Implementing Atomic Sections Using Lock Inference Final Report

By

Khilan Gudka <kg103@doc.ic.ac.uk>

Supervisor: Susan Eisenbach

June 23, 2007

Abstract

Atomicity is an important property for concurrent software, as it provides a stronger guarantee against errors caused by unanticipated thread interactions than race-freedom does. However, concurrency control in general is tricky to get right because current techniques are too low-level and error-prone. Consequently, a new software abstraction is gaining popularity to take care of concurrency control and the enforcing of atomicity properties, called atomic sections.

We propose an implementation of atomic sections using lock inference, a technique that infers the locks required for atomicity and inserts them transparently into the user's program. The main advantages of our approach are (1) we infer path expressions for objects accessed inside the atomic section that resolve at run-time to the actual objects being accessed, (2) we use multi-granularity locking for when a finite number of fine grained locks cannot be inferred and per-object locks otherwise and (3) we ensure freedom from deadlock without requiring the need to coarsen the locking granularity at compile-time.

Finally, we evaluate our approach by implementing a portion of the STMBench7 benchmark in a prototype object-oriented language called Single-step. We show that on some operations, our implementation can perform better than coarse locking.

I dedicate this to Lord Swaminarayan and my Guru HDH Pramukh Swami Maharaj.

Acknowledgements

I would firstly like to sincerely thank my supervisor Professor Susan Eisenbach for allowing me to take on such an interesting project. Her constant enthusiasm, patience and advice really saved me from failing what could have been a tremendously risky project.

I would also like to thank Dave Cunningham, a PhD student at DoC currently in his second year, for allowing me to work with him on this challenging problem. His helpful nature, ability to think very quickly and motto of "functionality first" inspired me in many ways.

I also thank my second supervisor Dr Paul Kelly for the very interesting discussion and his invaluable insights and suggestions for what I should do in this project.

Many thanks also to Dr Emil Lupu for his great advice over the years as my personal tutor.

Finally, I would like to thank the MEng group for making it a very fun four years.

Contents

Chapter 1

Introduction

1.1 Motivation

Concurrent software frequently exhibits erroneous behaviour due to unanticipated thread interactions. Traditionally, programmers try to avoid this by ensuring that their programs are race-free. However, race-freedom may not be sufficient to ensure the absence of such errors. As an example, consider the Java class for bank accounts given in Figure 1.1.

```
class Account {
    private int balance = 10; // initial balance
    synchronized int getBalance() {
        return balance; }
    synchronized void withdraw(int amount) {
        if(amount <= balance)
            balance = balance - amount; }
    synchronized void deposit(int amount) {
        balance = balance + amount; }
    synchronized void transferFrom(Account from, int amount) {
        int fromBalance = from.getBalance();
        if(amount <= fromBalance) {
            from.withdraw(amount);
            deposit(amount); }}
}
```

Figure 1.1: A Java class for bank accounts.

Each method within this class is *synchronised*, meaning that the lock on the receiver object must be acquired before it can be executed. This prevents multiple threads from simultaneously accessing the same account, which can lead to lost updates or race conditions. However, even though this program is race-free, it is still possible for errors to occur. Consider the example concurrent execution shown in Figure 1.2.

There are two accounts `a1` and `a2` both with an initial balance of £10. Two threads T1 and T2 are executing in parallel. T1 is transferring £10 from `a2` to `a1` and T2 is withdrawing £10 from `a2`. Figure 1.2(b) gives an example interleaving of their actions: T1 checks the balance of account `a2` and finds that it has sufficient funds for the transfer but is pre-empted shortly after. T2 then performs the withdrawal. T1 hasn't yet withdrawn from `a2`, therefore sufficient funds exist and the withdrawal completes successfully. The balance for `a2` is now £0. Meanwhile,

```
Account a1 = new Account();
Account a2 = new Account();

Thread T1:
    a1.transferFrom(a2,10);
Thread T2:
    a2.withdraw(10);
```

(a)

	Thread T1	Thread T2
1	a2.getBalance()	
2		a2.withdraw(10)
3	a2.withdraw(10)	
4	a1.deposit(10)	

(b)

Figure 1.2: A concurrent execution showing that thread interference can still occur if a program has no races.

T1 is resumed but remains unaware of this change still thinking the transfer can go ahead. It proceeds to withdraw £10 from `a2` (which has no effect) and deposits £10 for 'free' into `a1`.

This incorrect behaviour occurred because T2 was able to modify `a2` while the transfer was taking place. This was possible because although the invocations of `getBalance` and `withdraw` ensure mutually exclusive access to account `a2`, their composition does not. As a result, conflicting operations can be interleaved between them and lead to interference. Note that this program is race-free, as all methods are synchronised.

To assert that such interferences do not occur, we need a stronger property that ensures that threads cannot interleave conflicting operations while a block of code is executing, that is the *atomicity of code blocks*. A code block is said to be atomic if the result of any concurrent execution involving it is equivalent to the sequential case. This means that in the bank account example of Figure 1.2, the result of T1 and T2 executing concurrently would be the same as if T1 and T2 executed one after the other. Atomicity is a very powerful concept, as it enables us to reason about a program's behaviour at a simpler level. It abstracts away the interleavings of different threads (even though in reality, interleaving will still occur) enabling us to think about a program sequentially.

A number of techniques exist to verify atomicity of code blocks such as: type checking [16, 15], type inference [14], model checking [29], theorem proving [20] and run-time analysis [13, 52]. However, enforcing atomicity is still left to the programmer, usually using locks. Consider the steps required to make `transferFrom` atomic: Being a synchronised method, no other thread can access the current account until execution completes and the lock is released. However, to ensure that other threads cannot access account `from`, its lock must be acquired and held throughout. This prevents all but the transferring thread from accessing the two accounts involved making `transferFrom` atomic. The updated version is given in Figure 1.3.

```
synchronized void transferFrom(Account from, int amount) {
    synchronized(from) {
        int fromBalance = from.getBalance();
        if(amount <= fromBalance) {
            from.withdraw(amount);
            deposit(amount); }}}
```

Figure 1.3: Atomic version of `transferFrom` using locks.

However, the improved implementation has introduced another problem: the potential for deadlock. This could occur if transfers between two accounts are performed in both directions at the same time. To circumvent this, the programmer has to ensure that transfers do not occur in parallel. However, this is completely unrelated to the business logic and just adds unnecessary complexity to the program. This is only one of a handful of problems associated with locks; others include priority inversion [37], livelock [17] and convoying [10, 55].

Furthermore, it may not always be possible to ensure atomicity. Consider if instead we were invoking a method on an object that was an instance of some API class. Acquiring a lock on this object may not be sufficient if the method accesses other objects via instance fields, as we would need to acquire locks on those too in case they are accessible from other threads. However, accessing those fields would break encapsulation and might not even be possible if they are *private*. One solution would be for the class to provide a `Lock()` method that locks all its fields. However, this *breaks abstraction* and *reduces cohesion* because now the class has to provide operations that are not directly related to its purpose.

In summary, although atomicity allows us to more confidently assert the absence of errors due to thread interactions, programmers are still responsible for ensuring it. With current abstractions, this may not even be possible due to language features such as encapsulation. In fact, even if it is possible, modularity is broken thus increasing the complexity of code maintenance, while other problems such as deadlock are also increasingly likely.

1.1.1 A better abstraction for concurrency

This has led to a new abstraction that takes care of concurrency control and the enforcing of atomicity properties. *Atomic sections* [39] are blocks of code that execute as if in a single-step, with the details taken care of by the programming language. This is a significant improvement over current abstractions as atomic sections completely relieve the programmer from worrying about concurrency control. Figure 1.4 shows an implementation of `transferFrom` using atomic sections.

```
void transferFrom(Account from, int amount) {
    atomic {
        int fromBalance = from.getBalance();
        if(amount <= fromBalance) {
            from.withdraw(amount);
            deposit(amount); }}}
```

Figure 1.4: Implementing `transferFrom` using atomic sections.

