Final Field Semantics

Jeremy Manson and William Pugh

April 7, 2003, 12:27pm

Final fields are fields that are initialized once and then never changed. The detailed semantics of final fields are somewhat different from those of normal fields. In particular, we provide the compiler with great freedom to move reads of final fields across synchronization barriers and calls to arbitrary or unknown methods. Correspondingly, we also allow the compiler to keep the value of a final field cached in a register and not reload it from memory in situations where a non-final field would have to be reloaded.

Final fields also provide a way to create thread-safe immutable objects that do not require synchronization. A thread-safe immutable object is seen as immutable by all threads, even if a data race is used to pass references to the immutable object between threads.

In the abstract, the guarantees for final fields are as follows. When we say an object is "reachable" from a final field, that means that the field is a reference, and the object can be found by following a chain of references from that field. When we say the "correctly initialized" value of a final field, we mean both the value of the field itself, and, if it is a reference, all objects reachable from that field.

- At the end of an object's constructor, all of its final fields are "frozen" by an implicit "freeze" action.
- If a thread only reads references to an object that were written after the last freeze of its final fields, that thread is always guaranteed to see the frozen value of the object's final fields. Such references are called *correctly published*, because they are published after the object is initialized. There may be objects that are reachable by following a chain of references from such a final field. Reads of those objects will be up to date as of the the freeze of the final field.

- Conversely, if a thread reads a reference to an object written before a freeze, that thread is not automatically guaranteed to see the correctly initialized value of the object's final fields. Similarly, if a thread reads a reference to an object reachable from the final field without reaching it by following pointers from that final field, the thread is not automatically guaranteed to see the value of that object when the field was frozen.

- If a thread is not guaranteed to see a correct value for a final field or anything reachable from that field, the guarantees can be enforced by a normal happens-before relationship. In other words, those guarantees can be enforced by normal synchronization techniques.

- When you freeze a final which points to an object, then freeze a final field of that object, there is a happens-before relationship between the first freeze and the second.

Complications Retrofitting the semantics to the existing Java programming language requires that we deal with a number of complications:

- Using serialization in Java to read an object requires that the object first be constructed, initializing the final fields of the object. Then deserialization code is invoked to set the object to what is specified in the serialization stream. This means that the semantics must allow for final fields to change after objects have been constructed.

  Although our semantics allows for this, the guarantees we make are somewhat limited; they are specialized to deserialization. These guarantees are not intended to be part of a general and widely used mechanism for changing final fields. In particular, using JNI to modify final fields is illegal; the use of this technique will invalidate the semantics of the VM.

  To formalize the semantics for multiple writes/initializations of a final field, we allow multiple freezes. A second freeze action might, for example, take place after deserialization is complete.

- The final static fields System.in, System.out and System.err are defined to be mutable by public methods. Thus, we treat these three fields (and only these three fields) as write-protected, rather than final.

1 Introduction to Semantics

The semantics are laid out somewhat more formally than this. Accesses to final fields have the same semantics as accesses to ordinary fields. The only difference in their respective semantics is that the happens-before relationship includes a set writesBeforeDereference of additional writes that are visible when executing a given action. Most of this document discusses the construction of that set.

2 Full Semantics

We present here the formal semantics in full. We will then deconstruct them more carefully.

$\stackrel{hb}{\rightarrow}$ The notation $a \stackrel{hb}{\rightarrow} b$ denotes that a happens before b because of the semantics of non-final fields.

2.1 Freezes Associated with Writes

When an address a is stored in the heap by thread t at write w, it is stored as a pair $\langle a, G \rangle$, where G is a set of freeze operations defined as:

$$G = \{f \mid f \stackrel{hb}{\rightarrow} w\} \cup \text{freezesBeforeDereference}(t, a)$$

The set freezesBeforeDereference(t, a) is the set of freezes associated with the address a in thread t, as defined below.

2.2 Effect of Reads

A read r in thread t of field x of the object at address c returns a tuple $\langle a, G \rangle$; each such read has two corresponding sets. The first, the set freezeBeforeRead(r), is a set of freezes associated with the read. The second, the set writesBeforeRead(r), is a set of writes associated with the read. These sets are used to compute the values that are legal to see for final fields.

