

Thread Safety Through Partitions and Effect Agreements

Nicholas D. Matsakis
ETH Zurich
nicholas.matsakis@inf.ethz.ch

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

ABSTRACT

This paper describes a type and effect system for object-oriented programs which allows programmers to divide their heap into disjoint *partitions* at a field-level granularity, analyze the effects of common statements upon these partitions, and then use this information to enforce safety properties in their programs. Our flow-sensitive effect system requires methods to disclose which partitions of the heap they will read or write, and also allows them to specify an *effect agreement* which can be used to limit the conditions in which a method can be called. To show the flexibility of such a system for enforcing semantic constraints, we describe a safe multi-threading system with support for many common threading patterns, including unsynchronized access to thread-local and shared read-only data, as well as limited migration of data between threads.

1. INTRODUCTION

Widespread use of aliases in object-oriented programs can cause many difficulties for programmers. Aliases create invisible links between objects; seemingly innocuous changes made to an object in one context may affect other objects in unexpected ways.

Aliasing becomes an even bigger problem when combined with concurrency. If the synchronization discipline in the program is not used consistently, then aliases make it possible for an object to change during the execution of a method even when (from the point of view of the thread executing the method) no methods have been invoked on it.

Mainstream programming languages provide no automated means for preventing or controlling aliasing. Access modifiers like Java’s `public` and `private` can be used to control access to a field, but do nothing to regulate which objects are placed into the field.

Numerous researchers have devised mechanisms to fill this gap. One approach is to enforce the notion of encapsulation directly in the type system. Encapsulation boundaries can be defined in various ways, ranging from the class [1] or package level [2] all the way down to individual objects [3, 4, 5, 6]. This stronger notion of encapsulation can also be used to make much stronger guarantees in the type system, such as that a program obeys a consistent lock discipline and is deadlock-free [7]. Another approach is to annotate references with capabilities [8, 9] which control what can be done with that reference.

This paper attempts to refine previous approaches by providing generic mechanisms for aliasing control in a layered fashion. The contributions of the paper are as follows:

1. We present an abstraction, *partitions*, which can be used by programmers to declare possible aliasing in their programs, and describe a polymorphic type system for checking that partitions are respected.
2. We introduce *effect agreements*, which allow methods to prohibit specified events from happening either before or after the method is invoked. Effect agreements are checked using a flow-sensitive effect system that operates at a partition granularity.
3. To demonstrate the expressiveness of effect agreements, we describe how they can be used to ensure that multi-threaded programs follow safe conventions which guarantee the program is free of race conditions.

The paper begins with a brief overview of our approach, introducing partitions in more detail and also describing our effect system. After a brief example, we then delve into the details of our type system and effect checking algorithm. Finally, we show how to use the effect mechanisms to enforce thread safety.

2. PARTITIONS AT A GLANCE

A *partition* is a compile-time abstraction that describes a portion of the heap at a field-level granularity. In other words, if we define the heap $\mathcal{H}(o, f) \mapsto o$ as a mapping from an (object id, field) pairs to another object id, a partition is a set of such (object id, field) pairs. Partitions are similar to memory regions, except that we do not use them for memory management, but rather for alias tracking.

In our system, class and method definitions are parameterized by a set of *partition parameters*, similar to generic type variables. When the class is instantiated or the method is invoked, each partition parameter will be mapped to a fixed partition.

As a simple example, consider the following definition of the `IntWrapper` class:

```
class IntWrapper @P {  
  @P int field;  
  int get() { return field; }  
  void set(int i) { field = i; }  
}
```

The class has a single partition parameter, named `P`, which is indicated by the `@P` which follows the class name. The `@P` preceding the declaration for the field `field` indicates that the field is placed into the partition `P`. In addition to partition parameters, a partition can be defined by using standard set operations such as union and intersection to

```

class Class {
  void method() {
    new @P;
    IntWrapper@P x = new IntWrapper@P ();
    x.set(5);
  }
}

```

Figure 1: A method which creates a new partition, and places a new `IntWrapper` object inside it.

combine partition parameters into a new, anonymous partition. This is referred to as a *partition expression*, and is expressed with standard syntax like $P \cap Q$ or $P \cup Q$.

The syntax for parameterized method definitions and calls is similar to Java’s generic syntax:

```

@P void method(Object@P argument) { ... }
object.method@P(null);

```

The `@P` that appears before the return type is a partition parameter in method scope. When a method is invoked, the values of any partition parameters must be provided after the method name, as shown on the second line.

As with other parameterized type systems, when a class is instantiated or a method is invoked, concrete partitions must be provided for each partition parameter. In general, the partitions provided as values for each parameter to a method or class need not be disjoint; however, non-disjoint parameters may sometimes be rejected by the type checker if it would violate program safety properties.¹

2.1 Extent and Scope of a Partition

Although partitions are purely a compile-time construct, it is useful to examine how they would behave at runtime. Partitions are created by a `new @P` statement, as shown in Figure 1, but they are never explicitly destroyed.

When a partition is initially created, it contains no data and is disjoint from all other partitions. Data is added to a partition by creating new objects that contain fields located in that partition. In Figure 1, a new `IntWrapper` is constructed with `P` as its partition parameter. The field `field` of this instance will therefore be the first field placed into partition `P`, since `P` was freshly created on the line before.

Although partitions are initially disjoint, they can be made to intersect by creating data that exists in their intersection. As an example, consider the following method:

```

1 @Q void method() {
2   new @P;
3   IntWrapper@(P∩Q) x =
4     new IntWrapper@(P∩Q)();
5 }

```

While `P` and `Q` were disjoint before, the `new` expression on line 4 creates a field that exists in their intersection, and which therefore belongs to both `P` and `Q`.

In general, a partition can be specified either through a partition parameter, as shown initially, or a partition expression, as in the previous example. Partition expressions define a new, anonymous partition; unlike named partitions which are initially empty, however, this partition’s contents are determined by the set operations used to define it. Anonymous partitions grow as their component partitions grow;

¹Potential intersection of partitions is tracked using the effect system. See Section 4.1 for details.

```

class IntWrapper @P {
  @P int field;
  !Rd(P) int get() { return field; }
  !Wr(P) void set(int i) { field = i; }
}

```

Figure 2: The class `IntWrapper` which was shown before, annotated with effects.

therefore, a partition defined by $P \cup Q$ always contains the contents of both `P` and `Q`, even as new items are added into `P`.

