb this.c;
6   this.b hb this.c;
7   this.c locks this.l;
8 }
```

Figure 2. Asynchronous subintervals. Each interval is drawn as two inward facing triangles. The lines (both dashed and solid) represent the *happens-before* relation.

2. Summary of the Intervals Model

In this section, we introduce the core ideas of intervals. The model we present here is based on earlier work [21–23] but modified to better support the static type system that is the focus of this paper.

Intervals are first-class objects representing the slice of program time used to execute some block of code. The code associated with an interval can be anything from a single statement to an entire method.

Intervals can be nested inside of one another, forming a tree. An interval will not begin execution until its parent has begun, and it must finish before its parent can finish. In addition, intervals may be related to one another through a *happens-before* relation. If interval i *happens before* interval j, then i must complete before j can begin.

There are two distinct kinds of intervals: *Inline intervals*, presented in Figure 1, represent the control flow of sequential code. *Asynchronous intervals*, presented in Figure 2, represent parallel tasks. We explain each in turn.

2.1 Inline Intervals: Sequential Code

Inline intervals represent the execution of individual statements within a method. They allow the programmer to reference and name the time in which different portions of the method will execute.

Figure 1 contains a method `example()` and a diagram showing the intervals for its statements. An interval is depicted with two inward-facing triangles. Lines (both dashed and solid) represent the *happens-before* relation. The dashed box depicts asynchronous intervals. It will be explained in the next section, along with its contents.

The statements from the method `example()` each correspond to an interval in the diagram (labeled by line number). These intervals are fully ordered with respect to one another, reflecting the ordering of the statements in the method body.

The nesting of inline intervals matches the nesting of substatements within compound statements. The statement labeled `middle` on line 3, for example, is a code block. Its interval is therefore the parent of the intervals for its substatements.

The various inline intervals are all contained within a larger interval labeled `method`. `method` is a special variable representing the current activation of the method in which it is used, much as `this` represents the current receiver. The `method` interval is always the parent of the inline subintervals from the method body.

Labeling a statement gives a name to its inline interval. It also creates a local variable of type `Interval`, whose value is the corresponding interval. The interval for the statement `middle`, for example, is used in an expression on line 4.

Inline intervals are usually used only during compilation. They need only be instantiated at runtime if the interval is used in an expression.

2.2 Asynchronous Intervals: Parallel Tasks

Asynchronous intervals are used to express parallel tasks. Asynchronous intervals and any dependencies between them are declared as members of a class, as shown in Figure 2. When an instance of the class is created, its interval members are instantiated as well and added to the scheduler for eventual execution.

The syntax used in Figure 2 bears some explanation. First, we make use of a streamlined, Scala-like [28] constructor notation, where the constructor arguments appear directly after the class name. These arguments are implicitly stored into `final` fields of the same name. Line 1 of Figure 2 would therefore be written in standard Java as:

```
class Subintervals {
  final Interval parent;
  final Lock l;
  Subintervals(Interval parent, Lock l) {
      this.parent = parent;
      this.l = l;
  }
}
```

Lines 2–4 declare the interval members. The declaration `interval a(this.parent)`, for example, creates an asynchronous interval named `a` whose parent interval is the field `this.parent`. This interval `a` is then accessible as a field, like `this.a`.

Note that the parent of the three subintervals comes from the class constructor. In this way, the class defines a bundle of related intervals, but the code which instantiates it decides where those intervals fit into the overall schedule.

Asynchronous intervals run during a second phase of their parent interval. First, the parent interval runs its associated code. Once the interval's code has finished, its asynchronous subintervals begin to execute. In the diagram in Figure 2, this second phase is depicted by the dashed box. It is right-justified to indicate that the phase occurs after the execution of the code associated with `parent`.

To better understand how the execution of asynchronous intervals works, let us return to line 4 of Figure 1. Here, the class `Subintervals` is instantiated with the interval `middle` as the value for the `parent` parameter. The resulting schedule is shown in the diagram above the code: after lines 4 and 5 have executed, but before the block as a whole is complete, the subintervals `a`, `b`, and `c` (now members of `sub`) will execute. All three intervals will have finished before the interval for line 7 executes. An inline interval can thus be used as a lexically scoped fork-join construct.

The final lines in Figure 2 declare special requirements to the scheduler. Lines 5 and 6 extend the *happens-before* relation with two new edges, `a`→`c` and `b`→`c`. These edges ensure that `a` and `b` will complete before `c` begins (`a` and `b` remain unordered with respect to one another).

Line 7 associates the interval `c` with the lock `l`. As a result, the runtime will automatically acquire the lock `l` before `c` starts and release it after `c` ends.

3. The Intervals Type System

This section gives an overview of the intervals type system. We give a number of examples featuring common parallel patterns, such as point-to-point synchronization or fork-join processing, and show how each case could be typed. These examples serve not only to illustrate the features of our system but also to demonstrate the variety of programs which we can handle.

3.1 Data-Race Protection

Figure 3 presents a modified version of the `Subintervals` class where each interval generates a result. The `c` interval

```
1  class Subintervals(Interval parent) {
2    int aResult guardedBy this.a;
3    interval a(this.parent) {
4      aResult = ...;
5    }
6
7    int bResult guardedBy this.b;
8    interval b(this.parent) {
9      bResult = ...;
10   }
11
12   int cResult guardedBy this.c;
13   this.a hb this.c;
14   this.b hb this.c;
15   interval c(this.parent) {
16     cResult = process(aResult, bResult);
17   }
18 }
```

Figure 3. Point-to-point synchronization with intervals as guards.

uses the results generated by `a` and `b` to compute its own result, which is safe due to the edges `a`→`c` and `b`→`c`. This example demonstrates a pattern similar to futures [16], where a future is represented as the pair of an interval with its result (`a` and `aResult`, for example).

Each field declaration is annotated with a *guard object* [23]. The guard object is a gatekeeper: it determines the intervals that are permitted to read or write the fields it protects. The guards in Figure 3 are all intervals. In the next section we will show how other kinds of objects, such as locks, can be used as guards.

When a field is guarded by an interval i, it means that the field can only be written by i or by an inline subinterval of i. Asynchronous subintervals cannot write to the field because there can be multiple asynchronous subintervals active at one time. However, because the parent's code will have finished execution before they begin, asynchronous subintervals may safely *read* fields guarded by their parents.

In addition to subintervals, fields guarded by an interval i can be read by any other interval j which *happens after* i. Such an interval j can also be sure that the fields guarded by i are immutable, because they could only have been written by i and its subintervals, which have already finished.

3.1.1 The `Guard` Interface

Objects which will serve as guards must implement the interface `Guard`. The `Guard` interface defines three methods that can be used to dynamically check whether the guard permits reads or writes from a particular interval, as well as whether the data protected by the guard is immutable at a particular point:

- g`.permitsWr(`i`)` checks whether the guard g permits i to write to the fields guarded by g.

```
class Interval implements Guard {
  boolean permitsWr(Interval current) {
    return current == this
        || current.inlineSubOf(this);
  }
  boolean permitsRd(Interval current) {
    return current == this
        || current.subOf(this)
        || ensuresImm(current);
  }
  boolean ensuresImm(Interval current) {
    return this.hb(current);
  }
}

class Lock implements Guard {
  boolean permitsWr(Interval current) {
    return current.holdsLock(this);
  }
  boolean permitsRd(Interval current) {
    if(current.holdsLock(this)) return true;
    for(Interval i : current.superIntervals())
      if(i.holdsLock(this)) return true;
    return false;
  }
  boolean ensuresImm(Interval current) {
    return false;
  }
}
```

Figure 4. Implementation of the `Guard` interface for intervals and locks.

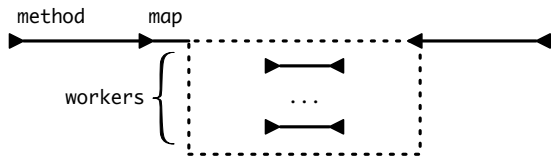
- g`.permitsRd(`i`)` is the same but for reads.
- g`.ensuresImm(`i`)` checks whether the fields protected by g are immutable for the interval i. This means that by the time i executes, they will have reached their final value and will not change again. Typically this occurs because the guard is an interval which *happens before* i and thus will have finished when i is active.

Figure 4 demonstrates how `Interval` and `Lock` objects implement the `Guard` interface. `Lock` guards permit writes from intervals holding the lock and their inline subintervals[1]. Reads are permitted from any subinterval of an interval which holds the lock.

Normally, the built-in guards' methods are never called: the compiler can figure out for itself whether they will succeed or not. They can still be useful, however, as a form of dynamic "escape hatch". The user can assert that the methods will return true, and the compiler will permit the corresponding accesses. If the assertion is false at runtime, an exception will be thrown.

Our built-in guards ensure that writes are fully ordered with respect to all other accesses. In some cases, however, users may wish to use a weaker ordering guarantee for better performance. This can be achieved through the use of a custom guard, as discussed in Section 5.2.

[1] Note that `holdsLock()` is true if an interval holds the lock directly or if it is an inline subinterval of one that does.




```
class Tree(Guard wr) {
  int value guardedBy this.wr;
  Tree left, right guardedBy this.wr;
}
class Compute {
  void computeValue(Tree tree) {
    assert tree.wr.permitsWr(method);

    map: {
      /* create async. subintervals of map
         that read tree.value, tree.left, etc */
    }

    tree.value = /* reduce results from workers */;
  }
}
```

Figure 5. Map-reduce-like pattern using a dynamic assertion.

No matter what ordering guarantees they make, all guards must ensure that the following properties hold:

1. *Authorization to read or write cannot be revoked*: If a check method succeeds for a given interval, then subsequent checks for the same interval will also succeed. This implies that there is never a need to invoke a check method twice and that checks and accesses do not need to be performed atomically.
2. *Write and immutable imply read*: If an interval has permission to write, or if the guarded data is immutable, then the interval also has permission to read.
3. *Inline subintervals can write*: If an interval has permission to write, then its inline subintervals have permission to write.
4. *All subintervals can read*: If an interval has permission to read or write, then both its asynchronous and inline subintervals have permission to read.

These constraints make it possible to write generic code that works with any kind of guard. This same code can then be applied to any data, no matter how it is protected. This code can even contain parallelism. The ability to write such code is essential to allow reusable libraries.

Figure 5 gives an example of reusable code implementing a simple MapReduce-like [10] pattern. The `computeValue()` routine takes a `Tree` argument and updates its `value` field.