As you can see, all we do is specify that we want the body of the method to be atomic. That's it! We don't have to worry about how this is done, the language just guarantees it. The language implementation that guarantees atomicity to be useful, must also ensure freedom from deadlock.

1.1.2 Implementing the semantics of atomicity

There are a number of proposed ways for implementing their semantics. The most popular of which is to execute them like database-style transactions. The idea here is to execute atomic sections on a private copy of memory and only commit the changes if another thread hasn't modified the locations that were accessed. If another thread has, then the would-be updates are discarded and the atomic section is re-executed. However, this can have a number of drawbacks

including: poor support for irreversible operations such as I/O, high run-time overheads and a large amount of wasted computation.

A more promising approach is to infer the locks necessary to ensure atomicity. This has a number of advantages: firstly, it doesn't limit expressiveness, secondly, it provides excellent performance in the common case of where there is no contention and thirdly, it has little run-time overhead. Initially, it may seem that we are re-inviting the problems associated with locks, however a combination of static analyses and run-time support are typically used to overcome them.

There are a number of challenges that this approach faces:

- Maximising concurrency
- Ensuring freedom from deadlock
- Minimising the number of inferred locks

To maximise concurrency, the locks should be fine grained and the number of locks should scale with the number of objects. But there are problems: the number of locks needed may be unbounded or unknown at compile-time. This project is about trying to solve these challenges.

The key ideas that we employ are:

- When fine grained locks can be inferred then do it.
- When a finite number of fine grained locks cannot be inferred then a single lock that subsumes the infinite/unknown number of locks should be inferred.
- A thread should not be allowed to wait on a lock if doing so would potentially cause a deadlock. To enable progress, it should be forced to give up all its locks and retry.

1.2 Contributions

The contributions made by this project are:

- We detail an implementation of atomic sections for object-oriented languages that:
  - Infers the locks necessary for atomicity with one lock per object where possible.
  - Requires no annotations from the programmer.
  - Is free from deadlock.
  - Has small run-time overhead (only for deadlock prevention).
- We describe a novel technique for statically inferring which objects are being accessed in an atomic section (Chapter 3). The advantage of our approach is that we do not coarsen object granularity.
- We provide an algorithm for inferring locks from object accesses. Our approach uses multi-granularity locking for when a finite number of fine grained locks cannot be inferred and per-object locks otherwise (Chapter 4).
- We give a novel algorithm for preventing deadlock at run-time. Our approach works when locks of differing granularities have been acquired (Chapter 5).
- We have built a prototype object-oriented language *Single-step* with features such as threading, inheritance and arrays to show feasibility (Chapter 6).

- We evaluate our approach by implementing a portion of the STMBench7 benchmark [23]. We show that on some operations, our implementation can perform better than coarse locking (Chapter 7).

Chapter 2

Background

2.1 Fundamentals of concurrency control using locks

Concurrent programs consist of multiple threads of execution. These threads are designed to work together, so they share memory. However, if care is not taken to control these shared accesses, they can lead to interference. Over the last thirty years, the primary way of doing this has been to use locks.

2.1.1 What is a lock?

A lock is a data structure that can be in one of two states: *acquired* and *free*. It also has two operations `lock()` and `unlock()` allowing it to be acquired and released respectively. A thread acquires the lock associated with a shared object before accessing it. If the lock is held by another thread, it must wait until that thread releases it. In this way, threads are prevented from performing conflicting operations and interfering with each other.

2.1.2 The complexities of using locks

In object-oriented languages, each object is typically protected by its own lock. However, in general the relationship between locks and objects is flexible. The number of objects protected by a lock is known as the *locking granularity*. This presents a tradeoff between simplicity and parallelism. A *coarse* granularity requires few locks, but permits less concurrency because threads are more likely to contend for the same lock. Conversely, *fine grained* locks protect fewer objects resulting in a larger number of locks but allow more accesses to proceed in parallel.

Programmers aim to get the best performance out of their software. However, the complexity of concurrency control can increase dramatically with the number of locks:

- Forgetting to acquire a lock re-invites the problem of interference (safety violation).
- Acquiring locks in the wrong order can lead to deadlock (progress violation).

Other problems that can occur due to locks include:

- **Priority inversion:** occurs when a high priority thread T_{high} is made to wait on a lower priority thread T_{low}. This is of particular concern in real-time systems or systems that use locks that busy-wait instead of blocking the thread. This is because T_{high} will be run in favour of T_{low} and the lock will never be released. Solutions include raising the priority of T_{low} to that of T_{high} (priority inheritance protocol) or the highest priority thread in the program (priority ceiling protocol) [37].

- **Convoying:** can occur in scenarios where multiple threads with similar behaviour are executing concurrently (e.g. worker threads in a web server). Each thread will be at a different stage in its work cycle. They will also be operating on shared data and thus will acquire and release locks as and when appropriate. Suppose one of the threads, T, currently possesses lock L and is pre-empted. While it is off the CPU, the other threads will continue to execute and effectively catch up with T up to the point where they need to acquire lock L to progress. Given that T is currently holding this lock, they will block. When T releases L, only one of these waiting threads will be allowed to continue, thus the effect of a convoy will be created as each waiting thread will be resumed one at a time and only after the previous waiting thread has released L [10, 55].

- **Livelock:** similar to deadlock in that no progress occurs, but where threads are not blocked. This may occur when locks that busy-wait are used.

These problems are hard to detect at compile-time and their impact at run-time can be disastrous [36, 38, 45].

2.2 Atomic sections

Given the problems associated with locks and that programmers face an inevitable turn towards concurrency [51], a lot of effort is being made to make concurrent programming easier.

In the introductory chapter, we mentioned an increasingly popular new abstraction called *atomic sections*. This is a declarative approach to concurrency control that allows programmers to specify that they want code to run atomically without worrying about how this is achieved. This is akin to SQL queries in relational databases.

There are a number of proposed ways for implementing atomic sections. The predominant approach is to execute them like database-style transactions. However, as we shall see, this has a number of shortcomings. These shortcomings are why we have chosen an alternative approach which instead infers the locks necessary for atomicity.

2.2.1 Database-style transactions

This technique is called *transactional memory* in the literature [30]. The key idea is to buffer memory updates during execution and only perform them on memory at the end if the locations that were accessed have not been modified by another thread. If they have, the would-be updates are discarded (termed *rollback*) and the transaction is re-executed [27].

To prevent interference while a transaction is updating memory (termed *commit*), it first acquires ownership of the locations to be updated. If a location is owned by another transaction, it will rollback and be re-executed. Ownership is released when the updates have been completed. In this way, atomicity is achieved without blocking threads [26].

Transactional memory provides a number of advantages over traditional blocking primitives such as locks:

- **No deadlock, priority inversion or convoying:** as there are no locks! Although, in theory a different form of priority inversion could occur if a high priority thread was rolled back due to an update made by a low priority thread.

- **More concurrency:** recall that with locks, the amount of concurrency possible is dependent on the locking granularity. Transactional memory provides the finest possible granularity (at the memory-word level), resulting in optimal parallelism. However, this comes at the cost of buffering overheads.

- **Automatic error handling:** Memory updates are automatically undone upon rollback, reducing the need for error handling code [24].
- **No starvation:** transactions are not held up waiting for blocked/non-terminating transactions, as they are allowed to proceed in parallel even if they perform conflicting operations.

However, these advantages rely on being able to rollback. This proves to be a huge limitation for atomic sections as it leads to the following drawbacks:

- **Irreversible operations:** Transactions cannot contain irreversible operations such as I/O because they cannot be rolled-back. Buffering is one proposed solution [25, 33], but requires rewriting I/O libraries and is not applicable in all situations. For example, consider a handshake with a remote server. Some implementations forbid irreversible actions using the type system [27], while others throw exceptions [46]. However, these are not practical in general.
- **Performance overhead:** Significant overhead is incurred due to buffering, validating that accessed locations have not been modified and committing [26].
- **Wasted computation:** A transaction that is later rolled-back is wasted computation. In one benchmark [34], tens of rollbacks occurred per second.

