Appeared in the proceedings of ECOOP '99, LNCS 1628, Springer Verlag, pp. 205–229

An Object-Oriented Effects System*

Aaron Greenhouse[1] and John Boyland[2]

[1] Carnegie Mellon University, Pittsburgh, PA 15213, aarong@cs.cmu.edu
[2] University of Wisconsin–Milwaukee, Milwaukee, WI 53201, boyland@cs.uwm.edu

Abstract. An effects systems describes how state may be accessed during the execution of some program component. This information is used to assist reasoning about a program, such as determining whether data dependencies may exist between two computations. We define an effects system for Java that preserves the abstraction facilities that make object-oriented programming languages attractive. Specifically, a subclass may extend abstract regions of mutable state inherited from the superclass. The effects system also permits an object's state to contain the state of wholly-owned subsidiary objects. In this paper, we describe a set of annotations for declaring permitted effects in method headers, and show how the actual effects in a method body can be checked against the permitted effects.

1 Introduction

The *effects* of a computation include the reading and writing of mutable state. An effects system is an adjunct to a type system and includes the ability to infer the effects of a computation, to declare the permitted effects of a computation, and to check that the inferred effects are within the set of permitted effects.

The effects system we describe here is motivated by our desire to perform semantics-preserving program manipulations on Java source code. Many of the transformations we wish to implement change the order in which computations are executed. Assuming no other computations intervene and that each computation is single-entry single-exit, it is sufficient to require that the effects of the two computations do not *interfere*: one computation does not write state that is read or written by the other. Therefore our system only tracks reads and writes of mutable state, although other effects can be interesting (e.g., the allocation effect of FX [6]). Because Java supports separate compilation and dynamic loading of code, one of the goals of the project is to carry out transformations on

* This material is based upon work supported by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0241. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government. The work of John Boyland is supported by an equipment grant from Sun Microsystems, Inc.

incomplete programs. In earlier work [4], we proposed a system of program annotations including ones to declare the permitted effects of missing code. The effects system described in this paper uses and expands upon these annotations.

Declaring the (permitted) effects of a method necessarily constrains the implementation of the method and any method that overrides it, but we do not want to "break abstraction," by revealing the implementations details of the program's classes. One of the requirements for a useful object-oriented effects system is that it still provide the ability to hide names of private fields. It should use abstract names that map to multiple mutable locations. This paper introduces a concept of a named "region" in an object. The regions of an object provide a hierarchical covering of the notional state of the object. This idea is extended to objects which have the sole reference to other ("unique") objects; the state of such owned objects is treated as extending the state of the owning object. As a result, a complex of numerous objects, such as a hash table, can be handled as a single notional object comprising a few regions of mutable state.

Our effects system uses declared annotations on fields and methods in Java. These annotations are non-executable, and can be ignored when implementing the program, although an optimizing compiler may find the information useful. The exact representation of the annotations is not important, but for the purposes of this paper, they are shown using an extension to Java syntax. The syntax of the effects annotations is given in Section 2.

A method body can be checked against its annotation using the annotations on the methods it calls. Similar to type-checking in the context of separate compilation, if all method bodies are checked at some point, the program as a whole obeys its annotations.

In the following section of the paper, we describe the basic concept behind the effects system: an abstract region of mutable state within an object. We also describe the technique for inferring the effects of a computation and checking inferred effects against declared effects. In the following section, we show how the regions of unique objects can be mapped into their owners. The precise description of the system is given in the appendix.

2 Regions of Objects

A *region* is an encapsulation of mutable state. The read and write effects of a method are reported with respect to the regions visible to the caller. In this section, we describe the general properties of a region, and how regions are specified by the programmer. Then we describe how methods are annotated, how the effects are computed, and how the annotation on a method can be checked for correctness.

2.1 Properties of Regions

The regions of a program are a hierarchy: at the root of the hierarchy there is a single region `All` that includes the complete mutable state of a program; at the leaves of the hierarchy are all the (mutable) fields which again comprise the entire mutable state of the program.[1] Thus we see that each field is itself a region without child regions. The hierarchy permits us to describe the state accessed by a computation with varying precision: most precise is listing the individual fields accessed, least precise is `All`.

Java fields are declared in classes and are either `static` or not. In the latter case (instance fields), the declaration actually implies multiple fields, one per object of the class in which the field was declared. Our generalization of fields, regions, are similarly declared in classes in two ways: *static* declarations add one new region; *instance* declarations add a set of disjoint regions, one for each object of the class in which the region is declared. We call the non-field regions *abstract regions*.[2]

Each region is declared to be inside a parent region, thus creating the hierarchy. The parent region specified in a static region declaration must not refer to an instance region; such a static region would in effect have multiple parents. The root region `All` is declared static; one of its child regions is an instance region `Instance` declared in the root class `Object`.

Instance regions are inherited by subclasses, and a subclass may declare its own regions, possibly within an inherited (non-field) region. Being able to extend existing regions is critical to being able to specify the effects of a method and later being able to meaningfully override the method (see Example 2).

Arrays are treated as instances of a class `Array`, which has the instance region `[]` (pronounced "element") as a subregion of `Instance`.[3] When a subscripted element of an array is accessed, it is treated as an access of region `[]`.

The same accessibility modifiers that apply to field declarations (in Java, `public`, `protected`, default (package) access, or `private`) also apply to region declarations. The root region `All`, and its descendants `Instance` and `[]`, are `public`.

2.2 Specifying Regions

New syntax is used to declare abstract regions in class definitions. The syntax `region` *region* declares abstract instance regions. Adding the keyword `static` will declare an abstract static region. The parent of a region may be specified in the declaration by appending the annotation "`in` *parent*" to the field or abstract region declaration. Region declarations without an explicit parent are assigned to `All` or `Instance` respectively if they are static or not.

[1] External state (such as file state) is beyond the scope of this paper. Suffice it that special regions under `All` can be used to model this external state.

[2] We use the term "abstract" to emphasize the implementation hiding characteristics of these regions, not in the usual sense of Java "abstract methods" which impose implementation requirements on subclasses.

[3] More precision could be obtained by distinguishing the different types of arrays, especially arrays of primitive types.

```
class Point {
  public region Position;
  private int x in Position;
  private int y in Position;
  public scale(int sc)
    reads nothing writes Position
  {
    x *= sc;
    y *= sc;
  }
}

class ColorPoint extends Point {
  public region Appearance;
  private int color in Appearance;
}
```

(a)

```
class Point3D extends Point {
  private int z in Position;
  public void scale(int sc)
    reads nothing writes Position
  {
    super.scale(sc);
    z *= sc;
  }
}
```

(b)

Fig. 1. The definitions of (a) classes `Point` and `ColorPoint`, and (b) class `Point3D`

The Java field selection syntax (*object*`.`*field* for instance fields and *Class*`.`*field* for static fields, as well as the provision for omitting `this` or the current class) is extended to all regions.

Example 1. Consider the class `Point` in Figure 1a. It declares the fields `x` and `y`, and the abstract region `Position` as their parent region. An instance o of class `Point` has four instance regions: o`.Instance`, which contains o`.Position`, which in turn contains o`.x` and o`.y`. The class `Point` has a method `scale`. This method is correctly annotated with `writes Position` (short for `writes this.Position`). The annotation `writes this.x, this.y` should not be used because the fields are private but the method is public. The annotation `writes Instance` would be legal, but less precise.

The class `ColorPoint` (also in Figure 1a) inherits from `Point`, declaring field `color` in a new abstract region `Appearance`, implicitly a subregion of `Instance`. If `ColorPoint` overrode `scale(int)`, the overriding method could not access `color` because `color` is not in `Position`, which is reasonable because color is not a geometric property. Given the less precise annotation on `scale(int)` mentioned above, the limitation would be lifted, because `color` is indeed in `Instance`.

We can create a second subclass of `Point`, `Point3D`, that adds the new field `z` to `Position` allowing it to be changed in the overriding of `scale(int)` as shown in Figure 1b.

2.3 Regions and Effects

As stated earlier, effects on objects are reported in terms of the regions that are affected. We distinguish between two kinds of effects: *read effects*, those that

$$
\begin{array}{rcll}
\mathit{annotation} & \rightarrow & \texttt{reads}\ \mathit{targets}\ \texttt{writes}\ \mathit{targets} & \\
\mathit{targets} & \rightarrow & \texttt{nothing} \mid \mathit{targetSeq} & \\
\mathit{targetSeq} & \rightarrow & \mathit{target} \mid \mathit{target},\ \mathit{targetSeq} & \\
\mathit{target} & \rightarrow & \ \ \mathit{var.region} & \text{region of a specific instance} \\
 & & \mid \texttt{any}(\mathit{class}).\mathit{region} & \text{region of any instance of class } \mathit{class} \\
 & & \mid \mathit{class.region} & \text{(static) region of a class}
\end{array}
$$

var is restricted to being one of the method parameters or the receiver.

Fig. 2. The syntax of method annotations used in this paper.

may read the contents of a region; and *write effects*, those that *may* change *or read* the contents of a region. Making writing include reading is useful when overriding methods, as demonstrated in Examples 1 and 2. It also closely follows how an optional write would be modeled using static single-assignment form (SSA): as a merge of a read and a write.