```
1  class Tree {
2    ghost Wr;
3    int value guardedBy this.Wr;
4    Tree[Wr=this.Wr] left, right guardedBy this.Wr;
5  }
6  class Compute {
7    void computeValue(
8      Tree[Wr permitsWr method] tree)
9    {
10     // "tree.Wr" is writable within the method.
11   }
12 }
```

Figure 6. Ghost fields and wildcards.

As part of these calculations, the method creates various asynchronous subintervals that use `value` in a read-only fashion.

`tree.value` is declared on line 2 as being guarded by the guard `tree.wr`, which could be any kind of guard. Nonetheless, all the accesses to `tree.value` are considered safe by the compiler due to line 7, which asserts that the guard `tree.wr` permits writes from `method`, the variable representing the interval for the current method invocation. This enables the write on lines 14. It also implies that asynchronous subintervals of `method` may read `tree.value`.

3.1.2 Ghost Fields

The class `Tree` in Figure 5 used a field `wr` to store its guard. Most classes, however, do not enforce synchronization themselves, but rather rely on their caller to ensure they are used in a proper manner. In this case, storing the guard in a field is not an optimal solution. Besides the obvious memory overhead, it provides the caller with no way to verify statically that the guard is writable.

Ghost fields[2] are a technique for rectifying this problem. A ghost field is a field that is erased at runtime. The values for an object's ghost fields are specified when it is first created and become part of its type. Because the caller knows the callee's type, it also knows the values of the callee's ghost fields, and so it can use those values as needed to verify statically that writes and reads are permitted.

It can also be helpful to think of ghost fields as a mixture of a field and a generic type argument: like a field, they refer to objects and are inherited by subclasses. Like a generic type argument, their value is carried through the type of the object. All ghost fields for a class must be bound to a specific object when an instance of the class is created and cannot be changed thereafter.

Figure 6 demonstrates the use of ghost fields. The constructor argument `wr` from class `Tree` has been converted into a ghost field `Wr`, declared on line 2. Ghost fields are untyped, so the declaration consists solely of a name. We use capitalized names for ghost fields to distinguish them from normal, reified fields.

From the perspective of the type checker, ghost fields and reified fields are generally interchangeable. The main distinction is that ghost fields cannot be used within an expression. This is because the ghost field is not present at runtime, so there would be no way to find the corresponding object. Another distinction is that, because ghost fields are untyped, they must be the final element in any path (for example, the path `this.Wr` is legal, but `this.Wr.f` is not).

The value of a ghost field can be specified by using a type annotation such as `Tree[Wr=`$path$`]`. Here, the `[Wr=`$path$`]` annotation indicates that the `Wr` field of the `Tree` instance is equal to the object reached by following the path $path$. An example of such a type annotation appears on line 4, which indicates that the `left` and `right` fields both point to `Tree` objects whose `Wr` field is the same as `this.Wr`. Ghost type annotations are always optional. The unadorned type `Tree` refers to any `Tree` object, regardless of the value of its `Wr` ghost field.

Frequently, however, we do not wish to specify a precise value for a ghost field, but we do wish to impose some constraints. As an example, consider the method `computeValue()` defined on line 7 of Figure 6. This method takes as argument a binary tree `tree` and performs some modifications to it. The precise value of the `tree.Wr` ghost field associated with the `tree` is not important to the method. What is important is that `tree.Wr` should be writable.

To handle such scenarios, we support wildcard type annotations, which are similar to the wildcard type arguments used in Java generics or the type constraints from Constrained Types [26]. Line 8 gives an example. The wildcard annotation `[Wr permitsWr method]` specifies that the ghost field `Wr` is bound to some guard which permits writes from `method`, but does not specify exactly which guard that is.[3] Similar annotations can be used for the other guard methods as well, as shown in the formalization in Section 4.

3.1.3 Method Requirements

Ghost fields can be used to impose conditions on a method's arguments, but sometimes a method may wish to specify other conditions that do not relate to its parameters. Such conditions can be specified with a *method requirement*.

Two examples appear in Figure 7. The first requirement, `this.wr permitsWr method`, specifies that the method may only be invoked if it is permitted to write fields guarded by `this.wr`. The variable `method` here refers to the callee.

The second requirement, `method inlineSubOf this.i`, states that the callee must be an inline subinterval of `this.i`.

[2] We originally adopted the term ghost fields from RccJava [2], but our ghost fields are somewhat different, in that they are inherited by subtypes. The Related Work section explains why we no longer use the purely parametric model found in RccJava.

[3] Although ghosts are untyped, the ghost `Wr` here could only be bound to a guard object because only instance of `Guard` can permit writes.

```
1  class Compute(Guard wr, Interval i) {
2    int value guardedBy wr;
3
4    void computeValue()
5    requires this.wr permitsWr method
6    { /* may write to fields guarded by this.wr */ }
7
8    void duringI()
9    requires method inlineSubOf this.i
10   { /* may only be invoked during this.i */ }
11 }
```

Figure 7. Method requirements.

In effect, this means that the method can only be invoked during the interval `this.i`.

Method requirements always relate two paths. These paths may begin with `this`, the variable `method`, or any of the method parameters. The full set of relations that can appear in a method requirement is defined in our formalization in Section 4.

When a method that has requirements is overridden by a subtype, the subtype is not allowed to declare additional requirements beyond those of the base class. This is necessary for type safety, as the method could always be invoked through a pointer typed as the base class. The caller would then be unaware of the additional requirements imposed by the subtype. It is legal for subtypes to declare fewer requirements than their supertypes, however.

3.2 Program Phases

So far we have seen how to use intervals and guards to protect against data races. However, the same machinery is also useful in a purely sequential context. In this section, we show how to use intervals to separate the initialization phase for an object from its normal usage.

Ideally, the class constructor should already encapsulate the initialization phase for every object. In practice, however, many objects offer methods for additional setup and configuration even after the constructor has returned. The Java type system is not strong enough to ensure that the normal methods of an object are not used until configuration is complete; nor it is able to ensure that methods which should only be used in configuration are not used at other times. Using intervals, it is possible to guarantee both of these properties.

The basic idea is to use an interval to represent the initialization phase. This interval is normally associated with an object using a ghost field, as shown in Figure 8. The class `Phases` defines a ghost field `Init` which stores the initialization interval. It uses method requirements to control when each method can be invoked relative to this initialization interval.

The field `configField` on line 3 is worthy of special mention. Because it is guarded by `this.Init`, `configField` will be mutable during initialization but immutable afterwards. This pattern is the intervals equivalent to a Java `final` field. Unlike the `final` keyword, however, an interval guard can support fields which are immutable during normal operation but which cannot be initialized within the constructor for whatever reason.

```
1  class Phases {
2    ghost Init;
3    int configField guardedBy this.Init;
4
5    void setupMethod()
6    requires method inlineSubOf this.Init
7    { /* Init in progress. */ }
8
9    void eitherMethod()
10   requires this.Init permitsRd method
11   { /* Init may have completed. */ }
12
13   void constructedMethod()
14   requires this.Init hb method
15   { /* Init must have completed. */ }
16 }
17
18 class Creator {
19   void construct() {
20     Phases[Init=init] obj;
21     init: {
22       obj = new Phases[Init=init]();
23       obj.setupMethod();       // OK.
24       obj.eitherMethod();      // OK.
25       obj.constructedMethod(); // ERROR.
26     }
27     obj.setupMethod();       // ERROR.
28     obj.eitherMethod();      // OK.
29     obj.constructedMethod(); // OK.
30     return obj;
31   }
32 }
```

Figure 8. Use of the `Constructor` interval.

The second class in Figure 8, `Creator`, demonstrates how to create and initialize an instance of `Phases`. It designates an inline interval `init` as the value for the `Init` ghost (note that inline intervals may be referred to at any point in the method). Within `init`, any attempt to invoke `constructedMethod()` results in an error, because the method requirement is not met. After `init` has completed, however, `constructedMethod()` may be freely invoked, but `setupMethod()` cannot.

Although the class `Phases` only defines a single `Init` phase, this technique could easily be extended to multiple object phases by adding more ghost fields or interval members.

4. Formalization

This section presents the highlights of our type checker. Rather than give an exhaustive summary of each rule, we

cdecl	:=	`class` `c`(*tys* `fs`) `extends` `c`(*paths*) { *members* }
member	:=	*gdecl* | *fdecl* | *mdecl* | *idecl* | *ldecl* | *hbdecl*
gdecl	:=	`ghost` `f`
fdecl	:=	*ty* `f` `guardedBy` *path*
mdecl	:=	`void` `m`(*tys* `xs`) *reqs* { *lstmts* }
req	:=	*path rel path*
idecl	:=	`interval` `f`(*path*) { *lstmts* }
ldecl	:=	*path* `locks` *path*
hbdecl	:=	*path* `hb` *path*
path	:=	`x`.`fs`
rel	:=	*trel* | *wcrel* | `locks`
trel	:=	`subOf` | `inlineSubOf` | `hb`
wcrel	:=	`permitsWr` | `permitsRd` | `ensuresImm` | `eq`
lstmt	:=	`x` : *stmt*
stmt	:=	`x`.`f` = `x`
	|	`x` = `x`.`f`
	|	`x` = `new` `c`[`fs` `eq` *paths*] (`xs`)
	|	`x`.`m`(`xs`)
	|	`assert` `x` *wcrel* `x`
ty	:=	`c`[`fs` *wcrels paths*]

Figure 9. Grammar for Inter

have chosen to explain at a high-level how two representative examples would be typed. The first example demonstrates how the type checker validates loading and storing into fields. The second example concerns dynamic assertions and method requirements. The complete type rules can be found in the Appendix.

Our type system guarantees that all field accesses would be approved by the appropriate guard object and that all method requirements are satisfied. We have proven that programs which use interval or lock guards are data-race free. Of course, if the user provides their own guards, those guards may permit data races. This is in fact a strength of our system, as some programs can permit race conditions to improve performance without harming correctness.

The type checker is based on a language Inter, which is a highly simplified version of the syntax we have been presenting in our examples. To keep the type rules concise, we have omitted any language feature that is not strictly necessary. This includes such common features as compound statements like `if` or `while` and even method return values. Inter also removes all syntactic sugar. References to fields must explicitly mention the `this` pointer, for example, and all statements are labeled with a name for the corresponding inline interval.

One minor difference in notation between Inter and our examples is that the type `Tree[Wr=method]` would be written `Tree[Wr eq method]` in our formal grammar. This is because we wish to reserve the symbol "=" for lexicographic equality.

The full grammar for Inter is shown in Figure 9. We use the following lexical conventions: The terminals `c`, `f`, `m`, and `x` stand for class, field, method and local variable names, respectively. Keywords are shown in `typewriter` font and non-terminals in *italics*. Note that `this` and `method` are not keywords in the grammar but treated as normal local variable names.

Sequences are indicated by a trailing `s`, so `xs` stands for zero or more local variable names. Unless separated by a semicolon (;), adjacent sequences indicate a sequence of tuples. This notation is widespread but often surprising to those who have not seen it before. As an example, "*tys* `xs`" does not indicate a sequence of types followed by a sequence of local variable names, but rather a sequence of "*ty* `x`" pairs. Similarly, "`fs eq` *pairs*" indicates a sequence of "`f eq` *pair*" tuples.

An Inter program consists of a series of class declarations. A class declaration *cdecl* for a class `c` defines its constructor arguments, supertypes, and members. A class member can be a ghost declaration (*gdecl*), a reified field (*fdecl*), an interval member (*idecl*), a lock or *happens-before* declaration (*ldecl*, *hbdecl*), or a method (*mdecl*).

A method requirement *req* relates two paths by a relation *rel*. The relations are further refined into transitive relations *trel* and wildcard relations *wcrel*. Only wildcard relations can appear in a type. The meanings of the various relations are:

- p `subOf` q: Interval p is a subinterval of q.
- p `inlineSubOf` q: Interval p is an inline subinterval of q.
- p `hb` q: The interval p *happens before* q.
- g `permitsWr` i: g`.permitsWr(`i`)` would succeed.
- g `permitsRd` i: g`.permitsRd(`i`)` would succeed.
- g `ensuresImm` i: g`.ensuresImm(`i`)` would succeed.
- p `eq` q: The paths p and q evaluate to the same object.
- i `locks` l: The interval i acquires the lock l.

The nonterminals *lstmt* and *stmt* correspond to statements with and without a label. As there are no nested expressions, any intermediate results must be stored into a local variable. There are five kinds of statements: Field stores and loads set or get the value of reified fields. `new` statements create new objects. The full type of the new object must be specified, including precise (not wildcard) values for all ghost fields, and the constructor arguments must all be supplied. A method call `x.m(xs)` invokes a method `m` on the receiver `x` with the arguments `xs`. Finally, assertions that a guard permits writes or reads are converted into a special form, the `assert` statement.

Inter generally assumes that all names are unique and methods are given in SSA form. The names of declared fields, interval members, and constructor arguments all live in the same namespace and must be unique from other names in the same class or any of its superclasses. Methods in a subclass may of course override methods in a superclass. Finally, the names of all local variables, parameters, statement labels must be unique within a method or interval body.

