Alias Killing: Unique Variables Without Destructive Reads

John Boyland[*]

Abstract

An unshared object can be accessed without regard to possible conflicts with other parts of a system, whether concurrent or single-threaded. A *unique variable* (sometimes known as a "free" or "linear" variable) is one that either is null or else refers to an unshared object. Being able to declare and check which variables are unique improves a programmer's ability to avoid program faults.

In previously described uniqueness extensions to imperative languages, a unique variable can be accessed only with a *destructive read*, which nullifies it after the value is obtained. This approach suffers from several disadvantages: the use of destructive reads increases the complexity of the program which must continually restore nullified values; adding destructive reads changes the semantics of the programming language; and many of the nullifications are actually unnecessary.

We demonstrate instead that uniqueness can be preserved through the use of existing language features. We give a modular analysis that checks (nonexecutable) uniqueness annotations superimposed on an imperative programming language without destructive reads.

1 Introduction

Several researchers (including Wadler [20], Hogg [11], Baker [3] and Minsky [14]) have proposed extending programming languages to express the concept of a unique or linear variable. Such a variable is either empty or else its value is the sole reference to an object. The object is then known as an *unshared* object. An unshared object can be destructively updated without affecting any other variable. Destructive updates on unshared objects can thus be permitted even in a language without side-effects. In an imperative language, the holder of an unshared object knows that no other part of the system can mutate the object. For instance, if a module has sole access to a file stream object, then its implementation can be changed to buffer input from the file, with the client of the module observing no change of behavior. Being able to declare which variables are expected to be unique and to have these declarations checked can make code more understandable and maintainable.

Extra rules are needed to ensure that the objects referred to by unique variables are indeed unshared. An object is unshared when first created and it will remain unshared as long as the reference is stored in only one variable. Similarly, if a unique variable is read at most once, any resulting object will be unshared. Baker proposes that unique variables be designated "use once" with a simple syntactic check to ensure the variable is used exactly once in every execution path during its lifetime. This restriction comes directly from Girard's linear logic [9].

The "use once" restriction can be enforced syntactically except for the case of unique fields of possibly shared objects because a shared object can be accessed multiple times. Furthermore, objects usually have indefinite lifetime and thus a single syntactic use of a field may translate into multiple uses at runtime. In order to handle this case, one can maintain the status of unique variables instead by requiring that they be read *destructively*, that is, after reading the value of a unique variable, the variable is *nullified* (made empty).[1] Emptiness is marked using a special "null" or "void" pointer. This latter restriction is the one used by Hogg and Minsky in their respective proposals [11, 14]. Baker

[*]Author's address: Department of Electrical Engineering and Computer Science, University of Wisconsin - Milwaukee, Milwaukee, WI 53201, boyland@cs.uwm.edu. Work supported in part by the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0241, and by an equipment grant from Sun Microsystems, Inc.

[1]Instead of a destructive read, an atomic swap can be used to solve the same problem in much the same way. A rather different approach is to "deep copy" the value.

also briefly mentions destructive reads when discussing how unshared objects can be stored in possibly shared objects.

Requiring reads of unique variables to be destructive is sufficient to prevent them from being *compromised*, that is aliased with another variable. But destructive reads of unique fields, by their very nature, modify the object containing the field. This fact has several unfortunate consequences. First, programs are more complex, as seen in the following section. Second, destructive reads may make a method or procedure appear to have more side-effects than it conceptually performs. Third, it would seem that adding unique variables to a programming language necessitates the further addition of destructive reads.

This paper proposes an alternate check for uniqueness, *alias killing*: when a unique field is read, any existing aliases are required to be dead. Together with restrictions on aliasing across procedure calls, alias killing can be checked by a static analysis in a modular fashion. If a program passes the check, then it has the same behavior with or without the uniqueness annotations. Thus we achieve the benefits of uniqueness annotations without having to extend the base language implementation.

In the following section, we give a taste of how destructive reads behave, and how one can relax the uniqueness invariant to apply only when it is needed. Section 3 describes an analysis that can determine that unique variables are used correctly. Related work is reviewed in Section 4. Section 5 concludes the paper.

2 Example

Figure 1 gives portions of a lexer class that reads characters from a file through the use of a buffered stream class. In particular, the constructor takes a file parameter and uses it to initialize a buffer object. We include one method to query the status of the lexer and another method to replace the file currently used with another one. This second method returns the former file with its file pointer set to exactly where the lexer last stopped; the `sync` method on the buffer accomplishes this requirement. This method could be used to implement an "include" command in the language being lexed; the client of the lexer (perhaps a parsing module) would recognize the include command, and then open the included file and pass it to the lexer. The returned file would be saved on a stack and when the included file concluded, the saved file would be substituted back, and lexing would continue directly after the include command.

```
public class Buffer {
  private File file;
  public Buffer(File f) { file = f; }
  ⋮
  public boolean atEOF() { ... }
  public void sync() { ... }
  public File getFile() {
    try { return file; }
    finally { file = null; }
  }
}

public class Lexer {
  private Buffer buf;
  public Lexer(File f) {
    buf = new Buffer(f);
  }
  ⋮
  public boolean isDone() {
    return buf.atEOF();
  }
  public File replace(File n) {
    buf.sync();
    File old = buf.getFile();
    Buffer nb = new Buffer(n);
    buf = nb;
    return old;
  }
}
```

Figure 1: A simple lexer class.

2.1 Destructive reads

Suppose we extend Java with "unique" variables and destructive reads in the style of Minsky's Eiffel* (for now, without the "nonconsumable" extension). For now, we shall say a *unique* variable is one whose value is null or else refers to an *unshared* object, one referred to by no other variables. This situation is called the *uniqueness invariant*. Parameters, receivers and return values may be declared `unique`.

A use of a unique variable is a *destructive read*: the variable is atomically set to null at the same time that the value is read. The destruction preserves the uniqueness invariant. Here we discuss how our example would be expressed using this extension.

If the file used for lexing is accessible from outside the lexer's buffer, then this outside code could move the file pointer (by performing a read or a seek) seriously confusing the buffer object. Thus it is desirable for the lexer class to require that callers ensure that the file passed to the constructor or to the `replace` method is unshared. In other words, the file should be a unique parameter. Similarly, the file parameter for the constructor for `Buffer` should also be unique.

The buffer for a lexer should also be unshared. It might seem that this fact is easily controlled since the lexer class creates the buffer object and does not return it to its client. In Java, however, a constructor can keep a reference to the object it is creating in a globally accessible location; not all constructors create unshared objects. Thus a constructor that *does* create an unshared object should be declared as such. Similarly, the return value for `replace` should be declared as unique.

If we declare `buf` as a unique field, however, all of its reads are destructive. In particular, even the use in `isDone` nullifies the field! So we need a new version of `atEOF` that returns the buffer as well as the boolean result. In particular, we need to extend Java with multiple return values as well. This new version of `atEOF` takes a unique receiver buffer and return a tuple of a unique buffer and a boolean. Both unshared and shared buffers will presumably require an `atEOF` method and since a shared buffer cannot be unique, we will need two methods. For the same reason, we will need two versions of `sync`. The unique versions of each method have an appended `U` in Figure 2. We only need one version of `getFile`, which will require the receiver to be unique; after the call is done, the buffer object is garbage and can be collected. Figure 2 shows how the example looks if we extend Java with unique variables, destructive reads and multiple return values. Receivers are declared unique by putting the keyword `unique` right before the body of the method.