Methods are labeled with their permitted effects using a notation similar to the Java `throws` clause. Figure 2 gives the syntax of method effect annotations. Again, analogous to `throws` clauses, the declared effects for a method describe everything that the method *or any of its overriding implementations* might affect. It is an error, therefore, for an overriding method to have more effects than declared for the method it overrides. The soundness of static analysis of the effects system would otherwise be compromised.

Example 2. This example shows the importance of abstract regions and the usefulness of having "write" include reading. Consider the class `Var` shown below, that encapsulates the reading and writing of a value. First we will show a set of annotations that do not permit a useful subclass, even assuming it were legal to expose names of private fields (regions) in annotations:

```
class Var {
  private int val;
  public void set( int x ) reads nothing writes this.val { val = x; }
  public int get() reads val writes nothing { return val; }
}
```

Suppose we now extended the class to remember its previous value. The annotations below are illegal because the declared effects of `UndoableVar.set()` are greater than the declared effects of `Var.set()`.

```
class UndoableVar extends Var {
  private int saved;
  public void set( int x ) reads nothing writes this.val, this.saved {
    saved = get(); super.set( x );
  }
  public void undo() reads this.saved writes this.val {
    super.set(saved);
  }
}
```

By instead placing the fields `val` and `saved` inside of a new abstract region `Value`, we are able to produce legal effect declarations:

```
class Var {
  public region Value;
  private int val in Value;
  public void set( int x ) reads nothing writes this.Value { val = x; }
  public int get() reads this.Value writes nothing { return val; }
}

class UndoableVar extends Var {
  private int saved in Value;
  public void set( int x ) reads nothing writes this.Value {
    saved = get(); super.set( x );
  }
  public void undo() reads nothing writes this.Value {
    super.set(saved);
  }
}
```

The implementation of `UndoableVar.set()` would not be legal if writing did not include the possibility of reading.

Computing Method Effects Effects are computed in a bottom-up traversal of the syntax tree. Every statement and expression has an associated set of effects. An effect is produced by any expression that reads or writes a mutable variable or mutable object field (in Java, immutability is indicated by the `final` modifier). The effects of expressions and statements include the union of the effects of all their sub-expressions. The computed effects use the most specific regions possible.

The analysis is intraprocedural; effects of method calls are obtained from the annotation on the called method, rather than from an analysis of the method's implementation. Because the method's declared effects are with respect to the formal parameters of the method, the effects of a particular method invocation are obtained by substituting the actual parameters for the formal parameters in the declared effects.

Checking Method Effects To check that a method has no more effects than permitted, each effect computed for the body of the method is checked against the effects permitted in the method's annotation. Any computed effect other than effects on local variables (which are irrelevant once the method returns) and effects on newly created objects must be *included* in at least one of the permitted effects in the annotation.[4] The hierarchical nature of regions ensures that if an effect is included in a union of other effects then it must be included in one of the individual effects. Constructor annotations are not required to include write effects on the `Instance` region of the newly created object.

[4] These exceptions can be seen as an instance of *effects masking* as defined for FX [6].

A read effect e_1 includes another read effect e_2 if e_1's target includes e_2's target. A write effect e_1 includes an effect e_2 if e_1's target includes e_2's target; e_2 may be a read or write effect. The target *var*`.`*region* includes any target for the same instance and a region included in its region. The target `any(`*class*`).`*region* includes any instance target referring to a region included in its region.[5] The target *class*`.`*region* includes any target using a region included by its region. Appendix A.3 defines effect inclusion precisely.

Example 3. Consider the body of `set` in class `UndoableVar`. In inferring the effects of the method body, we collect the following effects:

`writes this.saved` from `this.saved = ...`
`reads this.Value` from `this.get()`
`writes this.Value` from `super.set(...)`
`reads x` from `x`

The last effect can be ignored (it only involves local variables). The other three effects are all included in the effect `writes this.Value` in the annotation. This annotation is the same as that of the method it overrides

3 Unshared Fields

In this section we describe an additional feature of our effects system that permits the private state of an object to be further abstracted, hiding implementation details of an abstract data type from the user of the class. Ideally the user of the class `Vector` in Figure 3 should not care that it is implemented using an array, but any valid annotation of the `addElement` method based on what has been presented so far must reveal that its effects interfere with any access of an existing array. In this example, the array used to implement the vector is entirely encapsulated within the object, and so it is impossible that effects on this internal array could interfere with any other array.

Therefore, in addition to effect annotations, we now add the annotation `unshared` to the language as a modifier on field declarations. Unshared fields have the semantics of unique fields described in our earlier work [4] and formalized in a recently submitted paper [2]. Essentially an unshared field is one that has no aliases at all when it is read, that is, any object read from it is not accessible through any other variable (local, parameter or field).[6] An analysis ensures this property partly by keeping track of ("dynamic") aliases created by a read; the analysis rejects programs that may not uphold the property. We also use `unique` annotations on formal parameters and return values. The previously mentioned analysis ensures that the corresponding actual parameters are unaliased at the point of call and that a unique return value is unaliased when a method returns. For a more detailed comparison with the related terms used by other researchers, please see our related paper [2].

[5] The class is used to resolve the region but not to define inclusion.
[6] More precisely, the object *will not* be accessed through any other variable.

```
public class Vector {
  public region Elements, Size;
  private Object[] list in Elements;
  private int count in Size;

  public Vector() reads nothing writes nothing
  {
    this.list = new Object[10];
    this.count = 0;
  }

  public void addElement( Object obj )
      reads nothing writes this.Size, this.Elements, any(Array).[]
  {
    if( this.count == this.list.length ) {
      Object[] temp = new Object[this.list.length << 1];
      for( int i = 0; i < this.list.length; i++ )
        temp[i] = this.list[i];
      this.list = temp;
    }
    this.list[this.count++] = obj;
  }
  ...
}
```

Fig. 3. Dynamic array class implemented *without* unshared fields. Notice the exposure of `writes any(Array).[]` in method `addElement()`.

3.1 Using Unshared Fields

Let p refer to an object with an unshared field f that refers to a unique object. Because the object referenced by $p.f$ is *always* unaliased, it is reasonable to think of the contents of the referenced object as contents of the object p. Therefore, the instance regions of the object $p.f$ can be mapped into the regions of the object p. In this way, the existence of the object $p.f$ can be abstracted away. By default, the instance regions of the object referred to by an unshared field are mapped into the region that contains the unshared field (equivalent to mapping the region $p.f.$`Instance` into the region $p.f$). This can be changed by the programmer, however. In the examples, we extend the syntax of unshared fields to map regions of the object referenced by the unshared field into regions of the object containing the field. The mapping is enclosed in braces `{}`. (See the declaration of the field `list` in Figure 4.)

The mapping of instance regions of the unshared field (more precisely, those regions of the object under `Instance`) must preserve the region hierarchy. Consider an unshared field f of object o that maps region $f.q$ into region $o.p$. If region $f.q'$ is a descendant of $f.q$, then it must be mapped into a descendant of $o.p$.

```
public class Vector {
  public region Elements, Size;
  private unshared Object[] list in Elements {Instance in Elements};
  private int count in Size;

  public Vector() reads nothing writes nothing
  {
    this.list = new Object[10];
    this.count = 0;
  }

  public void addElement( Object obj )
      reads nothing writes this.Size, this.Elements
  {
    if( this.count == this.list.length ) {
      Object[] temp = new Object[this.list.length << 1];
      for( int i = 0; i < this.list.length; i++ )
        temp[i] = this.list[i];
      this.list = temp;
    }
    this.list[this.count++] = obj;
  }
  ...
}
```

Fig. 4. Dynamic array class implemented using unshared fields. Differences from Figure 3 are underlined. Now `writes any(Array).[]` is absent from the annotation of method `addElement`.

Example 4. Consider a class `Vector`, shown in Figure 3 that implements a dynamic array. Because nothing is known about the possible aliasing of the field `list`, it must be assumed that the array is aliased, and that any use of an element of the array referenced by `list` interferes with any other array. There are two main reasons that this is undesirable. First, by requiring an effect annotation using the region `[]`, it is revealed that the dynamic array is implemented with an array. There is no reason that the user of the class needs to know this. Second, the required effect annotation prevents the following two statements from being swapped, even when `vector1` and `vector2` are not aliased (recall that we intend to use this analysis to support semantics-preserving program manipulations).

```
vector1.addElement( obj1 );
vector2.addElement( obj2 );
```

There is no reason for the user of the class to expect that two distinct dynamic arrays should interfere with each other. The problem is that the implementation of `Vector` does not indicate that two instantiations are necessarily independent, although a simple inspection of the code would reveal that they are. By making the `list` field `unshared`, as in Figure 4, this information is provided. Now the

`Instance` region (which includes the `[]` region) of `list` is mapped into into the region `Elements` of the `Vector` object, and it is no longer necessary to include the effect `writes Array.[]` in the annotation of `addElement`.