Performance has been the main focus of transactional memory research over the last few years. A lot of progress has been made, such as removing the requirement that transactions be non-blocking [7, 9, 28, 34, 48], coarsening the granularity from memory-word to object [42, 19, 18, 31, 2, 34, 28] and making application-specific the decision made when a transaction detects that its accessed locations have been modified [49].

However, current state-of-the-art implementations still require rollback for avoiding deadlock and starvation [34]. Furthermore, the problem of I/O remains.

2.2.2 Inferring locks

A promising alternative to transactions is to infer the locks necessary for atomicity. This is called *lock inference* in the literature [5, 41, 32]. In this approach, a compile-time analysis works out what objects are accessed in the atomic section and inserts `lock()` and `unlock()` statements for their corresponding locks. To prevent deadlock, locks are always acquired in the same order. If an ordering cannot be found, the program may be rejected or the locking granularity coarsened [41]. Figure 2.1 shows a Java example of lock inference.

```
atomic {                          synchronized(obj) {
    obj.n = 10; }    --apply analysis-->      obj.n = 10; }
```

Figure 2.1: A Java example of lock inference.

This technique has a number of advantages over transactional memory:

- **Does not limit expressiveness:** Lock inference does not require rollback. Therefore, irreversible operations such as I/O are okay.
- **Better performance when there is no contention:** Acquiring a free lock can be as cheap as setting a bit [3, 1, 53], whereas transactional memory performs buffering regardless of whether objects are contended [53].

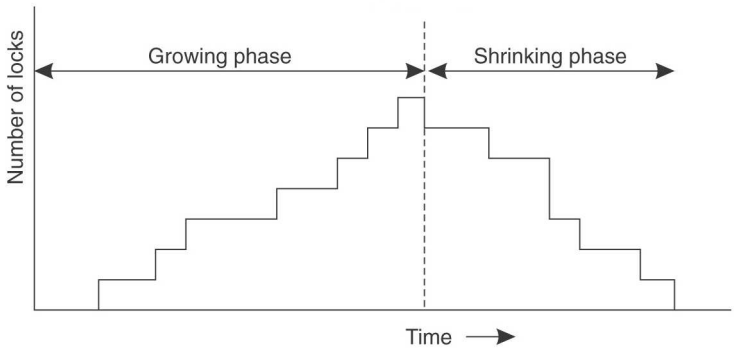


Figure 2.2: Locking policies that adhere to two-phase locking (2PL) will guarantee atomicity. Image adapted from http://rainbow.mimuw.edu.pl/SO/Wyklady-html/Tanenbaum/05-26.jpg.

- **Less run-time overhead:** Lock inference may be able to ensure freedom from deadlock at compile-time [41]. Then, the only run-time overheads are the `lock()` and `unlock()` operations. This can be extremely small if the lock is not contended (as mentioned above). When there is contention, techniques can be used to prevent suspending threads waiting for locks soon to be released [21].

In the following sections, we describe the steps involved in lock inference and the issues that complicate them. In summary, the steps are:

1. Inferring which objects are accessed
2. Mapping objects to locks
3. Ordering locks to prevent deadlock

2.2.2.1 Required restriction for atomicity

In this section we refer to a result from database theory [11]. However, it is important to note that atomicity has a different meaning there. We regard atomicity to mean 'in a single-step,' while in databases it means to execute completely or not at all [56]. These definitions are different. The former says that interleavings from threads do not affect the final result, while the latter says that execution does not stall half-way. The databases term for the semantics of atomic sections is *serialisability* [41]. All remaining uses of 'atomic' in this section should be read as 'serialisable.'

Database theory says that a sequence of `lock()` and `unlock()` operations (called a *locking policy*) will guarantee atomicity provided no locks are acquired after the first `unlock()`. This implies that in general, our inferred locking policy consist of a phase during which locks are acquired followed by a phase where locks are released. Such a restriction is called *two-phase locking* (2PL). The two phases are called *growing* and *shrinking* respectively. Figure 2.2 provides a graphical illustration.

A simple example would be a policy that acquired all necessary locks at the start of the atomic section and released them at the end. However, this will drastically impact concurrency, especially when objects are required for a short period of time and other atomic sections are waiting to access them. Additionally, atomic sections that require a large number of locks may have to wait a long time before they can start. In the worst case, they may never get to execute. To enable more parallelism, several variations of this basic policy exist [11]:

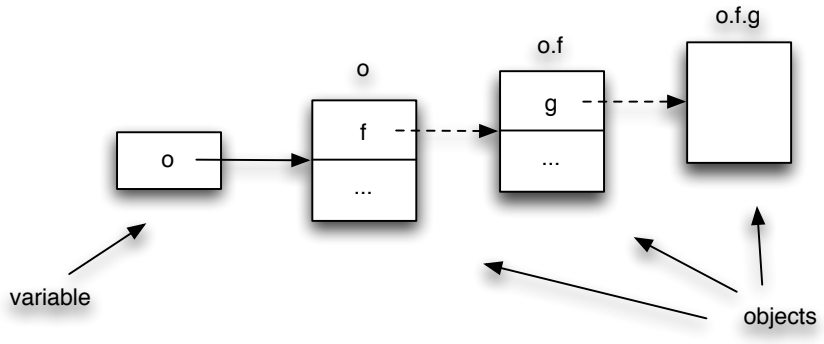


Figure 2.3: An example path `o.f.g`: `o` is a variable and `f` and `g` are field lookups.

- **Late locking:** Delays acquiring a lock until absolutely necessary and releases them all at the end. For example, each lock is acquired just before the object it protects is accessed for the first time. The advantage is that atomic sections spend less time waiting to start. However, late locking is complicated by the ordering on locks for avoiding deadlock. In the worst case, the resulting policy can be the same as the basic one.
- **Early unlocking:** Locks are acquired at the beginning of the atomic section, but are released when no longer required. This can achieve more parallelism than the basic policy, however it requires knowing when objects are no longer needed. This can be difficult at compile-time.
- **Late locking and early unlocking:** Locks are acquired only when they are needed, and once no more locks need to be acquired, they are released as they are no longer required. This policy can achieve more parallelism than the above two but it is complicated by their respective issues.

2.2.2.2 Inferring which objects are accessed

The first step in lock inference is to identify which objects are being accessed in the atomic section. This will have to cover all executions because at compile-time, we have no or little information about what paths will be taken through the program. Additionally, objects are allocated and deallocated at run-time, therefore there may not be a bound on how many are accessed. This is not good because we want the algorithm to terminate. We need to map the potentially infinite number of objects at run-time to a finite set at compile-time.

There are two approaches that can be taken here: one is to treat each occurrence of object creation `new` as representing an object [32]. They are sometimes called *abstract objects*. This ensures finiteness because the number of such statements is finite. However, all objects created with the same `new` are considered to be the same object. We shall see later that this has the disadvantage of resulting in coarse locks. An inference algorithm using this technique determines which of these abstract objects are pointed to by variables inside the atomic section. This is known as a *points-to* analysis [44].

There is a second better approach that is less conservative. In object-oriented languages, objects are referenced using path expressions. An example of a path is `o.f.g` where `o` is a variable and `f` and `g` are field lookups. Figure 2.3 shows a graphical representation. These paths resolve at run-time to the objects being accessed. Therefore, they provide a compile-time

```
atomic {
  me.account = you.account;
  me.account.balance = 0; }
```

(a)

```
atomic {
  me.account = you.account;
  khilan.account.balance = 0; }
```

(b)

Figure 2.4: Assignments (a) and aliasing (b) affect which objects are inferred.

representation which does not coalesce the real objects. This allows fine-grained locking. An inference algorithm using this technique determines which path expressions are referenced inside the atomic section. This is known as *path inference* [5, 41].

We now consider some of the issues that make inferring objects hard:

Assignments Variables can be assigned to one or more times in an atomic section. As a result, the object being referred to at an access may not be the same as where locks are acquired. This is more a problem for the path inference approach because the path itself represents the object. Consider the example in Figure 2.4(a). Here we assume locks are acquired at the start of the atomic section and released at the end. The object being updated in the second line is `me.account`. Therefore, we might naively conclude that `me.account` should be locked. However, `me.account` is assigned to `you.account` before the update. Hence, with respect to the start of the atomic section, the object being updated is actually `you.account`.