2.2.1 Freezes Seen as a Result of Reads

Consider a read r in thread t of field x of the object at address c that returns a tuple $\langle a, G \rangle$. The set of freezes freezeBeforeRead(r) associated with a read r of address a is:

$$\text{freezeBeforeRead}(r) = G \cup \{f \mid f \xrightarrow{hb} r\} \cup \text{freezesBeforeDereference}(t, c)$$

The set freezesBeforeDereference(t, a) is the intersection of the sets of freezes freezeBeforeRead(r) that that thread saw every time it read a reference to o. Let sawAddress(t, a) be the set of reads in thread t that returned the address a.

$$\text{freezesBeforeDereference}(t, a) = \bigcap_{r \in \text{sawAddress}(t,a)} \text{freezeBeforeRead}(r)$$

If a thread t allocated a (including all situations where sawAddress(t, a) is empty), then the set freezesBeforeDereference(t, a) is empty. Because the definition of freezesBeforeDereference(t, a) uses freezeBeforeRead(t, c), the actual freezesBeforeDereference sets are defined by the least fixed point solution to these equations (i.e., the smallest sets that satisfy these equations). This is explained more fully in Appendix A.

2.2.2 Writes Visible at a Given Read

For any read instruction r, there is a set of writes, writesBeforeRead(r), that is known to be ordered before r due to the special semantics of final fields. These ordering constraints are taking into account in determining which writes are visible to the read r. However, these ordering constraints do not otherwise compose with the standard happens before ordering constraints.

The set writesBeforeRead(r) is defined in terms the writes that are known to occur before any dereference of the address c by thread t, which is given by writesBeforeDereference(t, c). These equations are recursive so the solution is defined to be the least fixed point solution.

Result set for non-final fields or array elements Consider a read r in thread t of non-field field or element x of the object at address c. The set of writes writesBeforeRead(r) is defined as:

$$\text{writesBeforeRead}(r) = \text{writesBeforeDereference}(t, c)$$

Result set for final fields Consider a read r in thread t of final field x of the object at address c. The set of writes writesBeforeRead(r) is defined as:

$$\begin{array}{l}\text{writesBeforeRead}(r) = \\ \quad \text{writesBeforeDereference}(t, c) \cup \\ \qquad \{w \mid \exists f \text{ s.t. } f \in \text{freezesBeforeDereference}(t, c) \\ \qquad\qquad \wedge \ f \text{ is a freeze of } c.x \\ \qquad\qquad \wedge \ w \xrightarrow{hb} f\}\end{array}$$

Result set for static fields The set writesBeforeRead(r) associated with a read r of a static field is the empty set.

Visible Write Set The set writesBeforeDereference(t, a) is defined to be the intersection of the writesBeforeRead sets for all reads that see the value a.

$$\text{writesBeforeDereference}(t, a) = \bigcap_{r \in \text{sawAddress}(t,a)} \text{writesBeforeRead}(r)$$

If a thread t allocated a (any situations where sawAddress(t, a) is empty are included in this), then writesBeforeDereference(t, a) is empty. As with freezesBeforeDereference, these equations are recursive and the solution is defined to be the least fixed point solution to the equations (i.e., the smallest sets that satisfy these equations). This is explained more fully in Appendix A.

When a read r examines the contents of any field $a.x$ in thread t, all of the writes in writesBeforeRead(r) are considered to be ordered before r. In addition, if $a.x$ is a final static field, then r will always return $a.x$'s correctly constructed value, unless r happens in the thread that performed the class initialization, before the field was written. These constraints, along with the normal happens before constraints, are used to determine the legal values for the read.

2.3 Single Threaded Guarantees

For cases where a final field is set once in the constructor, the rules are simple: the reads and writes of the final field in the constructing thread are ordered according to program order.

We must treat the cases (such as deserialization) where a final field can be modified after the constructor is completed a little differently. Before modifying a frozen final field, the system must call a `realloc()` function, passing in a reference to the object, and getting out a reference to the object through which the final fields can be reassigned. The only appropriate way to use this `realloc()` function is to pass the only live reference to the object to the `realloc()` function, and only to use that value `realloc()` returns to refer to the object after that call.

After getting back a "fresh" copy from `realloc()`, the final fields can be modified and refrozen. The `realloc()` function will likely be implemented as a no-op, but it can be thought of as a function that might decide to perform a shallow copy.

In more detail, each reference within a thread essentially has a version number. Passing a reference through `realloc()` increments that version number. A read of a final field is ordered according to program order with all writes to that field using the same or smaller version number.

Note that two references to the same object but with different version numbers should not be compared for equality. If one reference is ever compared to a reference with a lower version number, then that read and all reads of final fields from that reference are treated as if they have the lower version number.

2.4 Write Protected Fields

`System.in`, `System.out`, and `System.err` are final static fields that can be changed by the methods `System. setIn()`, `System. setOut()` and `System. setErr()`. These methods call native code that can modify the respective fields; we refer to them as "write-protected" fields to distinguish them from ordinary final fields.