3.2 Checking Method Effects (Reprise)

In the previous section, we mentioned in passing that effects on local variables could be omitted when checking whether a method had no more effects than permitted. Effects on newly created objects can also be omitted, and similarly effects on unique references passed in as parameters can be omitted because by virtue of passing them, the caller retains no access (similar to "linear" objects) and thus any effects are irrelevant.

Unshared fields affect how we check inferred effects against permitted effects. Specifically, the declared mappings are used to convert any effects on regions of the unshared field into effects on the regions of the owner. This process is called "elaboration" and is detailed in Appendix A.4. The irrelevant effects are then masked from the elaborated effects and the remaining effects checked against those in the method's annotation.

Example 5. Because we can use unshared fields to abstract away the implementation of a class, we can implement the same abstract data type in two different ways, but with both classes' methods having the same annotations. Consider the implementation of a dynamic array shown in Figure 5. The field `head` in the class `LinkedVector` is unshared. This captures the notion that each dynamic array maintains a unique reference to its list of elements. This is necessary, but not sufficient, to make the effect annotations on `addElement` correct. It is not sufficient because it only indicates that the *first* element of the linked list is unshared; it says nothing about any of the other elements. Because it is the intention that each dynamic array maintains a distinct linked list, the `next` field of `Node` is declared to be `unshared` as well.

Let us verify the annotation of the method `addElement()` of class `Node`: `reads nothing writes this.Next`. The effect of the condition of the `if` statement is `reads this.next`, and the effect of the `then`-branch is `reads x, writes this.next`. The effect on the parameter `x` can be ignored outside the method, and the other effects are covered by the declared effect `writes this.Next`. The `else`-branch is more interesting, having a recursive call to `addElement()`. Looking up the declared effects of `addElement()`, we find `reads nothing writes this.Next`. The actual value of the receiver must now be substituted into the method effects. The receiver in this case is `this.next`, which is an unshared field whose `Next` region is mapped into region `this.Next`. The effect of the method call is thus `writes this.Next` (instead of a write to region `Next` of the object `this.next`), which is covered by the declared effects.

Checking that the declared effects of `addElement()` of `LinkedVector` are correct (and the same as the effects of `Vector.addElement()`) is now straightforward. The creation of a new node does not produce effects noticeable outside of the method. The conditional statement reads and writes `this.head`,

```
class Node {
  public region Obj, Next;
  public Object obj in Obj;
  public unshared Node next in Next
    {Instance in Next};

  public unique Node( Object o, Node n ) reads nothing writes nothing
  {
    this.obj = o;
    this.next = n;
  }

  public void addElement( unique Node x )
      reads nothing writes this.Next
  {
    if( this.next == null ) this.next = x;
    else this.next.addElement( x );
  }
}

class LinkedVector {
  public region Elements, Size;
  private unshared Node head in Elements
    { Instance in Elements };
  private int count in Size;

  public LinkedVector() reads nothing writes nothing
  {
    this.head = null;
    this.count = 0;
  }

  public void addElement( Object obj )
      reads nothing writes this.Elements, this.Size
  {
    Node tail = new Node( obj, null );
    if( this.head == null )
      this.head = tail;
    else
      this.head.addElement( tail );
    this.count += 1;
  }
  ...
}
```

Fig. 5. A dynamic array implemented using a linked list. Note that the annotation on `addElement(Object)` is the same as in Figure 4.

and calls `addElement()` on `this.head`. The first two effects are covered by the declared effect `writes this.Elements`, while the effect of the method call is `writes this.Elements` because the method call writes the `Next` region of the unshared field `this.head`, which maps `Next` to `this.Elements`. Finally, incrementing `this.count` has the effect `writes this.count`, which is covered by the declared effect `writes this.Size`.

Example 6. Figure 6 gives a simple implementation of a dictionary using association lists with annotations that abstract away the implementation details. It demonstrates the utility of being able to map the regions of an unshared field into multiple regions of the owner. The same set of annotations could be used for a hash table using the built-in hash code method `System.identityHashCode()`.

Assuming we had a variable `t` declared with type `Table`, the annotations permit the reordering of `t.containsKey()` and `t.pair()`, but do not permit reordering of `t.containsKey()` with `t.clear()`. The annotation on `clear()` is the most precise possible.

4 Using Effect Analysis

We will now briefly describe how the effects analysis can be used when applying semantics-preserving program manipulations. A precondition of many such manipulations (typically those that cause code motion) is that there not be any data dependencies between the parts of the program affected by the manipulation. In general, the two computations interfere if one mutates state also accessed by the other. Assuming the effects system is sound (see Section 6), then if the computations interfere, there is an effect from one computation that "conflicts" with an effect from the second. Two effects conflict if at least one is a write effect and they involve targets that may "overlap," that is, may refer to the same mutable state at run time.

Two targets may overlap only if they refer to overlapping regions, and the hierarchical nature of regions ensures that regions overlap only if one is included in the other. Furthermore, an instance target (a use of an instance region of a particular object) can only overlap another instance target if the objects could be identical. The effect inference system computes effects (and targets) using the expressions in the computation in context. Thus equality cannot be compared devoid of context. This observation has led us to formalize the desired notion of equality in a new alias question $MayEqual(e, e')$ [3]. Steensgaard's "points-to" analysis [11] may be used as a conservative approximation. These considerations are further complicated by unshared fields. The details may be seen in the Appendix.

5 Related Work

Reynolds [10] showed how interference in Algol-like programs could be restricted using rules that prevent aliasing. His technique, while simple and general, requires access to the bodies of procedures being called in order to check whether

```
class Node {
  Object key, val;
  unshared Node next { Instance in Instance, key in key, val in val, next in next };

  unique Node(Object k, Object v, unique Node n) reads nothing writes nothing
  { key=k; val=v; next=n; }

  int count() reads next writes nothing
  { if (next == null) return 1; else return next.count()+1; }
  Object get(Object k) reads key, val, next writes nothing
  { if (k == key) return val; else if (next == null) return null;
    else return next.get(k); }
  .
  .
  .
  boolean containsKey(Object k) reads key, next writes nothing
  { return k == key || next != null && next.containsKey(k); }
  void pair() reads key, next writes val
  { val = key; if (next != null) next.pair(); else ;}
}

public class Table {
  region Table;
  region Key in Table, Value in Table, Structure in Table;

  private unshared Node list in Structure
    { Instance in Instance, key in Key, val in Val, next in Structure };

  /** Return number of (key,value) pairs */
  public int count() reads Structure writes nothing
  { if (list == null) return 0;
    else return list.count(); }

  /** Remove all entries from table. */
  public void clear() reads nothing writes Structure { list = null; }

  public Object get(Object k) reads Table writes nothing
  { if (list == null) return null;
    else return list.get(k); }

  public void put(Object k, Object v) reads nothing writes Table { ... }
  public void remove(Object k) reads nothing writes Table { ... }
  public boolean contains(Object v) reads Val,Structure writes nothing { ... }

  public boolean containsKey(Object k) reads Key,Structure writes nothing
  { return list != null && list.containsKey(k); }

  /** Pair each existing key with itself. */
  public void pair() reads Key,Structure writes Value
  { if (list != null) list.pair(); else ; }
}
```

Fig. 6. Simple dictionary class with effects annotations.

they operate on overlapping global variables. The work includes a type system with mutable records, but the records cannot have recursive type (lists and trees are not possible).

Jackson's Aspect system [7] uses a similar abstraction mechanism to our regions in order to specify the effects of routines on abstract data types. He uses specifications for each procedure in order to allow checking to proceed in a modular fashion. In his work, however, the specifications are *necessary* effects and are used to check for missing functionality.

Effects were first studied in the FX system [6], a higher-order functional language with reference cells. The burden of manual specification of effects is lifted through the use of effects inference as studied by Gifford, Jouvelot, and Talpin [8, 12]. The work was motivated by a desire to use stack allocation instead of heap allocation in mostly pure functional programs, and also to assist parallel code generation. The approach was demonstrated successfully for the former purpose in later work [1, 13]. These researchers also make use of a concept of disjoint "regions" of mutable state, but these regions are global, as opposed to within objects. In the original FX system, effects can be verified and exploited in separately developed program components.

Leino [9] defines "data groups," which are sets of instance fields in a class. Each method declares which data groups it could modify. As with regions, a subclass may add new instance fields to existing data groups, but unlike our proposal, a field may be added to more than one data group. In their system, this ability is sound because the modifies clause is only used to see if a given method will modify a given field, not to see if effects of two methods could interfere.

Numerous researchers have examined the question of unique objects, as detailed in a paper of ours [2]. Most recently, Clarke, Potter and Noble have have formalized a notion of "owner" that separates ownership from uniqueness [5]. Their notion of ownership appears useful for the kind of effects analysis we present here, although the precise mechanism remains further work.

6 Further Work

The work described in this paper is ongoing. We are continuing to extend our effects system to cover all of Java. We also intend to use the system to infer suggested annotations on methods. Ultimately, the effects system will be used to determine possible data dependencies between statements, which will require a form of alias analysis. This section sketches our ideas for further work.