In [5], paths are rewritten as they are pushed up the atomic section, while [12] is conservative and restricts paths to variables and `final` fields.

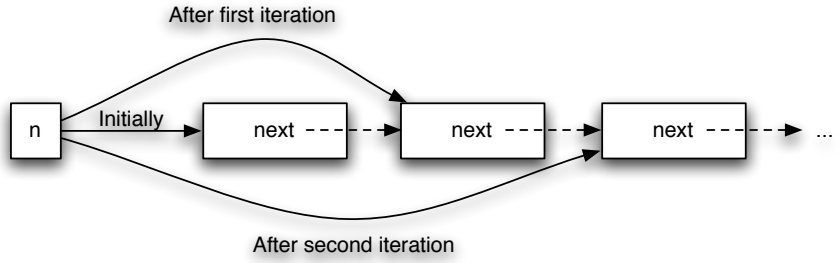
Aliasing Two variables are *aliases* if they refer to the same object. This complicates things further because an assignment to an object's field accessed through one alias may change the object being referred to when an access involving the other one occurs. For example, in Figure 2.4(b) `me` and `khilan` are aliases. Consequently, `you.account`'s balance is being updated in the second line. Aliases are usually computed using a points-to analysis. However, if this information is not available, all we can do is be conservative and assume that `me`, `you` and `khilan` could all alias each other. This is because our lock inference analysis must be correct for all executions. The approaches in the literature all have conservative alias analyses.

Autolocker [41] assumes that all non-global paths of the same type are aliases, while [5] treats the receivers of paths that have the same final field as possible aliases. For example, potential aliases in paths `x.f.g.s.g` and `q.g` are: `x.f`, `x.f.g.s` and `q`. Finally, the approach in [32] uses coarse locks when aliasing makes it unclear which objects are being accessed.

Recursive data-structures Recursive data-structures such as linked lists can be grown or shrunk as needed. This has the advantages that firstly, space is not wasted and secondly, their size is not bounded. However, this latter characteristic creates a problem. Consider the code in Figure 2.5(a) that traverses a linked list. We cannot (in general) infer at compile-time how many nodes will be accessed. To ensure our analysis is correct and covers all cases, we can only assume that this number is infinite. This is okay if we are inferring abstract objects because these are finite, but the paths approach generates an infinite set of paths! Assume we are locking at the start of the atomic section. Then, the set of objects accessed would be {`n`,

```
Node n = list.head;
atomic {
    while(n.next != null)
        n = n.next; }
```

(a)



(b)

Figure 2.5: Traversing a linked list.

`n.next`, `n.next.next`, ...}. Consider the linked list in Figure 2.5(b) to understand why. The diagram shows the node pointed to by `n` after each iteration. The key thing to note is that the object `n` points to after an iteration is `n.next` with respect to what it previously pointed to. As we are locking at the start of the atomic section, we want all paths to be in terms of what `n` points to there. This is the aforementioned set. But how do we represent such infinite sets? One proposed solution is to use regular expressions [4, 5]. For example, the set above can be written as `n(.next)*`.

2.2.2.3 Mapping objects to locks

Having inferred which objects are being accessed in the atomic section, the next step is to determine the locks that protect them. There are two options here:

1. Let the programmer specify which locks protect what objects (using annotations) [41], or
2. Let the implementation do it [32, 5]

The first approach has the advantage that it gives developers more control over performance as they can control the granularity of lock. However, it adds the overhead of annotations and also relies on the programmer using them correctly. The second approach is the more commonly used one therefore we shall continue with it only.

If we took the points-to approach and inferred abstract objects, generating the locks to protect them is easy: assign a fresh one to each object. However, notice that because each abstract object corresponds to several run-time objects, all of these will require the same lock. This means locks are coarse, which leads to significantly less concurrency.

What we really want is that, locks should be fine-grained and the number of locks should scale with the number of run-time objects. This is not possible with the points-to approach because it coalesces objects. However, it *is* possible with paths.

In object-oriented languages, each object is typically protected by its own lock. This lock may be stored in one of its fields or the run-time may keep a mapping. This implies that an object determines its lock. Furthermore, recall that paths resolve to objects at run-time. Therefore, the path is all we need! In fact, this is *exactly* how fine-grained locking is achieved in these languages. Figure 2.6 shows two examples to illustrate.

Java:	C#:
synchronized(o.f.g) { }	**lock**(o.f.g) { }

Figure 2.6: Examples of how objects are locked in Java and C#.

Recursive data-structures Recall that regular expressions can be used to represent paths accessed while traversing a recursive data-structure. But how do we lock them? We could expand the regular expression but this is no good as paths can be infinitely long. This amounts to acquiring an infinite number of locks, which is impossible. One proposed solution is to instead lock 'something' that the objects have in common. This could be a class, another object, place of construction, etc. Locking these kinds of things has the effect of locking all objects associated with them. This is called *multi-granularity locking*. For example, locking a class effectively locks all its instances. So one way of locking `n(.next)*` is to lock the `Node` class. Note that there is an issue of granularity because locking the `Node` class locks *all* its instances, including those which are not part of the linked list we are traversing. A better approach is to lock another object, such as the `list` object. This requires an ownership relation between objects [6, 8].

2.2.2.4 Order of acquiring and releasing locks

We have seen that there are two fundamental requirements for ensuring atomicity using locks:

1. Objects must be locked before being accessed
2. The locking policy must be two-phase

However, there are restrictions on the order locks are acquired, otherwise problems can occur:

Deadlock If two or more threads try to acquire the same locks but in different orders, it can lead to a state where they wait for each other called *deadlock*. Consider the example given in Figure 2.7. Two threads T1 and T2 try to lock the same objects `o1` and `o2` but in different orders. Figure 2.7(b) shows an example execution that leads to deadlock. Initially no locks have been acquired. T1 locks `o1` but is then pre-empted. T2 successfully locks `o2` but then blocks waiting for T1 to unlock `o1`. When T1 is resumed it finds the lock on `o2` is not free. Therefore, it blocks waiting for T2 to release it. Note that we have T1 waiting for T2 and T2 waiting for T1. The two threads are deadlocked. This can be visualised as shown in Figure 2.7(c). Nodes represent threads, and an edge T1 → T2 means that T1 is waiting for T2 to release a lock. This is called a *waits-for graph*. Deadlock corresponds to a cycle in the graph - in this case T1 → T2 → T1.

One way to avoid deadlock is to ensure locks are always acquired in the same order. When the number of locks is finite, it is possible to determine this ordering at compile-time. This is because all locks to be acquired are known.

However, if we infer paths, this isn't possible without being overly conservative. For example, [41] imposes an ordering at compile-time by treating all paths with the same type as aliases.

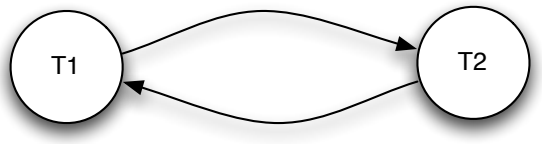
```
Thread T1:
  synchronized(o1) {
      synchronized(o2) {
          o1.i = o2.i; }}

Thread T2:
  synchronized(o2) {
      synchronized(o1) {
          o1.i = o2.i; }}
```

(a)

	T1	T2
1	lock o1	
2		lock o2
3		lock o1
4	lock o2	*waiting*
5	*waiting*	*waiting*

(b)



(c)

Figure 2.7: A Java example of deadlock. (a) shows a simple program where two threads T1 and T2 try to acquire the same locks but in different orders. (b) shows an example execution that leads to deadlock. (c) gives a graphical representation of deadlock using a waits-for graph. The nodes are threads and an edge T1 → T2 means that T1 is waiting for a lock held by T2. A deadlock is shown by the cycle T1 → T2 → T1.

```
Object a[n] = {path_1 ... path_n};
sort(a);
lock(a[0]) ; ... ; lock(a[n]);
```

(a)

```
boolean locked = false;

while(!locked) {
   // lock in address order
   Object a[n] = {path_1 ... path_n};
   sort(a)
   lock(a[0]) ; ... ; lock(a[n]);

   // check that paths still point to the same objects
   Object a_after[n] = {path_1 ... path_n};
   if(!a.equals(a_after))
      unlock(a[0]) ; ... ; unlock(a[n]);
   else
      locked = true; }
```

(b)

Figure 2.8: Example code suggested in [5] for avoiding deadlock at run-time when locks are fine-grained. Note that (a) may lead to the wrong locks being acquired if objects are not locked in path prefix order. This may occur because prefixes of paths are not guaranteed to be ordered by address. (b) checks that paths point to the same objects.