The compiler needs to treat these fields differently from other final fields. A read of an ordinary final field is "immune" from synchronization: the barrier involved in an acquire does not affect what value is read from the final field. Since these other fields may be seen to change, synchronization

f1 is a final field; its default value is 0

Thread 1	Thread 2	Thread 3
o.f1 = 42	r = p;	s = q;
p = o;	i = r.f1;	j = s.f1;
freeze o.f1	t = q;	
q = o;	if (t == r)	
	k = t.f1;	

We assume r and s do not see the value null. i and k can be 0 or 42, and j must be 42.

Figure 1: Example of Simple Final Semantics

events should have an affect on them.

Therefore, the semantics dictate that these fields be treated as normal fields that cannot be changed by bytecode, unless that bytecode is in the constructor of the `System` class. Since these fields do not have the semantics of final fields, but can only be changed by special code, we call them *write-protected* fields.

3 Simple Semantics

We can now go more carefully into the details of how the semantics work. We will first discuss how the notion of a freeze is communicated between threads.

Consider Figure 1. A freeze, for the moment, is simply what happens at the end of a constructor. Although s, r and t can see the value null, we will not concern ourselves with that; that just leads to a null pointer exception.

The reference q is correctly published after the end of o's constructor. Our semantics guarantee that threads that only see correctly published references to o will see the correct value for o's final fields. We therefore want to construct a special happens before relationship between the freeze of $o.f1$ and the read of it as $q.f1$ in Thread 3.

The read of $p.f1$ in Thread 2 is a different case. Thread 2 sees p, an incorrectly published reference to object o; it was made visible before the end of o's constructor. A read of $p.f1$ could easily see the default value for that field, if a compiler decided to reorder the write to p with the write to

$o.f1$. No read of $p.f1$ should be guaranteed to see the correctly constructed value of the final field.

What about the read of $q.f1$ in Thread 2? Is that guaranteed to see the correct value for the final field? A compiler could determine that p and q point to the same object, and therefore reuse the same value for both $p.f1$ and $q.f1$ for that thread. We want to allow the compiler to remove redundant reads of final fields wherever possible, so we allow k to see the value 0. The object can be thought of as being "tainted"; the thread is never guaranteed to see its correctly constructed final fields.

In addition to compiler transformations allowing this behavior, a DSM-like implementation would allow it. More generally, if a thread t reads an incorrectly published reference to an object o, thread t forever sees a tainted version of o without any guarantees of seeing the correct value for the final fields of o.

These properties also need to be discussed in terms of multiple freeze operations; we will allow multiple freezes for various implementation dependent issues. The tainting process therefore becomes slightly more complicated. Each reference to an object is associated with some set of guarantees about what will be seen. These guarantees tell which freeze operations are visible to the read. A thread only gets those guarantees that are associated with all of the references it sees.

In Figure 1, Thread 1 is provided with the guarantees from the p reference, because p in Thread 1 provides the fewest guarantees. Thread 2 is provided with the guarantees from the q reference, because, in Thread 2, q provides the fewest guarantees. We can now present this formally: we will show the relevant portions of the full semantics.

Freezes are ordered with respect to writes of references When an address a is stored in the heap by thread t at write w, it is stored as a pair $\langle a, G \rangle$, where G is a set of freeze operations defined as:

$$G = \{f \mid f \overset{hb}{\rightarrow} w\}$$

Freezes that are associated with a read r of a variable v (which is $o.f$) by thread t Consider a reference a to an object o. A thread t may read a multiple times: each such read r will be a member of the set sawAddress(t, a). The set of freezes freezeBeforeRead(r) associated with one

$o.f2$ is a final reference to the class of object p. $p.b$ is a reference to an integer array. a is a length 1 array whose contents are initialized to 0s.

Thread 1	Thread 2	Thread 3
o.f2 = p;	i = a[0];	s1 = pub.f2;
p.b = a;	r1 = pub.f2;	s2 = s1.b;
a[0] = 42;	r2 = r1.b;	s3 = s2[0];
freeze o.f2;	r3 = r2[0];	
pub = o;		

We assume $r1$ and $s1$ do not see the value null.
$r2$ and $s2$ must both see the correct pointer to array a.
$s3$ must be 42, but $r3$ does not have to be 42.

Figure 2: Example of Transitive Final Semantics

of these reads r is G:

$$\text{freezeBeforeRead}(r) = G$$

The set $\text{freezesBeforeDereference}(t, a)$ is the intersection of the sets of freezes $\text{freezeBeforeRead}(r)$ that that thread saw at r.

$$\text{freezesBeforeDereference}(t, a) = \bigcap_{r \in \text{sawAddress}(t,a)} \text{freezeBeforeRead}(r)$$

If a thread t allocated a (including all situations where $\text{sawAddress}(t, a)$ is empty), then the set $\text{freezesBeforeDereference}(t, a)$ is empty.