In addition to creating new data in the intersection, partitions can be made to intersect via *partition extension*. Partition extension adds data that exists in one partition into another partition, without removing it from the old partition. The data simply exists in both partitions. Of course, this requires conservative assumptions on the part of the type system, because any data in the two partitions is now indistinguishable. For this reason, partition extension is most useful when the data currently exists in a partition that will shortly be going out of scope.

2.2 Effects

Partitions allow the program’s data to be divided into distinct logical sections, but the effect system regulates how those partitions may be used. As in other object-oriented effect systems [10], programmers annotate methods to describe which partitions they affect and how.

Figure 2 shows the `IntWrapper` class defined earlier, but annotated with effects. Effect declarations precede each method declaration and are part of the method’s interface. `get()` is annotated with `!Rd(P)`, as it reads a field in partition `P`, while `set()` has a corresponding write effect.

The full set of effects we use in this paper are as follows:

1. `Rd(w)`: indicates that data in partition `w` is read.
2. `Wr(w)`: indicates that data in partition `w` is written.²
3. `ARd(w)`: indicates that data in partition `w` is read within an atomic block (defined in Section 3.1).
4. `AWr(w)`: indicates that data in partition `w` is written within an atomic block (defined in Section 3.1).
5. `Inter(w1, w2)`: indicates that the partitions `w1` and `w2` were made to intersect.

It is the programmer’s responsibility to declare the effects that a method may have. The effect checker statically ensures that the method body cannot affect any partition parameter in a way that is not declared. Note that it is not necessary to declare effects for partitions which the method creates. The method shown in Figure 1, for example, does not need to declare a write effect for partition `P`, because that partition is disjoint from all partition parameters.

2.3 Effect Agreements

While effect declarations allow a programmer to limit the effects a method may have, it is sometimes useful for a method to be able to limit the effects of its caller. For example, if a particular method is only sound if two partitions

²Unlike OOFX [10], write effects do not imply read effects.

P and Q are disjoint, it would be useful to declare that the method may only be invoked if no `Inter(P,Q)` effect has occurred (i.e., if P and Q have not been made to intersect).

To enable these sort of guarantees, methods may be annotated with *effect agreements* that constrain what can happen before the method is called or after it returns. These effects form a kind of agreement between the caller and the callee. We use a flow-sensitive effect checker to enforce them statically.

Agreements are subtly different from Design By Contract [11]. In DBC, methods declare *preconditions*, which the caller must guarantee for the callee to function properly, and *postconditions*, which the callee promises to bring about or maintain. In this way, contract obligations flow in both directions.

In contrast, all effect agreements are obligations the callee imposes on the caller. These obligations always take the form of effects which are not permitted. Each effect agreement has a *time span* that determines precisely when the events are not permitted to occur.

In this paper, we use two different time spans for effect agreements, `pre` and `par`. These two time spans are depicted graphically in Figure 3, and their meaning is explained here:

- `pre` agreements, written `pre -F(w)`³, indicate that the effect $F(\bar{w})$ must not have occurred before this method is called. In Figure 3a, the `pre` time span covers all points in time before the callee is invoked.
- `par` agreements, written `par -F(w)`, indicate that a parallel thread has started which requires that effects $F(w)$ do not occur during its execution. Generally, we do not know when a thread will finish executing, and therefore a `par` agreement would apply for all future time, as shown in Figure 3a.

In some cases, however, a thread’s lifetime can be bounded. For example, in Section 5, we describe a `forkjoin` keyword that can be attached to a statement block to guarantee that any threads which were (transitively) started during the statement block will be joined before the statement block finishes execution. Using such a `forkjoin` block enables us to limit the time that the `par` agreement applies. This is depicted in Figure 3b, where the caller knows that the thread will finish before or by the end of the `forkjoin` region.

To return to our original motivating example, to declare that its partition parameters P and Q must be disjoint, a method could use the effect agreement `pre -Inter(P,Q)`.

Effect agreements are considered binding on the current thread; however, a thread is also responsible for the behavior of any thread which it (transitively) starts. Therefore, if a thread T invokes a method which prohibits it from later writing to a partition, then T may not start another thread which writes to that same partition.

2.4 Effects and Inheritance

Because effects and effect agreements form part of the interface of a method, it is important to describe how they interact with inheritance. In accordance with the Liskov

³The reason we use a minus sign - and not a ! before the effect is to indicate that agreements describe effects which are forbidden, not permitted.

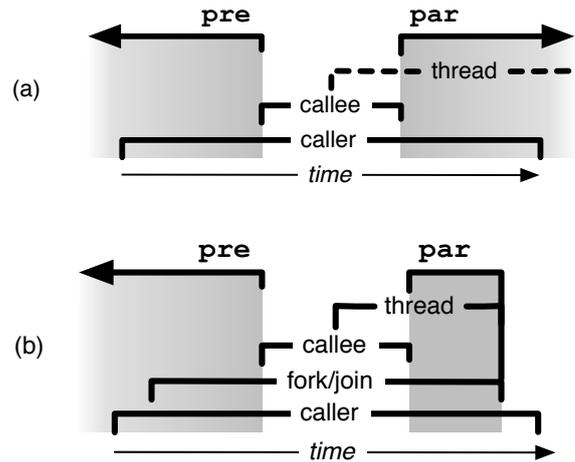


Figure 3: A timeline demonstrating the time spans of `pre` and `par` effect agreements in two different scenarios.

Substitution Principle [12], it must be safe for any subtype to be used where a supertype is expected. Therefore, an overloaded method in a subtype may not (a) have more effects than the method it overrides; or (b) prohibit an effect via an effect agreement which its supertype allows. The type checker verifies these conditions statically.