An important aspect of a static effect system is for it to be *sound*, that is, it does not say two computations do not interfere through the access of some shared state when in fact they do. Proving soundness requires a run-time definition of interference which we must then show is conservatively approximated by our analysis. The run-time state of the program we are interested in includes fields of reachable objects as well as the state of local variables and temporaries in all activation frames. The effects of a computation can then be seen as reads and writes on elements of the state. These effects can then be compared to that of the second computation. Interference occurs when an element of state is written by one computation and read or written by the other. We have begun formulating a proof of correctness using the concept of what run-time state is accounted for by a target (is "covered" by a target), proving that if a target includes another target than it covers everything covered by the other.

Java permits a limited form of multiple inheritance in the form of code-less "interfaces." In order to permit useful annotations on method headers in interfaces, we must be able to add regions to interfaces and thus we must handle multiple inheritance of instance regions. (As with static fields, static regions are not inherited and thus do not complicate matters.) Multiple inheritance of instance regions is handled by permitting a class or interface inheriting regions from an interface to map them into other regions as long as the hierarchy is preserved. Any two regions, both of which are visible in a superclass or superinterface of the classes or interfaces performing the mapping, must have the same relation ($<$, $>$, or unrelated) in the latter class or interface. This restriction ensures that reasoning about regions remains sound. Conflicting relations in superclasses or superinterfaces may forbid certain inheritance combinations.

Java has *blank final* fields which are fields that are initialized in each constructor of the class that declared them but otherwise follow the restrictions on final fields—they may not be assigned. The Java language does not forbid methods from *reading* the value of a final field, even if the method is called in the constructor before the field is initialized. This situation is aggravated by classes which override methods called by the constructor in the superclass. In order to treat blank finals the same as finals (as immutable fields) and yet determine data dependencies correctly, extra annotations are needed on methods that may be called by a constructor. In particular, these methods must be annotated with the list of blank final fields that may be read. This annotation breaks abstraction to some degree but is needed only when checking the class itself and its subclasses.

Java supports multiple threads. So far we have ignored the possibility that state could be modified by a concurrent thread. As explained in a recent JavaWorld article [14], thread safety for the access of a field can be ensured in three situations:

1. the access is protected by a synchronization on the owning object;
2. the field is immutable ("final");
3. the object has a "thread-safe" wrapper.

The first two cases can be easily checked through syntactic checks. The third condition is satisfied when the field is accessed through unshared or unique references.

As discussed in our earlier work [4], the task of adding annotations can be made less burdensome if it can be semi-automated. The same technique used to test whether a method stays within the permitted effects annotation can help to generate an annotation for an unannotated method. Essentially, the set of effects inferred from the method is culled by removing redundant effects, and effects on regions inaccessible to the caller are promoted to effects on the nearest accessible parent region.

7 Conclusion

We have introduced an effects system for Java that permits an implementor to express the effects of a method in terms of abstract regions instead of in terms of mutable private fields. The effects system properly treats wholly-owned subsidiary objects as part of the owning object. We also introduce a way of checking the effects inferred from the body of a method against the effects permitted in a method's annotation. As described in this paper, this effects system can be used to check consistency of annotations. It can also be used to determine whether there may be data dependencies between two computations involving separately compiled code.

Acknowledgments

We thank our colleagues at CMU (William L. Scherlis and Edwin C. Chan) and UWM (Adam Webber) for comments on drafts of the paper and numerous conversations on the topic. We also thank the anonymous reviewers.

References

1. Alexander Aiken, Manuel Fähndrich, and Raph Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, San Diego, California, USA, *ACM SIGPLAN Notices*, 30(6):174–185, June 1995.
2. John Boyland. Deferring destruction when reading unique variables. Submitted to IWAOOS '99, March 1999.
3. John Boyland and Aaron Greenhouse. MayEqual: A new alias question. Submitted to IWAOOS '99, March 1999.
4. Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE '98)*, Kyoto, Japan, April 19–25, pages 167–176. IEEE Computer Society, Los Alamitos, California, 1998.
5. David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, October 18–22, *ACM SIGPLAN Notices*, 33(10):48–64, October 1998.
6. D. K. Gifford, P. Jouvelot, J. M. Lucassen, and M. A. Sheldon. FX-87 reference manual. Technical Report MIT/LCS/TR-407, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1987.
7. Daniel Jackson. Aspect: Detecting bugs with abstract dependencies. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.

8. Pierre Jouvelot and David K. Gifford. Algebraic reconstruction of types and effects. In *Conference Record of the Eighteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 303–310. ACM Press, New York, 1991.
9. K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *OOPSLA'98 Conference Proceedings—Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, October 18–22, *ACM SIGPLAN Notices*, 33(10), October 1998.
10. John C. Reynolds. Syntactic control of interference. In *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, pages 39–46. ACM Press, New York, January 1978.
11. Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the Twenty-third Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, St. Petersburg, Florida, USA, January 21–24, pages 32–41. ACM Press, New York, 1996.
12. Jean-Pierre Talpin and Pierre Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2(3):245–271, July 1992.
13. Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Conference Record of the Twenty-first Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, Portland, Oregon, USA, January 17–21, pages 188–201. ACM Press, New York, 1994.
14. Bill Venners. Design for thread safety: Design tips on when and how to use synchronization, immutable objects, and thread-safe wrappers. *JavaWorld*, August 1998.

A Definitions

Throughout the appendix, class names/types and variable names are represented as elements of a set of identifiers **Ide**. We assume that facilities are available for determining the type/class of an expression/variable, and that the class hierarchy is available for determining if two classes are related. We use the phrase "class c is a descendant of class c'" to mean that class c is equal to c' or c's superclass is a descendant of c'.

$$\begin{array}{rr} \mathit{var} \to \mathbf{Ide} & \mathit{type} \to \mathbf{Ide} \\ \mathit{name} \to \mathbf{Ide} & \mathit{class} \to \mathbf{Ide} \end{array}$$

A.1 Syntax of Programming Language

Several language features used in the text have been omitted for simplicity; the remaining features can express the omitted features. The method receiver object is never implicit. It is illegal to write to a formal parameter. Parent regions must always be specified, and there are no default mappings for unshared fields. We require instance fields/regions to be explicitly declared as such using the keyword `instance`. A preprocessing pass can convert a program from the language presented in the body of the paper into the one presented here.

$$
\begin{aligned}
\textit{program} &\rightarrow \textit{classDecl}^* \\
\textit{classDecl} &\rightarrow \texttt{class}\ \textit{name}\ \texttt{extends}\ \textit{name}\ \textit{classBody} \\
\textit{classBody} &\rightarrow \{\ \textit{classBodyDecl}^*\ \} \\
\textit{classBodyDecl} &\rightarrow \textit{region} \mid \textit{field} \mid \textit{method} \mid \textit{constructor} \\
\textit{region} &\rightarrow \textit{kind}\ \texttt{region}\ \textit{name}\ \texttt{in}\ \textit{name}\texttt{;} \\
\textit{field} &\rightarrow \textit{kind}\ \textit{type}\ \textit{name}\ \texttt{in}\ \textit{name}\texttt{;} \\
&\mid\ \textit{kind}\ \texttt{unshared}\ \textit{type}\ \textit{name}\ \texttt{in}\ \textit{name}\ \{\textit{mappings}\}\texttt{;} \\
\textit{mappings} &\rightarrow \textit{mapping} \mid \textit{mapping}\texttt{,}\ \textit{mappings} \\
\textit{mapping} &\rightarrow \textit{name}\ \texttt{in}\ \textit{name} \\
\textit{kind} &\rightarrow \texttt{instance} \mid \texttt{static} \\
\textit{method} &\rightarrow \textit{type}\ \textit{name}\ \texttt{(}\ \textit{params}\ \texttt{)}\ [\textit{annotation}]\ \textit{block} \\
\textit{constructor} &\rightarrow \textit{name}\ \texttt{(}\ \textit{params}\ \texttt{)}\ [\textit{annotation}]\ \textit{block} \\
\textit{annotation} &\rightarrow \texttt{reads}\ \textit{locations}\ \texttt{writes}\ \textit{locations} \\
\textit{locations} &\rightarrow \texttt{nothing} \mid \textit{targetSeq} \\
\textit{targetSeq} &\rightarrow \textit{location} \mid \textit{location}\texttt{,}\ \textit{targetSeq} \\
\textit{location} &\rightarrow \textit{var}\texttt{.}\textit{region} \mid \texttt{any(}\textit{class}\texttt{).}\textit{region} \mid \textit{class}\texttt{.}\textit{region} \\
\textit{params} &\rightarrow \epsilon \mid \textit{paramDecls} \\
\textit{paramDecls} &\rightarrow \textit{paramDecl} \mid \textit{paramDecl}\texttt{,}\ \textit{paramDecls} \\
\textit{paramDecl} &\rightarrow \textit{type}\ \textit{name} \\
\textit{block} &\rightarrow \{\ \textit{stmt}^*\ \} \\
\textit{stmt} &\rightarrow \textit{block} \mid \texttt{;} \mid \texttt{return;} \mid \texttt{return}\ \textit{expr}\ \texttt{;} \mid \textit{type}\ \textit{name}\texttt{;} \mid \\
&\quad\ \textit{expr}\ \texttt{=}\ \textit{expr}\texttt{;} \mid \textit{stmtExpr}\texttt{;} \mid \textit{constuctObj}\texttt{(}\textit{args}\texttt{);} \mid \\
&\quad\ \texttt{if(}\ \textit{expr}\ \texttt{)}\ \textit{stmt}\ \texttt{else}\ \textit{stmt} \mid \texttt{while(}\ \textit{expr}\ \texttt{)}\ \textit{stmt} \\
\textit{expr} &\rightarrow \textit{name} \mid \textit{rexpr} \mid \textit{vexpr} \\
\textit{rexpr} &\rightarrow \textit{stmtExpr} \mid \textit{allocExpr} \mid \textit{constuctObj} \mid \textit{expr}\ \texttt{[}\textit{expr}\texttt{]} \mid \textit{expr}\texttt{.}\textit{field} \\
\textit{vexpr} &\rightarrow \textit{expr} \oplus \textit{expr} \mid \ominus \textit{expr} \mid \texttt{true} \mid \texttt{1} \mid \texttt{null} \mid \textit{type} \\
\textit{args} &\rightarrow \epsilon \mid \textit{argsSeq} \\
\textit{argsSeq} &\rightarrow \textit{expr} \mid \textit{expr}\texttt{,}\ \textit{argsSeq} \\
\textit{stmtExpr} &\rightarrow \textit{expr}\texttt{.}\textit{method}\texttt{(}\textit{args}\texttt{)} \\
\textit{allocExpr} &\rightarrow \texttt{new}\ \textit{type}\ (\texttt{[}\textit{expr}\texttt{]})^+ \mid \texttt{new}\ \textit{class}\texttt{(}\textit{args}\texttt{)} \\
\textit{constuctObj} &\rightarrow \texttt{super} \mid \texttt{this}
\end{aligned}
$$