Programming in this way with destructive reads is awkward, and so in Eiffel* (and with Hogg's Islands), the uniqueness invariant is relaxed to per-

```
public class Buffer {
  private unique File file;
  public unique Buffer(unique File f) {
    file = f;
  }
  ⋮
  public boolean atEOF() { ... }
  public (unique Buffer,boolean) atEOFU()
      unique { ... }

  public void sync() { ... }
  public unique Buffer syncU() unique
  { ... }

  public unique File getFile() unique {
    return file;
  }
}

public class Lexer {
  private unique Buffer buf;
  public unique Lexer(unique File f) {
    buf = new Buffer(f);
  }
  ⋮
  public boolean isDone() {
    boolean done;
    (buf,done) = buf.atEOFU();
    return done;
  }

  public unique File replace(unique File n)
  { buf = buf.syncU();
    unique File old = buf.getFile();
    unique Buffer nb = new Buffer(n);
    buf = nb;
    return old;
  }
}
```

Figure 2: A lexer class using uniqueness extensions. (Destructive reads are underlined for emphasis.)

mit so-called "dynamic aliases." An *alias* of a variable for our purposes is a variable that refers to the same object as the first variable. This definition is nonstandard, but is convenient for languages such as Java that do not have explicit dereferencing. A *static alias* is a field variable; a *dynamic alias* is a receiver, parameter or local variable alias.[2] A dynamic alias of an otherwise unique variable is also known as a "nonconsumable," "uncaptured," or "limited" variable. Such a variable may be used for computation and copied into other local variables or parameters (which become dynamic aliases) but must never be stored in a field. In this paper, we refer to variables whose values may never be stored (or returned) as *borrowed* variables. Intuitively, a borrowed variable is one whose value may be "owned" elsewhere (referenced by a unique field). Dynamic aliases are easier to reason about than unrestricted aliases. Consider the situation where a reference to an unshared object is passed as a borrowed receiver or parameter: when the method returns (assuming the return value is not the same object), we know that all the dynamic aliases have died and thus the object is unshared again. Most methods in fact do not store their receivers in objects and many parameters are likewise used only temporarily. These receivers and parameters can be declared borrowed. With this extension, we no longer need two versions of **`atEOF`** and **`sync`**; instead we have one each with a borrowed receiver.

Weakening our uniqueness variant to admit dynamic aliases makes it easier to program with unique variables than with destructive reads alone, but this weakening comes with a cost. One main purpose in avoiding aliasing was to make it easier to track mutations; dynamic aliases make it harder to track down all mutations. Another purpose in avoiding aliasing is to make it easier to reason about the effects of a computation. A dynamic alias may make an unwelcome appearance as a parameter to a method of an object with the otherwise unique field, leading to unexpected data dependencies between writes through the unique field and reads through the parameter. Thus by weakening our unique invariant, we have made the invariant less useful for analysis and programmers alike.

[2] Almeida's "balloons," which use copy-on-assignment rather than destructive reads, permit return values to be dynamic aliases as well.

2.2 Our contribution

Our alternative is conceptually simple: we keep the strong invariant but only require it to be true when it is needed; we express uniqueness with nonexecutable annotations layered on top of an unchanged language; we use annotations (such as "borrowed") in method interfaces to carry out the analysis intraprocedurally, that is, in the context of separate compilation. We call the basic idea behind the analysis "alias killing," which will be explained in the following section. In this section, we show how one can understand our approach as a variant of the one using destructive reads. We start with Figure 2 which uses destructive reads to maintain a strong uniqueness invariant.

Extending a programming language has a number of disadvantages. Among them are that it is not possible to apply standard tools to code using the extensions, and that extended code cannot be reused by people who do not accept the extensions. A better approach is to use supra-lingual annotations to express the desired properties and to check these annotations separate from the implementation. This approach can handle destructive reads by expressing them using regular reads and assignment in certain fixed patterns acceptable to the analyzer.

Suppose one consistently expresses destructive reads with calls to a (polymorphic) method **`prog1`** that simply returns its first argument. For instance, one can replace code from the **`replace`** method (*old code* ⟶ *new code*):

```
buf = buf.syncU()                       ⟶
buf = prog1(buf,buf=null).syncU();

return old;                             ⟶
return prog1(old,old=null);
```

(Ignore the ugliness of this syntax for now; it simply demonstrates that a destructive read can be expressed using normal reads and writes.) The interesting point is that (assuming we consider actual parameters to be aliases), the unique variables have fleeting dynamic aliases. The aliasing disappears as soon as the field is nullified and is completely harmless, assuming locking or some other condition ensures that the variable isn't being accessed by another thread. The uniqueness invariant is thus not actually true at every point in the program, but the points when it is false are "uninteresting."

This situation is very common in the use of object invariants. Typically an invariant will be violated by some actions and then recovered soon thereafter. Our approach to uniqueness is in the same spirit: the invariant for a unique field must be true whenever it would be useful (when the field is read) but otherwise can be safely violated. This reasoning allows us to expand the region where the invariant is not true. For instance, the nullification of `old` in `replace` can be removed:

```
return prog1(old,old=null);        ⟶
return old;
```

More subtly, if we are assured that the action of `syncU` does not invoke any methods on the lexer, and thus the field `buf` will not be read, then the nullification of `buf` can be deferred and then seen as dead:

```
buf = prog1(buf,buf=null).syncU(); ⟶
buf = prog1(buf.syncU(),buf=null); ⟶
buf = buf.syncU();
```

In fact, under reasonable assumptions, we can defer almost all nullifications one would use to emulate destructive reads to the point that they become dead and can be removed.[3] We have thus demonstrated that it is possible to support a strong uniqueness invariant (at the points it is needed) without atomic destructive reads. Next we show how we can extend this idea to encompass "borrowing" reads without weakening the invariant.

Many of the methods taking a unique receiver such as `atEOFU` and `syncU` simply return the receiver back to the caller unchanged. This extra return value forces us to handle multiple return values, and it also means the unique field is constantly being rewritten whenever it is used. Knowing this, we can then re-implement these methods so as not to return the receiver, and re-implement their callers to reuse the receiver. These methods are simply restricted not to store the receiver anywhere and not to return it. These rules ensure that the method *could* be implemented to take a unique receiver and return an extra unique result identical to the receiver. With these changes, `atEOFU` ends up being identical to `atEOF` and can be folded into it. Similarly, `sync` and `syncU`. The restrictions are identical to those for borrowed receivers (and parameters) that we saw earlier. What we have here is an alternative explanation for the restrictions.

Figure 3 shows the result of using our approach. Instead of extending the language with unique types and destructive reads, we use non-executable annotations (shown *slanted*) and extra static checks. The underlying program is identical to our original program in Figure 1. There is another difference between Figure 2 and Figure 3: constructors in Java are not responsible for creating an object, just initializing it. Therefore constructors that "return" unique objects are annotated as having borrowed receivers.

```
public class Buffer {
  private unique File file;
  public Buffer(unique File f) borrowed {
    file = f;
  }
  ⋮
  public boolean atEOF() borrowed { ... }
  public void sync() borrowed { ... }
  public unique File getFile() unique {
    try { return file; }
    finally { file = null; }
  }
}

public class Lexer {
  private unique Buffer buf;
  public Lexer(unique File f) borrowed {
    buf = new Buffer(f);
  }
  ⋮
  public boolean isDone() borrowed {
    return buf.atEOF();
  }
  public unique File replace(unique File n)
      borrowed
  { buf.sync();
    File old =  buf.getFile();
    Buffer nb = new Buffer(n);
    buf = nb;
    return old;
  }
}
```

Figure 3: A lexer class using uniqueness annotations.

[3] We do not remove the nullification of `file` in `getFile` because it may be accessed during finalization.

```
public class Vector {
  private int size;
  private unique Object[] contents;

  public Vector() borrowed {
    size = 0;
    contents = new Object[10];
  }

  public int size() borrowed {
    return size;
  }

  public synchronized Object elementAt(int i)
      borrowed
  {
    return contents[i]; // error check omitted
  }

  public synchronized
  void ensureCapacity(int min)
      borrowed
  {
    if (min <= contents.length) return;
    if (min <= contents.length*2)
      min = contents.length*2;
    Object[] newContents = new Object[min];
    for (int i=0; i < size; ++i) {
      newContents[i] = contents[i];
    }
    // "contents" can be reclaimed here
    contents = newContents;
  }

  public synchronized
  void addElement(Object elem)
      borrowed
  {
    ensureCapacity(size+1);
    contents[size] = elem;
    ++size;
  }
  ⋮
}
```

Figure 4: Vectors implemented with unique arrays.

Figure 4 shows another example of uniqueness annotations in an implementation of `Vector` using a unique array. The uniqueness invariant ensures that accesses to different vectors can be carried out without interference. It also permits the vector to replace the array when it is not large enough without affecting any other code. The code also shows how uniqueness could be used by to perform "compile-time garbage collection."