As a first approximation, we make the following guarantee: if a freeze of an object is associated with a given read r of one of that object's final fields, r will return the correctly constructed value for that final field. This will be refined in the next section.

4 Reachability Guarantees

4.1 Standard Visibility Rules

Now consider Figure 2. In this example, the final field $o.f2$ is a reference. In this case, we want to make some additional guarantees. It would not be

Thread 1	Thread 2
o.f = p;	i = pub.x;
p.g = 42;	j = pub.y;
pub.x = o;	k = j.f;
freeze p.g;	l = k.g;
pub.y = o;	

Figure 3: Example of Reachability

very useful if we only guaranteed that the values read for references were correct, without also making some guarantees about the objects to which those references point. In this case, we need to make guarantees for $o.f2$, the object p to which it points and the array pointed to by $p.b$.

We make a very simple guarantee: if a final reference is published correctly, and its correct value was guaranteed to be seen by an accessing thread (as described in Section 3), everything *transitively reachable* from that final reference is also guaranteed to be up to date as of the freeze. In Figure 2, o's reference to p, p's reference to a and the contents of a are all guaranteed to be seen by Thread 3. We call this idiom a *dereference chain*.

We make one exception to this rule. In Figure 2, Thread 2 reads $a[0]$ through two different references. A compiler might statically determine that these references are the same, and reuse i for $r3$. Here, a reference reachable from a final field is read by a thread in a way that does not provide guarantees; it is not read through the final field. If this happens, the thread "gives up" its guarantees from that point in the dereference chain; the address is now tainted. In this example, the read of $a[0]$ in Thread 2 can return the value 0.

4.2 More about Reachability

The definition of reachability is a little more subtle than might immediately be obvious. Consider Figure 3. It may seem that the final field $p.g$ can only be reached through one dereference chain. However, consider the read of $pub.x$. A global analysis may indicate that it is feasible to reuse its value for j. $o.f$ and $p.g$ may then be read without the guarantees that are provided when they are reached from $pub.y$.

The upshot of this is that a reachability chain is not solely based on syntactic rules about where dereferences occur. There is a link in a dereference

chain from any dynamic read of a value to any action that dereferences that value, no matter where the dereference occurs in the code.

We can now provide detailed semantics for the way in which writes reachable from final fields are made visible to other threads; this was deferred from the previous section.

4.3 Formal Reachability Guarantees

If a variable is only accessed via a series of dereferences from final fields, it garners the guarantees made by those final fields. Otherwise, the correct value must be seen as a result of some other happens-before relationship.

We define a set writesBeforeDereference(t, a) that contains the writes that happen before reads of an address a in thread t because of the special semantics of final fields. As we have seen, if a is the value returned by a read of some variable $o.x$, then the writes guaranteed to be visible when reading a depend on the writes that were guaranteed to be seen when reading o. This is because o is the previous element on the dereference chain. The set visible at the read of o is writesBeforeDereference(t, o).

To define the set writesBeforeDereference(t, a) formally, we need to examine what values might be seen at any read that might return the value a. For a given read r, this is the set writesBeforeRead(r).

When a read r is of a static variable, the set writesBeforeRead(r) is the empty set. All reads of static final fields f are guaranteed to see the correctly constructed value for that field, unless those reads occur before the write to f.

For a read r of a non-final field $o.x$ that sees a, the set writesBeforeRead(r) is simply the set of writes available to be seen when we read o, the last element in the dereference chain before $o.x$. The set writesBeforeDereference(t, o) is defined in this way.

For a read of a final field $c.x$ that sees a, the set writesBeforeRead(r) consists partly of the set of writes available to be seen when we read o; this is the set writesBeforeDereference(t, o). For the additional guarantees for final fields, we also include all writes that happen before freezes of $o.f$ associated with o.

$$\text{writesBeforeRead}(r) = \text{writesBeforeDereference}(t, c) \cup \{w \mid w \xrightarrow{hb} f \wedge$$

Thread 1	Thread 2	Thread 3
o.f1 = 42;	r1 = Global.a;	s1 = Global.b;
freeze o.f1;	Global.b = r2;	s2 = s1.f1;
Global.a = o;		

$s2$ is guaranteed to see 42, if $s1$ is a reference to o.

Figure 4: Freezes are Passed Between Threads

$$f \in \text{freezesBeforeDereference}(t, c) \wedge$$
$$f \text{ is a freeze of } c.x\}$$

The notation $a \overset{hb}{\rightarrow} b$ denotes that a happens before b. As a reminder, the set freezesBeforeDereference(t, a) is the set of freezes associated with a.