A.2 Tracking locals

We need to track the mutable state references that could occur in local variables. The analysis here computes a relation between local variables and such reference expressions (*rexpr* in the preceding grammar) that each variable may be equal to. The relation also permits a local to be paired to the initial value of a formal parameter. It is not necessary that these expressions are available at the analysis point or even side-effect-free for this purpose. This analysis is very similar to standard def-use dataflow analyses. The relations will be drawn from a set **VB** for "variable bindings."

$$
\begin{aligned}
\mathbf{Bindings} &= \textit{rexpr} + \mathbf{Ide} \\
\mathbf{VB} &= 2^{\mathbf{Ide} \times \mathbf{Bindings}}
\end{aligned}
$$

$$B \;:\; expr \to \mathbf{VB} \to \mathbf{Bindings}$$
$$B\; name\; V = \{b \mid (name, b) \in V\}$$
$$B\; rexpr\; V = \{rexpr\}$$
$$B\; vexpr\; V = \{\}$$

$$V \backslash name = V - \{(name, b) \mid b \in \mathbf{Bindings}\}$$

For each statement s, we define two sets V_s^- and V_s^+ as the least fixed point solution to the following set of equations defined with regard to the syntax of each method or constructor:

$\ldots\; name$ `(` $params$ `)` $\ldots\; block$
$\quad V_{block}^- = \{(var, var) \mid var \in params\}$
$block \to$ `{` $stmt_1 \;\ldots\; stmt_n$ `}`
$\quad V_{stmt_1}^-, \ldots, V_{stmt_n}^-, V_{block}^+ = V_{block}^-, V_{stmt_1}^+, \ldots, V_{stmt_n}^+$
$stmt \to block$
$\quad V_{block}^-, V_{stmt}^+ = V_{stmt}^-, V_{block}^+$
$stmt \to$ `;`
$\quad V_{stmt}^+ = V_{stmt}^-$
$stmt \to type\;\; name$`;`
$\quad V_{stmt}^+ = V_{stmt}^- \backslash name$
$stmt \to name$ `=` $expr$`;`
$\quad V_{stmt}^+ = (V_{stmt}^- \backslash name) \cup \left\{(name, b) \mid b \in B\; expr\; V_{stmt}^-\right\}$
$stmt \to expr$ `=` $expr$`;` (Otherwise)
$\quad V_{stmt}^+ = V_{stmt}^-$
$stmt \to stmtExpr$`;` $\mid$ $constructObj$`(`$args$`)`
$\quad V_{stmt}^+ = V_{stmt}^-$
$stmt \to$ `if (` $expr$ `)` $stmt_1$ `else` $stmt_2$
$\quad V_{stmt_1}^-, V_{stmt_2}^- = V_{stmt}^-, V_{stmt}^- \qquad V_{stmt}^+ = V_{stmt_1}^+ \cup V_{stmt_2}^+$
$stmt \to$ `while (` $expr$ `)` $stmt'$
$\quad V_{stmt'}^-, V_{stmt}^+ = V_{stmt}^- \cup V_{stmt'}^+, V_{stmt}^- \cup V_{stmt'}^+$

When analyzing a target involving a local variable v, we will need the set of binding for that use. Let $B\,v$ be shorthand for $B\,v\,V_s^-$ where s is the immediately enclosing statement.

A.3 Formalization of Regions and Effects

A region is a triple $(class, name, tag)$, where $class$ is the class in which the region is declared, $name$ is the name of the region, and tag indicates whether the region is static or instance. We require the name of each region to be unique.

$$\mathbf{Regions}_I = \mathbf{Ide} \times \mathbf{Ide} \times \{\mathsf{instance}\} \text{ (Set of instance regions)}$$
$$\mathbf{Regions}_S = \mathbf{Ide} \times \mathbf{Ide} \times \{\mathsf{static}\} \text{ (Set of static regions)}$$
$$\mathbf{Regions} = \mathbf{Regions}_I \cup \mathbf{Regions}_S \;=\; \mathbf{Ide} \times \mathbf{Ide} \times \{\mathsf{static}, \mathsf{instance}\}$$

An effect is a read of or write to one of four kinds of targets, which represent different granularities of state.

$$\begin{array}{c} \mathit{effect} \to \mathsf{read}(\mathit{target}) \mid \mathsf{write}(\mathit{target}) \\ \mathit{target} \to \mathsf{local}\,\mathbf{Ide} \mid \mathsf{instance}\,\mathit{expr}\times\mathbf{Regions}_I \mid \mathsf{anyInstance}\,\mathbf{Regions}_I \mid \mathsf{static}\,\mathbf{Regions}_S \end{array}$$

The region hierarchy is a triple $(\mathit{regions}, \mathsf{parent}_R, \mathit{unshared})$, where *regions* is the set of regions in the program, parent_R is the parent ordering of the regions; $r\,\mathsf{parent}_R\,s \Leftrightarrow s$ is the super region of r; and *unshared* is the mapping of regions of unshared fields to regions of the class. The mapping is a set of 4-tuples $(\mathit{class}, \mathit{field}, \mathit{region}_u, \mathit{region}_c)$ with the interpretation that *field* is an unshared field of instances of class *class*, and region region_u of the object referenced the field is mapped into region region_c of the object containing *field*.

$$\begin{array}{rl} \mathbf{Map} &= \mathbf{Ide}\times\mathbf{Ide}\times\mathbf{Regions}\times\mathbf{Regions} \\ \mathbf{Hier} &= 2^{\mathbf{Regions}}\times(\mathbf{Regions}\to\mathbf{Regions})\times 2^{\mathbf{Map}} \\ \mathsf{All} &= (\texttt{Object}, \texttt{All}, \mathsf{static}) \\ \mathsf{Instance} &= (\texttt{Object}, \texttt{Instance}, \mathsf{instance}) \\ \mathsf{Element} &= (\texttt{Array}, \texttt{[]}, \mathsf{instance}) \\ \mathbf{H}_0 &= (\{\mathsf{All}, \mathsf{Instance}, \mathsf{Element}\}, \{(\mathsf{Instance}, \mathsf{All}), (\mathsf{Element}, \mathsf{Instance})\}, \emptyset) \end{array}$$

$$\begin{array}{rcl} \mathsf{lookup} &:& 2^{\mathbf{Regions}}\to\mathbf{Ide}\to\mathbf{Ide}\to\mathbf{Regions} \\ \mathsf{lookup}\ r\ c\ f &=& (c', f, t)\in r \text{ s.t. class } c \text{ is a descendant of class } c' \end{array}$$

The region hierarchy $(\mathit{rgns}, \mathsf{parent}_R, m)$ is built from $\mathbf{H}_0$, the initial hierarchy, by analyzing the field and region declarations in the program's class definitions. It must have the following properties: no region may have more than one parent; the region `Instance` of an unshared field must be mapped into a region of the owning object; the unshared mappings must respect the tree structure; only instance regions of unshared fields may be mapped; and regions of a static unshared field may only be mapped into static regions.

$$\begin{array}{c} \forall(c,r,p)\in \mathit{rgns}.\,\forall(c,r,p')\in\mathit{rgns}.\ p=p' \\ (\exists(c,f,r_u,r_c)\in m)\Rightarrow(\exists(c,f,\mathsf{Instance},q)\in m) \\ \forall(c,f,r_u,r_c)\in m.\left(\begin{array}{c}(\forall(c,f,r'_u,r'_c)\in m.\ r'_u\le_R r_u\Rightarrow r'_c\le_R r_c) \\ \wedge\,(r_u\le_R \mathsf{Instance})\wedge(f\in\mathbf{Regions}_S\Rightarrow r_c\in\mathbf{Regions}_S)\end{array}\right) \end{array}$$