2.5 Example: Multithreaded Server

One common multithreaded application is a server. Figure 4 shows a simple server which has one thread listening for connections on a given port, defined by the class `ListenerThread`. When a connection arrives, the server initializes an object describing the new connection and creates a `HandlerThread` to handle it. The handler is given control of the connection object and started in parallel. From that point on, the connection object is considered thread-local data for the handler thread, and should not be used by the listening thread anymore. In this example, we show how the effect checker, combined with effect agreements, can be used to verify that the connection object is safely transferred to the new thread. In Section 5, we will expand the technique shown here into a more general solution.

The `ListenerThread` class does not have any partition parameters. Instead, within the `run()` method it creates a fresh partition, P, which contains the listening `Socket` instance. The `HandlerThread` class is parameterized by a single partition L for its thread-local data. In its `accept()` method, the `ListenerThread` creates a new partition H which it gives to each new `HandlerThread` to use for its thread-local data.

The error we are trying to prevent is that the `ListenerThread` continues to write to the partition H after it has started the `HandlerThread`. To prevent this, the `HandlerThread` has declared an effect agreement on its `start()` method which prohibits its local partition L (H, from the `ListenerThread`’s point of view) from being read or written. This agreement is given `par` scope so that it is in effect as long as the thread may execute.⁴

⁴A sharp-eyed reader will note that, because the `start()`

```

class ListenerThread extends Thread {
  public void run() {
    new @P;
    Socket@P socket = new Socket@P();
    while(true)
      accept@P(socket);
  }

  !Rd(P) !Wr(P)
  @P void accept(Socket@P socket) {
    new @H;
    Socket@H connection =
      socket.accept@H();
    init@H(connection);
    new HandlerThread@H(connection).start();
  }

  !Wr(H)
  @H void init(Socket@H conn) { ... }
}

class HandlerThread@L extends Thread {
  @L Socket@L connection;

  HandlerThread(Socket@L c) {
    connection = c;
  }

  !Rd(L) !Wr(L)
  par -Rd(L) -Wr(L)
  public void start() { ... }

  !Rd(L) !Wr(L)
  public void run() { ... }
}

```

Figure 4: The skeleton of a simple server which spawns a new thread to handle each incoming request.

To verify that effect agreements are respected, the effect checker will compute what effects may occur before and after each statement in the method. In this case, because the effect agreements in question are restrictions on what can happen *after* the `HandlerThread`'s start method returns, we are interested in the effects that can occur after each statement.

Figure 5 shows the `accept()` method of `ListenerThread`, annotated at each step with the set of future effects for each statement. Since we are computing what events are to come, the analysis is done starting at the end of the method and working backwards. Therefore, line 12 depicts the initial set of effects. Because there are no effect agreements declared on this method concerning `P`, we must make the conservative assumption (which happens to be true, in this case) that data in partition `P` may be both read and written in the future.

Note that there no conservative assumptions are needed for `H`. This is because it is newly created in this method. Data in `H` can only escape the method in two ways. The first is by intersection with a partition parameter, such as `P`: in this case, as we will see later, we would conservatively assume that any modifications to `P` may also affect `H`. The second is through an unbounded wildcard, which in effect is a type where some partitions are unspecified. This latter case does not present a problem because it is not allowed to

method is inherited from `Thread`, `HandlerThread` cannot add effect agreements in this fashion. We resolve this in Section 5 by modifying the thread class itself.

```

1  !Rd(L) !Wr(P)
2  @P void accept(Socket@P socket) {
3    // { Rd/Wr(P) }
4    new @H;
5    // { Rd/Wr(P) Rd/Wr(H) }
6    Socket@H connection =
7      socket.accept@H();
8    // { Rd/Wr(P) Rd/Wr(H) }
9    init@H(connection);
10   // { Rd/Wr(P) Rd/Wr(H) }
11   new HandlerThread@H(connection).start();
12   // { Rd/Wr(P) }
13 }

```

Figure 5: The results of a flow-sensitive effect analysis of the `accept()` method from Figure 4.

have effects which affect unbounded wildcard partitions.

Line 11 contains the call which starts the `HandlerThread`. This is where we must verify the effect agreements for `start()`: to do so, we compare the set of events to come from line 12 with the forbidden events, and determine that the method call is permitted. Since `start()` declares that it reads and writes `HandlerThread`'s partition parameter, we determine that `H` may be both read and written at this point and add those effects to the set, yielding line 10.

From line 9 back to line 5, the effect set is unchanged because it already contains reads and writes. No methods are invoked which declare an effect agreement, so there is no need to check for conflicts. Finally, we reach line 4 which creates the `H` partition. This kills any effects which affect only `H`, leaving only the effects on `P` in line 3. Since this set is within the declared bounds of the bound, we decide the method is effect safe and that the thread will run without interference.

3. JPART: JAVA WITH PARTITIONS

This section describes the partition type system and effect checker in more detail. While a full formal treatment is beyond the scope of this paper, we cover its most interesting and unique features.

3.1 Support for Concurrency

For the most part, JPart looks and feels like Java. The primary differences are:

- `forkjoin { \bar{s} }` statements replaces Java's `join()` method with a simple, lexically scoped mechanism. `forkjoin` executes the statements \bar{s} and dynamically tracks the set of threads which they start. Once \bar{s} have executed, the `forkjoin` statement waits for all threads started in \bar{s} , and any threads that they (transitively) have started, to finish before continuing. `forkjoin` statements are simpler than Java's mechanism `join()` methods, but can still express real-world examples of Fork/Join parallelism [13, 14].
- `atomic { \bar{s} }` statements replace locks as a means of managing concurrency. The semantics of `atomic` are that it executes the sub-statements \bar{s} atomically, meaning that it must appear as though they have not been interrupted by any other thread. Atomic statements are discussed in detail in [15].
- Casts can be used for partition extension as well as their traditional purpose. If a variable of type `C@P` is

```

cdecl   := class C@ $\bar{P}$  extends t {  $\overline{fdecl}$   $\overline{mdecl}$  }
fdecl   := @ $w$  t x
mdecl   := @ $\bar{P}$  ! $\bar{f}$   $\bar{c}$  t  $\overline{m(t\ x)}$  { mbody }
mbody   := new @ $\bar{P}$ ;  $\bar{t}$  x;  $\bar{b}$ 
c       := pre  $\bar{f}$  | par  $\bar{f}$ 
b       := L :  $\bar{s}$  goto  $\bar{L}$ ;
s       := x = e ; | x.x = e ; | return e ;
          | forkjoin {  $\bar{s}$  } | atomic {  $\bar{s}$  }
e       := x | x.x | x.m@ $\bar{w}$ (x) | new C@ $\bar{w}$  | (t) x
t       := C@ $\bar{q}$ 
f       := F( $\bar{w}$ )
q       := ? | ?  $\subseteq$  w | w
w       := P |  $\emptyset$  | w  $\cup$  w | w  $\cap$  w

```

Figure 6: The grammar for a JPart class declaration.

cast to the same class type, but with different partition parameters, such as $C@Q$, then those fields in the object that are located in P are added to Q so that they exist in both.