```
1  class Subintervals(Interval parent) {
                          E_class
       this : Subintervals
       this.a subOf this.parent
       this.b subOf this.parent
       this.a hb this.b

2    int aResult guardedBy this.a;
3    interval a(this.parent) {...}
4
5    int bResult guardedBy this.b;
6    this.a hb this.b;
7    interval b(this.parent) {
                  E_b (also includes E_class)
         b1 : Interval
         b2 : Interval
         b1 inlineSubOf this.b
         b2 inlineSubOf this.b
         b1 hb b2

8      b1: r = this.aResult;
                  E_b1 (also includes E_b)
         r : int
         r eq this.aResult

9      b2: this.bResult = r;
10   }
11 }
```

Figure 10. A simplified version of the class `Subintervals` from Figure 3, along with the environment that is deduced by the type checker at each point.

These limitations simply make the type rules easier to understand, they do not change the set of programs we can type in any meaningful way.

4.1 Example 1: Field Loads and Field Stores

The first example we consider appears in Figure 10. It consists of a class with two interval members, `a` and `b`. Both generate a result. `b` *happens after* `a` and copies `a`'s result from the field `aResult` into `bResult`. We will explain step-by-step how the type checker checks the interval member `b`.

4.1.1 The Environment $\mathcal{E}$

The environment $\mathcal{E}$ is the heart of the type checker. It records facts that the compiler has been able to deduce about the code being checked. As the compiler proceeds through the program, it adds new facts to the environment. Nothing is ever removed.

The facts in the environment take the form of tuples. These tuples can take two forms. Tuples like *path rel path* relate two paths. Tuples like `x` : *ty* record the type for a local variable.

In Figure 10, the new tuples that are deduced from each statement or declaration are shown beneath it. When checking a particular statement, the environment in scope includes all the tuples deduced from previous statements, as well as those from the interval/method and class the statement is contained in.

4.1.2 Class and Interval Environments

We begin the example by looking at the class environment $\mathcal{E}_{class}$. This environment contains all the information that the compiler can deduce from the class declaration. The contents of $\mathcal{E}_{class}$ will be in scope when checking all members of the class.

The first tuple in $\mathcal{E}_{class}$ is `this : Subintervals`. This simply records the fact that the local variable `this` has the type `Subintervals`.

The next two tuples declare that `this.a` and `this.b` are both subintervals of `this.parent`. These tuples were deduced by examining the parent declarations for each interval member.

The final tuple `this.a hb this.b` is based on the *happens-before* declaration on line 6. If there were any lock declarations, they would appear in the environment as well.

Next, we proceed to the interval `b` declared on line 7. The box labeled $\mathcal{E}_{\texttt{b}}$ shows the tuples added to the environment from `b`'s declaration. These tuples all relate to the inline subintervals for `b`'s two statements, `b1` and `b2`. Each statement has a corresponding local variable of type `Interval`, as indicated by the first two tuples. The next two tuples specify that `b1` and `b2` are both inline subintervals of `this.b`. Finally, the last tuple gives the *happens-before* ordering.

4.1.3 Stable Paths

Because we never remove any tuples from the environment, we must be sure that tuples in the environment reflect facts that are always true as the program executes, and will not become invalidated as fields are updated. This is done by ensuring that all paths referenced from the environment are *stable*. Intuitively, a path is stable if the object it will evaluate to at runtime cannot change.

Some paths are always stable. For example, a path that consists only of a local variable, like `x` or `this`, can never change, since we do not allow variables to be reassigned. Similarly, the path `this.parent` on line 7 is stable because the field `parent` is a constructor argument and therefore immutable.

Other parts are only stable at certain times. For example, consider the field `aResult`, which is guarded by the interval `a`. When the class is first created, the interval `a` has not yet executed, and so `aResult` is not stable. Therefore we could not add a path involving `aResult` to the class environment. During the interval `b`, however, we know that `a` has finished, and therefore the field `aResult` is immutable. That is why the tuple `r eq this.aResult` that appears in $\mathcal{E}_{\texttt{b1}}$ is safe.

In the rules that follow, the judgement $\mathcal{E} \vdash$ *path* `stableBy` $\texttt{x}_i$ declares that *path* is stable during the interval $\texttt{x}_i$. A path which is stable during $\texttt{x}_i$ will also be stable for all intervals

STMT-LOAD

$$\frac{\mathcal{E} \vdash \mathtt{x}_o : \mathtt{c} \quad \mathit{reified}(\mathtt{c}, \mathtt{f}) = (ty_\mathtt{f}; path_g) \quad \Theta = [\mathtt{this} \to \mathtt{x}_o] \quad \mathcal{E} \vdash \Theta(path_g)\ \mathtt{stableBy}\ \mathtt{x}_l \quad \mathcal{E} \vdash \Theta(path_g)\ \mathtt{permitsRd}\ \mathtt{x}_l}{\mathcal{E} \vdash \mathtt{x}_l : (\mathtt{x}_d = \mathtt{x}_o.\mathtt{f}) \to \mathcal{E} + (\mathtt{x}_d : \Theta(ty_\mathtt{f}))}$$

STMT-LOAD-IMMUTABLE

$$\frac{\mathcal{E} \vdash \mathtt{x}_o : \mathtt{c} \quad \mathit{reified}(\mathtt{c}, \mathtt{f}) = (ty_\mathtt{f}; path_g) \quad \Theta = [\mathtt{this} \to \mathtt{x}_o] \quad \mathcal{E} \vdash \Theta(path_g)\ \mathtt{stableBy}\ \mathtt{x}_l \quad \mathcal{E} \vdash \Theta(path_g)\ \mathtt{ensuresImm}\ \mathtt{x}_l}{\mathcal{E} \vdash \mathtt{x}_l : (\mathtt{x}_d = \mathtt{x}_o.\mathtt{f}) \to \mathcal{E} + (\mathtt{x}_d : \Theta(ty_\mathtt{f})) + (\mathtt{x}_d\ \mathtt{eq}\ \mathtt{x}_o.\mathtt{f})}$$

STMT-STORE

$$\frac{\mathcal{E} \vdash \mathtt{x}_o : \mathtt{c} \quad \mathit{reified}(\mathtt{c}, \mathtt{f}) = (ty_\mathtt{f}; path_g) \quad \Theta = [\mathtt{this} \to \mathtt{x}_o] \quad \mathcal{E} \vdash \Theta(path_g)\ \mathtt{stableBy}\ \mathtt{x}_l \quad \mathcal{E} \vdash \Theta(path_g)\ \mathtt{permitsWr}\ \mathtt{x}_l \quad \mathcal{E} \vdash \mathtt{x}_v : \Theta(ty_\mathtt{f})}{\mathcal{E} \vdash \mathtt{x}_l : (\mathtt{x}_o.\mathtt{f} = \mathtt{x}_v) \to \mathcal{E}}$$

Figure 11. Rules for checking the statements from Figure 10. Θ represents a substitution.

that *happen after* $\mathtt{x}_i$. The definition of this judgement can be found in the Appendix.

4.1.4 Checking Statements

Figure 11 gives the formal type rules for checking loads and stores. Statements are checked with the judgement $\mathcal{E} \vdash \mathit{lstmt} \to \mathcal{E}'$, where $\mathcal{E}$ represents the environment when the statement starts and $\mathcal{E}'$ represents the environment after the statement has finished. This judgement allows a statement to add tuples to the environment that may be used by later statements.

There are three rules in Figure 11. The first two rules cover field loads. STMT-LOAD can be used for any field load. STMT-LOAD-IMMUTABLE is an improved version that only applies to loads of immutable fields. It is always preferable to apply the rule STMT-LOAD-IMMUTABLE when possible. The final rule, STMT-STORE, covers field stores.

All three rules are very similar. They begin by invoking the helper function *reified*(c, f) to determine the declared type $ty_\mathtt{f}$ and guard path $path_g$ of the field f being accessed.

The notation $\Theta = [\mathtt{this} \to \mathtt{x}_o]$ defines a substitution function Θ which replaces instances of this with the field owner $\mathtt{x}_o$. Θ is used to convert from the viewpoint of the class declaration to the viewpoint of the method.

In order to access a field, the path to its guard must be stable during the current statement $\mathtt{x}_l$. This ensures that the guard for a field cannot be reassigned while we are accessing the field itself. Furthermore, the guard must either permit reads, ensure immutability, or permit writes, depending on the rule. In the case of a store, there is the additional requirement that the value $\mathtt{x}_v$ to be stored must have a compatible type with the field.

Assuming all checks are successful, both STMT-LOAD and STMT-LOAD-IMMUTABLE add a new tuple $\mathtt{x}_d : \Theta(ty_\mathtt{f})$ to the environment. This tuple declares that the newly defined variable $\mathtt{x}_d$ has the same type as the field which was just loaded.

The rule STMT-LOAD-IMMUTABLE adds an additional tuple ($\mathtt{x}_d$ eq $\mathtt{x}_o$.f) to the environment. This tuple states that $\mathtt{x}_d$ always contains the same object as $\mathtt{x}_o$.f. This tuple is safe because the field f is immutable, and therefore any later load of $\mathtt{x}_o$.f would yield the same result.

Returning to Figure 10, the statement b1 on line 8 loads the field this.aResult into the variable r. Because aResult is immutable, this statement can be checked using the rule STMT-LOAD-IMMUTABLE.

The first step is to find the guard path for aResult, which is declared on line 2 as this.a. The this pointer here is relative to the owner whose field is being accessed; in this case, the owner is also this, so the guard path does not change after substitution.

Proving that the guard this.a ensures immutability for the interval b1 can be done by observing two facts: first, this.a *happens before* this.b. Second, the interval b1 is a subinterval of this.b. Therefore, this.a must have finished when b1 is active.

Because immutable fields are always readable, this statement could also have been checked using the rule STMT-LOAD. The only difference is that the resulting environment would not include the tuple r eq this.aResult. In this example, this makes no difference, but we will see a case in the next section where it is necessary.

Processing the store to bResult in statement b2 is very similar. The guard path in this case is this.b. Here the type checker deduces that this.b permits writes from the interval b2 because b2 is an inline subinterval of this.b.

4.2 Example 2: Assertions and Method Requirements

Figure 12 presents the second example, which shows the interaction of dynamic assertions, method requirements, and equivalence relations. As before, we have interspersed the environment deduced at each point with the code itself. This time there are no interval members but instead the class defines two methods, requireWritable() and assertWritable(). The first method modifies the field value. It requires that the guard wr permit writes in order to be invoked. The second method first asserts dynamically that wr permits writes and then invokes the first. We examine each method in turn.

4.2.1 The Method requireWritable()

The method requireWritable() is interesting because it declares a requirement that this.wr permitsWr method. This requirement is added to the method's initial environment $\mathcal{E}_{\mathtt{rw()}}$, along with the tuples relating to the inline subinterval for the statement rw1. As we will see in the next section, in order to invoke requireWritable(), a caller must first ensure that the requirement is true. Therefore, it is safe for requireWritable() to assume that the requirement holds when it executes.