Finally, the set writesBeforeDereference(t, a) consists of the intersection, for all reads in thread t of the sets writesBeforeRead(r), where v is any variable. This is defined as:

$$\text{writesBeforeDereference}(t, a) = \bigcap_{r \in \text{sawAddress}(t,a)} \text{writesBeforeRead}(r)$$

The set sawAddress(t, a) is the set of reads in thread t that returned the address a. If a thread t allocated a (including all situations where sawAddress(t, a) is null), then the set writesBeforeDereference(t, a) is empty. When reading any field $a.x$ in thread t, writesBeforeRead(t, a) is used for determining the legal values; it is the set of actions that are seen to happen before this read due to the special semantics of final fields.

The full semantics of final fields is discussed in Section 2. That discussion also includes additional guarantees made for static fields.

5 Freezes Are Passed Between Threads

Consider Figure 4. If $s1$ is a reference to o, should $s2$ have to see 42? The answer to this lies in the way in which Thread 3 saw the reference to o.

Thread 1 correctly published a reference to o, which Thread 2 then observed. Had Thread 2 then read a final field of o, it would have seen the correct value for that field; the thread would have to have ensured that it

saw all of the updates made by Thread 1. To do this on SMP systems, Thread 2 does not need to know that it was Thread 1 that performed the writes to the final variable, it needs only to know that updates were performed. On systems with weaker memory constraints (such as DSMs), Thread 2 would need this information; we shall discuss implementation issues for these machines later.

How does this impact Thread 3? Well, like Thread 2, Thread 3 cannot see a reference to o until the freeze has occurred. Like Thread 2, Thread 3 is guaranteed to see all of the writes to o that occurred prior to the freeze. There is therefore no reason not to provide Thread 3 with the same guarantees with which we provide Thread 2.

This causes a slight change to the formal semantics. Remember that there are a set of freezes associated with each write of a reference. Until now, this set only included the freezes that explicitly happened before that write. It must now include any freezes that are guaranteed to be seen by the reference we are writing out. In Figure 4, the write to *Global.b* will include the freezes that *Global.a* sees. The change is a simple one. Recall that when we write out an address, we write it out as a pair $\langle a, G \rangle$: G is the set of freezes associated with that address. We simply add the set freezesBeforeDereference(t, a) to G:

$$G = \text{freezesBeforeDereference}(t, a) \cup \{f \mid f \overset{hb}{\rightarrow} w\}$$

6 Additional Information for Reads

6.1 Semantics' Interaction with Happens Before Edges

Now consider Figure 5. We want to describe the interaction between ordinary happens-before relationships and final field guarantees. In Thread 1, o is published incorrectly (before the freeze). However, if the code in Thread 1 happens before the code in Thread 2, the normal happens-before relationships ensure that Thread 2 will see all of the correctly published values. As a result, j will be 42.

What about the reads in Thread 3? We assume that k does not see a null value: should the normal guarantees for final fields be made? We can answer this by noting that the write to *Global.b* in Thread 2 is the same as a correct publication of o, as it is guaranteed to happen after the freeze. We

p.x is initialized to 42.
o.f is final.

Thread 1	Thread 2	Thread 3
lock m;	lock m;	k = Global.b;
Global.a = o;	i = Global.a;	l = k.x;
o.f = p;	Global.b = i;	
freeze o.f;	j = i.x;	
unlock m	unlock m;	

If the unlock in Thread 1 happens before the unlock in Thread 2:
i will see o.
j will see 42.
k will see o or null.
l will see 42 or throw a null pointer exception..

Figure 5: Example of Happens Before Interaction

therefore make the same guarantees for any read of *Global.b* that sees o as we do for a read of any other correct publication of o.

A freeze can piggyback on a happens-before relationship Given a freeze f and a read r of a final variable v, where the read r happens in thread t, if $f \overset{hb}{\rightarrow} r$, the set containing that freeze counts as one of the sets associated with the the read of the object pointing to v.

We can now define the changes to the formalism that reflect this. The difference is in the set of freezes read at an access r: the set freezeBeforeRead(r). The formulation is now as follows:

$$\text{freezeBeforeRead}(r) = G \cup \{f \mid f \overset{hb}{\rightarrow} w\}$$

7 Reads and Writes of Final Fields in the Same Thread

Up to this point, we have only made guarantees about the contents of final fields for reads that have seen freezes of those final fields. This implies that

The `deserialize()` method sets the final field $p.x$ to 42 and then performs a freeze on $p.x$. It passes back a reference to the p object. This is done in native code.