3.2 Grammar

The type system and effect checker are defined in terms of the language JPart, which is a simplification of Java similar in spirit to Featherweight Java [16], but extended with the partition, effect, and agreement annotations we have seen.

The grammar for a JPart class description is contained in Figure 6. Like Featherweight Java, we use an overline to indicate repetition. Terminals are represented in normal font, nonterminals in *italics*, and keywords in **bold**.

Class declarations include a set of formal partition parameters \bar{P} following the class name. Type references, including the super type declaration, are also extended with a corresponding list of partition expressions.

Fields are declared by specifying a partition expression q , which indicates the partition in which the field is located, a type t , and a name x .

Method declarations are prefaced with a set of partition parameters \bar{P} , effects \bar{f} , and agreements \bar{c} . For simplicity, we restrict the new partition declarations (**new** \bar{P}) we showed earlier to the beginning of a method.

Method bodies are defined as a series of basic blocks which encode a control flow graph. Each block has a label L and a set of successors, specified after a **goto** statement. In between the block has a series of statements \bar{s} , which should be familiar from Java.

There are two nonterminals for partition expressions. The first, w , allows for the combination of named partitions (P) with common set operations such as union and intersection. The partition parameters provided to a type, however, make use of a second nonterminal, q , which extends w with *wildcards*, written $? \subseteq w$. A type which makes use of a wildcard parameter indicates that the exact partition for that parameter is not known; however, the partition provided is known to be a subset of w .

3.3 Subtyping and Partition Wildcards

The design of JPart’s type system is based on Java; in particular, our partition wildcards are a simplification of the wildcards used by Java generics [17]. To simplify the presentation, JPart omits primitive types such as `int` and `double` and includes only class types.

```

class ObjectWrapper@P {
  @P Object@P field;
  Object@P get() { return field; }
  void set(Object@P i) { field = i; }
}

```

Figure 7: Definition of ObjectWrapper

JPart is a single-inheritance system. With the exception of `Object`, each JPart class must define a supertype. As JPart includes a parameterized type system, type references includes not only a class name, but a partition expression for each partition parameter of the class.

So long as no wildcards are employed, JPart types are non-variant with respect to their partition parameters, meaning that two types $C@P$ and $C@Q$ are not subtypes of one another, even if $P \subseteq Q$. This allows partition parameters to be referenced in both co-variant positions, such as the types of return values, or contra-variant positions, such as the type of a method parameter.

Use-site variance can be achieved through the use of one or more partition wildcards. Using a wildcard for a particular partition parameter indicates that the precise value of that parameter is unknown; however, the value can be bounded by some partition which is a superset of the actual value. A type $C@(? \subseteq P)$ is a supertype of some type $C@(? \subseteq Q)$ if $P \subseteq Q$. In addition, $C@(? \subseteq P)$ is a supertype of $C@P$.

One final note in the area of subtyping concerns downcasts. Because partitions are erased at runtime, there is no way to dynamically verify that a downcast is safe. Therefore, we require that the partition parameters on the target type of the cast be unbounded wildcards, unless a more precise partition can be derived by examining the specific subtype and supertype involved.

Deriving a partitioning for a subtype C from a supertype D can be a complicated process, particularly in the face of partition expressions. One simple technique which should handle the most common cases is to find those partition expressions in the **extends** clause of C which consist of only one partition parameter in C , and do not contain unions or intersections. The precise value of that partition parameter can then be derived from the corresponding partition on the static type of the variable being cast.

3.4 Fields and Methods Calls with Wildcards

Fields and methods of variables whose types contain wildcard partitions may still be used in co-variant positions. It is a static error to use a type when a wildcard partition would appear, after substitution, in a contra-variant position such as a method parameter or the type of a field being modified.

As an example, consider the class `ObjectWrapper` shown in Figure 7. The class takes one partition parameter P which is used in both a covariant position (`get()`) and a contravariant position (`set()`). Both the `get()` and `set()` methods be invoked on a variable of type `ObjectWrapper@Q`. If, however, the variable had type `ObjectWrapper@(? \subseteq Q)`, then it would be illegal to invoke the `set()` method, as the parameter P has a wildcard value but appears in a contravariant position. The `get()` method could still be invoked, however, although its result would have type `Object@(? \subseteq Q)`.

4. EFFECT CHECKER

```

class Class1@A {
  @A Class1@A field;
  !Rd(P) !Wr(Q)
  @P @Q void method(Class1@P p, Class1@Q q) {
    q.field = ...;
    p.field = ...;
  }
}

```

Figure 8: A simple method with two effects.

The purpose of the effect checker is two-fold. First, it should verify that the method declares any effects it might have which can affect a partition parameter in its interface. Second, it must guarantee that the effect agreements on any invoked methods are upheld.

4.1 Intersection and Partition Extension

For the most part, the effect checker algorithm does not take the effect label into account when propagating effects. A read effect is treated identically to a write effect. However, in order to properly account for all possible interactions, the effect checker must know when two partitions have been made to intersect.

Two partitions can be made to intersect either by creating data in their intersection or by partition extension from one partition to the other. As is described in Section 4.5, code which does either of those two things also generates an effect, $\text{Inter}(w_1, w_2)$, which indicates that the partitions w_1 and w_2 potentially share data.