```
1  class Compute(Guard wr) {
                    E_class
       this : Compute

2    int value guardedBy this.wr;
3
4    void requireWritable()
5    requires this.wr permitsWr method
                E_rw()  (also includes E_class)
         this.wr permitsWr method
         rw1 : Interval
         rw1 inlineSubOf method

6    {
7      rw1: this.value = ...;
8    }
9
10   void assertWritable() {
                E_aw()  (also includes E_class)
         ...
         aw3 : Interval
         aw3 inlineSubOf method

11     aw1: wr = this.wr;
              E_aw1  (also includes E_aw())
         wr : Guard
         wr eq this.wr

12     aw2: assert wr permitsWr method;
                E_aw2  (also includes E_aw1)
         wr permitsWr method

13     aw3: this.requireWritable();
14   }
15 }
```

Figure 12. A mixture of the class `Compute` from figures 5 and 7, along with the environment deduced at each point.

The statement `rw1` itself is a store to the field `value`. As before, the checker must verify that `value`'s guard `this.wr` permits writes from the statement `rw1`. It is able to do so by combining the tuple `this.wr permitsWr method` with the tuple `rw1 inlineSubOf method`. Without the method requirement, however, the type check would not have succeeded, because there would be no way to show that `this.wr permitsWr method`.

4.2.2 The Method `assertWritable()`

`assertWritable()` begins on line 11 with a load of the field `this.wr`. The rule STMT-LOAD-IMMUTABLE from Figure 11 can be used, because the field being loaded is an immutable constructor argument. Two tuples are therefore added to the environment, one declaring the variable `wr` and another equating it with the immutable path `this.wr`.

Line 12 then performs a dynamic check that `wr` is writable. The type rule for `assert` statements is called STMT-ASSERT

$$\textsc{Stmt-Assert}\quad \frac{\mathcal{E} \vdash \mathtt{x}_g : \mathtt{Guard} \qquad \mathcal{E} \vdash \mathtt{x}_i : \mathtt{Interval}}{\mathcal{E} \vdash \mathtt{x}_l : (\mathtt{assert}\ \mathtt{x}_g\ \mathit{wcrel}\ \mathtt{x}_i) \rightarrow \mathcal{E} + (\mathtt{x}_g\ \mathit{wcrel}\ \mathtt{x}_i)}$$

$$\textsc{Stmt-MthdCall}\quad \frac{\begin{array}{c}\mathcal{E} \vdash \mathtt{x}_r : \mathtt{c} \qquad \mathit{signature}(\mathtt{c}, \mathtt{m}) = (\mathit{tys}_{\mathtt{m}}\ \mathtt{xs}_{\mathtt{m}})\ \mathit{reqs}_{\mathtt{m}} \\ \Theta = [\mathtt{this} \rightarrow \mathtt{x}_r, \mathtt{method} \rightarrow \mathtt{x}_l, \mathtt{xs}_{\mathtt{m}} \rightarrow \mathtt{xs}_a] \\ \forall i.\, \mathcal{E} \vdash \Theta(\mathit{reqs}_{\mathtt{m}}(i)) \qquad \forall i.\, \mathcal{E} \vdash \mathtt{xs}_a(i) : \Theta(\mathit{tys}_{\mathtt{m}}(i))\end{array}}{\mathcal{E} \vdash \mathtt{x}_l : (\mathtt{x}_r.\mathtt{m}(\mathtt{xs}_a)) \rightarrow \mathcal{E}}$$

Figure 13. Rules for checking the method `assertWritable()`.

and appears in Figure 13. It simply assumes that the call will succeed and adds the asserted tuple to the environment.

Finally, line 13 calls `requireWritable()`. Method calls are typed by the rule STMT-MTHDCALL. This rule must validate that (a) the method requirements are met and (b) the arguments to the method are of the correct type.

The function *signature*(`c`, `m`) returns the types $\mathit{tys}_{\mathtt{m}}$ and names $\mathtt{xs}_{\mathtt{m}}$ of the arguments as well as the method requirements $\mathit{reqs}_{\mathtt{m}}$. The substitution Θ is then used to convert to the viewpoint of the caller: references to `this` are converted to the method receiver; the interval `method` is converted to the interval for the call statement; and the formal arguments $\mathtt{xs}_{\mathtt{m}}$ are converted to their actual values $\mathtt{xs}_a$.

The final two clauses ensure that the environment $\mathcal{E}$ supports all method requirements and that the arguments have the correct types. Note that we treat sequences as a function from their index to their items. Therefore, the notation $\mathit{reqs}_{\mathtt{m}}(i)$, for example, refers to the ith method requirement.

Returning to the call on line 13 of Figure 12, we see that the invoked method has a single method requirement. After substitution, the requirement is: `this.wr permitsWr aw3`. The type checker can deduce that this requirement is satisfied by combining several available facts: (1) the variable `wr` and the path `this.wr` both refer to the same object; (2) `wr` permits writes by `method`; and (3) `aw3` is an inline subinterval of `method`.

This example highlights the importance of distinguishing immutable fields. We were able to type this example because we could reliably link the local variable `wr` to the path `this.wr`.

4.3 Rules Concerning Ghosts

One feature that did not appear in our examples was ghost fields. Ghosts come into play in two places: checking types and judging relations between paths. The key rules are presented in Figure 14 and explained in this section.

4.3.1 Checking Types

Because of its dependent type system, Inter does not have rules that compare two types in isolation. Instead, the judgement $\mathcal{E} \vdash \mathit{path} : \mathit{ty}$ is used to state that the path *path* has the

$$\text{T-VAR}\quad \frac{(\texttt{x} : ty) \in \mathcal{E}}{\mathcal{E} \vdash \texttt{x} : ty} \qquad \text{T-FIELD}\quad \frac{\mathcal{E} \vdash path : \texttt{c} \qquad reified(\texttt{c}, \texttt{f}) = (ty; \ldots)}{\mathcal{E} \vdash path.\texttt{f} : [\texttt{this} \rightarrow path]\, ty}$$

$$\text{T-SUB}\quad \frac{\mathcal{E} \vdash path : \texttt{c}_{sub} \qquad \texttt{c}_{sub}\ \texttt{is a subclass of}\ \texttt{c}_{sup} \qquad \forall i.\, \mathcal{E} \vdash path.\texttt{fs}(i)\ wcrels(i)\ paths(i)}{\mathcal{E} \vdash path : \texttt{c}_{sup}[\texttt{fs}\ wcrels\ paths]}$$

$$\text{REL-WILDCARD}\quad \frac{\mathcal{E} \vdash path_1 : \texttt{c}[\ldots, \texttt{f}\ wcrel\ path_2, \ldots]}{\mathcal{E} \vdash path_1.\texttt{f}\ wcrel\ path_2}$$

Figure 14. Rules related to ghost fields.

type *ty*. The first three rules in Figure 14 define this judgement.

The rules T-VAR and T-FIELD determine the type of local variables and reified fields and should be self-explanatory. The final rule T-SUB defines the subtyping relation. It states that *path* can be assigned the type $\texttt{c}_{sup}[\texttt{fs}\ wcrels\ paths]$ so long as (a) *path* is typable as some subclass $\texttt{c}_{sub}$ of $\texttt{c}_{sup}$; and (b) all the ghosts `fs` *wcrels paths* apply to *path*. For example, given a path `x.y` and a type `Tree[Wr permitsWr method]`, we must ensure that `x.y.Wr permitsWr Method` is supported by the environment $\mathcal{E}$.

4.3.2 Judging Relations Between Paths

The final rule in Figure 14, called REL-WILDCARD, is used to determine that relation holds based on a wildcard. For example, given a path `x.y` with the type `Tree[Wr permitsWr method]`, this rule would allow us to conclude that `x.y.Wr permitsWr method`. This rule is therefore the inverse of the rule T-SUB we saw earlier: where T-SUB only allows a type to be given if a relation holds, REL-WILDCARD assumes that a relation holds based on a type.

4.4 Remaining Rules

We have only presented the highlights of the full Inter type system here. The complete rules are published in the appendix.

4.5 Proof Outline

In this section, we briefly outline the proof that our type system prevents data races. The complete proof, along with the operational semantics of Inter, is available on our website [1]. For the purpose of the proof, we assume that users are only using the built-in guards.

A machine state in the operational semantics is represented as a tuple $(\mathcal{E}; \mathcal{O}; \mathcal{A})$:

- $\mathcal{E}$ is a restricted subset of the environment used in the type check rules. It records the names of intervals and other objects that have been created along with the *happens-before* relation, interval hierarchy, and what locks must be held.
- $\mathcal{O}$ contains the set of points that have occurred thus far in the execution. A point has the form `x.f` where `f` is either `start`, `mid`, or `end`.
  - When `x.start` has occurred, the interval `x` has commenced execution.
  - When `x.mid` has occurred, the interval's code block is complete and so asynchronous subintervals may commence execution.
  - When `x.end` has occurred, the interval `x` has completed execution in its entirety.
- $\mathcal{A} : \texttt{x} \rightarrow$ *lstmts* maps an asynchronous interval `x` to its statements *lstmts*.

The data-race theorem itself rests on three main lemmas. The first is a standard preservation lemma showing that every step from a well-formed machine state results in another well-formed machine state. The second lemma states that given a well-formed machine state $(\mathcal{E}; \mathcal{O}; \mathcal{A})$, a guard $\texttt{x}_g$, and two intervals $\texttt{x}, \texttt{x}'$ such that $\mathcal{E} \vdash \texttt{x}_g$ `permitsWr x` and $\mathcal{E} \vdash \texttt{x}_g$ `permitsRd x'`, `x` must be ordered with respect to $\texttt{x}'$. That is, any interval permitted to write must be ordered with respect to all other intervals granted access (read or write). The final lemma is analogous to the second, but applies to guards specified by a full path $path_g$, not only a single local variable. These three lemmas are then combined to show that, in every execution beginning from a well-formed machine state, all writes to a given field are ordered with respect to all other accesses of that field.

The proof is largely straight-forward, but there is one subtle point concerning `null` pointers. Imagine a method with a parameter `x` that had the type `Tree[Wr permitsWr method]`. If `null` were given as a parameter of this method, then rule REL-WILDCARD can still be used to judge that `x.Wr permitsWr method`, even though the value of `x` is in fact `null` (and thus `x.Wr` was never bound to a specific object). The reason this does not permit race conditions is that REL-WILDCARD is the only rule which can apply to such a path, and therefore a non-existent guard can only allow writes by at most one interval (`method`, in this case).

5. Experience

Prototype versions of the intervals runtime and type checker are available on our website [1]. We have implemented a complete checker for our type system based on an expanded version of Inter, the language used in our formalization. Our type checker supports the core features of Java, such as generic types with wildcards or multiple inheritance in the form of interfaces. It omits features, like anonymous classes, that can be emulated by source-to-source translation.

We have used our implementation to check a number of representative examples of different parallel patterns, including fork-join, point-to-point, and lock-based synchroniza-

tion patterns. None of the examples require dynamic checks or assertions of any kind.

1. BBPC is a bounded-buffer producer-consumer. Interestingly, although such a design is typically implemented with locks, the intervals implementation is lock-free. It relies solely on creating *happens-before* edges. Section 5.1 will explore BBPC in detail.
2. TSP is a parallel solver for the travelling salesman problem. It uses asynchronous intervals to exhaustively explore all possible paths and find the shortest one. The TSP example is interesting because it makes use of many different kinds of guards, including a custom guard that permits data races under controlled circumstances. Section 5.2 will cover these guards in detail.
3. SOR is an implementation of the successive over-relaxation method for solving linear systems of equations. It is implemented in a fork-join style, where each round is processed to completion before beginning the next round. The matrix is divided into two parts, red and black. Each round first writes to red while reading from black, and then does the opposite. This example makes use of a library class for encapsulating arrays which we have not discussed in this paper.
4. Life is an implementation of Conway's Game of Life. We divide the board into tiles and create an interval per tile per round. Each interval *happens after* the neighboring intervals from previous rounds, thus allowing it to read the results which they generate.

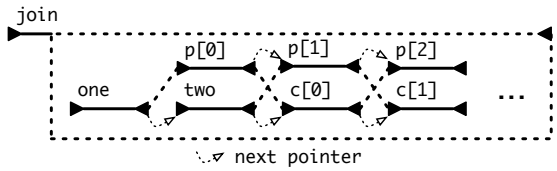
5.1 Bounded-Buffer Producer-Consumer

Bounded-buffer producer-consumers are very common in parallel systems. The idea is to have two independent parallel tasks. Traditionally, the producer task writes data into a buffer and the consumer reads it out. If the producer produces items faster than the consumer can consume them, the buffer will eventually become full. This causes the producer to wait until the consumer has caught up. Most implementations use locks on the buffer to coordinate the two workers.

The intervals version, shown in Figure 15, performs the same task, but does so quite differently. There is no central buffer and thus there are no locks. The producer and consumer are represented as streams of intervals; each interval represents the time to produce or consume one particular item. In the diagram, these intervals are given names like `p[`i`]`, meaning the producer of the ith item, and `c[`i`]`, meaning the consumer of the ith item.

Happens-before edges are used instead of locks to coordinate the timing between producer and consumer:

1. Edges like `p[`i`]`→`c[`i`]` ensure that the consumer of item i will wait for the producer of item i. These edges result from the declaration on line 9.