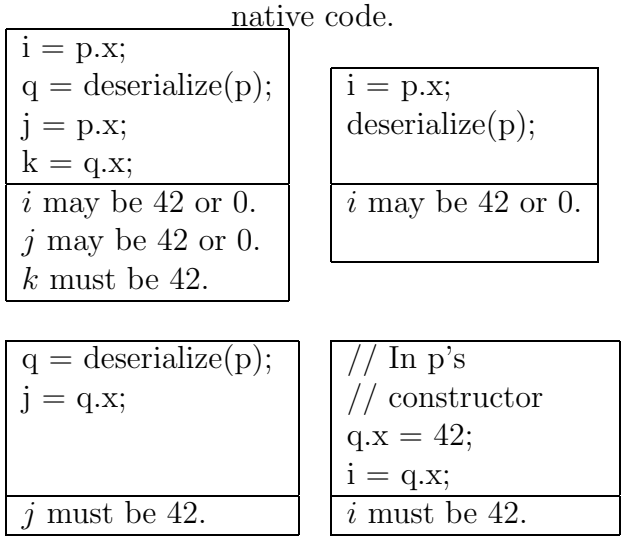

Figure 6: Four Examples of Final Field Optimization

a read of a final field in the same thread as the write, but before a freeze, might not see the correctly constructed value of that field.

Sometimes this behavior is acceptable, and sometimes it is not. We have four examples of how such reads could occur in Figure 6. In three of the examples, a final field is written via deserialization; in one, it is written in a constructor.

We wish to preserve the ability of compiler writers to optimize reads of final fields wherever possible. When the programs shown in Figure 6 access $p.x$ before calling the `deserialize()` method, they may see the uninitialized value of $p.x$. However, because the compiler may wish to reorder reads of final fields around method calls, we allow reads of $p.x$ to see either 0 or 42, the correctly written value.

On the other hand, we do want to maintain the programmer's ability to see the correctly constructed results of writes to final fields. We have a simple metric: if the reference through which you are accessing the final field was not used before the method that sets the final field, then you are guaranteed to see the last write to the final field. We call such a reference a *new* reference to the object.

This rule allows us to see the correctly constructed value for $q.x$. Because the reference being returned from `deserialize()` is a new reference to the same object, it provides the correct guarantees.

For cases where a final field is set once in the constructor, the rules are simple: the reads and writes of the final field in the constructing thread are ordered according to program order.

We must treat the cases (such as deserialization) where a final field can be modified after the constructor is completed a little differently. Before modifying a frozen final field, the system must call a `realloc()` function, passing in a reference to the object, and getting out a reference to the object through which the final fields can be reassigned. The only appropriate way to use this `realloc()` function is to pass the only live reference to the object to the `realloc()` function, and only to use that value `realloc()` returns to refer to the object after that call.

After getting back a "fresh" copy from `realloc()`, the final fields can be modified and refrozen. The `realloc()` function will likely be implemented as a no-op, but it can be thought of as a function that might decide to perform a shallow copy.

In more detail, each reference within a thread essentially has a version number. Passing a reference through `realloc()` increments that version number. A read of a final field is ordered according to program order with all writes to that field using the same or smaller version number.

Note that two references to the same object but with different version numbers should not be compared for equality. If one reference is ever compared to a reference with a lower version number, then that read and all reads of final fields from that reference are treated as if they have the lower version number.

We can now see how these rules apply to the examples in Figure 6. Any reference to a final field via q will be treated as a "new" version number, and see the correct value for the field. Any reference to a final field via p will have the "old" version number, and not be guaranteed to see the correctly constructed value.

This final change provides us with the full semantics; we now focus on additional examples and implementation issues.

a is an array whose first element is initialized to 0.
o is of a class that has two final fields, f and g.
$o.b$ is not a final field.

Thread 1	Thread 2	Thread 3
Global.b = a;	m = Global.x;	n = Global.x;
a[0] = 1;	i = m.f[0];	k = n.f[0];
o.f = a;	j = m.g[0];	l = Global.b[0];
o.g = a;		
freeze o.f;		
freeze o.g		
Global.x = o;		

Figure 7: Thread 2 must see a[0] == 1, Thread 3 may not

8 More Examples

8.1 Reachability from Two Distinct Pointers

In Figure 7, we have three references to the same array; two are final, and one is not. As with our other examples, we shall assume that no reads return a null value. In Thread 2, both array references are final fields; both references should have guarantees about the contents of the array. In Thread 3, only one of the array references is final. A compiler could reorder the read of Global.b[0] to the beginning of Thread 3, see the value 0 for $a[0]$, and then reuse that value for $n.f[0]$. Thus, no guarantees are made for Thread 3.

How does this play out in the semantics? We first consider every dereference chain through which we accessed a variable. In Thread 2, the array is accessed through both $m.f$ and $m.g$. We then intersect the freezes seen by both of these chains of access: since $m.f$ and $m.g$ see the same set of freezes, the read of $a[0]$ is provided a full set of guarantees.