This has the side-effect that because of other dependencies on the locking order created by assignments and the fact they use late locking, it is highly likely deadlock freedom will not be achievable. Their approach either rejects the program or uses coarser locks in this case.

To obtain an accurate ordering for such approaches, we need to use the (unique) address of the object [5]. This means the order locks are acquired cannot be determined until run-time. Furthermore, all objects must be known at the start of the atomic section otherwise we cannot sort them. Example code for deadlock free lock acquisition at run-time is shown in Figure 2.8(a) [5].

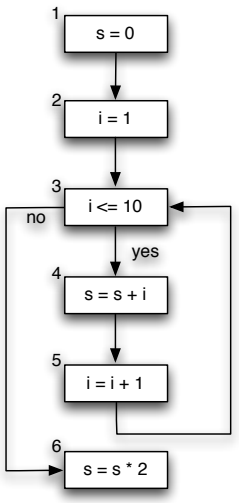
Wrong locks being acquired A prefix of a path p is any portion of p starting from the left. For example, the prefixes of `o.f.g` are `o` and `o.f`. Note that paths are not prefixes of themselves.

The object referenced by a path can change if any of its prefixes are assigned to. This is of significance because if after locking `o.f.g`, another thread assigns to `o.f`, all subsequent accesses of `o.f.g` may not correspond to the object locked. Hence, we would have acquired the wrong lock. To prevent this, we must additionally lock all prefixes in increasing order of length. This seems intuitive because accessing `o.f.g` requires accessing field `f` of `o` and field `g` of `o.f`.

However, this introduces the possibility of deadlock again, as locks are not necessarily acquired in address order. We can overcome this by detecting when deadlock occurs, release locks

```
s = 0;
i = 1;
while(i <= 10) {
      s = s + i;
      i = i + 1; }
s = s * 2;
```

(a)



(b)

Figure 2.9: (a) is a program that calculates double the sum of 1 to 10. (b) is its control flow graph.

already acquired and retry. However, this is only possible if locks are acquired at the start of the atomic section, otherwise we would need rollback support. Nevertheless, it has the advantage that run-time costs are lower because no sorting is required.

In Figure 2.8(a), locks are acquired in address order, so it is possible for the wrong locks to be acquired. One way this can be overcome is to check that paths refer to the same objects after all locks have been acquired. This is shown in Figure 2.8(b).

2.3 Program analysis

Program analysis allows us to approximate run-time behaviours of programs at compile-time. It is used in lock inference to determine which objects are being accessed and what their corresponding locks are. This section provides a very brief overview of relevant concepts. For a detailed account, please refer to [43].

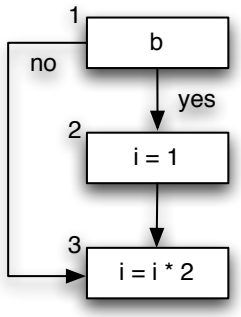
2.3.1 Data flow analysis

The approach to program analysis that is of relevance to this project is *data flow analysis*. In this technique, it is customary to think of a program as a graph: the nodes are simple statements or expressions and the edges describe how control might pass from one simple statement to another. This is called a *control flow graph*. Figure 2.9(b) shows an example graph for a program that calculates double the sum of 1 to 10. Nodes are labeled uniquely. Notice the two edges coming out of the while condition corresponding to where flow goes to when it is true or false. At compile-time, we typically cannot determine exactly which edges will be followed, therefore we must consider all of them. 'If' statements are similar.

In a nutshell, data flow analysis works by pushing sets of 'things' through the graph until they stabilise. When a node receives data from immediately preceding nodes, called *entry information*, it applies the effect of its statement and passes the resulting set, called *exit information*, to its immediate successors. If a node has multiple predecessors, like 3 in Figure 2.9(b),

```
int i;
if(b)
    i = 1;
i = i * 2;
```

(a)



(b)

Figure 2.10: Simple program to demonstrate the difference between may and must analyses.

the incoming data are first combined using set union or intersection depending on the type of analysis.

There are broadly four types of data flow analyses depending on whether (a) we want information that is valid along all paths from the start of the program to a node or only some of them and (b) we want to know something about code before nodes or after them. For (a), consider the example given in Figure 2.10 and suppose we want to know whether variable `i` has been initialised before reaching 3. The start node of the program is 1. There are two paths from 1 to 3: $1 \rightarrow 2 \rightarrow 3$ and $1 \rightarrow 3$. `i` is initialised along the first but not the second. Therefore, we deduce `i` may not be initialised. This is called a *must analysis* because we only assert `i` is initialised if all paths from 1 to 3 initialise `i`. The key point here is that we consider all paths. In this type of analysis, data from immediate predecessors are combined using set intersection. If instead we wish to determine what value `i` might have, we union the result of each path. This is called a *may analysis*. In this, data from immediate predecessors are combined using set union.

Sometimes we will want to calculate information about paths reaching a node and other times about paths leaving a node. For example, determining if `i` is initialised at 3 in Figure 2.10 requires looking at paths reaching it. On the other hand, determining what objects are accessed in an atomic section, requires looking at paths leaving the start of the section. In the former, data is passed from the start of the program downwards. This is called a *forwards analysis*. In the latter, data is passed from the end of the program upwards. This is called a *backwards analysis*. Note that the predecessor of a node in a forwards analysis will be the successor of a node in a backwards analysis. Do not confuse this with control flow, we are talking about *data flow*.

This leads to the following four types of data flow analyses:

- Forwards, must
- Forwards, may
- Backwards, must
- Backwards, may

Entry and exit information are commonly referred to as the entry and exit sets of the node. In a forwards analysis, the entry sets give us the final information we want, while in a backwards

analysis it is the exit sets. Note that while the notion of predecessor and successor get swapped around, entry and exit sets do not. That is, in a backwards analysis the exit set is calculated by combining the results of its predecessors and the entry set is calculated by pushing this data 'through' the node. This implies that the notion of 'entry' and 'exit' remain consistent with the control flow graph.

To calculate the final entry and exit sets, an iterative algorithm is used. Figure 2.11 gives it in pseudo-code.

```
while(entry and exit sets change) {
    for each node n {
      // calculate new entry set
      entry'(n) = { };
      for each predecessor node p of n
          entry'(n) = entry'(n) + exit(p);

      // calculate new exit set
      exit'(n) = f_n(entry(n)); } }
```

Figure 2.11: Iterative algorithm for computing entry and exit sets.

Here we use '′' to distinguish between the current and previous iterations. The function f_n applies the effect of n's statement to its previous entry set. It is known as a *transfer function*. This function will typically kill some incoming data and add any additional information created by this node. These are called its *kill* and *gen* sets respectively. One can express f_n in terms of these sets as follows:

$$f_n(d) = (d \setminus kill_n(d)) \cup gen_n(d)$$

The algorithm terminates when every entry and exit set does not change between iterations. This is referred to as having reached a *fixed point*.

2.3.1.1 Intraprocedural versus interprocedural

So far we have only looked at data flow analysis in a single method. This is known as *intraprocedural data flow analysis*. Lock inference also needs to determine object accesses in methods called from atomic sections because these need to be protected too. When we consider data flow across methods, this is called *interprocedural data flow analysis*.