Arrays containing unique references as seen in our third example (Figure 5) are also useful. As opposed to the vector example, here the array is not only referenced through a unique field but its contents are unique as well. As a result, bucket accesses are guaranteed to be non-interfering with bucket accesses of any other hashtable. The flexibility of uniqueness can seen in that the unique bucket references can be reused in a new array when the hash table is resized.

In summary, a strong uniqueness invariant can be maintained where it is useful through the addition of nonexecutable checkable annotations. The annotations are needed only on fields and in method interfaces: in particular we don't need to distinguish destructive reads from borrowing reads and do not need to annotate local variables.

3 Details

In this section, we first describe a simple imperative language. Then we add a simple rule that handles uniqueness with alias killing. As long as the execution of the extended language does not go "wrong," unique fields are used correctly. We use interface annotations in order to be able to analyze a single procedure at a time. This section describes the rules informally. A precise description is given in the appendices. Finally, we describe how we apply these ideas to the Java programming language.

3.1 Base language

Our base language (see Figure 6) is a simple imperative language with objects (updateable and extensible records) but without dynamic dispatching. We ignore typing issues; any object can have any field. Procedures declare the names of parameters and also a special name for the return value. New locals may be declared in any block. New objects

```
class Bucket {
  Object key, value;
  unique Bucket next;
  Bucket(Object k, Object v, unique Bucket n) borrowed {
     key=k; value=v; next=n;
  }
}

public class Hashtable {
  private unique Bucket unique[] buckets;
  private int size = 0;
  public Hashtable() borrowed { buckets = new Buckets[10]; }
  public synchronized Object get(borrowed Object k) borrowed {
    for (Bucket b = buckets[k.hashCode()%buckets.length]; b != null; b=b.next) {
      if (b.key.equals(k)) return b.value;
    }
    return null;
  }
  public synchronized Object put(Object k, Object v) borrowed {
    int h = k.hashCode() % buckets.length;
    for (Bucket b = buckets[h]; b != null; b=b.next) {
      if (b.key.equals(k)) {
        Object old = b.value;
        b.value = v;
        return old;
      }
    }
    buckets[h] = new Bucket(k,v,buckets[h]);
    if (++size > buckets.length) rehash();
    return null;
  }
  protected void rehash() borrowed {
    Bucket unique[] newBuckets = new Bucket[buckets.length*2];
    for (int i=0; i < buckets.length; ++i) {
      while (buckets[i] != null) {
        Bucket b = buckets[i];
        buckets[i] = b.next;
        int h = b.key.hashCode() % newBuckets.length;
        b.next = newBuckets[h];
        newBuckets[h] = b;
      }
    }
    // "buckets" can be reclaimed here
    buckets = newBuckets;
  }
  ⋮
}
```

Figure 5: Hash tables implemented with unique arrays of unique references.

$$
\begin{array}{rcl}
\mathit{program} & \longrightarrow & \mathit{decl}^* \\[1em]
\mathit{decl} & \longrightarrow & \texttt{field}\ f \\
& | & \texttt{proc}\ p(l_1, \ldots, l_n)\ l_0\ \mathit{block}\ \texttt{end} \\[1em]
\mathit{block} & \longrightarrow & \mathit{local}^*\ \mathit{stmt}^* \\[1em]
\mathit{local} & \longrightarrow & \texttt{local}\ l \\[1em]
\mathit{stmt} & \longrightarrow & \texttt{if}\ \mathit{test}\ \texttt{then}\ \mathit{block}\ \texttt{else}\ \mathit{block}\ \texttt{end} \\
& | & \texttt{while}\ \mathit{test}\ \texttt{do}\ \mathit{block}\ \texttt{end} \\
& | & \mathit{lhs}\ \texttt{=}\ \mathit{expr} \\[1em]
\mathit{test} & \longrightarrow & \mathit{expr}_0\ \texttt{==}\ \mathit{expr}_1 \\[1em]
\mathit{expr} & \longrightarrow & \mathit{lhs} \\
& | & \texttt{new} \\
& | & \texttt{null} \\
& | & p(\mathit{expr}_1, \ldots, \mathit{expr}_n) \\[1em]
\mathit{lhs} & \longrightarrow & l \\
& | & \mathit{expr}\texttt{.}f \\[1em]
l, p, f & \in & \mathit{Id}\quad \mathit{(identifiers)}
\end{array}
$$

Figure 6: Syntax of Base Language.

```
proc replace(lex,n) old
  local ignore
  ignore = sync(lex.buf)
  old = getFile(lex.buf)
  lex.buf = newBuffer(n)
end
```

Figure 7: Sample Procedure.

are created with the special expression `new`. Additionally, we have a constant value `null`; it is an error to assign or select a field from this value. Figure 7 shows how the Java method `replace` from our example is rendered.

A *variable* is a field of an object, or an instance of a local, parameter, or return value. When a variable is created, it is first *undefined*: a variable is created for each declared field when a `new` expression is evaluated; a variable is created for a parameter or return value when a procedure call starts to be evaluated; a variable is created for a local when its block starts to be evaluated. The formal parameters are initially defined using the values of the corresponding actual parameters. A variable can be defined by assigning a value to it. It is an error to read an undefined variable. The return variable must be defined at the end of a procedure.

A defined variable is either `null` or refers to an object. If it not null, its *aliases* are all the other defined variables that refer to the same object.

3.2 Alias Killing

We extend the base language by adding an optional annotation on fields: a field may be declared as *unique*. We add one extra rule to the semantics of the language that ensures that unique fields are indeed unique. The single useful feature of unique variables in general is that when one reads the value of the variable, one knows that if the value is not `null`, the object referenced is not accessible through any other variable. This intuition is captured by the following rule (alias killing):

> When a unique field of an object is read,
> all aliases of the field are made undefined.

This rule ensures that if a program completes successfully (that is, without an error), then all unique fields were indeed unique. It does not forbid the field from having any other aliases, it simply requires that they all be *dead*, that is, assigned before being used again, if ever.

This rule does *not* involve a destructive read. In particular, it does not use a legal value (null) to avoid aliasing, instead aliases are made *undefined*. If the unique field is later read, this is legal only if the previous read was a "borrowing" one; all previously obtained aliases must be dead. If the unique field is assigned before next being read, then one

$$
\begin{array}{rc}
\top = & \text{undefined} \\
 & \sqcup \\
\flat = & \begin{array}{c}\text{borrowed}\\ \text{shared}\end{array} \\
 & \sqcup \\
\sharp = & \begin{array}{c}\text{owned}\\ \text{shared}\end{array} \\
 & \sqcup \\
\bot = & \begin{array}{c}\text{owned}\\ \text{unique}\end{array}
\end{array}
$$

Figure 8: Lattice V for annotations and variable state.

may consider the read to have been destructive; the destruction was simply deferred.

Most importantly, if the execution of a program using the extended semantics completes without error, it has precisely the same evaluation as under the base semantics. In other words, our extension does not require a different execution environment for well-behaved programs. Therefore, if we can determine statically that a program will not read aliases killed by our rule, no dynamic checks will be necessary.

As with almost all static analysis problems, the most precise answer is incomputable (reduces to the halting problem) and so we use a conservative approximation. If the program passes the approximation then it definitely will not read dead aliases. We further require an analysis to be modular: it must be possible to check a program by individually testing the separately compiled modules. In the rest of this section, we describe an analysis that meets these requirements.

3.3 Annotating procedure interfaces

The simple "alias killing" rule suffers from having global effect: a read of a unique field can potentially make a great number of variables undefined, many not even in scope. Here we describe how the analysis uses checked annotations on procedures and preemptive alias killing so that it can work correctly knowing only about local aliases.