In the face of an $\text{Inter}(w_1, w_2)$ event, our effect checker makes worst-case assumptions. This means that it assumes that any effect which affects w_2 potentially affects w_1 as well. This gives rise to an operation $\text{closure}_i(f_i, f)$ which expands a set of effects f to include any indirect effects that may be caused by the intersections witnessed by intersection effects in f_i . More precisely, $\text{closure}_i(f_i, f)$ is the smallest set such that $\exists F(\bar{w}) \in f, j. (\text{Inter}(w_j, w') \in f_i) \implies \{F(\bar{w}), F([w_j/w']\bar{w})\} \subseteq \text{closure}_i(f_i, f)$, assuming that Inter is commutative.

In general, a method must declare any effect whose affected partition may intersect a partition parameter of the method or class. However, what about intersections that the method does not know about? Consider the method `method` declared in Figure 8, which has two partition parameters, P and Q, and copies a field from an object in partition P to an object in partition Q. Accordingly, it declares a $\text{Rd}(P)$ effect and $\text{Wr}(Q)$; but are these effects sufficient? If the actual partitions provided for the parameters P and Q were intersecting, then it may be that the method writes P as well. The answer is that, in our system, a method is responsible for declaring indirect effects due to intersections which it itself caused, but not those due to intersections among its partition parameters. Instead, the caller is responsible for applying the closure operation to the effects of the method once the actual partitions are known. This results in more precision.

4.2 Iterated Analysis

The effect checker is based around an iterated analysis which propagates sets of effects around the control flow graph until it reaches a fixed point. Similar algorithms are commonly used for data-flow analysis in compilers [18]. Be-

Algorithm 1 effect-flow

```

POST  $\leftarrow$  {  $\emptyset$  for each block }
POST'  $\leftarrow$  {  $\emptyset$  for each block }
PRE  $\leftarrow$  {  $\emptyset$  for each block }
initialize PRE and POST with starting assumptions
while PRE or POST continues to change do
  for all blocks  $b$  do
     $f_{f'} \leftarrow \cup(\{\text{POST}'_{b'} \text{ for each } b' \in \text{predecessors of } b\})$ 
     $f_f \leftarrow \cup(\{\text{POST}_{b'} \text{ for each } b' \in \text{predecessors of } b\})$ 
     $f_r \leftarrow \cup(\{\text{PRE}_{b'} \text{ for each } b' \in \text{successors of } b\})$ 
    for all statements  $s$  in  $b$  do
       $f_{f'} \leftarrow \text{effects}(f_{f'}, s)$ 
       $f_x \leftarrow \text{effects}(f_f, s)$ 
       $f_f \leftarrow f_f \cup f_x$ 
       $f_r \leftarrow f_r \cup f_x$ 
    end for
     $\text{POST}_b \leftarrow f_f$ 
     $\text{POST}'_{b'} \leftarrow f_{f'}$ 
     $\text{PRE}_b \leftarrow f_r$ 
  end for
end while

```

cause effect agreement can constrain what happens before a method call as well as what happens afterwards, we have to actually do two analyses: one propagating events forward along the dataflow graph, and one in reverse. In both cases, we use set union as the confluence operator for joining multiple control flows.

The complete effect flow algorithm is shown as Algorithm 1. It computes three sets of events per block b : POST_b and POST'_b , which indicate those events that will have occurred after the block finishes executing (but before its successors), and PRE_b , which indicates the events that can occur in the future, at the point immediately before the block is executed. The difference between POST and POST' is that the former makes conservative assumptions about potential intersection between partitions, and the latter does not. POST is used for checking effect agreements, which require the worst-case analysis, but POST' is used for determining whether a method declares all potential effects.

To seed the analysis, it is necessary to begin with conservative assumptions. We assume that any effect which is not specifically forbidden by an effect agreement is possible. Let f_{all} be the set of all events affecting the partition parameters, and f_{pre}/f_{par} be the events forbidden by pre- and par- agreements respectively. Then we seed the POST set for the initial block with $f_{all} \setminus f_{pre}$, and the PRE set for each block ending in a `return` statement with $f_{all} \setminus f_{par}$. In addition, the PRE set contains an intersection event $\text{Inter}(P, P)$ for every partition parameter P.

To process a basic block, we determine the initial set of events by taking the union of the POST/POST' and PRE events from the predecessors and successors of this block, respectively. We then use the function $\text{effects}(f, s)$ (defined below) to compute the effect of each statement and add them to the appropriate event set. Note that computing the effects is a function not only of the statement, but of the set of effects f that may have occurred so far: f is needed to account for possible intersections.

This whole process is repeated until the sets reach a fixed point. Because there are only a finite number of partitions and events to be generated, and the various effect sets only

Algorithm 2 `check-reverse-stmt(f, s)`

```
if  $s$  is a forkjoin statement then
   $f_t = \emptyset$ 
else
   $f_t = f$ 
end if
if  $s$  has substatements then
  for all substatements  $s'$  in  $s$  do
    check-reverse-stmt( $f_t, s'$ )
     $f_t \leftarrow f_t \cup \text{effects}(s')$ 
  end for
end if
check that par-contracts in  $s$  do not occur in  $f$ 
```

```
class Class@P@Q {
  @(P∩Q) int field;
}
```

Figure 9: A class whose field is declared in the intersection of two partitions.

grow, the algorithm must eventually terminate.

4.3 Checking the Method Interface

The effects POST' for each block are used to check the method interface. For every effect $F(w)$ listed in some set of POST' where a partition parameter is among the free variables of w , there must be a declared effect $F(w')$ where $w \subseteq w'$. According to this rule, effects which cannot affect any partition parameter but only newly-created partitions do not have to be declared. Intersection events do not need to be listed if one partition is a newly-created partition.

4.4 Checking Effect Agreements

After Algorithm 1 completes, using the PRE and POST sets to check effect agreements is straightforward. We check agreements with pre time span first by walking through each basic block, examining its statements in order, and ensuring that the set of events which may have occurred up to that point is disjoint from the events listed in any pre -agreements.