```
1  abstract class Stream {
2    Stream next guardedBy inter;
3    abstract interval inter;
4  }
5
6  // this == c[i], prod == p[i]
7  class Consumer(Interval parent, Producer prod)
8  extends Stream {
9    prod.inter hb inter; // p[i] -> c[i]
10   interval inter(parent) {
11     /* consume prod.result */
12     next = new Consumer(parent, (Producer)prod.next);
13 } }
14
15 // this == p[i], cons == c[i-2]
16 class Producer(Interval parent, Stream cons)
17 extends Stream {
18   Object result guardedBy inter;
19   cons.inter hb inter; // c[i-2] -> p[i]
20   interval inter(parent) {
21     result = /* produce item */;
22     next = new Producer(parent, cons.next);
23 } }
24
25 class DummyConsumer(Interval parent)
26 extends Stream {
27   Stream link guardedBy parent;
28   interval inter(parent) {
29     next = link;
30 } }
31
32 class Start {
33   void main() {
34     join: {
35       DummyConsumer one = new DummyConsumer(join);
36       DummyConsumer two = new DummyConsumer(join);
37       Producer prod = new Producer(join, one);
38       one.link = two;
39       two.link = new Consumer(join, prod);
40 } } }
```

Figure 15. BBPC

2. Edges like `c[`$i-2$`]`→`p[`i`]` ensures that the producer for item i will wait until item $i-2$ has been consumed. This corresponds to a bounder buffer size of 2. These edges result from the declaration on line 19.

The base class `Stream` is used to define a single link in a stream of intervals. Each link has a corresponding interval `inter`, which is defined abstractly and therefore must also

be defined in each subclass. When `inter` executes, it will create the next link in the stream and store it into the field `next`. That means that the producer `p[`i`]`, for example, creates `p[`$i+1$`]`, which in turn creates `p[`$i+2$`]`, and so on.

The classes `Producer`, `Consumer`, and `DummyConsumer` all extend `Stream`. We first describe how `Producer` and `Consumer` work in the steady state, then discuss `DummyConsumer`, which is used only to bootstrap the system.

The `Producer` and `Consumer` classes are each parameterized by a link from the opposite stream. This link is used to create the incoming *happens-before* edges. Each consumer `c[`i`]` expects to be created with `p[`i`]` as argument, whereas each producer `p[`i`]` expects the consumer `c[`$i-2$`]` from two items back.

When the `Consumer`'s interval executes, it begins on line 11 by reading the data which was produced produced. It then creates the next link in the `Consumer` stream, `c[`$i+1$`]`. It uses `prod.next` to obtain the next producer.[4] `Producer` is similar but it begins by producing data instead.

The class `DummyConsumer` is used to bootstrap the producer and consumer streams. The problem is that a `Producer`, when created, expects to be given the `Consumer` from two items back. But for the first two producers, there is no such consumer! Therefore, we create two instances of the class `DummyConsumer`. A `DummyConsumer` interval doesn't do any actual work, it simply sets the `next` pointer to the value in its field `link`. By creating more `DummyConsumer` instances, the example could be adjusted to allow an arbitrary "buffer size" instead of only 2.

`DummyConsumer` expects the field `link` to be initialized by the interval `parent`. `DummyConsumer`'s interval can therefore read the field safely, because it does not begin execution until its parent's code has executed. In the class `Start`, two `DummyConsumer` instances are instantiated. The parent for both is the inline interval `join`. On lines 38 and 39, the two link fields are initialized as shown in the diagram.

5.2 Use of Guards in the TSP Solver

The parallel solver for the Travelling Salesman Problem is the most complex of the example implementations. One of the interesting aspects of TSP is that it uses a variety of techniques for protecting the shared data, including custom guards that permit controlled data races.

The solver is based around asynchronous worker intervals that share access to a priority queue storing partially explored paths. Each worker extracts the most promising path from the queue and extends it by one step, resulting in a set of new paths (one for each node that is not yet in the path). Each new path is initialized and then inserted into the priority queue for other workers to extend. Once a path is inserted in the queue, it is never modified again.

[4] Here we use a downcast from `Stream` to `Producer`. The actual implementation uses generic types instead, but we avoided them here to keep the example simple.

Whenever a worker finds a complete path that visits all nodes, it acquires a lock and checks to see whether this is the shortest path found so far. At the same time, it updates a heuristic field that stores the length of this path.

If a worker ever finds that the path it is exploring is already longer than the shortest path, then it simply stops and proceeds to the next path in the priority queue. This avoids expanding paths that cannot possibly be shorter than the current solution.

The TSP example uses three kinds of guards in total:

Interval Guards: Because each partial path is only mutable during initialization, and are shared freely afterwards, they are perfect candidates for an interval guard. The type checker ensures that the interval guard must have completed before the object is placed into the queue, and so all shared paths are immutable.

Lock Guards: Because they must be modified repeatedly by concurrent intervals, the shared queue and field storing the shortest path are protected by a lock.

Custom Guards: A custom guard is used for the field which stores the shortest path length. This guard allows access from any thread. This guard permits data races, but as the field is used merely a heuristic, this cannot affect the program's overall correctness. Because this field is read very frequently, acquiring a lock for every access would be far too expensive.

6. Related Work

6.1 Lock-based Type Systems

The closest ancestor to our system are the type systems for enforcing the consistent use of locks found in systems like RccJava [2], Cyclone [15], or SafeJava [7]. Our work generalizes these approaches to support a variety of data-race protection schemes, rather than merely locks. We also integrate the *happens-before* relation into the type system, allowing us to check lock-free programs as well. One of the effects of this is that we do not need special case treatment for thread-local data or immutable objects.

Another difference between our work and earlier systems is our treatment of ghost fields. All of the systems cited above used a dependent type system of some kind that allowed types to be parameterized by objects. Most of these systems were modeled on traditional generic types, only with objects substituted for types. We initially tried this approach but found that it interacted poorly with other Java features, particularly generic types. Inherited ghost fields as presented here were designed to circumvent the problems we encountered.

Figure 16 demonstrates the most severe problem we encountered with a parametric system, which concerned interfaces like `Comparable<T>`. Annotating such an interface in our current design is straightforward (as shown). In a parameterization-based system, however, there is no way to

```
// The inherited ghost field system we use
// can easily refer to a ghost field of o:
interface Comparable<T> {
  int compareTo(T o)
  requires this.Wr permitsRd method
  requires o.Wr permitsRd method;
}