We add annotations on procedure parameters and return values: a parameter or return value may be declared unique; a *shared* (that is, not unique) parameter may be declared borrowed. Return values may never be borrowed. Figure 8 shows the lattice of values used in annotations. The annotation on a parameter must be from the set $\{\bot, \sharp, \flat\}$; the annotation on a return value must be one of $\{\bot, \sharp\}$. The object referred to by a *borrowed* parameter may not be returned from a procedure, assigned to a field, or passed as an *owned* (that is, not borrowed) actual parameter. The same lattice provides values for field annotations. The annotation v_f on a field f must be from the set $\{\bot, \sharp\}$. We do not put annotations on local variables. The analysis also uses (but here does not check) an annotation on procedures that indicates which variables it may read.[4]

The interface annotations may be read as obligations and dual privileges:

1. The caller of a procedure must ensure that parameters declared unique are indeed unaliased. A procedure may assume unique parameters are unaliased.

2. A procedure must ensure that a borrowed parameter is not further aliased when it returns. A caller may assume that a borrowed actual parameter will not be further aliased when the procedure returns.

3. A procedure must ensure that a return value declared unique is indeed unaliased. A caller may assume that a unique return value is unaliased.

4. The caller of a procedure must ensure that any unique field read or written by the procedure (or by a procedure it calls, recursively) is unaliased at the point of call. A procedure may assume that the first time it reads or writes a unique field, the field has no aliases.

5. A procedure must ensure that any unique field it reads or writes during its execution is unaliased upon procedure exit, and that no fields have been made undefined. A caller may assume that a procedure does not create any

[4] A listing of fields possibly accessed during the execution of the procedure would serve well and would also be easy to check modularly. We would prefer however to use an approach which permitted better information hiding, such as obtained using our object-oriented effects system [10].

aliases of unique fields or make fields undefined.

If all procedures fulfill these responsibilities, then if a non-dead variable is made undefined by a unique field read in a procedure, then the unique field must have been compromised by this procedure. Thus assuming we can check a procedure, we can check a program.

3.4 Checking a procedure

In this section, we summarize an analysis for checking uniqueness annotation on a per-procedure basis. (The full description is given in the appendices.) The analysis uses an abstract store semantics modeled after Sagiv, Reps, and Wilhelm's shape analysis [18]. The analysis keeps track of aliases of originally unshared objects. It must also determine when a reference to the object has been *irretrievably compromised*, that is stored in a field, or passed as an owned parameter. The analysis also keeps track of objects whose unique fields are undefined (by virtue of being irretrievably compromised), and may not be read. These fields must be be restored before the procedure returns or calls another procedure that may read the field.

A unique field may only be written with null or an object with a known set of aliases. The aliases are all immediately assumed undefined after the unique field is written. This action anticipates them being made undefined when the field is read. Similar rules are used for unique parameters and return values.

3.5 Alias killing in Java

Thus far in this section, we have discussed how to check annotations in the context of a toy language. Here we sketch the application of our approach to Java:

- Receivers.

  The receiver `this` of a method is considered an extra parameter with its own annotation.

- Static methods.

  A static method does not have a receiver and thus is viewed as a simple procedure.

- Overriding.

  A parameter in the overriding method must not be given an annotation lower in the lattice than the parameter in the overridden method, and the return value must not be given an annotation higher in the lattice than the return value in the overridden method. (This rule is an application of the standard function subtyping rule.)

- Constructors.

  A constructor is treated as a method without a return value whose receiver is a newly created, uninitialized object. If the receiver is annotated borrowed, the caller may be sure that the newly created object remains unaliased after the constructor has returned. The first action of a constructor (other than the one in `Object`) is to call the constructor in the superclass.

- Static (global) variables.

  A global variable is considered a special field in a shared global state object passed to every procedure. Global variables may be declared unique.

- Return statements.

  The `return` statement is considered a combination of assignment to the return value followed by a jump out of the body of the method. The extra complexities of control-flow (for instance the numerous places where an exception may be raised or propagated) can be handled as with any other analysis; one must be careful to handle the `finally` clause correctly.[5]

- Throw statements.

  A `throw` statement is treated as an assignment to a special (per thread) global variable, followed by the effect of `return null;` for the purposes of control-flow.

- Arrays.

  Array references are treated as shared field references. Arrays of unshared objects are also

[5] A finally clause can, for instance, prevent a `return` statement from taking effect. Hence the setting of the return value must be considered separately from the control-flow aspects.

useful (as seen in the hash table in Figure 5) but require special treatment because the elements are anonymous.

- Reflection.

  Java's reflection capabilities can be used to subvert many checks, including the ones outlined here.

We have implemented the analysis for Java as part of the ACT/Fluid project at Carnegie Mellon University and University of Wisconsin–Milwaukee. Our prototype does not currently handle arrays of unshared objects; it can check the vector in Figure 4 but not the hash table in Figure 5.

The analysis described here is for single-threaded code, but unique variables are particularly useful in multi-threaded applications, because synchronization is not needed when accessing fields of unique objects.[6] Of course, accessing unique fields of possibly shared objects must still be properly synchronized in order to prevent race conditions. When the lock is released, the unique field must be uncompromised and have no live aliases. If a language has asynchronous calls (Java does not), then we must add the rule that borrowed values cannot be passed in such calls.

4 Related Work

Reynolds [17] discusses a technique for determining when statements would never *interfere*, that is, could be meaningfully executed in parallel. His paper included a simple object system but had no concept corresponding to a unique pointer.

In the denotational semantics of an imperative language, the store is reified as a value, but the uses of the store are such that it can be modified in place, that is, it is *single-threaded*. Schmidt [19] details rules that ensure a store is single-threaded. The rules permit a store to be used in parallel in two different contexts if neither updates the store nor returns the store. This condition allows single-threading to be less onerous, and roughly corresponds to what we call a borrowed parameter.

Wadler [20] gives a type system based on Girard's linear logic [9] for typing programs using both linear and nonlinear types. The type system does not permit linear values to be embedded in shared values, because, as mentioned earlier, then it becomes difficult to determine that the value is used exactly once. Following Schmidt, the type system *does* permit shared read-only access to the linear value as long as the value computed in this scope does not include any references to the linear values being accessed.[7]

Chen and Hudak [5] show how a linear abstract datatype (ADT) can be encapsulated in a *monad* thus restricting the use of linear types to the module implementing the linear ADT. Monads enable *imperative functional programming* [16] by ensuring effects on uncopyable resources such as file systems are performed in order. References to mutable state encapsulated in a monad can be treated as normal values [12], but then of course these references can be duplicated, thus making it difficult to ensure linearity. Syntactic sugar enables monadic programming to look almost like an imperative program. The rules on the uses of the monad are enforced by the type system, and thus a function must be written to work with either a monadic value or a regular value, not both. Also, unlike Wadler's type system, there is no concept corresponding to "borrowed" parameters.

Concurrent Clean, a lazy functional language, uses "unique types" to represent the current state of the "world" [1]. As in this work (and contrasting with linear logic), a "unique" parameter is guaranteed by the caller to be unique but the callee may make aliases if desired, without the need for a special sharing primitive. The type system checks that actual parameters and results are unique if so declared. Apparently, Concurrent Clean has no facility for declaring "borrowed" parameters, but Wadler's check could be adapted for this purpose.

A number of proposals to add some form of uniqueness to object-oriented languages are similar to this work. Table 1 compares the terminology used in each case. With Hogg's Islands [11], a class with an interface that satisfies certain simple properties can be ascertained to be a "bridge" class whose instances allow exclusive access to an "island" of objects. All the objects accessible through a bridge are protected in that they are not accessi-

[6] Uniqueness is a sufficient, but not necessary condition; an object need only be thread-local.

[7] This condition is analogous to the condition that "borrowed" parameters cannot be returned. The read-only condition stems from the fact that the language does not permit side-effects.

Variable	This work	Islands	Eiffel*	Balloons	ESC	FAP
param/result	unique	free	u-variable	*none* [a]	virgin	free
field	unique	unique	u-attribute	balloon	pivot	rep [b]
field	shared	(normal)	(normal)	non-balloon	plenary	arg or var
parameter	borrowed	unique [c]	non-consumable [d]	balloon [e]	not captured	*none* [f]

[a] There is no mechanism for creating a free balloon. Storing a balloon in a field always (conceptually) copies it.
[b] May be privately duplicated.
[c] The actual parameter must be null or an alias of a unique field.
[d] Default. Only u-pointers permitted as actual parameters.
[e] A reference to a balloon can be returned as well. Undesirable dynamic aliasing may result.
[f] The type system can ensure the representation is not exposed, but it does not ensure it remains unique.