The key idea is that data to a node n that performs a method call (called a *caller node*) flows to the start of the corresponding method m and exit information from m's last node flows back to n. Calculating the entry set is the same as in the intraprocedural case, but the exit set is now calculated from both the entry set and the information flowing back from m. Figure 2.12 gives a graphical description. Here, d is the entry information for n and d' is the data flowing back from m.

Interprocedural analysis introduces two new functions $f_n^1(d)$ and $f_n^2(d, d')$. f_n^1 modifies the incoming data as required for passing to the method. This might include removing information about local variables and renaming arguments to the corresponding formal parameter names. f_n^2 modifies the data flowing back from the method as appropriate for returning from it and combines it with the entry information for n. The former might include renaming formal parameters. The entry information may also be modified based on the returning data.

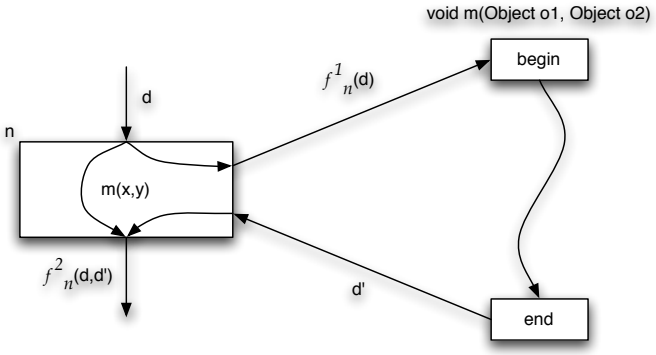
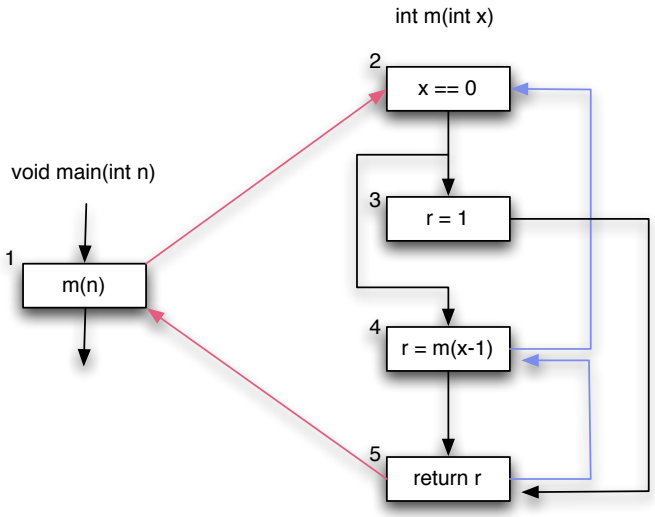


Figure 2.12: Interprocedural analysis.

```
int m(int x) {
    int r;
    if(x == 0)
        r = 1;
    else
        r = m(x-1);
    return r;
}

void main(int n) {
    m(n);
}
```

(a)



(b)

Figure 2.13: Problem of valid paths.

2.3.1.2 Valid paths through the program

Armed with these two functions, we could carry out the interprocedural analysis like in the intraprocedural case. However, this turns out to be rather naive because it allows data to flow along paths that do not correspond to a run of the program. Consider the example program in Figure 2.13. At run-time, there will typically be a stack of method calls that are waiting to be returned to. Execution always returns to the most recent one first, i.e. the one at the top of stack. However, notice that in Figure 2.13 there is nothing stopping the analysis from considering the path $1 \rightarrow 2 \rightarrow 4 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 1$ corresponding to calling m twice but returning only once. This is a problem because it can lead to incorrect solutions.

We can restrict consideration to valid paths by associating call stacks with data. This will typically be a string of labels corresponding to method call nodes with the most recent on the right. This string is called a *calling context*. Now, data is a set of functions mapping contexts to the data flow information for that context. Note that we may have several contexts at a time because there may be several ways of reaching a method from the start of the program. The two key things that make using contexts work are:

- Before passing data to a method, append the caller node's label to all contexts. This indicates that it is now the most recent method call.
- For all contexts passed back by the method, only keep those whose right most label is this node. This ensures that we don't pass data back along the wrong paths. To indicate we have returned from the call, remove this right most label.

An analysis that uses contexts is called *context-sensitive*.

2.4 Summary

In this chapter, we looked at background concepts relevant to this project.

Chapter 3

Inferring object accesses in atomic sections

The first step in lock inference is to determine the objects being accessed in the atomic section, as only then can we determine the locks that we need to acquire to protect them. The ultimate goal is to infer as fine-grained locks as possible so that we can permit higher levels of parallelism. However, the granularity of locks that our algorithm infers will depend directly on the granularity with which we infer object accesses. Some existing approaches treat all objects created at the same construction site as the same object. However, the effect of this is that each of these objects will be protected by the same lock! [5, 4] have suggested inferring path expressions such as `x.f` (as used with constructs such as `synchronized { }`) instead because they resolve at run-time to the actual object being accessed.

In this chapter, we present an algorithm that does the following: Given an atomic section, it outputs a finite state automaton describing path expressions for objects accessed. We call this a *paths graph*. As an example, Figure 3.1 shows the list traversal algorithm from Figure 2.5(a) and its corresponding graph.

```
Node n = list.head;
atomic {
    while(n.next != null)
        n = n.next; }
```

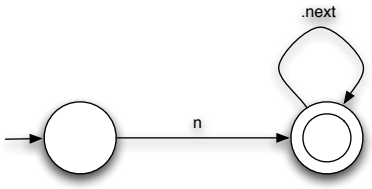


Figure 3.1: An example of our algorithm inferring object accesses inside a loop.

The ideas behind our algorithm came about after a first attempt using regular expressions. This provided some useful insights, which we consider. However, first we describe our language.

3.1 Our language

We base our work on object-oriented languages because they present a number of challenging problems for atomic sections. Our prototype language is called *Single-step* (see Appendix A).

To simplify the analysis, we first transform programs so all statements in the control flow graph are in one of the following forms (x, x_0,...,x_n, y are variables; f is a field name):

- A copy statement `x = y`

- A null statement `x = null`
- An object creation statement `x = new C` (where `C` is a class name)
  - Non-default constructors are converted to method calls:
    `x = new C(1)` $\longrightarrow$ `x = new C; C(x,1)`
- A load statement `x = y.f`
- A store statement `x.f = y`
- A method call `m(`x_0`,`x_1`,...,`x_n`)` (x_0 is receiver and x_n is result)
  - The result variable is passed by reference as the last parameter. Assigning to it in `m` causes its value to be updated in the calling method.
- A return statement `return`
- An array load statement `x = a[y]`
- An array store statement `a[x] = y`

This simplification is achieved using temporary variables.

3.2 First attempt: regular expressions

The key challenge we face is inferring object accesses inside loops. This is because we do not know how many times the loop will execute, so we have to consider that any number is possible. Although, this means we cannot in general put a bound on the number of objects being accessed in it. [5, 4] suggest regular expressions as a way to represent such accesses. For example, the accesses in the above program could be represented as `n(.next)*`. However, nobody has come up with a general algorithm for doing this.

This formed the starting point for the project: to find an algorithm for inferring regular expressions describing accesses inside loops. However, we quickly found that even simple examples would generate long and complicated expressions that could not be minimised to the form we were expecting. Consider the example in Figure 3.2.

```
while(b1) {
    if(b2)
        x = x.f;
    else
        x = x.g; }
```

Figure 3.2: Simple examples cause problems for regular expressions.

This program conditionally performs field lookups on two fields `f` and `g` starting from the object pointed to by `x`. Assume the heap before the while loop is executed is as shown in Figure 3.3. The figure also gives the path expressions for all objects reachable from `x` via one or more field lookups. Each vertical column of objects corresponds to what `x` may point to after 0, 1, 2 or more iterations (in the diagram it is up to 2) respectively.