// A traditional, parameterization-based approach
// cannot express that OWr is associated with o:
interface Comparable[Wr]<T> {
  [OWr] int compareTo(T o)
  requires Wr permitsRd method
  requires OWr permitsRd method;
}
```

Figure 16. The difference between guards inherited by default and those that must be passed explicitly through supertypes.

refer to the ghost fields of the parameter `o` unless type variables like `T` can themselves be parameterized (in most systems, only classes can have type parameters).

6.2 Effect Systems

Many projects have applied effect systems towards detecting data races [6, 14, 18, 25], including prior work by the current authors [20]. Effect systems have the characteristic that each method summarizes the regions of memory that it might access as part of its signature. When multiple threads are executed, the system must guarantee that the regions affected by different threads are disjoint. If they are not, an error is reported. However, and this is the key difference with our system, neither of the conflicting routines is wrong in and of itself. It is only their composition that is incorrect. In our system, in contrast, at least one of those conflicting routines must be accessing fields without permission from the guard. Therefore, each routine is only concerned about proving that it itself is safe, and not for what others might do.

Both approaches have their advantages. An effect system can theoretically permit greater re-use, assuming the various challenges towards effectively modularizing and abstracting effect and region declarations are overcome. Our system, on the other hand, (a) has no need of an alias analysis and (b) supports localized dynamic "escape hatches," which we view as an essential feature. In an effect system, in contrast, a method cannot check dynamically whether an access is conflicting unless the other potentially concurrent methods also use dynamic checks (otherwise the necessary information is not present at runtime).

6.3 Other Static Analyses

A number of whole program analysis techniques [11, 24] have been applied to data-race detection. Such analyses generally must process a large codebase with little to no annotation, so the speed of the analysis and the rate of false positives is often the primary concern. Our work in contrast is intended to be a modular type system that programmers use while writing parallel programs.

Rather than detecting data races, some work focuses on enforcing higher-level properties such as method atomicity [13] or higher-level data races [5, 9, 17, 31]. Our use of guards helps to avoid high-level data races, because related fields tend to be protected by the same guard and thus modified atomically.

SharC [4] defines annotations that can be used on C structures to identify thread-local, read-only, or shared data. They employ a simple means of parameterization, but for complex cases rely on a dynamic monitoring system.

Jade [29] and Serialization Sets [3] are two projects which use programmer-provided specifications to dynamically parallelize a program. Their specifications resemble the ones which we use, but their purpose is quite different.

Fractional permissions [8, 30] have been used as an alternative to effects for detecting races. Because fractional permissions allow exclusive access to be parceled out to many threads but later reclaimed, they can handle objects which are only temporarily shared: however, they are limited by the ability of the compiler to pair up which threads are forked and joined.

Delayed types [12] introduced a notion of time with the aim of addressing object initialization. In their system, each object has an associated time by which it will be fully initialized, similar to the `Constructor` interval discussed in Section 3.2. Their type system is otherwise quite different from ours. Times are not first-class objects and cannot be used to create parallelism. Furthermore, times are not exposed to the user but rather introduced by the type system in a stylized fashion.

Constrained types [26] allow users to annotate types with constraints that specify what values their fields possess or other criteria. These constraints are very similar to our wildcard type annotations, but applied to more general constraints such as the range of values an integer may possess. Their constraints also apply to reified fields, whereas ours are currently limited to ghost fields. Their work and ours complement each other and could be fruitfully combined.

Our use of inherited ghost fields is reminiscent of virtual types, an alternative to parametric polymorphism supported by languages such as Scala [27] and Beta [19]. In these languages, a class can declare an abstract type member that is inherited by subtypes.

7. Conclusion

Despite its importance, time is generally a second-class citizen in programming languages. It is possible to tell the compiler the types of a method's arguments, but not when it should be invoked. Similarly, the compiler knows the type for each field, but not when it will be initialized. At best, a language may offer ad hoc techniques such as constructors

or `final` fields, which encode a few common patterns but cannot be generalized.

In this paper, we have presented an approach for giving users the ability to name and describe time in their programs. Our system unifies sequential and parallel control flow into one construct, the interval. Through a static type system, we allow users to choose when and under what conditions a field should be modified or a method should be invoked. We have proven that our type system protects against data races. We demonstrate that these same techniques can be reapplied to to enforce phases in an object's lifetime.

We have focused on creating a practical system. Tools like ghost fields and method requirements help users enforce their desired protocols with zero runtime and memory overhead. Similarly, because no type system can describe all programs, we allow users to insert checked assertions about the program schedule. As demonstrated in our experience section, however, the type system is expressive enough that such checks are rarely needed.

References

[1] `http://intervals.inf.ethz.ch`.

[2] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2), 2006.

[3] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization sets: a dynamic dependence-based parallel execution model. In *PPoPP*. ACM, 2009. ISBN 978-1-60558-397-6.

[4] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: checking data sharing strategies for multithreaded C. In *PLDI*. ACM, 2008.

[5] N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of atomic blocks and typestate. *SIGPLAN Not.*, 43(10), 2008.

[6] 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 *OOPSLA*. ACM, 2009.

[7] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*. ACM, 2002. ISBN 1-58113-471-1.

[8] J. Boyland. Checking Interference with Fractional Permissions. In R. Cousot, editor, *Static Analysis: 10th International Symposium*, volume 2694 of *LNCS*. Springer, 2003.

[9] L. Ceze, C. von Praun, C. Caşcaval, P. Montesinos, and J. Torrellas. Concurrency control with data coloring. In *MSPC*. ACM, 2008. ISBN 978-1-60558-049-4.

[10] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI*. USENIX Association, 2004.

[11] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.*, 37(5):237–252, 2003. ISSN 0163-5980.

[12] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA*. ACM, 2007.

[13] C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for Java. *ACM Trans. Program. Lang. Syst.*, 30(4), 2008. ISSN 0164-0925.

[14] A. Greenhouse and W. L. Scherlis. Assuring and evolving concurrent programs: annotations and policy. In *ICSE*. ACM, 2002.

[15] D. Grossman. Type-safe Multithreading in Cyclone. In *TLDI*. ACM, 2003.

[16] R. H. Halstead, Jr. MULTILISP: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7 (4), 1985. ISSN 0164-0925.

[17] B. Jacobs, F. Piessens, K. R. M. Leino, and W. Schulte. Safe Concurrency for Aggregate Objects with Invariants. In *SEFM*. IEEE Computer Society, 2005.

[18] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL*. ACM, 1988.

[19] O. L. Madsen and B. Møller-Pedersen. Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA*. ACM, 1989.

[20] N. D. Matsakis and T. R. Gross. Thread Safety through Partitions and Effect Agreements. In *LCPC*, 2008.

[21] N. D. Matsakis and T. R. Gross. Programming with Intervals. In *LCPC*, 2009.