Table 1: Approximate comparison of uniqueness annotations.

ble through other objects outside the island. Hogg adds checked annotations and destructive reads to Smalltalk programs. As summarized in the table, Hogg's `unique` parameters correspond with our "borrowed" parameters but must be (aliases to) unique fields. Unfortunately, a method with a unique parameter cannot make use of the (partial) uniqueness since nothing prevents another alias of the field being created by reading the field again. In other words, in contrast with our work, the user of a borrowed reference may invoke code that uses the original reference (a breakdown of uniqueness). Finally, the island property is too strict for many purposes; usually it is desirable for some objects to be shared by the outside.

Minsky proposed Eiffel* [14], Eiffel with uniqueness extensions. Any variable (attribute, parameter or return value) can be designated a "u-pointer." Reading any such variable into another variable effects a destructive read. Certain parameters may be designated "nonconsumable u-pointers" which cannot be used except in certain nondestructive contexts. Such parameters correspond to our borrowed parameters, except that (apparently) only (consumable or nonconsumable) u-pointers can be used as actual parameters. Minsky's system does not check that a nonconsumable reference cannot be used by code that uses the original reference, nor does it permit consumable receivers such as in `buf.getFile()` in our example. Minsky's proposal also does not preserve uniqueness in cases such as `u.m(u)` [14, footnote 4].

Almeida's Balloons [2] were designed to provide strong encapsulation. Similar to a Hogg island, an instance of a balloon type has the only (stored) reference to any object reachable from it. This restriction is enforced in two different ways. First, references to internal non-balloon objects cannot be passed out of the scope of the balloon type. Secondly, no reference to a balloon object may be stored in an object. Balloons have a linear flavor; assigning a balloon field requires copying the balloon first. As a result, parameters (and return values) of balloon type are always borrowed. Almeida points out that the "copy" can often be delayed or omitted altogether by the runtime system if the balloon's contents are not changed by side-effects. Regular balloons can expose their internal balloons; which is taken as a feature in one example (a dictionary that functions somewhat like a library). Opaque balloons never expose their internals. Since parameters and return values are treated similarly, balloons lose the ability to control dynamic aliases: regular balloons are promiscuous and opaque balloons cannot have *any* dynamic aliases. The balloon property, like the island property, is too strict for many purposes.

The concept of "rep exposure," the exposure of an abstract datatype's internal representation was first introduced in CLU [13]. LCLint [8] provided a way to prevent references to internal state from escaping an abstraction, at least for mutation, but this rule proved too strict and led to numerous spurious warnings [7, page 6]. Detlefs, Leino and Nelson [7] use a theorem prover to diagnose rep exposure as part of their "extended static checking" (ESC) system. A "pivot" field is one whose validity is required for the containing object to be valid; roughly, all state transitions on an object stored in a pivot field must be carried out through the field itself. Other fields are known as "plenary" fields. The term "virgin" is used for references that have not yet been stored ("captured") in any field. It does not quite correspond to "free" or "unique" because even if a virgin reference is passed as an actual parameter used to initialize a pivot field, the caller may keep using the reference, thus exposing the representation. It appears this flaw could be fixed by forbidding the caller from using the reference after the call.

Noble, Vitek and Potter [15] overcome the limitations of Islands and Balloons in their solution to the problem of rep exposure. Each "alias-protected" container object divides up its internal objects into two classes: representation objects, and argument objects. The former are kept private and may be mutated, while the latter may be shared, but not mutated. *Flexible alias protection* (FAP), as their system is called, makes use of "free" (that is, unique) parameters and return values. Representation objects need not be unshared; they may be used in more than one place, but representation fields are often initialized with unique references. FAP also uses "roles" to partition argument types in a way that prevents a called method from exposing a representation object passed to it as an argument object. Roles provide much of the protection of borrowed parameters except that they do not prevent an unshared object from being (locally) shared.

Clarke, Potter and Noble [6] formalize the idea of "representation" objects as "owned" objects. These objects have a single owner for their entire lifetime: they do not need to be, nor can they be initialized with "free" objects, nor can ownership be transferred. Ownership is thus more restrictive in some ways than uniqueness, while remaining less restrictive in other ways (limited aliasing is permitted). For instance, ownership is not sufficient to handle our lexer example. On the other hand, uniqueness cannot be used to limit aliasing in doubly-linked lists.

In earlier work [4], we motivate the use of unique variables ("unique" parameters and "unshared" fields) for the purposes of verifying semantics-preserving program manipulation. In particular, aliasing can be computed much more precisely when more variables are known to be unique. The paper also defines a parameter mode called "limited unique." The actual parameters can be assumed distinct from any value accessible to the called procedure, but are borrowed, that is, cannot be stored or returned. The technique described in the current paper can be extended to handle this mode through additional checks.

5 Conclusion

In contrast to Eiffel* and Islands which use destructive reads, we add uniqueness to a base language only through the use of annotations and extra static checks. The benefit is that the annotations can be stripped out and ignored whenever the program is implemented and executed. As a result, existing compilers and run-time systems can be used unchanged. The major disadvantage is that a complex (albeit intraprocedural) analysis must be used to ensure that uniqueness is preserved. Apparently minor source-level changes may lead to code no longer satisfying these checks., with potentially confusing error messages. However, since we do not change the semantics of the base language, a new more sophisticated analysis that ensures the basic procedure-level semantics of Section 3.3 could be substituted for the one detailed in Section 3.4. For instance, a theorem prover could be used. The contribution of this work is not necessarily in the actual analysis, but instead the way in which checking for uniqueness (alias killing) can be performed in a modular fashion.

Using static analysis avoids another disadvantage of destructive reads: implicit nullification. Instead, with alias killing, if the field is never explicitly set to null, then it will never be null when read. Thus not only does the analysis ensure variables declared unique are indeed unique, it also ensures that a unique variable is never read at "unexpected" times.

Acknowledgments

This paper describes work carried out by the author in collaboration with Edwin Chan, Aaron Greenhouse and William Scherlis at CMU. The idea to use a lexer for an example came from Greg Nelson and K. Rustan M. Leino. Jan Vitek, Rowan Davies, Phil Wadler and numerous anonymous referees all provided useful comments on drafts of the paper.

References

[1] Peter Achten, John van Groningen, and Rinus Plasmeijer. High level specification of I/O in functional languages. In John Launchbury and P. Samson, editors, *Workshop on Functional Programming, Glasgow 1992*, Ayr, UK, July 6–8, Workshops in Computer Science, pages 1–17. Springer, Berlin, 1993.

[2] Paulo Sergio Almeida. Balloon types: Controlling sharing of state in data types. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming, 11th European Conference*, Jyväskylä, Finland, June 9–13, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer, Berlin, 1997.

[3] Henry G. Baker. 'Use-once' variables and linear objects—storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, January 1995.

[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, 1998.

[5] Chih-Peng Chen and Paul Hudak. Rolling your own mutable ADT: A connection between linear types and monads. In *Conference Record of the Twenty-fourth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, Paris, France, January 15–17, pages 54–66. ACM Press, New York, 1997.

[6] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA'98—Object-Oriented Programming Systems, Languages and Applications*, Vancouver, Canada, October 18–22, *ACM SIGPLAN Notices*, 33(10):48–64, October 1998.

[7] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Technical Report KRML 68, Digital Equipment Corporation, Systems Research Center, July 1996.

[8] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, New Orleans, Louisiana, USA, December 6–9, *ACM SIGSOFT Software Engineering Notes*, 19(5):87–96, December 1994.

[9] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.

[10] Aaron Greenhouse and John Boyland. An object-oriented effects system. In *ECOOP'99 — Object-Oriented Programming, 13th European Conference*. Springer, Berlin, To appear.

[11] John Hogg. Islands: Aliasing protection in object-oriented languages. In *OOPSLA'91—Object-Oriented Programming Systems, Languages and Applications*, Phoenix, Arizona, USA, October 6–11, *ACM SIGPLAN Notices*, 26(11):271–285, November 1991.

[12] John Launchbury and Simon L. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, 8:293–341, 1995.