A reasonable regular expression summarising these objects is `x(.f*.g*)*`. However, our analysis would instead generate `x(.f*.g*(.f.g)*(.g.f)*)*`. Attempting to minimise this using standard algorithms did not prove successful. Minimising regular expressions is a hard

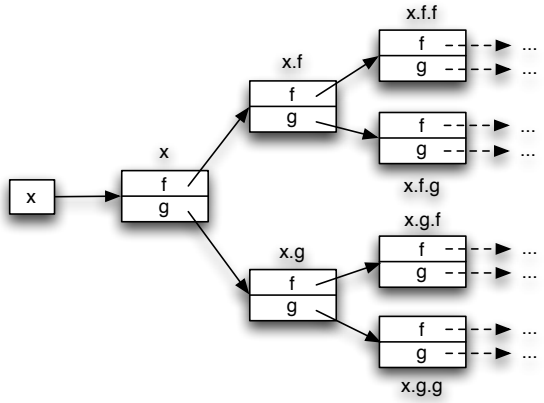


Figure 3.3: Heap.

problem because at each step there can be a large number of applicable simplifications [50]. With more complex programs, the expressions get worse and simplifying them becomes impractical.

Regular expressions are equivalent to non-deterministic finite state automata (NFA) with textbook algorithms around for converting between the two [35]. The advantage of the latter is it is easier to manipulate because it is a graph. One alternative then is to convert the regular expression to a NFA, simplify it and then convert back to a regular expression again. However, we found this to often lead to more complicated regular expressions than we first started with [47].

3.3 Second attempt: graph-based approach

We therefore tried generating NFAs directly and found it to work very well. For example, our algorithm produces the following for the program in Figure 3.2:

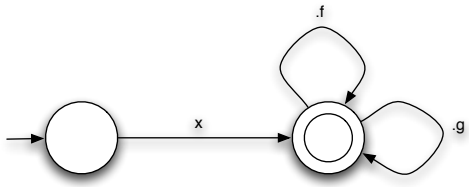


Figure 3.4: Paths graph for the program in Figure 3.2.

This approach has not been considered by existing lock inference approaches and is surprisingly simple. We now describe it in more detail. Figure 3.5 shows a diagrammatic overview.

3.3.1 Paths graph

A paths graph describes path expressions for objects accessed. For example, the graph in Figure 3.4 describes the set of paths {`x, x.f, x.g, x.f.f, x.f.g, ...`}. These are read from the graph by following transitions from the starting state to accepting states. Note that because we must infer all prefixes of paths too (see §2.2.2.4), all non-starting states are accepting.

Path graphs are represented as a set of transitions. There are two types of transition: *variable transitions* and *field transitions*. Variable transitions correspond to the left-most variables in

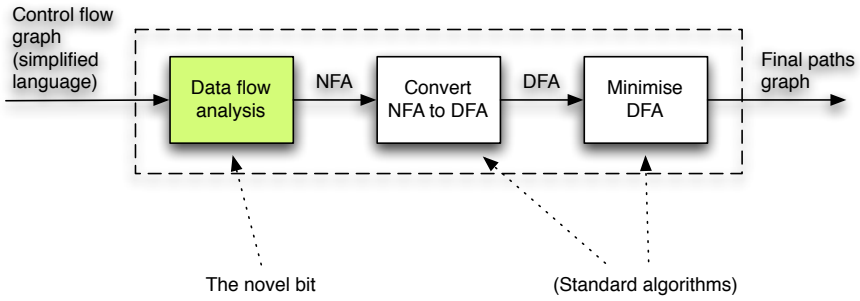


Figure 3.5: An overview of our algorithm for inferring object accesses inside atomic sections.

paths (e.g. $\xrightarrow{x}$ in Figure 3.4), therefore they always transition from the starting state s_0. Field transitions correspond to field lookups (e.g. $\xrightarrow{.f}$ in Figure 3.4). The start state of a paths graph is never accepting because paths have non-zero length.

3.3.2 Data flow analysis

The algorithm uses a context-sensitive interprocedural data flow analysis (see §2.3) to generate a paths graph at every statement in the program (at this stage, the graphs are NFAs).

The initial (empty) paths graph is propagated upwards from the bottom of the atomic section (backwards analysis), applying transfer functions and unioning graphs from predecessors (may analysis) as it passes through each statement in its control flow graph. This continues until all graphs stabilise. The final result is the computed paths graph at the top of the atomic section.

3.3.2.1 State names

The naming of states is crucial as it makes detecting looping accesses of the form shown in Figure 3.4 very easy. It will become clear later when we look at load statements why this is, but for now it is enough to say that states are named with the label of the program statement that generated them [54]. Although, these names are only of significance during data flow analysis. After this, they serve no purpose other than to distinguish between states.

3.3.2.2 Transfer functions

Transfer functions describe the effect of a statement on the paths graph (see §2.3.1). These are the key components of the data flow analysis. We now give the transfer function for each type of statement. **Keep in mind that this is a backwards analysis, therefore data is flowing from bottom to top!**

Copy statement A copy statement `x = y` makes `x` point to whatever object `y` points to. Figure 3.6 illustrates this graphically. The key issue here is, `x` may point to a different object before the statement than it does after. Consequently, paths starting with `x` may refer to different objects before the statement than they do after.

However, the graph must maintain which objects are being accessed, so to fix this we use the fact that whatever `x` points to just after the statement is the same as what `y` points to just before the statement. We essentially rename all paths starting with `x` to start with `y`. The kill and gen sets are:

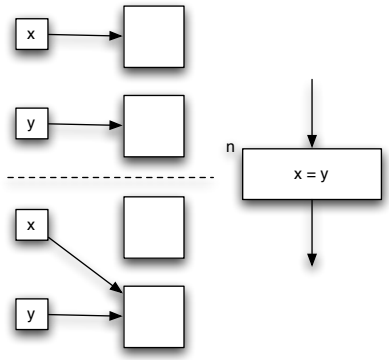


Figure 3.6: Copy statement.

$\text{kill}_n(\text{G}) = \{\ s_0 \xrightarrow{x} n' \mid s_0 \xrightarrow{x} n' \text{ is in G } \}$
$\text{gen}_n(\text{G}) = \{\ s_0 \xrightarrow{y} n' \mid s_0 \xrightarrow{x} n' \text{ is in G } \}$

There is a special case when `y` is `null`: we do not add any edges. $\text{gen}_n(\text{G})$ now becomes:

$\text{gen}_n(\text{G}) = \{\ s_0 \xrightarrow{y} n' \mid s_0 \xrightarrow{x} n' \text{ is in G and y is not } \texttt{null}\}$

Load statement A load statement `x = y.f` makes `x` point to the object referenced by `y.f`. Figure 3.7 gives a graphical illustration.

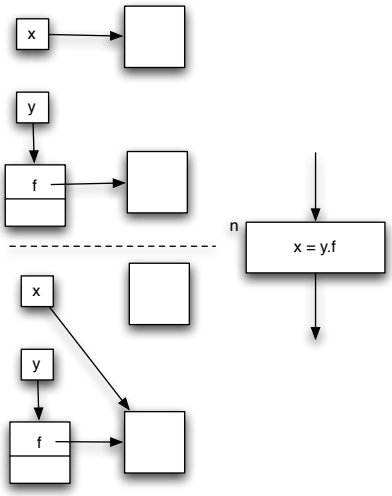


Figure 3.7: Load statement.

The object pointed to by `y` is being accessed, therefore we record this in the graph by adding a variable edge from the starting state. But what is the destination state? It turns out that the choice we make is crucial for detecting looping accesses. If this is a fresh state then we risk generating an infinite number of them when inside a loop. The solution lies in only generating at most one state per program statement [54]. A natural candidate then is a state named with the program statement's label. We assume that two or more states with the same name are the same state.

The reason why such a naming scheme works is because of the transformation required for load statements. Based on the transfer function for copy statements, we require rewriting paths starting with `x` to start with `y.f`. This 'rewrite' is done by replacing all edges $s_0 \xrightarrow{x} n'$ by $s_0 \xrightarrow{y} n \xrightarrow{.f} n'$.