[22] N. D. Matsakis and T. R. Gross. Handling Errors in Parallel Programs Based on Happens Before Relations. In *HIPS*, 2010.

[23] N. D. Matsakis and T. R. Gross. Reflective Parallel Programming. In *HotPar*, 2010.

[24] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. *SIGPLAN Not.*, 41(6):308–319, 2006. ISSN 0362-1340.

[25] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis. Contextual effects for version-consistent dynamic software updating and safe concurrent programming. *SIGPLAN Not.*, 43(1), 2008.

[26] N. Nystrom, V. Saraswat, J. Palsberg, and C. Grothoff. Constrained types for object-oriented languages. In *OOPSLA*. ACM, 2008.

[27] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *Proc. FOOL 10*, Jan. 2003.

[28] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 2008.

[29] M. Rinard. *The Design, Implementation and Evaluation of Jade, a Portable, Implicitly Parallel Programming Language*. PhD thesis, Dept. of CS, Stanford University, 1994.

[30] T. Terauchi. Checking race freedom via linear programming. In *PLDI*. ACM, 2008.

[31] M. Vaziri, F. Tip, and J. Dolby. Associating synchronization constraints with data in an object-oriented language. In *POPL*. ACM, 2006. ISBN 1-59593-027-2.

A. Complete Typing Rules for Inter

A.1 Lexical Conventions

- The terminals `c`, `f`, `m`, and `x` stand for class, field, method and local variable names, respectively. Non-terminals are shown in *italics*.
- Sequences are indicated by a trailing `s`. Unless separated by a semicolon (;), adjacent sequences indicate a sequence of tuples. Sequences are a function from their index to their items. Therefore, $\texttt{xs}(i)$ indicates the ith variable in the sequence. We use the notation $(F(i)|i)$ to mean a sequence whose items are $F(1), F(2)$, etc.
- The notation $\Theta = [\texttt{x} \to \mathit{path}]$ defines a substitution function Θ which replaces instances of the variable `x` with the path *path*. Θ can be applied to types ($\Theta(\mathit{ty})$) or method requirements ($\Theta(\mathit{req})$).
- The notation $\mathcal{E} \vdash \mathit{lstmts} \xrightarrow{\mathit{fold}} \mathcal{E}_N$ indicates apply the judgement $\mathcal{E} \vdash \mathit{lstmt} \to \mathcal{E}'$ to each statement in *lstmts* in turn. The initial environment is $\mathcal{E}$. Thereafter, the result $\mathcal{E}'$ from each statement is used as the initial environment for its successor. $\mathcal{E}_N$ represents the final environment.

A.2 Summary of Judgements

$\mathcal{E} \vdash \mathit{path} : \mathit{ty}$	*path* has type *ty* in $\mathcal{E}$
$\mathcal{E} \vdash \mathit{path}$ `stable`	Value of *path* cannot change
$\mathcal{E} \vdash \mathit{path}$ `stableBy` $\texttt{x}_i$	Value of *path* cannot change once interval $\texttt{x}_i$ starts
$\mathcal{E} \vdash \mathit{ty}$ `stableBy` $\texttt{x}_i$	Type *ty* only references paths that are stable during $\texttt{x}_i$.
$\mathcal{E} \vdash \mathit{req}$ `stableBy` $\texttt{x}_i$	Requirement *req* only references paths that are stable during $\texttt{x}_i$.
$\mathcal{E} \vdash \mathit{req}$	*req* = *path rel path* is supported by $\mathcal{E}$.
`c : OK`	Definition of class `c` is sound.
$\mathcal{E} \vdash \mathit{ldecl}$ `: OK`	Lock declaration *ldecl* is sound in $\mathcal{E}$.
$\mathcal{E} \vdash \mathit{hbdecl}$ `: OK`	*Happens-before* declaration *hbdecl* is sound in $\mathcal{E}$.
$\mathcal{E} \vdash \mathit{fdecl}$ `: OK`	Field declaration *fdecl* is sound in $\mathcal{E}$.
$\mathcal{E} \vdash \mathit{mdecl}$ `: OK`	Method declaration *mdecl* is sound in $\mathcal{E}$.
$\mathcal{E} \vdash (\mathit{tys}$ `xs`$)$ *reqs* `overrides` *mdecl*	Override of *mdecl* with signature $(\mathit{tys}$ `xs`$)$ *reqs* is sound.
$\mathcal{E} \vdash \mathit{lstmt} \to \mathcal{E}'$	Labeled statement *lstmt* is sound in $\mathcal{E}$.
	The next statement should be checked in $\mathcal{E}'$.

A.3 Helper Functions

- The functions *gdecls*(`c`), *fdecls*(`c`), *idecls*(`c`), *hbdecls*(`c`), and *ldecls*(`c`) return the corresponding declarations from the body of class `c`.
- *reified*(`c`, `f`) = (*ty*; *path*) yields the type *ty* and guard path *path* for the field `f` defined in the class `c` or a superclass. No substitutions are performed. If `f` is a constructor argument or interval member, then the special variable `final` is returned as its guard path.
- *signature*(`c`, `m`) = $(\mathit{tys}$ `xs`$)$ *reqs* yields the parameters $(\mathit{tys}$ `xs`$)$ and requirements *reqs* declared for the class `m` in the class `c` (or a superclass, if `c` does not override `m`). No substitutions are performed.
- *overrides*(`c`, `m`) = *mdecls* yields the list of method declarations with name `m` from superclasses of `c`.

A.4 Typing Paths

$$\text{T-VAR}\ \frac{(\texttt{x} : \mathit{ty}) \in \mathcal{E}}{\mathcal{E} \vdash \texttt{x} : \mathit{ty}} \qquad \text{T-FIELD}\ \frac{\mathcal{E} \vdash \mathit{path} : \texttt{c} \qquad \mathit{reified}(\texttt{c}, \texttt{f}) = (\mathit{ty}; \ldots)}{\mathcal{E} \vdash \mathit{path}.\texttt{f} : [\texttt{this} \to \mathit{path}]\,\mathit{ty}}$$

$$\text{T-SUB}\ \frac{\mathcal{E} \vdash \mathit{path} : \texttt{c}_{\mathit{sub}} \qquad \texttt{c}_{\mathit{sub}}\ \texttt{is a subclass of}\ \texttt{c}_{\mathit{sup}} \qquad \forall i.\, \mathcal{E} \vdash \mathit{path}.\texttt{fs}(i)\ \mathit{wcrels}(i)\ \mathit{paths}(i)}{\mathcal{E} \vdash \mathit{path} : \texttt{c}_{\mathit{sup}}[\texttt{fs}\ \mathit{wcrels}\ \mathit{paths}]}$$

A.5 Stability

PATH-STABLE-ALL
$$\frac{\mathtt{x}_i \textit{ fresh} \qquad \mathcal{E} \vdash \textit{path}\ \mathtt{stableBy}\ \mathtt{x}_i}{\mathcal{E} \vdash \textit{path}\ \mathtt{stable}}$$

PATH-STABLE-LV
$$\frac{(\mathtt{x} : \textit{ty}) \in \mathcal{E}}{\mathcal{E} \vdash \mathtt{x}\ \mathtt{stableBy}\ \mathtt{x}_i}$$

PATH-STABLE-GHOST
$$\frac{\mathcal{E} \vdash \textit{path}_o\ \mathtt{stableBy}\ \mathtt{x}_i \qquad \mathcal{E} \vdash \textit{path}_o : \mathtt{c} \qquad \mathtt{f} \in \textit{ghosts}(\mathtt{c})}{\mathcal{E} \vdash \textit{path}_o.\mathtt{f}\ \mathtt{stableBy}\ \mathtt{x}_i}$$

PATH-STABLE-REIFIED
$$\frac{\mathcal{E} \vdash \textit{path}_o : \mathtt{c} \qquad \textit{reified}(\mathtt{c}, \mathtt{f}) = (\ldots; \textit{path}_g) \qquad \Theta = [\mathtt{this} \to \textit{path}_o] \qquad \mathcal{E} \vdash \Theta(\textit{path}_g)\ \mathtt{stableBy}\ \mathtt{x}_i \qquad \mathcal{E} \vdash \Theta(\textit{path}_g)\ \mathtt{ensuresImm}\ \mathtt{x}_i}{\mathcal{E} \vdash \textit{path}_o.\mathtt{f}\ \mathtt{stableBy}\ \mathtt{x}_i}$$

TY-STABLE
$$\frac{\mathtt{fs} \subseteq \textit{ghosts}(\mathtt{c}) \qquad \forall i.\, \mathcal{E} \vdash \textit{paths}(i)\ \mathtt{stableBy}\ \mathtt{x}_i}{\mathcal{E} \vdash \mathtt{c}[\mathtt{fs}\ \textit{wcrels}\ \textit{paths}]\ \mathtt{stableBy}\ \mathtt{x}_i}$$

REQ-STABLE
$$\frac{\mathcal{E} \vdash \textit{path}_1\ \mathtt{stableBy}\ \mathtt{x}_i \qquad \mathcal{E} \vdash \textit{path}_2\ \mathtt{stableBy}\ \mathtt{x}_i}{\mathcal{E} \vdash (\textit{path}_1\ \textit{rel}\ \textit{path}_2)\ \mathtt{stableBy}\ \mathtt{x}_i}$$

A.6 Relations (All)

REL-ENV
$$\frac{(\textit{path}_1\ \textit{rel}\ \textit{path}_2) \in \mathcal{E}}{\mathcal{E} \vdash \textit{path}_1\ \textit{rel}\ \textit{path}_2}$$

REL-TRANS
$$\frac{\mathcal{E} \vdash \textit{path}_1\ \textit{trel}\ \textit{path}_2 \qquad \mathcal{E} \vdash \textit{path}_2\ \textit{trel}\ \textit{path}_3}{\mathcal{E} \vdash \textit{path}_1\ \textit{trel}\ \textit{path}_3}$$

REL-WILDCARD
$$\frac{\mathcal{E} \vdash \textit{path}_1 : \mathtt{c}[\ldots, \mathtt{f}\ \textit{wcrel}\ \textit{path}_2, \ldots]}{\mathcal{E} \vdash \textit{path}_1.\mathtt{f}\ \textit{wcrel}\ \textit{path}_2}$$

A.7 Relations (eq)

EQ-SELF
$$\mathcal{E} \vdash \textit{path}\ \mathtt{eq}\ \textit{path}$$

EQ-SYMMETRIC
$$\frac{\mathcal{E} \vdash \textit{path}_2\ \mathtt{eq}\ \textit{path}_1}{\mathcal{E} \vdash \textit{path}_1\ \mathtt{eq}\ \textit{path}_2}$$

EQ-EXTEND
$$\frac{\mathcal{E} \vdash \textit{path}_1\ \mathtt{eq}\ \textit{path}_2}{\mathcal{E} \vdash \textit{path}_1.\mathtt{f}\ \mathtt{eq}\ \textit{path}_2.\mathtt{f}}$$

EQ-REL
$$\frac{\mathcal{E} \vdash \textit{path}_1\ \mathtt{eq}\ \textit{path}_1' \qquad \mathcal{E} \vdash \textit{path}_2\ \mathtt{eq}\ \textit{path}_2' \qquad \mathcal{E} \vdash \textit{path}_1'\ \textit{rel}\ \textit{path}_2'}{\mathcal{E} \vdash \textit{path}_1\ \textit{rel}\ \textit{path}_2}$$

A.8 Relations (hb)

HB-SUB-LEFT