[13] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. The MIT Press, Cambridge, Massachussetts, USA and London, England, 1986.

[14] Naftaly Minsky. Towards alias-free pointers. In Pierre Cointe, editor, *ECOOP'96 — Object-Oriented Programming, 10th European Conference*, Linz, Austria, July 8–12, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209. Springer, Berlin, July 1996.

[15] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP'98 — Object-Oriented Programming, 12th European Conference*, Brussels, Belgium, July 20–24, volume 1445 of *Lecture Notes in Computer Science*. Springer, Berlin, 1998.

[16] Simon L. Peyton Jones and Philip Wadler. Imperative functional programming. In *Conference Record of the Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, Charleston, South Carolina, USA, January 10–13, pages 71–84. ACM Press, New York, 1993.

[17] 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.

[18] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[19] David A. Schmidt. Detecting global variables in denotational semantics. *ACM Transactions on Programming Languages and Systems*, 7(2):299–310, April 1985.

[20] Philip Wadler. Linear types can change the world! In *Programming Concepts and Methods*. Elsevier, North-Holland, 1990.

$$
\begin{array}{rcl}
\mathcal{F}, \mathcal{F}' & : & Id \to S \to S \\
\mathcal{B} & : & block \to S \to S \\
\mathcal{C} & : & stmt \to S \to S \\
\mathcal{T} & : & test \to S \to (S \times S) \\
\mathcal{E} & : & expr \to S \to S
\end{array}
$$

Figure 9: Evaluation function signatures.

$$
\begin{array}{rcl}
\mathsf{start} & : & S \\
\mathsf{save}, \mathsf{restore} & : & \mathcal{P}(L) \to S \to S \\
\mathsf{set}, \mathsf{get} & : & L \to S \to S \\
\mathsf{store}, \mathsf{load} & : & F \to S \to S \\
\mathsf{null}, \mathsf{new}^v, \mathsf{release}^v & : & S \to S \\
\mathsf{equal} & : & S \to (S \times S) \\
\sqcup & : & (S \times S) \to S
\end{array}
$$

(new^v for $v \neq \perp$ and $\mathsf{release}^v$ are only used in abstract semantics.)

Figure 10: Signatures of state primitives.

A Standard Semantics

A.1 Base Semantics

The base semantics is defined in two layers: a set of evaluation functions (whose signatures are given in Figure 9) that use primitives of a stack-based memory system (whose signatures are given in Figure 10). There are two evaluation functions for evaluating procedure calls, $\mathcal{F}$ and $\mathcal{F}'$; the latter is left to be defined with the store primitives because in the abstract semantics, we don't attempt to evaluate the whole program. In the standard semantics, however, the second is identified with the first. The definitions of the evaluation functions are given in Figure 11.

Most of the store primitives simply convert an input state into an output state: save and restore bracket the scope of local variables; set and get access local variables; store and load access fields; null and new push values onto the stack; release pops a value off the stack. The primitives that are not simple state transformers are start which is the starting state, equal which returns two states, one assuming

$\mathcal{F}\, p\, s = \mathsf{restore}\; L'\; s''$

where p declared $\texttt{proc}\ p(l_1, \ldots, l_n)\ l_0\ block\ \texttt{end}$

$$
\begin{array}{rcl}
s'' & = & (\mathsf{get}\, l_0)\,(\mathcal{B}\, block)\, s' \\
s' & = & (\mathsf{set}\, l_1) \ldots (\mathsf{set}\, l_n)\,(\mathsf{save}\, L')\, s \\
L' & = & \{l_i \mid 0 \leq i \leq n\}
\end{array}
$$

$\mathcal{B}[\![decl^*\ \ stmt^*]\!]\, s = \mathsf{restore}\; L'\; s''$

$$
\begin{array}{rcl}
\text{where } s'' & = & (\mathcal{C} stmt_n) \ldots (\mathcal{C} stmt_1)\, s' \\
s' & = & \mathsf{save}\; L'\; s \\
L' & = & \{l_i \mid \texttt{local}\ l_i = decl_i\}
\end{array}
$$

$\mathcal{C}[\![\texttt{if}\ test\ \texttt{then}\ block_1\ \texttt{else}\ block_2]\!]\, s = s_1' \sqcup s_2'$

$$
\begin{array}{rcl}
\text{where } s_i' & = & \mathcal{B}\, block_i\; s_i \\
(s_1, s_2) & = & \mathcal{T}\, test\; s
\end{array}
$$

$\mathcal{C}[\![\texttt{while}\ test\ \texttt{do}\ block\ \texttt{end}]\!]\, s = s''$

$$
\begin{array}{rcl}
\text{where } (s', s'') & = & \mathrm{fix}\, g\, (s, \perp) \\
g\, (s, t) & = & (\mathcal{B}\, block\; s_t,\, s_f \sqcup t) \\
& & \text{where } (s_t, s_f) = \mathcal{T}\ test\ s
\end{array}
$$

$\mathcal{C}[\![l\ \texttt{=}\ expr]\!]\, s = \mathsf{set}\, l\, (\mathcal{E}\ expr\ s)$

$\mathcal{C}[\![expr_0\,.\,f\ \texttt{=}\ expr_1]\!]\, s = \mathsf{store}\, f\; s''$

$$
\begin{array}{rcl}
\text{where } s'' & = & \mathcal{E}\ expr_1\ s' \\
s' & = & \mathcal{E}\ expr_0\ s
\end{array}
$$

$\mathcal{T}[\![expr_1\ \texttt{==}\ expr_2]\!]\, s = \mathsf{equal}\, s''$

$$
\begin{array}{rcl}
\text{where } s'' & = & \mathcal{E}\ expr_2\ s' \\
s' & = & \mathcal{E}\ expr_1\ s
\end{array}
$$

$\mathcal{E}[\![l]\!]\, s = \mathsf{get}\, l\, s$

$\mathcal{E}[\![expr\,.\,f]\!]\, s = \mathsf{load}\, f\, (\mathcal{E}\ expr)\ s$

$\mathcal{E}[\![\texttt{new}]\!]\, s = \mathsf{new}^{\perp}\ s$

$\mathcal{E}[\![\texttt{null}]\!]\, s = \mathsf{null}\, s$

$\mathcal{E}\,[\![p\,(expr_1\,,\, \ldots\,,\, expr_n)]\!]\, s = \mathcal{F}'\, p\, s'$

$$
\begin{array}{rcl}
\text{where } p & \text{has} & n \text{ parameters} \\
s' & = & (\mathcal{E}\ expr_n) \ldots (\mathcal{E}\ expr_1)\, s
\end{array}
$$

Figure 11: Evaluation functions.

$$
\begin{aligned}
S_0 &= (O_0 \times O_0^* \times I_0^* \times H_0)_\perp^\top \\
I_0 &= L \mapsto O_0 \\
H_0 &= (O_0 \times F) \mapsto O_0 \\
O_0 &= \mathbf{N}_0 \quad \text{(nonnegative integers)} \\
\mathcal{F}_0' &= \mathcal{F}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{start}_0 &= (1, (), ([]), []) \\
\mathsf{save}_0\, L'\, (n, \vec{o}, (i : \vec{i}), h) &= (n, \vec{o}, (i \backslash L', i : \vec{i}), h) \\
\mathsf{restore}_0\, L'\, (n, \vec{o}, (i, i' : \vec{i}), h) &= (n, \vec{o}, (i'[l \mapsto i[l] \mid l \notin L'] : \vec{i}), h) \\
\mathsf{get}_0\, l\, (n, \vec{o}, (i : \vec{i}), h) &= (n, (i[l] : \vec{o}), (i : \vec{i}), h) \\
\mathsf{set}_0\, l\, (n, (o : \vec{o}), (i : \vec{i}), h) &= (n, \vec{o}, (i[l \mapsto o] : \vec{i}), h) \\
\mathsf{load}_0\, f\, (n, (o : \vec{o}), \vec{i}, h) &= (n, (h[o, f] : \vec{o}), \vec{i}, h) \quad o \neq 0 \\
\mathsf{store}_0\, f\, (n, (o, o' : \vec{o}), \vec{i}, h) &= (n, \vec{o}, \vec{i}, h[(o', f) \mapsto o]) \quad o \neq 0 \\
\mathsf{null}_0\, (n, \vec{o}, \vec{i}, h) &= (n, (0 : \vec{o}), \vec{i}, h) \\
\mathsf{new}_0^\perp\, (n, \vec{o}, \vec{i}, h) &= (n + 1, (n : \vec{o}), \vec{i}, h) \\
\mathsf{equal}_0\, (n, (o, o : \vec{o}), \vec{i}, h) &= ((n, \vec{o}, \vec{i}, h), \perp) \\
\mathsf{equal}_0\, (n, (o, o' : \vec{o}), \vec{i}, h) &= (\perp, (n, \vec{o}, \vec{i}, h)) \quad o \neq o'
\end{aligned}
$$

$$
\begin{aligned}
s \sqcup_0 \perp = \perp \sqcup_0 s &= s \\
s \sqcup_0 \top = \top \sqcup_0 s &= \top
\end{aligned}
$$

(Undefined mappings or operator calls result in $\top$. All the operations are strict in $\perp$ and $\top$.)