In Thread 3, the array is accessed through both $n.f$ and $Global.b$. We intersect the set of freezes see by both "approaches" to the array. Even though $n.f$ provides guarantees, $Global.b$ provides none: the intersection of these guarantees is empty. Therefore, no guarantees are made for Thread 3.

p.g is final in the first and second example, and not final in the second.

Thread 1	Thread 2
p.g = a;	i = Global.x;
a[0] = 1;	j = i.f;
freeze p.g;	k = j.g;
o.f = p;	l = k[0];
freeze o.f;	
Global.x = o;	

Thread 1	Thread 2
p.g = a;	i = Global.x;
freeze p.g;	j = i.f;
a[0] = 1;	k = j.g;
o.f = p;	l = k[0];
freeze o.f;	
Global.x = o;	

Thread 1	Thread 2
p.g = a;	i = Global.x;
a[0] = 1;	j = i.f;
o.f = p;	k = j.g;
freeze o.f;	l = k[0];
Global.x = o;	

Figure 8: Both Examples are Guaranteed to See $a[0] == 1$

8.2 Reachability through Multiple Final Fields

Now consider Figure 8. In all of these examples, l in Thread 2 is guaranteed to be the value 1 (again, assuming that the read of $Global.x$ does not return a null value). This is a reasonable expectation for each thread. In both the example in which $p.g$ is not final and the example in which the write to the array occurs after the freeze of $p.g$, the read of a should be protected by the freeze of $o.f$.

The way that this plays itself out in the formal semantics is very simple; the freezes that are seen after you have followed a dereference chain are the freezes seen by all of the final fields in that dereference chain.

9 Implementation Issues

9.1 Permitted Optimizations

The fundamental question under the aegis of implementation issues is a simple one: what reorderings can a compiler writer prise out of final fields? To be more precise, we must address two issues: first, what are the reorderings

we are not allowed to perform that we might perform on normal fields? Second, what are the reorderings we are allowed to perform that we may not perform on normal fields?

9.1.1 Prohibited Reorderings

The most important guarantee that we make for the use of final fields is that if an object is only made visible to other threads after its constructor ends, then those other threads will see the correctly initialized values for its final fields. It is therefore of paramount importance that a write of a reference to a memory location where it might become visible to another thread never be reordered with respect to a write to a final field. In addition, such a write should never be reordered with anything reachable from a final field that was written before the end of the constructor.

As an example of this, we look back at Figure 2. In this figure, the write to *pub* in Thread 1 must never be reordered with respect to anything that takes place before the read. If such a reordering occurred, then Thread 3 might be able to see the reference to the object without seeing the correctly initialized final fields.

9.1.2 Enabled Reorderings

There is one principles that guides whether a reordering is legal for final fields: the notion that "all references are created equal". If a thread reads a final field via multiple references to its containing object, it doesn't matter which one of those references is used to access the final field. None of those references will make more guarantees about the contents of that final field than any other. The upshot of this is that as soon as a thread sees a reference to an object, it may load all of that object's final fields, and reuse those values regardless of intervening control flow, data flow, or synchronization operations.

Consider once more the code in Figure 3. As soon as the read of $pub.x$ occurs, all of the loads of o's final fields may occur; the reference $pub.x$ of o is "equal" to the reference $pub.y$ of o. This might cause uninitialized values to be seen for $p.g$ and $o.f$, as the read of $pub.x$ can occur before the freeze of $p.g$.

This should not be taken to mean that normal fields reachable through final fields can always be treated in the same way. Consider Figure 9. As

ready is a boolean volatile field, initialized to `false`.
a is an array.

Thread 1	Thread 2
a = 1,2,3;	i = pub.x;
ready = true;	j = i.f;
o.f = a;	if (ready) {
pub.x = o;	k = j[0];
freeze o.f;	}

Figure 9: Happens Before Does Matter

a reminder, a happens before ordering is enforced between a write to and a read of a volatile. In this figure, the volatile enforces a happens before ordering between the write to the array and the read of $j[0]$: assuming that the other reads see the correct values (which is not guaranteed), then k is required to have the value 1.

9.2 Implementation on Weak Memory Orders

One of the problems with guaranteeing where writes are seen without explicit lock and unlock actions to provide ordering is that it is not always immediately obvious how the implementation will work. One useful thing to do is consider how this approach might be implemented on a system where few guarantees are given about memory coherence.

Imagine a Lazy Release Consistent (LRC) machine: a processor acquires data when a lock action occurs, and releases it when an unlock action occurs. The data "piggyback" on the lock acquire and release messages in the form of "diffs", a listing of the differences made to a given page since the last acquire of that memory location.

Let us assume that each object with a final field is allocated in space that had previously been free. The only way for a second processor to see a pointer to that object at all is to perform an acquire after the processor constructing the object performed a release. If the release and the acquire do not happen, the second processor will never see a pointer to that object: in this case, neither the object's final fields nor anything reachable in a dereference chain from its final fields will appear to be incorrectly initialized.