Inclusion Given the region hierarchy $(\mathit{rgns}, \mathsf{parent}_R, m)$, we create the reflexive transitive closure of parent_R to define the inclusion relation over regions.

$$\begin{array}{l} \le_R\ \subseteq \mathbf{Regions}\times\mathbf{Regions} \\ \le_R\ = (\mathsf{parent}_R \cup \{(r,r)\mid r\in\mathit{rgns}\})^* \end{array}$$

The inclusion relation over targets, $\le_T$, is defined to be the reflexive transitive closure of a relation, parent_T.

$$
\begin{aligned}
\mathsf{unshared}\, e &\Leftrightarrow e = e'\,.\,f \wedge \exists(c, f, r_u, r_c) \in m \\
\mathsf{shared}\, e &\Leftrightarrow \neg \mathsf{unshared}\, e
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{parent}_T &\subseteq \ \textit{target} \times \textit{target} \\
\mathsf{local}\, v \ \mathsf{parent}_T \ \mathsf{local}\, w &\Leftrightarrow v = w \\
\mathsf{instance}\,(e, r) \ \mathsf{parent}_T \ \mathsf{instance}\,(e, r') &\Leftarrow r \ \mathsf{parent}_R \ r' \\
\mathsf{instance}\,(e\,.\,f, r) \ \mathsf{parent}_T \ \mathsf{instance}\,(e, r') &\Leftarrow \mathsf{lookup}\ \textit{rgns}\ (\mathsf{typeof}\ e)\ f \in \mathbf{Regions}_I \\
&\quad \wedge\ \exists(c, f, r, r') \in m \\
\mathsf{instance}\,(e, r) \ \mathsf{parent}_T \ \mathsf{anyInstance}\, r &\Leftarrow \mathsf{shared}\, e \\
\mathsf{instance}\,(e\,.\,f, r) \ \mathsf{parent}_T \ \mathsf{static}\, s &\Leftarrow \mathsf{lookup}\ \textit{rgns}\ (\mathsf{typeof}\ e)\ f \in \mathbf{Regions}_S \\
&\quad \wedge\ \exists(c, f, r, s) \in m \\
\mathsf{anyInstance}\, r \ \mathsf{parent}_T \ \mathsf{anyInstance}\, r' &\Leftarrow r \ \mathsf{parent}_R \ r' \\
\mathsf{anyInstance}\, r \ \mathsf{parent}_T \ \mathsf{static}\, s &\Leftarrow r \ \mathsf{parent}_R \ s \\
\mathsf{static}\, s \ \mathsf{parent}_T \ \mathsf{static}\, s' &\Leftarrow s \ \mathsf{parent}_R \ s'
\end{aligned}
$$

$$
\begin{aligned}
\leq_T &\subseteq \ \textit{target} \times \textit{target} \\
\leq_T &= \ (\mathsf{parent}_T \cup \{(t, t) \mid t \in \textit{target}\})^*
\end{aligned}
$$

Finally, effect inclusion is defined using $\leq_T$ in the obvious manner.

$$
\begin{aligned}
\leq_E &\subseteq \ \textit{effect} \times \textit{effect} \\
\mathsf{read}(t) \leq_E \mathsf{read}(t') &\Leftrightarrow t \leq_T t' \\
\mathsf{write}(t) \leq_E \mathsf{write}(t') &\Leftrightarrow t \leq_T t' \\
\mathsf{read}(t) \leq_E \mathsf{write}(t') &\Leftrightarrow t \leq_T t'
\end{aligned}
$$

Method Annotations The method annotations are used to construct a method dictionary: a set of tuples $(\textit{class}, \textit{method}, \textit{params}, \textit{effects})$, where $\textit{class}.\textit{method}$ is the method (or constructor) name, *params* is a vector of type–identifier pairs that captures the method's signature and formal parameter names, and *effects* is the set of effects that the method produces. The method dictionary is produced by analyzing the annotations of the methods in each defined class. For a program p, with region hierarchy $(r, \mathsf{parent}_R, m)$, the set of method dictionary is $\mathcal{A}_P[\![p]\!]\, r\, \emptyset$.

$$
\begin{aligned}
\mathbf{Method} &= \mathbf{Ide} \times \mathbf{Ide} \times (\mathbf{Ide} \times \mathbf{Ide})^* \times 2^{\textit{effect}} \\
\mathcal{A}_P &: \textit{program} \rightarrow 2^{\mathbf{Regions}} \rightarrow 2^{\mathbf{Method}} \rightarrow 2^{\mathbf{Method}} \\
\mathcal{A}_C &: \textit{classDecl} \rightarrow 2^{\mathbf{Regions}} \rightarrow 2^{\mathbf{Method}} \rightarrow 2^{\mathbf{Method}} \\
\mathcal{A}_B &: \textit{classBody} \rightarrow 2^{\mathbf{Regions}} \rightarrow \mathbf{Ide} \rightarrow 2^{\mathbf{Method}} \rightarrow 2^{\mathbf{Method}} \\
\mathcal{A}_D &: \textit{classBodyDecl} \rightarrow 2^{\mathbf{Regions}} \rightarrow \mathbf{Ide} \rightarrow 2^{\mathbf{Method}} \rightarrow 2^{\mathbf{Method}} \\
\mathcal{A}_A &: \textit{annotation} \rightarrow 2^{\mathbf{Regions}} \rightarrow 2^{\textit{effect}} \\
\mathcal{A}_T &: \textit{locations} \rightarrow 2^{\mathbf{Regions}} \rightarrow 2^{\textit{target}} \\
\mathcal{A}_F &: \textit{params} \rightarrow (\mathbf{Ide} \times \mathbf{Ide})^*
\end{aligned}
$$

$$\begin{aligned}
\mathcal{A}_P[\![\mathit{classDecl}^*]\!]\, r\, a \;&=\; (\mathcal{A}_C[\![\mathit{classDecl}_n]\!] r) \cdots (\mathcal{A}_C[\![\mathit{classDecl}_1]\!] r\, a)\\
\mathcal{A}_C[\![\texttt{class}\ c\ \texttt{extends}\ p\ \mathit{classBody}]\!]\, r\, a \;&=\; \mathcal{A}_B[\![\mathit{classBody}]\!] r\, c\, a\\
\mathcal{A}_B[\![\{\, \mathit{classBodyDecl}^* \}]\!]\, r\, c\, a &= (\mathcal{A}_D[\![\mathit{classBodyDecl}_n]\!] r\, c)\\
&\quad \cdots (\mathcal{A}_D[\![\mathit{classBodyDecl}_1]\!] r\, c)\, a\\
\mathcal{A}_D[\![\mathit{region}]\!]\, r\, c\, a &= a\\
\mathcal{A}_D[\![\mathit{field}]\!]\, r\, c\, a &= a\\
\mathcal{A}_D[\![\mathit{type}\ \mathit{name}\ \texttt{(}\mathit{params}\texttt{)}\ \mathit{anno}\ b]\!]\, r\, c\, a &= a\, \cup\, \{(c, \mathit{name}, \mathcal{A}_F[\![\mathit{params}]\!], \mathcal{A}_A[\![\mathit{anno}]\!] r)\}\\
\mathcal{A}_D[\![c\ \texttt{(}\mathit{params}\texttt{)}\ \mathit{anno}\ b]\!]\, r\, c\, a &= a\, \cup\, \{(c, c, \mathcal{A}_F[\![\mathit{params}]\!], \mathcal{A}_A[\![\mathit{anno}]\!] r)\}\\
\mathcal{A}_A[\![\epsilon]\!]\, r &= \{\mathsf{writes}(\mathsf{All})\}\\
\mathcal{A}_A[\![\texttt{reads}\ \mathit{locs}_R\ \texttt{writes}\ \mathit{locs}_W]\!]\, r \;&=\; \bigcup\nolimits_{t \in \mathcal{A}_T[\![\mathit{locs}_R]\!] r} \mathsf{read}(t)\, \cup\, \bigcup\nolimits_{t \in \mathcal{A}_T[\![\mathit{locs}_W]\!] r} \mathsf{write}(t)\\
\mathcal{A}_T[\![\texttt{nothing}]\!]\, r &= \emptyset\\
\mathcal{A}_T[\![\mathit{location}\,\texttt{,}\ \mathit{targetSeq}]\!]\, r &= \mathcal{A}_T[\![\mathit{location}]\!] r\, \cup\, \mathcal{A}_T[\![\mathit{targetSeq}]\!] r\\
\mathcal{A}_T[\![\mathit{var}\,\texttt{.}\,\mathit{region}]\!]\, r &= \{\mathsf{instance}\,(\mathit{var}, \mathsf{lookup}\ r\ (\mathsf{typeof}\ \mathit{var})\ \mathit{region})\}\\
\mathcal{A}_T[\![\mathit{class}\,\texttt{.}\,\mathit{region}]\!]\, r &= \{\mathsf{static}\,(\mathsf{lookup}\ r\ \mathit{class}\ \mathit{region})\}\\
\mathcal{A}_T[\![\texttt{any(}\mathit{class}\texttt{)}\,\texttt{.}\,\mathit{region}]\!]\, r &= \{\mathsf{anyInstance}\,(\mathsf{lookup}\ r\ \mathit{class}\ \mathit{region})\}\\
\mathcal{A}_F[\![\epsilon]\!] &= ()\\
\mathcal{A}_F[\![\mathit{type}\ \mathit{name}]\!] &= (\mathit{type}, \mathit{name})\\
\mathcal{A}_F[\![\mathit{type}\ \mathit{name}\,\texttt{,}\ \mathit{params}]\!] &= ((\mathit{type}, \mathit{name}) : \mathcal{A}_P[\![\mathit{params}]\!])
\end{aligned}$$

$$\begin{aligned}
\mathsf{typeof}\ &:\ \mathit{expr} \to \mathbf{Ide}\\
\mathsf{typeof}\ e\ &=\ \text{static class of expression } e
\end{aligned}$$