Figure 12: Definition of standard store.

$$
(S_1, I_1, H_1, O_1, \mathcal{F}_2') = (S_0, I_0, H_0, O_0, \mathcal{F})
$$

$$
\begin{aligned}
\mathsf{load}_1\, f &= \mathsf{load}_0\, f \quad (v_f = \sharp) \\
\mathsf{load}_1\, f\, (n, (o : \vec{o}), \vec{i}, h) &= (n, (o' : \vec{o}), \vec{i}', h') \quad (v_f = \perp) \\
\text{where } i_j' &= i_j \backslash \{l \mid i_j[l] = o'\} \\
h' &= h \backslash (\{(o'', f') \mid h[o'', f'] = o'\} \backslash (o, f)) \\
o' &= h[o, f] \\
\text{and } o' &\notin \vec{o} \\
(\mathsf{start}_1, \mathsf{save}_1, \mathsf{restore}_1, \ldots) &= (\mathsf{start}_0, \mathsf{save}_0, \mathsf{restore}_0, \ldots)
\end{aligned}
$$

Figure 13: Store extended with alias killing.

the top two stack values are equal and other assuming they are not, and $\sqcup$ which combines two states.

The store for the base semantics is given in Figure 12. It consists of four parts, the last "address" to be allocated, a stack of values, a sequence of local variable bindings (one for each active instance of a scope) and an object store which defines what objects are stored in which fields. The notation $(x, x', x'', \ldots : \vec{x})$ is used to access the first elements of a sequence, where $\vec{x}$ is the remainder of the sequence.

A.2 Extended Semantics

Extending the base semantics with the rule for alias killing is handled by defining a new store (see Figure 13) identical to the base store except for the load primitive. The only difference is that the load may make some variables undefined. As a result, the state of the program when using the extended semantics is always included in ($\sqsubseteq$) the state of the program when using the base semantics.

B Abstract Semantics

Our abstract semantics differs from the concrete and extended semantics in that it only models the current procedure's call stack. Before we can prove the correctness of the analysis, we will have to rectify this situation and then also tackle the fact that then the abstract store will be modelled with an infinite lattice, which leads to a function lattice which is infinite in height. We are designing a new analysis framework to handle this situation.

The abstract store keeps track of which unique fields have been read, and into which variables. We adapt the abstract store of Sagiv, Reps and Wilhelm [18] because it is one of the only analyses in the literature that provides "strong nullification." This property permits the abstract store to record when a field is overwritten with a new value; most other analyses can only add the new value to a set of possible values. It also has the property that each abstract object is identified with the list of local variables that refer to it; thus the abstract "mapping" of locals to objects need only include the set of objects.

$$
\begin{aligned}
S_2 &= (\mathbf{N}_0 \times I_2 \times H_2)_\bot^\top \\
I_2 &= \mathcal{P}(O_2) \\
H_2 &= \mathcal{P}((O_2 \times F) \times O_2) \\
O_2 &= \mathcal{P}(X_2) \\
X_2 &= L \cup \mathbf{N} \cup \{x^\sharp, x^\flat, x^\top\}
\end{aligned}
$$

Figure 14: Abstract store types.

We adapt the analysis for our purposes in several major ways:

- Least significantly, we adapt it to work with a stack of evaluated objects.

- As it is defined, the analysis becomes almost useless if any object not referred to by local variables is shared. The authors suggest one possibility, to divide the "primordial soup" into two pots, one of shared and one of unshared objects. For various reasons, including the ability to represent borrowed and undefined values, we instead use three pseudo-variables $x^\top$, $x^\flat$ and $x^\sharp$. Any object with potentially unknown aliases is pointed to by $x^\sharp$, which keeps it from falling into the soup of objects not pointed to by any variable. The other pseudo-variables, $x^\top$ and $x^\flat$ are used to identify undefined and borrowed objects respectively. An implication of using $x^\sharp$ in this way means that the definition of *compatible*$^\sharp$ in the paper cannot be used. This implication has little consequence because of the restrictions on unique fields that we impose.

- Instead of using an "is-shared" flag for each abstract node, we can simply see which pseudo-variables point to it and use that to compute a lattice value $v \in V$.

- The analysis as defined does not handle procedure calls. The authors sketch how it could be extended to interprocedural analysis; here we instead adapt it for intraprocedural analysis.

As a result of these changes, it will be necessary to prove correctness again; we have not yet done so.

$$
\begin{array}{rcl}
\mathsf{start}_2 & = & (0, \{\{\}, \{x^\top\}, \{x^\flat\}, \{x^\flat, x^\sharp\}\}, \{\}) \\
\mathsf{save}_2\, L'\, s & = & \mathsf{apply}\,(\mathsf{add}\; x^\top\, L')\, s \\
\mathsf{restore}_2\, L'\, s & = & \mathsf{apply}\,(\mathsf{remove}\; L')\, s \\
\mathsf{get}_2\, l\,(n-1, i, h) & = & \mathsf{apply}\,(\mathsf{add}\; l\, \{n\})\,(n, i, h) \quad i(l) \neq \top \\
\mathsf{set}_2\, l\,(n, i, h) & = & \mathsf{release}^\flat_2\, \mathsf{apply}\,(\mathsf{add}\; n\, \{l\})\, \mathsf{apply}\,(\mathsf{remove}\; \{l\})\,(n, i, h) \\
\mathsf{load}_2\, f\, s & = & \mathsf{new}^\sharp_2\, \mathsf{release}^\flat_2\, s \quad v_f = \sharp \\
\mathsf{load}_2\, f\,(n, i, h) & = & \mathsf{set}_2\, n\,(n+1, i', h'') \quad v_f = \bot \\
\text{where } h'' & = & h' \cup \{((o, f), \{n+1\}) \mid n \in o \in i'\} \\
(n+1, i', h') & = & \mathsf{new}^\bot_2\, \mathsf{apply}\,(\mathsf{add}\; x^\top\, X')\, \mathsf{apply}\,(\mathsf{remove}\; X')(n, i, h) \\
X' & = & \displaystyle\bigcup_{n \in o \in i, ((o, f), o') \in h} o' \\
\text{and } \{x^\top, x^\flat, x^\sharp\} \cap X' & = & \emptyset \\
\mathsf{store}_2\, f\,(n, i, h) & = & \mathsf{release}^\flat_2\, \mathsf{release}^{v_f}_2\,(n, i, h - \{((o, f), o') \mid o' \in O_2, n-1 \in o \in i\}) \\
\mathsf{null}_2\,(n-1, i, h) & = & (n, i, h) \\
\mathsf{new}^\bot_2\,(n-1, i, h) & = & (n, i \cup \{\{n\}\}, h \cup \{((\{n\}, f), o) \mid ((\{\}, f), o) \in h\}) \\
\mathsf{new}^v_2\,(n-1, i, h) & = & (n, i, h) \sqcup (\mathsf{apply}\,(\mathsf{add}\; x^v\, \{n\})\,(n, i, h)) \quad v \in \{\sharp, \flat\} \\
\mathsf{release}^\flat_2\,(n, i, h) & = & \mathsf{apply}\,(\mathsf{remove}\; \{n\})\,(n-1, i, h) \quad i(n) \sqsubseteq \flat \\
\mathsf{release}^\sharp_2\,(n, i, h) & = & \mathsf{apply}\,(\mathsf{remove}\; \{n\})\, \mathsf{apply}\,(\mathsf{add}\; n\, \{x^\sharp\})\,(n-1, i, h) \quad i(n) \sqsubseteq \sharp \\
\mathsf{release}^\bot_2\,(n, i, h) & = & \mathsf{apply}\,(\mathsf{remove}\; \{n\})\, \mathsf{apply}\,(\mathsf{add}\; n\, \{x^\top\})\,(n-1, i, h) \quad i(n) \sqsubseteq \bot \\
\mathsf{equal}_2\, s & = & (s'_=, s'_\neq) \quad \text{where } s'_R = \mathsf{release}^\flat_2\, \mathsf{release}^\flat_2\, \mathsf{filter}\,(\mathsf{eq}_R\; n\; n-1)\, s \\
(n, i, h) \sqcup_2 (n, i', h') & = & (n, i \cup i', h \cup h') \\
 & & \\
i(x) & = & \displaystyle\bigsqcup_{x \in o \in i} \begin{cases} \top & \text{if } x^\top \in o \\ \flat & \text{if } x^\flat \in o, x^\sharp \notin o \\ \sharp & \text{if } x^\sharp \in o \\ \bot & \text{otherwise} \end{cases} \\
\mathsf{add}\; x\, X\, X' & = & X' \cup X \quad (x \in X') \\
\mathsf{add}\; x\, X\, X' & = & X' \quad (x \notin X') \\
\mathsf{remove}\; X'\, X'' & = & X'' - X' \\
\mathsf{apply}\; c\,(n, i, h) & = & \mathsf{check}\,(n, \{c\, o \mid o \in i\}, \{((c\, o, f), c\, o') \mid ((o, f), o') \in h, c\, o' \neq \{\}\}) \\
\mathsf{eq}_R\; x\, x'\, X' & = & (x \in X') R (x' \in X') \quad R \in \{=, \neq\} \\
\mathsf{filter}\; p\,(n, i, h) & = & \mathsf{check}\,(n, \{o \mid o \in i, p\, o\}, \{((o, f), o') \mid ((o, f), o') \in h, p\, o, p\, o'\}) \\
\mathsf{check}\,(n, i, h) & = & (n, i, h) \quad (\{x^\top, x^\flat, x^\sharp\} \supseteq o \in i, ((o, f), o') \in h) \Rightarrow o' \cap \{x^\top, x^\flat, x^\sharp\} = \emptyset
\end{array}
$$