Checking par -agreements is similar, but with a slight twist. The twist occurs because par -agreements actually restrict events that can happen concurrently, but what we have computed so far is the set of events that can happen at any point in the future. If a `forkjoin` statement is encountered, we know statically that any threads started within will be joined before it finishes. Therefore, when we process the substatements of the `forkjoin`, we start with an empty set of “future events”. The effects of any statements within the `forkjoin` itself are still significant, however. Algorithm 2 displays a recursive method `check-reverse-stmt(f, s)` which, when invoked with a statement s and a set f of events that can occur afterwards, checks that any par -agreements are satisfied.

4.5 Effects of Statements and Expressions

This section describes the `effects(f, s)` function used by Algorithm 1 to compute the effects caused by a given statement. There are five statements which cause effects in JPart to consider:

1. Reading and writing fields: Directly accessing a field causes an appropriate read or write event to be generated for the partition which contains the field. The partition is determined by consulting the formal declaration for the field and substituting the appropriate partitions for the class partition parameters. The events generated are either Rd and Wr or ARd and AWr depending on whether this is a substatement of an atomic block.
2. Creating new instances: Creating new objects can potentially cause intersection events if any of the fields of the new object are located in the intersection of two partitions. This can happen in two ways: either an expression like $\text{P}\cap\text{Q}$ is given as a partition parameter, or the class contains a field that is declared in the intersection of two partitions, as shown in Figure 9.
3. Partition extension: When an existing object is cast from one set of partition parameters to another, the new partitions are extended to include the existing object, causing intersection events to be generated. For example, casting an object from type `IntWrapper@P` to `IntWrapper@Q` would cause an `Inter(P,Q)` event.
4. Method calls: The effects of a method are determined by consulting the effects listed in its declaration and substituting appropriate values for the formal partition parameters. If the method call takes place within an atomic statement, then any Rd and Wr effects must be translated into ARd and AWr effects.

If any effect is generated which affects a partition described by a wildcard, the effect is “promoted” so as to affect the bound of the wildcard. For example, if an effect $\text{Rd}(?\subseteq\text{Q})$ occurs, then the effect is considered to be a read on the entire partition Q . It is a static error for there to be an effect on an unbounded wildcard such as $\text{Rd}(?)$.

Once the base effects f_b are generated, the `closure(f, f_b)` operation is used to take into account any partitions that may have intersected before this statement occurred.

5. ENFORCING THREAD-SAFETY

We can use effect agreements to ensure that a partition is never accessed by multiple threads in an incompatible fashion. The core idea is to use the type of the `Thread` object, from which all threads must derive, to guarantee that threads only use partitions in one of several pre-approved ways.

We guarantee that any partition the thread may access falls into one of the following categories:

1. Thread-Local: Thread-local partitions are read and written by the thread outside of `atomic` sections. Such a partition may not be simultaneously used in any way by any other thread.
2. Read-Only: Read-only partitions are never written by the thread and are read outside of `atomic` sections. Such partitions may not be written by other threads.
3. Shared: Shared partitions are actively modified by multiple threads simultaneously. Any access to such a partition, read or write, must take place within an `atomic` section.

```

1  abstract class Thread@S@R@L {
2
3      !ARd(S ∪ R ∪ L)
4      !AWr(S ∪ L)
5      !Rd(R ∪ L)
6      !Wr(L)
7      abstract void run();
8
9      !ARd(S ∪ R ∪ L)
10     !AWr(S ∪ L)
11     !Rd(R ∪ L)
12     !Wr(L)
13     pre -Inter(S,R) -Inter(S,L) -Inter(R,L)
14     par -Wr(S ∪ R ∪ L)
15     par -Rd(S ∪ L)
16     par -AWr(R ∪ L)
17     par -ARd(L)
18     abstract void start();
19 }
20 }

```

Figure 10: The definition of legal effects for a Thread. Using these effects guarantees that all partitions modified by a thread are used in a safe fashion.

We assume that threads are started by invoking the method `start()` defined in the `Thread` class. The actions of a thread are defined by its `run()` method; invoking `run()` directly, however, does not start the thread, but is merely a normal method call that runs in the current thread.

Therefore, we can use the declared interface on the `start()` and `run()` methods to control what threads are permitted to do. The desired interface for class `Thread` is shown in Figure 10. The idea is to parameterize the thread by three partitions, `S`, `R`, and `L`, which contain the shared, read-only, and thread-local data that this thread may access. As we will see, the definition of the `run()` and `start()` methods do not permit data in these partitions to be used in any way other than those outlined above. Furthermore, because a method must list all of its effects in its interface, we can be sure that this thread will not affect any partitions other than `S`, `R`, or `L`.

To get a better understanding of the definition in Figure 10, let us examine it line by line:

- Line 1 declares the partition parameters, `S`, `R`, and `L`.
- Lines 3–7 define the `run()` method. The effect declarations on lines 3–6 describe how the thread is allowed to access `S`, `R`, and `L`. Line 3, for example, indicates that any partition may be atomically read, whereas line 4 restricts atomic writes to shared (`S`) and thread-local (`L`) data. Line 5 allows read-only (`R`) and thread-local (`L`) data to be read non-atomically, but only `L` may be modified in a non-atomic fashion, as indicated on line 6.
- Lines 9–12 indicate that the `start()` method has the same effects as the `run()` method.
- Lines 13–17 define the effect agreement for the `start()` method. These agreements highlight the important difference between `start()` and `run()`: invoking `run()` does not actually start a second thread. `start()`, however, performs its actions in parallel with the current thread, and therefore it has to place constraints on the current thread. Note the use of the `par` time span for these effect agreements, which guarantees that the

```

class Main {
    @A !Rd(A)
    public void mapreduce(List@A objects) {
        new @O;
        List@O results = new List@O();
        forkjoin {
            for (Object@A object : objects) {
                new @L;
                MapThread@O@A@L mt =
                    new MapThread@O@A@L();
                mt.object = object;
                mt.results = results;
                mt.start();
            }
        } // reduce results, without atomic
    }
}

class MapThread@S@A@L extends Thread@S@A@L {
    @L Object@A object;
    @L List@S results;

    !Rd(L) !Wr(L) !Rd(A) !ARd(S) !AWr(S)
    public void run() {
        new @R;
        Object@R r = map@R(object);
        Object@S r2 = (Object@S) r; // part. extension
        atomic { results.add(r2); }
    }

    @R !Rd(A) !Rd(R) !Wr(R)
    public Object@R map(Object@A o) {...}
}

```

Figure 11: Map-reduce example

forbidden events will not occur in parallel with this thread, though they may occur after the thread is known to have terminated.