A.4 Computing Effects

The effects of an expression e in program p, with region hierarchy $(r, \mathsf{parent}_R, m)$ and effects dictionary d, are $\mathcal{E}[\![e]\!]\ (r, \mathsf{parent}_R, m)\ d$.

$$\mathcal{E}[\![-]\!] : \mathbf{Hier} \to 2^{\mathbf{Method}} \to 2^{\mathit{effect}}$$

Features that do not have effects

$$\begin{array}{rrr}
\mathcal{E}[\![\texttt{;}]\!]\ h\ d = \emptyset & \mathcal{E}[\![\mathit{type}]\!]\ h\ d = \emptyset & \mathcal{E}[\![\texttt{true}]\!]\ h\ d = \emptyset\\
\mathcal{E}[\![\texttt{return;}]\!]\ h\ d = \emptyset & \mathcal{E}[\![\texttt{1}]\!]\ h\ d = \emptyset & \mathcal{E}[\![\texttt{null}]\!]\ h\ d = \emptyset
\end{array}$$

Features that do not have direct effects

$$\begin{aligned}
\mathcal{E}[\![\texttt{\{}\ \mathit{stmt}^*\ \texttt{\}}]\!]\ h\ d &= \bigcup \mathcal{E}[\![\mathit{stmt}_i]\!] h\ d\\
\mathcal{E}[\![\texttt{return}\ \mathit{expr}\texttt{;}]\!]\ h\ d &= \mathcal{E}[\![\mathit{expr}]\!] h\ d\\
\mathcal{E}[\![\mathit{expr}_1\ \texttt{=}\ \mathit{expr}_2\texttt{;}]\!]\ h\ d &= \mathcal{E}_{\text{LVal}}[\![\mathit{expr}_1]\!] h\ d\, \cup\, \mathcal{E}[\![\mathit{expr}_2]\!] h\ d\\
\mathcal{E}[\![\texttt{if(}\ \mathit{expr}\ \texttt{)}\ \mathit{stmt}_1\ \texttt{else}\ \mathit{stmt}_2]\!]\ h\ d &= \mathcal{E}[\![\mathit{expr}]\!] h\ d\, \cup\, \mathcal{E}[\![\mathit{stmt}_1]\!] h\ d\, \cup\, \mathcal{E}[\![\mathit{stmt}_2]\!] h\ d\\
\mathcal{E}[\![\texttt{while(}\ \mathit{expr}\ \texttt{)}\ \mathit{stmt}]\!]\ h\ d &= \mathcal{E}[\![\mathit{expr}]\!] h\ d\, \cup\, \mathcal{E}[\![\mathit{stmt}]\!] h\ d\\
\mathcal{E}[\![\mathit{expr}_1 \oplus \mathit{expr}_2]\!]\ h\ d &= \mathcal{E}[\![\mathit{expr}_1]\!] h\ d\, \cup\, \mathcal{E}[\![\mathit{expr}_2]\!] h\ d\\
\mathcal{E}[\![\ominus \mathit{expr}]\!]\ h\ d &= \mathcal{E}[\![\mathit{expr}]\!] h\ d\\
\mathcal{E}[\![\mathit{expr}\,\texttt{,}\ \mathit{argsSeq}]\!]\ h\ d &= \mathcal{E}[\![\mathit{expr}]\!] h\ d\, \cup\, \mathcal{E}[\![\mathit{argsSeq}]\!] h\ d\\
\mathcal{E}[\![\texttt{new}\ \mathit{type}\,(\texttt{[}\mathit{expr}\texttt{]})^+]\!]\ h\ d &= \bigcup \mathcal{E}[\![\mathit{expr}_i]\!] h\ d
\end{aligned}$$

Features that have direct effects Reading or writing to a local variable produces an effect on a local target. Array subscripting produces an effect on the array's region `[]`. Accessing a field produces an effect on an instance or static target, depending on whether the field is an instance or static field, respectively.

$$
\begin{aligned}
\mathcal{E}[\![\mathit{var}]\!]\ h\ d &= \{\mathsf{read}(\mathsf{local}\ \mathit{var})\}\\
\mathcal{E}[\![\mathit{constructObj}]\!]\ h\ d &= \{\mathsf{read}(\mathsf{local}\,\mathtt{this})\}\\
\mathcal{E}[\![\mathit{expr}_1\,\mathtt{[}\mathit{expr}_2\mathtt{]}]\!]\ h\ d &= \mathcal{E}[\![\mathit{expr}_1]\!]h\ d \cup \mathcal{E}[\![\mathit{expr}_2]\!]h\ d\\
&\quad \cup \{\mathsf{read}(\mathsf{instance}\,(\mathit{expr}_1, \mathsf{Element}))\}\\
\mathcal{E}[\![\mathit{expr}\,\mathtt{.}\,\mathit{field}]\!]\ (r, \mathsf{parent}_R, m)\ d &= \mathcal{E}[\![\mathit{expr}]\!](r, \mathsf{parent}_R, m)\ d \cup \{\mathsf{read}(t)\}\\
\text{where } \mathit{rgn} &= \mathsf{lookup}\ r\ (\mathsf{typeof}\ \ \mathit{expr})\ \mathit{field}\\
t &= \begin{cases} \mathsf{instance}\,(\mathit{expr}, \mathit{rgn}) & \text{if } \mathit{rgn} \in \mathbf{Regions}_I\\ \mathsf{static}\,\mathit{rgn} & \text{if } \mathit{rgn} \in \mathbf{Regions}_S \end{cases}
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}_{\mathrm{LVal}}[\![\mathit{var}]\!]\ h\ d &= \{\mathsf{write}(\mathsf{local}\ \mathit{var})\}\\
\mathcal{E}_{\mathrm{LVal}}[\![\mathit{expr}_1\,\mathtt{[}\mathit{expr}_2\mathtt{]}]\!]\ h\ d &= \mathcal{E}[\![\mathit{expr}_1]\!]h\ d \cup \mathcal{E}[\![\mathit{expr}_2]\!]h\ d\\
&\quad \cup \{\mathsf{write}(\mathsf{instance}\,(\mathit{expr}_1, \mathsf{Element}))\}\\
\mathcal{E}_{\mathrm{LVal}}[\![\mathit{expr}\,\mathtt{.}\,\mathit{field}]\!]\ (r, \mathsf{parent}_R, m)\ d &= \mathcal{E}[\![\mathit{expr}]\!](r, \mathsf{parent}_R, m)\ d \cup \{\mathsf{write}(t)\}\\
\text{where } \mathit{rgn} &= \mathsf{lookup}\ r\ (\mathsf{typeof}\ \ \mathit{expr})\ \mathit{field}\\
t &= \begin{cases} \mathsf{instance}\,(\mathit{expr}, \mathit{rgn}) & \text{if } \mathit{rgn} \in \mathbf{Regions}_I\\ \mathsf{static}\,\mathit{rgn} & \text{if } \mathit{rgn} \in \mathbf{Regions}_S \end{cases}
\end{aligned}
$$

Method and constructor calls In addition to any effects produced by evaluating the actual parameters and the receiver, method calls also have the effects produced by the method. These are found by first looking up the method being called in the method dictionary (using find, which uses the language specific static method resolution semantics), and then substituting the expression of the actual parameter for the formal in any effects with instance targets. The helper function process creates a vector of type–expression pairs out of the argument list; the types are used in find, while the expressions are used in pair. The function pair is used to create a set of formal–actual parameter pairs. The actual substitutions are made by the function replace.