$$\frac{\mathcal{E} \vdash \textit{path}_\mathtt{c}\ \mathtt{subOf}\ \textit{path}_p \qquad \mathcal{E} \vdash \textit{path}_p\ \mathtt{hb}\ \textit{path}}{\mathcal{E} \vdash \textit{path}_\mathtt{c}\ \mathtt{hb}\ \textit{path}}$$

HB-SUB-RIGHT
$$\frac{\mathcal{E} \vdash \textit{path}_\mathtt{c}\ \mathtt{subOf}\ \textit{path}_p \qquad \mathcal{E} \vdash \textit{path}\ \mathtt{hb}\ \textit{path}_p}{\mathcal{E} \vdash \textit{path}\ \mathtt{hb}\ \textit{path}_\mathtt{c}}$$

A.9 Relations (all guards)

RBY-WBY
$$\frac{\mathcal{E} \vdash \textit{path}_g\ \mathtt{permitsWr}\ \textit{path}_i}{\mathcal{E} \vdash \textit{path}_g\ \mathtt{permitsRd}\ \textit{path}_i}$$

RBY-IMMUTABLE
$$\frac{\mathcal{E} \vdash \textit{path}_g\ \mathtt{ensuresImm}\ \textit{path}_i}{\mathcal{E} \vdash \textit{path}_g\ \mathtt{permitsRd}\ \textit{path}_i}$$

WBY-INLINE
$$\frac{\mathcal{E} \vdash \textit{path}_i\ \mathtt{inlineSubOf}\ \textit{path}_p \qquad \mathcal{E} \vdash \textit{path}_g\ \mathtt{permitsWr}\ \textit{path}_p}{\mathcal{E} \vdash \textit{path}_g\ \mathtt{permitsWr}\ \textit{path}_i}$$

RBY-SUB
$$\frac{\mathcal{E} \vdash \textit{path}_i\ \mathtt{subOf}\ \textit{path}_p \qquad \mathcal{E} \vdash \textit{path}_g\ \mathtt{permitsRd}\ \textit{path}_p}{\mathcal{E} \vdash \textit{path}_g\ \mathtt{permitsRd}\ \textit{path}_i}$$

A.10 Relations (built-in guards)

WBY-INTERVAL
$$\mathcal{E} \vdash \textit{path}_g\ \mathtt{permitsWr}\ \textit{path}_g$$

IMM-INTERVAL
$$\frac{\mathcal{E} \vdash \textit{path}_g\ \mathtt{hb}\ \textit{path}_i}{\mathcal{E} \vdash \textit{path}_g\ \mathtt{ensuresImm}\ \textit{path}_i}$$

IMM-FINAL
$$\mathcal{E} \vdash \mathtt{final}\ \mathtt{ensuresImm}\ \textit{path}_i$$

WBY-LOCK
$$\frac{\mathcal{E} \vdash \textit{path}_i\ \mathtt{locks}\ \textit{path}_g}{\mathcal{E} \vdash \textit{path}_g\ \mathtt{permitsWr}\ \textit{path}_i}$$

A.11 Declarations

CLASS-OK

$$\frac{\begin{array}{c}\mathit{idecls}(\mathtt{c}) = \mathtt{interval\ fs}(\mathit{paths})\ \{\ \ldots\ \} \\ \mathcal{E}_\mathtt{c} = [\mathtt{this} : \mathtt{c}] + \mathit{ldecls}(\mathtt{c}) + \mathit{hbdecls}(\mathtt{c}) + (\mathtt{this.fs}(i)\ \mathtt{subOf}\ \mathit{paths}(i) \mid i) \\ \forall i.\ \mathcal{E}_\mathtt{c};\ \mathtt{c} \vdash \mathit{hbdecls}(\mathtt{c})(i) : \mathtt{OK} \\ \forall i.\ \mathcal{E}_\mathtt{c};\ \mathtt{c} \vdash \mathit{ldecls}(\mathtt{c})(i) : \mathtt{OK} \quad \forall i.\ \mathcal{E}_\mathtt{c};\ \mathtt{c} \vdash \mathit{fdecls}(\mathtt{c})(i) : \mathtt{OK} \quad \forall i.\ \mathcal{E}_\mathtt{c};\ \mathtt{c} \vdash \mathit{mdecls}(\mathtt{c})(i) : \mathtt{OK} \quad \forall i.\ \mathcal{E}_\mathtt{c};\ \mathtt{c} \vdash \mathit{idecls}(\mathtt{c})(i) : \mathtt{OK}\end{array}}{\mathtt{c} : \mathtt{OK}}$$

LOCK-DECL-OK

$$\frac{\mathcal{E} \vdash \mathit{path}_i\ \mathtt{stable} \quad \mathcal{E} \vdash \mathit{path}_l\ \mathtt{stable} \quad \mathcal{E} \vdash \mathit{path}_i : \mathtt{Interval} \quad \mathcal{E} \vdash \mathit{path}_l : \mathtt{Lock}}{\mathcal{E};\ \mathtt{c} : \mathit{path}_i\ \mathtt{locks}\ \mathit{path}_l\ \mathtt{OK}}$$

HB-DECL-OK

$$\frac{\mathcal{E} \vdash \mathit{path}_i\ \mathtt{stable} \quad \mathcal{E} \vdash \mathit{path}_l\ \mathtt{stable} \quad \mathcal{E} \vdash \mathit{path}_i : \mathtt{Interval} \quad \mathcal{E} \vdash \mathit{path}_l : \mathtt{Interval}}{\mathcal{E};\ \mathtt{c} : \mathit{path}_i\ \mathtt{hb}\ \mathit{path}_l\ \mathtt{OK}}$$

FIELD-OK

$$\frac{\mathcal{E} \vdash \mathit{path}\ \mathtt{stable} \quad \mathcal{E} \vdash \mathit{ty}\ \mathtt{stable}}{\mathcal{E} \vdash \mathit{ty}\ \mathtt{f\ guardedBy}\ \mathit{path} : \mathtt{OK}}$$

INTERVAL-OK

$$\frac{\mathcal{E} \vdash \mathit{path}\ \mathtt{stable} \quad \mathcal{E} \vdash \mathit{path} : \mathtt{Interval} \quad \mathtt{this.f};\ \mathcal{E} \vdash \mathit{lstmts} : \mathtt{OK}}{\mathcal{E};\ \mathtt{c} \vdash \mathtt{interval\ f}(\mathit{path})\ \{\ \mathit{lstmts}\ \} : \mathtt{OK}}$$

METHOD-OK

$$\frac{\begin{array}{c}\mathcal{E}_1 + (\mathtt{method} : \mathtt{Interval}) + (\mathtt{xs}(i) : \mathit{tys}(i) \mid i) + \mathit{reqs} = \mathcal{E}_\mathtt{m} \\ \forall i.\ \mathcal{E}_\mathtt{m} \vdash \mathit{reqs}(i)\ \mathtt{stableBy\ method} \quad \forall i.\ \mathcal{E}_\mathtt{m} \vdash \mathit{tys}(i)\ \mathtt{stableBy\ method} \\ \mathit{overrides}(\mathtt{c}, \mathtt{m}) = \mathit{methods}_o \quad \forall i.\ \mathcal{E}_\mathtt{m} \vdash (\mathit{tys}\ \mathtt{xs})\ \mathit{reqs}\ \mathtt{overrides}\ \mathit{methods}_o(i) \\ \mathtt{method};\ \mathcal{E}_\mathtt{m} \vdash \mathit{lstmts} : \mathtt{OK}\end{array}}{\mathcal{E}_1 \vdash \mathtt{void\ m}(\mathit{tys}\ \mathtt{xs})\ \mathit{reqs}\ \{\ \mathit{lstmts}\ \} : \mathtt{OK}}$$

METHOD-OVERRIDE-OK

$$\frac{\Theta = [\ \mathtt{xs}_p \to \mathtt{xs}_b\ ] \quad \forall i.\ \mathcal{E} + \Theta(\mathit{reqs}_p) \vdash \mathit{reqs}_b(i) \quad \forall i.\ \Theta(\mathit{tys}_p(i)) = \mathit{tys}_b(i)}{\mathcal{E} \vdash (\mathit{tys}_b\ \mathtt{xs}_b)\ \mathit{reqs}_b\ \mathtt{overrides\ void\ m}(\mathit{tys}_p\ \mathtt{xs}_p)\ \mathit{reqs}_p\ \{\ \ldots\ \}}$$

A.12 Statements

LSTMTS-OK

$$\frac{\begin{array}{c}\mathit{lstmts} = (\mathtt{xs} : \ldots) \\ \mathcal{E}_2 = \mathcal{E}_1 + (\mathtt{xs}(i) : \mathtt{Interval} \mid i) + (\mathtt{xs}(i)\ \mathtt{inlineSubOf}\ \mathit{path}_{par} \mid i) + (\mathtt{xs}(i-1)\ \mathtt{hb\ xs}(i) \mid i) \\ \mathcal{E}_2 \vdash \mathit{lstmts} \xrightarrow{\mathit{fold}} \mathcal{E}_N\end{array}}{\mathit{path}_{par};\ \mathcal{E}_1 \vdash \mathit{lstmts} : \mathtt{OK}}$$

STMT-STORE

$$\frac{\mathcal{E} \vdash \mathtt{x}_o : \mathtt{c} \quad \mathit{reified}(\mathtt{c}, \mathtt{f}) = (\mathit{ty}_\mathtt{f};\ \mathit{path}_g) \quad \Theta = [\mathtt{this} \to \mathtt{x}_o] \quad \mathcal{E} \vdash \Theta(\mathit{path}_g)\ \mathtt{stableBy\ x}_l \quad \mathcal{E} \vdash \Theta(\mathit{path}_g)\ \mathtt{permitsWr\ x}_l \quad \mathcal{E} \vdash \mathtt{x}_v : \Theta(\mathit{ty}_\mathtt{f})}{\mathcal{E} \vdash \mathtt{x}_l : (\mathtt{x}_o.\mathtt{f} = \mathtt{x}_v) \to \mathcal{E}}$$

STMT-LOAD

$$\frac{\mathcal{E} \vdash \mathtt{x}_o : \mathtt{c} \quad \mathit{reified}(\mathtt{c}, \mathtt{f}) = (\mathit{ty}_\mathtt{f};\ \mathit{path}_g) \quad \Theta = [\mathtt{this} \to \mathtt{x}_o] \quad \mathcal{E} \vdash \Theta(\mathit{path}_g)\ \mathtt{stableBy\ x}_l \quad \mathcal{E} \vdash \Theta(\mathit{path}_g)\ \mathtt{permitsRd\ x}_l}{\mathcal{E} \vdash \mathtt{x}_l : (\mathtt{x}_d = \mathtt{x}_o.\mathtt{f}) \to \mathcal{E} + (\mathtt{x}_d : \Theta(\mathit{ty}_\mathtt{f}))}$$

STMT-LOAD-IMMUTABLE

$$\frac{\mathcal{E} \vdash \mathtt{x}_o : \mathtt{c} \quad \mathit{reified}(\mathtt{c}, \mathtt{f}) = (\mathit{ty}_\mathtt{f};\ \mathit{path}_g) \quad \Theta = [\mathtt{this} \to \mathtt{x}_o] \quad \mathcal{E} \vdash \Theta(\mathit{path}_g)\ \mathtt{stableBy\ x}_l \quad \mathcal{E} \vdash \Theta(\mathit{path}_g)\ \mathtt{ensuresImm\ x}_l}{\mathcal{E} \vdash \mathtt{x}_l : (\mathtt{x}_d = \mathtt{x}_o.\mathtt{f}) \to \mathcal{E} + (\mathtt{x}_d : \Theta(\mathit{ty}_\mathtt{f})) + (\mathtt{x}_d\ \mathtt{eq}\ \mathtt{x}_o.\mathtt{f})}$$

STMT-NEW

$$\frac{\forall i.\ \mathcal{E} \vdash \mathit{paths}(i)\ \mathtt{stableBy\ x}_l \quad \mathcal{E} + (\mathtt{x}_d : \mathtt{c}[\mathtt{fs\ eq}\ \mathit{paths}]) = \mathcal{E}' \quad \mathit{ctor}(\mathtt{c}) = \mathit{tys}_\mathtt{c} \quad \forall i.\ \mathcal{E}' \vdash \mathtt{xs}_a(i) : [\mathtt{this} \to \mathtt{x}_d]\,\mathit{tys}_\mathtt{c}(i)}{\mathcal{E} \vdash \mathtt{x}_l : (\mathtt{x}_d = \mathtt{new\ c}[\mathtt{fs\ eq}\ \mathit{paths}])\ (\ \mathtt{xs}_a\ ) \to \mathcal{E}'}$$

STMT-MTHDCALL

$$\frac{\begin{array}{c}\mathcal{E} \vdash \mathtt{x}_r : \mathtt{c} \quad \mathit{signature}(\mathtt{c}, \mathtt{m}) = (\mathit{tys}_\mathtt{m}\ \mathtt{xs}_\mathtt{m})\ \mathit{reqs}_\mathtt{m} \\ \Theta = [\mathtt{this} \to \mathtt{x}_r, \mathtt{method} \to \mathtt{x}_l, \mathtt{xs}_\mathtt{m} \to \mathtt{xs}_a] \quad \forall i.\ \mathcal{E} \vdash \Theta(\mathit{reqs}_\mathtt{m}(i)) \quad \forall i.\ \mathcal{E} \vdash \mathtt{xs}_a(i) : \Theta(\mathit{tys}_\mathtt{m}(i))\end{array}}{\mathcal{E} \vdash \mathtt{x}_l : (\mathtt{x}_r.\mathtt{m}(\mathtt{xs}_a)) \to \mathcal{E}}$$

STMT-ASSERT

$$\frac{\mathcal{E} \vdash \mathtt{x}_g : \mathtt{Guard} \quad \mathcal{E} \vdash \mathtt{x}_i : \mathtt{Interval}}{\mathcal{E} \vdash \mathtt{x}_l : (\mathtt{assert\ x}_g\ \mathit{wcrel}\ \mathtt{x}_i) \to \mathcal{E} + (\mathtt{x}_g\ \mathit{wcrel}\ \mathtt{x}_i)}$$