- Line 13 guarantees that the partitions for shared, read-only, and thread-local data are all mutually disjoint.
- Line 14 guarantees that the parent thread does not write non-atomically to any partition that the child thread has access to.
- Line 15 guarantees that the parent thread does not try to read non-atomically from any partition which the child thread will be writing to.
- Line 16 guarantees that the parent thread will not make atomic writes to the `R` or `L` partitions.
- Line 17 guarantees that the thread-local data is not atomically read by the parent thread.

At first, it might seem stringent to require that every `Thread` class describe their data in exactly three partitions. However, due to the possibility of using partition expressions as parameters, this is not a real limitation. For example, to define a thread which has a shared partition `S`, no read-only partition, and two local partitions, `L1` and `L2`, one can extend `Thread@S@O@L1@L2`.

5.1 Example: Map Reduce

Figure 11 shows a more involved example following the well-known map reduce pattern [19]. The example contains two classes, `Main` and `MapThread`. The method `Main.mapreduce()` takes as an argument a list of objects in some partition `A` (we assume for now that both the list and the objects contained

in it are in the same partition), which it promises only to read. It then iterates over the array and creates and initializes a new `MapThread` for each object in the array. A new partition is created for the thread-local data of `MapThread`, and the input object (`object`) and output array (`results`) are also given to it before the thread is started.

The `MapThread`'s `run()` method begins by invoking its method `map()` with the object it was given. The `map()` method reads the object from the partition `A` and constructs a mapped version in the partition `R`. These reads and writes occur without locks or synchronization of any kind. This is safe because the `run()` method declares `!Rd(A)`, indicating that it reads partition `A` without synchronization, and because the partition `R` was created within the `run()` method and is not accessible to other threads.

The cast to `Object@S` which follows the call to `map` is very important, because it is in fact a partition extension. This cast has the effect of adding the data stored in the `R` partition to `S`. In effect, this makes what was previously thread-local data into shared data which can then be added to the shared `results` array. Because `results` is shared, this add happens within an `atomic` statement. Once all threads have finished, however, the `results` array can be read and written freely by the `Main` class without violating the `MapThread`'s effect agreement due to the `forkjoin` region.

6. RELATED WORK

The work in this paper is in many ways a synthesis of several existing techniques, and therefore touches on many different bodies of work.

6.1 Capability-Based Alias Control

Capability-based alias control uses capabilities that are associated with references to limit how that reference can be used [8, 20, 9, 21]. Generally these annotations are used to prevent writes and enforce uniqueness, but they could also be used more generally. Our work has no notion of capabilities; instead we rely on effects to describe what is permitted. Effects cannot be tracked on a per-reference basis, however, so it might be useful to integrate capabilities as well.

6.2 Ownership

Universe Types [4] are a form of ownership types used as part of a larger system for program verification. The system has recently been extended to support ownership transfer [22], which is similar to partition extension, but much more descriptive. It ensures that only one external alias to a cluster of objects exists by requiring that potentially invalidated fields be reassigned after method calls. Our partition extension system is much simpler, and mainly intended for allowing data to be transferred from a short-lived partition into a shared, persistent one.

Multiple Ownership [23] by Cameron et al. describes a generalization of a single ownership system that allows an object to have multiple owners simultaneously. Like us, they rely on the effect system to moderate what objects are permitted to be modified. Our system differs primarily in its use of a flow-sensitive effect system and effect agreements.

SafeJava [7] and SafeJavaML [24] use ownership to enforce many safety properties, including multi-threaded programming. They require an object's top-level owner be locked before it can be accessed, thus ensuring a consistent locking discipline. One downside of this approach is that it entan-

gles encapsulation and threading, which may require that one or the other be compromised. Fine-grained locking, for example, requires promoting representation objects to top-level owners so that they have their own lock. In contrast, our approach strives to separate the partitioning structure of the program from the thread safety check.

6.3 Regions and Effect Systems

The polymorphic effect system introduced by Lucassen and Gifford [25] combined first-class memory regions with an effect system to enable safe parallelism in a dataflow setting. It also allowed effects to be masked if they concerned only private regions, similar to how methods may create new, local partitions which are unobservable to their caller.

OOFX [10], along with Data Groups [26], later brought these ideas into an object-oriented context. To account for subtyping, both systems allow the fields of an object to be divided into regions or groups, similar to partitions, which can be extended by a subtype, and allow methods to declare which of these sets of fields are affected. Unlike partitions, OOFX and Data Groups defined regions per-instance, which means that a single region cannot contain state from multiple objects, although OOFX was later extended as described below.

Greenhouse and Scherlis [27] introduced annotations to OOFX which allow programmers to associate locks with mutable state. In addition to program verification, these annotations can be used to do systematic program refactoring. This paper also added parameterized regions which are similar in power to partition parameters, as they allow a single region to encompass state from multiple objects.

In the paper Ownership, Encapsulation and the Disjointness of Type and Effect [28], Clarke and Drossopoulou present an expressive combination of ownership and effects where each effect affects a region of the object graph called a *shape*. Shapes are a natural consequence of the ownership tree in their system.

Contextual Effect Systems [29] uses a flow-sensitive effect system to detect potential data races. They rely on a standard alias analysis to map pointers to abstract objects, rather than partitions defined by the programmer. Like us, they detect data races by contrasting the effects of a thread when it is started with the effects of the current thread from that point forward, though they lack a mechanism such as effect agreements that allows the user or library author to describe conflicts. Potential conflicts are used as inputs to the Locksmith tool [30], so as to help reduce false error reports.

6.4 Thread Analysis

Tools for determining whether multi-threaded programs are sound have been around almost as long as threads themselves, and therefore there exists a large body of work which we cannot even begin to do justice to. We will focus here on approaches that are based on type and effect systems.

Flanagan, et al. developed a type system to check atomicity [31] of method implementations rather than merely race freedom. Their work detects many subtle programming errors that can occur despite a correct lock discipline. Our work, in contrast, deals with a transactional memory system which already guarantees atomic execution.