Figure 15: Abstract primitives.

B.1 Abstract store

Figure 14 gives the types of the domains used in the abstract store. Each store is a lifted tuple of an integer, a set of nodes, and a set of possible field assignments. The integer gives the depth of the evaluation stack. In the concrete semantics, we needed a mapping from local variables to store nodes, but since each abstract node $o \in O_2$ is identified by the set of "simple" variables that alias it, such a mapping is unnecessary in the abstract semantics. Instead we simply keep I_2 the set of nodes under analysis. The set of *simple* variables X_2 is the union of the local variables, evaluation stack locations (represented by natural numbers) and pseudo-variables ($x^\top$, $x^\flat$ and $x^\sharp$). The field store H_2 is similar in structure to the concrete semantics, except it keeps a set of possibilities for a field's value, rather than its one true value. (Actually, because of the way nodes are identified, it keeps a set of aliasing possibilities.) Furthermore, since the analysis only tracks unique fields, H_2 only keeps possibilities for aliases of unique fields.

Figure 15 gives definition of the abstract primitives. In the starting state, the evaluation stack is empty ($n = 0$). The starting abstract store has four nodes: one unaliased, a node representing an undefined value, a node represented a borrowed value that is unique, and a node that represents shared values (possibly borrowed). We assume there are no aliases of unique fields at the start and thus H_2 is empty. Appendix B.2 shows how we use this state to start checking a procedure by making assumptions about the parameters.

The definitions of save and restore are relatively simple because we assume all local variables are distinct; save has all the locals refer to the undefined node (add adds aliases), and remove removes each local from any alias set. The get primitive adds the next stack location as an alias of every node to which the local may refer; set first removes the local from any alias sets and then makes it alias the top stack location, after which the stack is popped (using release).

If a shared field is accessed using load, we assume the result could be potentially aliased with any shared value in our abstract store; we ignore the reference to the object with the field (we pop it off the stack) and then push a reference to a "new" shared value. The load of a unique field is more interesting. Starting near the bottom of the definition, one sees we first get the set X' of variables that may alias the unique field. This set must not contain any pseudo-variables; if it does, the field has been compromised and may not be read (the result abstract store thus then defaults to $\top$). In the third line of the definition, reading from the right, we see that first these variables are nullified (removed) and then all made undefined (made to point to what $x^\top$ points to). The process of making these variables undefined causes the cell that the load may find to slide into the soup of anonymity (it has no aliases). This is as it should be, because another thread may have written the unique field since we last read it. Thus to perform the load, we materialize a new node onto the stack. The new field store h'' gets a field definition with pointers to the new node. This definition works even if stack slot n was one of the variables made undefined in the first step. The desired value is now on top of the stack but there is still a reference to the object in the next stack slot; set is used to rectify the situation.

When we process a store, we first remove the connection to any node pointed to before the store. Then the value being stored is processed according to the field annotation, for which see the explanation of release below. Finally, we discard the reference used to access the modified object.

Since the store does not model null, pushing the null value on the evaluation stack is very simple. The new primitive is used to allocate new unique, shared or borrowed values. A new unique value is materialized as a node referred to only by the (new) top of the stack. Additionally, since $\mathsf{new}^\perp$ is used for fetching unique fields and for allocating new objects, it also assumes that any aliases of unique fields of the anonymous object also may be aliases of the new objects fields. When allocating a "new" shared or borrowed node, we use the respective pseudo-variable as a model for what this variable may alias. The "join" operation essentially doubles the number of nodes referred to by the respective pseudo-variable.

A value popped of the top of the stack using release can be simply checked for definedness, or it can be made shared or it can be asserted as a unique reference. In the latter two cases, we modify the alias sets to make then shared or undefined respectively.

The equal primitive creates two abstract states, one in which the two top stack variables alias each other and one in which they do not. The join primitive ($\sqcup$) only joins two states with equal-depth evaluation stacks.

The $i(x)$ operation is used to compute the state of a simple variable x. It is unique if none of the objects to which it refers have pseudo-variable aliases. Otherwise, it is undefined if $x^\top$ is an alias, and shared if $x^\sharp$ is an alias. Only if it might alias a borrowed unique variable, do the restrictions on borrowed variables apply to it.

After every operation affecting the field store, we apply check to ensure we have not "lost track" of any objects with compromised unique fields.

B.2 Checking a Procedure

To check a procedure, we evaluate it with an abstract store that represents any concrete store where the caller keeps its obligations:

$$\mathsf{start} \sqsupseteq \mathsf{release}^{v_0}\,(\mathcal{F}\,p)\,\mathsf{new}^{v_n}\;\ldots\;\mathsf{new}^{v_1}\,\mathsf{start}$$

where v_i is the annotation on formal parameter l_i of p. The return value is checked against its annotation using $\mathsf{release}$. The resulting abstract store must have the same form as we started with, or be more restrictive; this inequality represents the fact that the procedure fulfills its responsibilities to its caller.

When the procedure calls another procedure, we check that it fulfills its responsibilities as a caller by first checking that it has restored everything read by the callee, and then using $\mathsf{release}$ to check each actual parameter. Finally, we assume the called procedure returns an appropriate value:

$$\mathcal{F}'\,p\,s = \mathsf{new}^{v_0}\,\mathsf{release}^{v_1}\;\ldots\;\mathsf{release}^{v_n}\,\mathsf{checkreads}\,s$$

where v_i is the annotation on formal parameter l_i of p. The primitive $\mathsf{checkreads}$ is an oracle that simulates the effect of reading any fields that the procedure may read.