$$
\begin{aligned}
\mathcal{E}[\![\mathit{expr}\,\mathtt{.}\,\mathit{method}\,\mathtt{(}\mathit{args}\mathtt{)}]\!]\ h\ d &= \mathcal{E}[\![\mathit{expr}]\!]h\ d \cup \mathcal{E}[\![\mathit{args}]\!]h\ d\\
&\quad \cup \mathsf{replace}\ e\ (\mathsf{pair}\ p\ l) \cup \{(\mathtt{this}, \mathit{expr})\}\\
\text{where } l &= \mathsf{process}\ \mathit{args}\\
(c, \mathit{method}, p, e) &= \mathsf{find}\ d\ (\mathsf{typeof}\ \ \mathit{expr})\ \mathit{method}\ l
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![\mathtt{new}\ \mathit{class}\,\mathtt{(}\mathit{args}\mathtt{)}]\!]\ h\ d &= \mathcal{E}[\![\mathit{args}]\!]h\ d\ \cup\ \mathsf{replace}\ e\ (\mathsf{pair}\ p\ l)\\
\text{where } l &= \mathsf{process}\ \mathit{args}\\
(\mathit{class}, \mathit{class}, p, e) &= \mathsf{find}\ d\ \mathit{class}\ \mathit{class}\ l
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}[\![\mathit{constructObj}\,\mathtt{(}\mathit{args}\mathtt{);}]\!]\ h\ d &= \mathcal{E}[\![\mathit{args}]\!]h\ d\ \cup\ \mathsf{replace}\ e\ (\mathsf{pair}\ p\ l)\\
\text{where } l &= \mathsf{process}\ \mathit{args}\\
(c, \mathit{method}, p, e) &= \mathsf{find}\ d\ (\mathsf{typeof}\ \ \mathit{constructObj})\ (\mathsf{typeof}\ \ \mathit{constructObj})\ l
\end{aligned}
$$

$$\begin{aligned}
\mathsf{find} &: 2^{\mathbf{Method}} \to \mathbf{Ide} \to \mathbf{Ide} \to (\mathbf{Ide} \times \mathbf{Ide})^* \to \mathbf{Method} \\
\mathsf{find}\ d\ c\ m\ l &= (c', m, p, e) \in d \text{ subject to language's} \\
&\quad\ \text{static method resolution mechanism}
\end{aligned}$$

$$\begin{aligned}
\mathsf{process} &: \mathit{args} \to (\mathbf{Ide} \times \mathit{expr})^* \\
\mathsf{process}\ \epsilon &= () \\
\mathsf{process}\ \mathit{expr} &= (\mathsf{typeof}\ \mathit{expr}, \mathit{expr}) \\
\mathsf{process}\ \mathit{expr}\ \mathbf{,}\ \mathit{argSeq} &= ((\mathsf{process}\ \mathit{expr}) : (\mathsf{process}\ \mathit{argSeq}))
\end{aligned}$$

$$\begin{aligned}
\mathsf{pair} &: (\mathbf{Ide} \times \mathbf{Ide})^* \to (\mathbf{Ide} \times \mathit{expr})^* \to 2^{\mathbf{Ide} \times \mathit{expr}} \\
\mathsf{pair}\ ()\ () &= \emptyset \\
\mathsf{pair}\ ((t, n) : p)\ ((t', e) : l) &= \{(n, e)\} \cup \mathsf{pair}\ p\ l
\end{aligned}$$

$$\begin{aligned}
\mathsf{replace} &: 2^{\mathit{effect}} \to 2^{\mathbf{Ide} \times \mathit{expr}} \to 2^{\mathit{effect}} \\
\mathsf{replace}\ e\ p &= R' \cup W' \cup (e - (R \cup W)) \\
\text{where } R &= \{\mathsf{read}((f, r)) \in e \mid (f, a) \in p\} \\
W &= \{\mathsf{write}((f, r)) \in e \mid (f, a) \in p\} \\
R' &= \{\mathsf{read}((a, r)) \mid \mathsf{read}((f, r)) \in R \wedge (f, a) \in p\} \\
W' &= \{\mathsf{write}((a, r)) \mid \mathsf{write}((f, r)) \in W \wedge (f, a) \in p\}
\end{aligned}$$

Elaborating Effects and Masking Sometimes we need to expand a set of effects to take into account the bindings of local variables and the mapping of unshared fields. In particular we need this *elaboration* when we check a method annotation.

$$\begin{aligned}
\mathsf{elaborate}(E) = &\left\{\mathsf{read}(t) \mid t \in \mathsf{elaborate}(\left\{t'\right\}), \mathsf{read}(t') \in E\right\} \\
\cup &\left\{\mathsf{write}(t) \mid t \in \mathsf{elaborate}(\left\{t'\right\}), \mathsf{write}(t') \in E\right\}
\end{aligned}$$

$$\begin{aligned}
\mathsf{elaborate}(T) = &\ \text{smallest set } T' \supseteq T \text{ such that} \\
&\quad (\mathsf{instance}\,(v, r) \in T', e \in B\, v) \Rightarrow \mathsf{instance}\,(e, r) \in T' \\
&\quad \mathsf{instance}\,(e.f, r) \in T' \Rightarrow \mathsf{instance}\,(e, r') \in T' \\
&\quad\ \text{where } r' = \mathit{min}_{\leq_R} \{r_c \mid (c, f, r_u, r_c) \in m, r \leq_R r_u\}
\end{aligned}$$

Effect masking is accomplished by dropping out effects on local variables, regions of local variables (handled in elaboration), regions in newly allocated objects, and regions of unshared fields (which are redundant after elaboration):

$$\begin{aligned}
\mathsf{mask}(E) = &\left\{\mathsf{read}(t) \mid t \in \mathsf{mask}(\left\{t'\right\}), \mathsf{read}(t') \in E\right\} \\
\cup &\left\{\mathsf{write}(t) \mid t \in \mathsf{mask}(\left\{t'\right\}), \mathsf{write}(t') \in E\right\}
\end{aligned}$$

$$\begin{aligned}\mathsf{mask}(T) = \{t \in T \mid t & \text{ not of the forms } \mathsf{local}\, v \text{ (where } v \text{ is any local or parameter),}\\ & \mathsf{instance}\,(v, r) \text{ (where } v \text{ is a local variable, but not a parameter)}\\ & \mathsf{instance}\,(\mathit{allocExpr}, r)\text{, or } \mathsf{instance}\,(e, r) \text{ (where } \mathsf{unshared}\, e)\}\end{aligned}$$

A.5 Checking Annotated Effects

The annotated effects of a method, E_A, need to be checked against the computed effects of the method, E_C. If E_A does not account for every possible effect of the method, the annotation is invalid. Formally, the annotated effects of a method are valid if and only if $\forall e \in \mathsf{mask}(\mathsf{elaborate}(E_C)).\, \exists e' \in E_A.\ e \leq_E e'$

A.6 Conflicts and Interference

Two effects *conflict* if and only if $\mathsf{read}(t_1)$ is included (by $\leq_E$) in one of the effects, $\mathsf{write}(t_2)$ is included in the other effect, and $t_1\, \mathsf{overlap}_T\, t_2$. This can be extended to sets of effects: E_1 and E_2 conflict if and only if $\exists e_1 \in \mathsf{elaborate}(E_1)$. $\exists e_2 \in \mathsf{elaborate}(E_2)$. e_1 and e_2 conflict. Finally, two computations *interfere* if and only if they have conflicting sets of effects. Target overlap, $\mathsf{overlap}_T$, is the symmetric closure of $\mathsf{overlap}_0$.

$$\begin{gathered}\mathsf{overlap}_R \subseteq \mathbf{Regions} \times \mathbf{Regions}\\ r\, \mathsf{overlap}_R\, r' \Leftrightarrow r \leq_R r' \vee r' \leq_R r\end{gathered}$$

$$\begin{gathered}\mathsf{overlap}_T \subseteq \mathit{target} \times \mathit{target}\\ \mathsf{overlap}_T = \mathsf{overlap}_0 \cup \{(t', t) \mid (t, t') \in \mathsf{overlap}_0\}\end{gathered}$$

$$\begin{aligned}\mathsf{overlap}_0 &\subseteq \mathit{target} \times \mathit{target}\\ \mathsf{local}\, v\ \mathsf{overlap}_0\ \mathsf{local}\, w &\Leftarrow v = w\\ \mathsf{instance}\,(e, r)\ \mathsf{overlap}_0\ \mathsf{instance}\,(e', r') &\Leftarrow \mathit{MayEqual}\,(e, e') \wedge r\, \mathsf{overlap}_R\, r'\\ \mathsf{instance}\,(e, r)\ \mathsf{overlap}_0\ \mathsf{anyInstance}\, r' &\Leftarrow \mathsf{shared}\, e \wedge r\, \mathsf{overlap}_R\, r'\\ \mathsf{instance}\,(e, r)\ \mathsf{overlap}_0\ \mathsf{static}\, s &\Leftarrow \mathsf{shared}\, e \wedge r\, \mathsf{overlap}_R\, s\\ \mathsf{anyInstance}\, r\ \mathsf{overlap}_0\ \mathsf{anyInstance}\, r' &\Leftarrow r\, \mathsf{overlap}_R\, r'\\ \mathsf{anyInstance}\, r\ \mathsf{overlap}_0\ \mathsf{static}\, s &\Leftarrow r\, \mathsf{overlap}_R\, s\\ \mathsf{static}\, s\ \mathsf{overlap}_0\ \mathsf{static}\, s' &\Leftarrow s\, \mathsf{overlap}_R\, s'\end{aligned}$$