Let us now assume that the acquire and release do happen. As long as

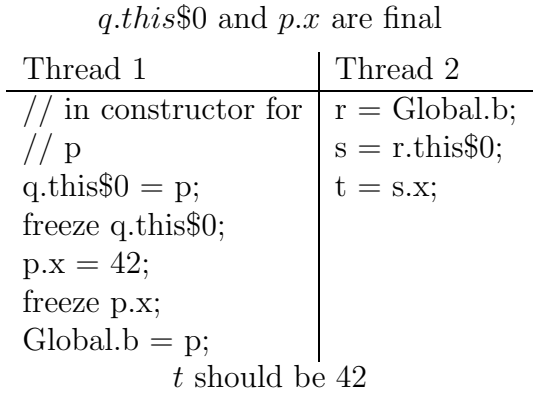

Figure 10: Guarantees Should be made via the Enclosing Object

these actions take place after object has been constructed (and there is no code motion around the end of the constructor), the diffs that the second processor acquires are guaranteed to reflect the correctly constructed object. This property makes implementation of final fields on a LRC-based DSM possible.

A Guarantees Made by Enclosing Objects

Consider Figure 10. In Thread 1, the inner object q is constructed inside the constructor for p. This allows a reference to p to be written before the freeze of $p.x$. The reference is now tainted, according to our semantics: any other thread reading it will not be guaranteed to see the correctly constructed values for $p.x$.

However, Thread 2 is guaranteed not to see the final fields of p until after p's constructor completes, because it can only see them through the correctly published variable *Global.b*. Therefore, it is not unreasonable to allow this thread to be guaranteed to see the correct value for $p.x$.

In general, we want to change the semantics so that a freeze for an object o is seen by a thread reading a final field $o.f$ if o is only read through a dereference chain starting at a reference that was written after the freeze of $o.f$.

The change to the semantics that allows this is simple. We state that when we read a reference to an object, the freezes seen by that read include

Initially, $a.ptr$ points to b, and $b.ptr$ points to a. $a.o$, $b.o$ and $obj.x$ are all final.

Thread 1	Thread 2
b.o = obj;	r1 = A.ptr;
freeze b.o;	r2 = r1.o;
a.o = obj;	r3 = r2.x
freeze a.o;	
obj.x = 42;	
freeze obj.x;	s1 = B.ptr;
A = a;	s2 = s1.o;
B = b;	s3 = s2.x;

Figure 11: Cyclic Definition Causes Problems

the freezes seen by the object dereferenced to reach that read. Again, we are changing the set freezeBeforeRead: the set of freezes seen at a read. If, at read r of an address a, a is field x of the object at address c, then freezeBeforeRead is:

$$\text{freezeBeforeRead}(r) = G \cup \{f \mid f \overset{hb}{\rightarrow} r\} \cup \text{freezesBeforeDereference}(t, c)$$

The set freezesBeforeDereference(t, c) above becomes the empty set if no object was dereferenced to read address a.

This formulation now makes the contents of freezesBeforeDereference(t, a) for some address a dependent on the freezesBeforeDereference(t, c) for an object c that has a reference to a. If c and a have references to each other, then we have a cyclic definition: freezesBeforeDereference(t, a) is defined in terms of freezesBeforeDereference(t, c), and freezesBeforeDereference(t, c) is defined in terms of freezesBeforeDereference(t, a). This is obviously undesirable.

To resolve this correctly, we state that the freezesBeforeDereference(t, a) set is the least fixed point solution to the equations that calculate both itself and freezesBeforeDereference(t, a); these sets will then consist of the smallest sets that satisfy these equations.

Consider Figure 11. Thread 2 correctly reads both A and B, and should, as a result, see the correct result for the final field $obj.x$, when it is read in both $r3$ and $s3$. This means that the freezesBeforeDereference(t, a) set and the freezesBeforeDereference(t, b) set must both contain the freeze of $obj.x$.

To calculate the freezesBeforeDereference(t, a) set, we must take into consideration all of the freezesBeforeDereference sets for the addresses through which a is accessed. It is accessed through a pointer for b when it is read at $s2$; we must therefore take into account the freezesBeforeDereference(t, b) set.

To calculate the freezesBeforeDereference(t, b) set, we must take into consideration all of the freezesBeforeDereference sets for the addresses through which b is accessed. It is accessed through a pointer for a when it is read at $r2$; we must therefore take into account the freezesBeforeDereference(t, a) set. This gives us a cycle.

To break this cycle, we calculate the least fixed point solution to the freezesBeforeDereference(t, a) equations. This should generate the appropriate sets without introducing additional, unintended guarantees.