Guava [32] is a dialect of Java that does not permit data races by construction. Guava divides classes into three cat-

egories: monitors, values, and objects. Only monitors are accessible to multiple threads. Values may not be referenced, and are instead passed by value. Objects act like traditional Java objects, except that they are local to a single thread. These categories closely resemble the shared, read-only, and local partition parameters we defined on the `Thread` class. In our case, however, a single object may transition between those categories during the flow of the graph. Furthermore, different instances of the same class may be used as values in some places, or objects in others. In Guava, on the other hand, these decisions apply to all instances of a given class.

7. CONCLUSION

In this paper, we have presented *partitions*, an abstraction for exposing the alias structure of a program, along with an accompanying flow-sensitive effect system with effect agreements. We also detailed one application of our effect system, which was checking that a multi-threaded program uses safe patterns for its synchronization.

Our long-term goal is to give programmers a simple and expressive way to check semantic properties of their own design. Rather than encoding a specific notion of correctness into the type system, we aim to develop generic mechanisms, such as partitions and effect agreements, that can be reused for a variety of purposes.

8. REFERENCES

- [1] Almeida, P.S.: Balloon types: Controlling sharing of state in data types. In: ECOOP '97. Volume 1241 of LNCS., Springer-Verlag (1997) 32–59
- [2] Vitek, J., Bokowski, B.: Confined types. In: OOPSLA, ACM (1999) 82–96
- [3] Hogg, J.: Islands: aliasing protection in object-oriented languages. SIGPLAN Not. **26** (1991) 271–285
- [4] Dietl, W., Drossopoulou, S., Müller, P.: Generic Universe Types. In Ernst, E., ed.: ECOOP. Volume 4609 of Lecture Notes in Computer Science., Springer-Verlag (2007) 28–53
- [5] Aldrich, J., Chambers, C.: Ownership domains: Separating aliasing policy from mechanism. In: ECOOP. (2004) 1–25
- [6] Potanin, A., Noble, J., Clarke, D., Biddle, R.: Generic ownership for generic Java. In: OOPSLA, ACM (2006) 311–324
- [7] Boyapati, C.: Safejava: a unified type system for safe programming. PhD thesis, Massachusetts Institute of Technology (2004) Supervisor-Martin C. Rinard.
- [8] Aldrich, J., Kostadinov, V., Chambers, C.: Alias annotations for program understanding. In: OOPSLA, ACM (2002) 311–330
- [9] Noble, J., Vitek, J., Potter, J.: Flexible Alias Protection. In: ECOOP, Springer-Verlag (1998) 158–185
- [10] Greenhouse, A., Boyland, J.: An Object-Oriented Effects System. In: ECOOP, Springer-Verlag (1999) 205–229
- [11] Meyer, B.: Object-Oriented Software Construction. Prentice Hall PTR (2000)
- [12] Liskov, B.H., Wing, J.M.: Behavioural subtyping using invariants and constraints. In: Formal methods for distributed processing: a survey of object-oriented approaches, Cambridge University Press (2001) 254–280
- [13] Bull, J.M., Kambites, M.E.: JOMP—an OpenMP-like interface for Java. In: JAVA, ACM (2000) 44–53
- [14] Lea, D.: A Java fork/join framework. In: JAVA, ACM (2000) 36–43
- [15] Harris, T., Fraser, K.: Language support for lightweight transactions. SIGPLAN Not. **38** (2003) 388–402
- [16] Igarashi, A., Pierce, B.C., Wadler, P.: Featherweight Java: a minimal core calculus for Java and GJ. ACM Trans. Program. Lang. Syst. **23** (2001) 396–450
- [17] Torgersen, M., Hansen, C.P., Ernst, E., von der Ahé, P., Bracha, G., Gafter, N.: Adding wildcards to the Java programming language. In: SAC, ACM (2004) 1289–1296
- [18] Aho, A.V., Ullman, J.D.: Principles of Compiler Design. Addison-Wesley Longman Publishing Co., Inc. (1977)
- [19] Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: OSDI, USENIX Association (2004)
- [20] Boyland, J., Noble, J., Retert, W.: Capabilities for Sharing: A Generalisation of Uniqueness and Read-Only. In: ECOOP, Springer-Verlag (2001) 2–27
- [21] Boyland, J.: Alias burying: unique variables without destructive reads. Softw. Pract. Exper. **31** (2001) 533–553
- [22] Müller, P., Rudich, A.: Ownership transfer in universe types. In: OOPSLA, ACM (2007) 461–478
- [23] Cameron, N.R., Drossopoulou, S., Noble, J., Smith, M.J.: Multiple ownership. In: OOPSLA, ACM (2007) 441–460
- [24] Permandla, P., Roberson, M., Boyapati, C.: A type system for preventing data races and deadlocks in the java virtual machine language: 1. In: LCTES, ACM (2007) 10
- [25] Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: POPL '88, ACM (1988) 47–57
- [26] Rustan, K., Leino, M.: Data groups: specifying the modification of extended state. In: OOPSLA, ACM (1998) 144–153
- [27] Greenhouse, A., Scherlis, W.L.: Assuring and evolving concurrent programs: annotations and policy. In: ICSE '02, ACM (2002) 453–463
- [28] Clarke, D., Drossopoulou, S.: Ownership, encapsulation and the disjointness of type and effect. In: OOPSLA, ACM (2002) 292–310
- [29] Neamtiu, I., Hicks, M., Foster, J.S., Pratikakis, P.: Contextual effects for version-consistent dynamic software updating and safe concurrent programming. SIGPLAN Not. **43** (2008) 37–49
- [30] Pratikakis, P., Foster, J.S., Hicks, M.: Locksmith: context-sensitive correlation analysis for race detection. In: PLDI '06, ACM (2006) 320–331
- [31] Flanagan, C., Qadeer, S.: A type and effect system for atomicity. SIGPLAN Not. **38** (2003) 338–349
- [32] Bacon, D.F., Strom, R.E., Tarafdar, A.: Guava: a dialect of Java without data races. In: OOPSLA, ACM (2000) 382–400