We can now explain why it works: if `y` was `x` (so instead we had `x = x.f`), then all edges $s_0 \xrightarrow{x} n'$ would be replace by $s_0 \xrightarrow{x} n \xrightarrow{.f} n'$. If inside a loop, $s_0 \xrightarrow{x} n$ will have already been generated by this statement in a previous iteration. Therefore, this will get rewritten to $s_0 \xrightarrow{x} n \xrightarrow{.f} n$ creating the necessary loop! This is only possible because we use the same state each time when generating the variable edge! The kill and gen sets are:

$\text{kill}_n(\text{G}) = \{\ s_0 \xrightarrow{x} n' \mid s_0 \xrightarrow{x} n' \text{ is in G}\ \}$
$\text{gen}_n(\text{G}) = \{\ s_0 \xrightarrow{y} n\ \} \cup \{\ n \xrightarrow{.f} n' \mid s_0 \xrightarrow{x} n' \text{ is in G}\ \}$

Store statement A store statement of the form `x.f = y` makes the `f` field of the object pointed to by `x` refer to the same object as `y`. Figure 3.8 shows this graphically.

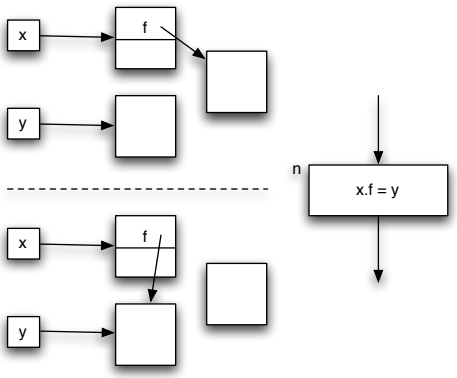


Figure 3.8: Store statement.

Again we require a graph transformation to make sure paths refer to the same objects before the statement as they do after it. We might assume all we need to do is rewrite paths starting with `x.f` to start with `y`. However, it is not as easy as this.

Consider if `z` and `x` are aliases. Then, by assigning to `x.f` we are also assigning to `z.f`. Hence, to ensure paths starting with `z.f` refer to the same objects they also need to be rewritten to start with `y`. However, if we are not sure whether `z` and `x` are aliases, we can only assume that both are possible: in the case where they are aliases all paths starting with `z.f` need to be rewritten to start with `y`. When they are not aliases, paths starting with `z.f` are not affected by this statement and should not be rewritten. Figure 3.9 shows these two cases graphically.

Our algorithm does not infer what aliases what (we consider having better alias information an optimisation rather than being crucial to the approach). Therefore, we assume any potential alias for `x` may or may not be an alias. We take the approach from [5] that considers the largest prefixes of paths that have the same final field as potential aliases. For example, for the two paths `p.f` and `p'.f`, `p` and `p'` may be aliases.

Hence, this statement may affect all paths starting with `p.f` where `p` is an arbitrary path. We add an edge $s_0 \xrightarrow{y} n''$, whenever we have $s_0 \xrightarrow{p} n' \xrightarrow{.f} n''$. The result of this is, original paths are kept in case `p` and `x` are not aliases and rewritten versions of the paths also exist in case

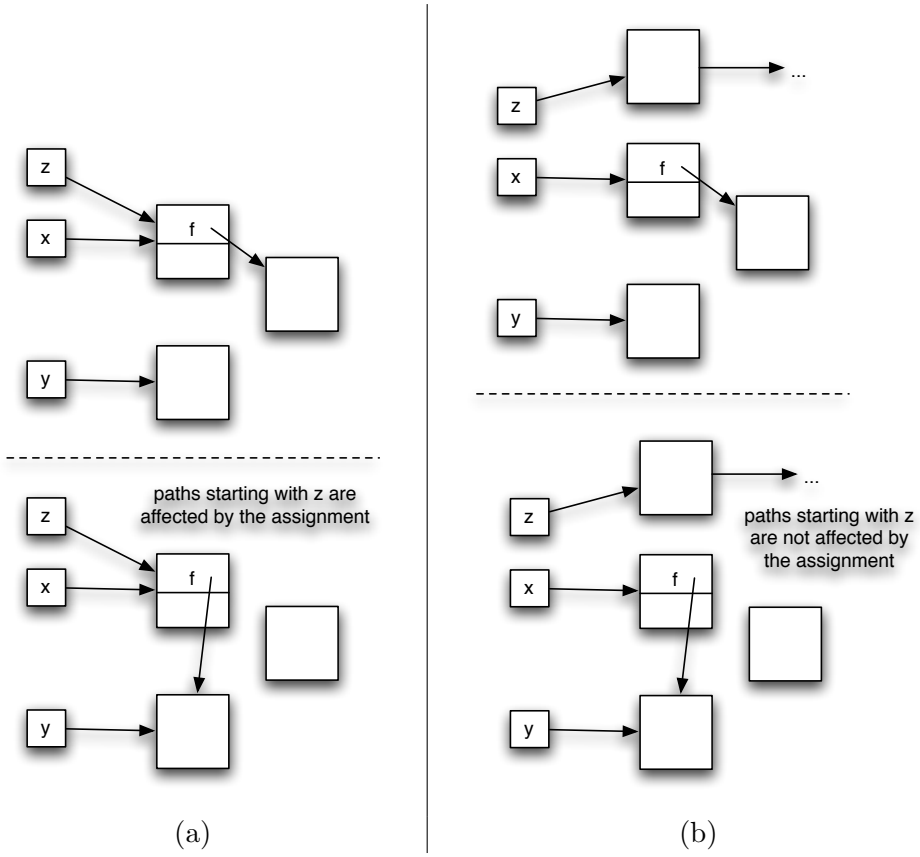


Figure 3.9: The effects of a store statement when variables `z` and `x` are aliases (a) and when they aren't (b).

they are. Notice that by adding the edge $s_0 \xrightarrow{y} n''$, field lookups after `.f` automatically become field lookups of `y`.

This store statement also accesses the object pointed by `x`, therefore we add $s_0 \xrightarrow{x} n$ (as explained in the transfer function for load statements). The kill and gen sets are:

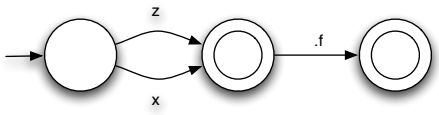
$\text{kill}_n(\text{G}) = \{\ \}$
$\text{Gen}_n(\text{G}) = \{\ s_0 \xrightarrow{x} n\ \} \cup \{\ s_0 \xrightarrow{y} n'' \mid n' \xrightarrow{.f} n''$ is in G $\}$

There is one exception when we know we can rewrite the path—if the path starts with `x.f`! This is because there is no question of `x` possibly not being an alias of itself! In this case, edges of the form $s_0 \xrightarrow{x} n' \xrightarrow{.f} n''$ are replaced with $s_0 \xrightarrow{y} n''$.

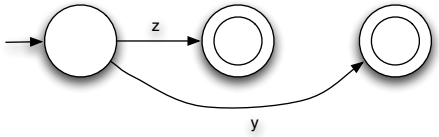
However, there is a further complication. Consider if we had the paths graph as shown in Figure 3.10(a). It describes the paths accessed are: `x`, `z`, `x.f` and `z.f`. Suppose we remove $s_0 \xrightarrow{x} n' \xrightarrow{.f} n''$ and add $s_0 \xrightarrow{y} n''$ resulting in the paths graph given in Figure 3.10(b). The paths described are { `z`, `y` }—we have lost `z.f`!

The problem is there are other potentially unaffected paths using the $\xrightarrow{.f}$ edge. Thus, we must check for these first. The kill set is:

$\text{kill}_n(\text{G}) = \{\ s_0 \xrightarrow{x} n' \xrightarrow{.f} n'' \mid s_0 \xrightarrow{x} n' \xrightarrow{.f} n''$ is in G and there does not exist an edge $n''' \xrightarrow{.g} n'$ for any g and there does not exist an edge $s_0 \xrightarrow{w} n'$ for all w $\neq$ x $\}$



(a)



(b)

Figure 3.10: Example of paths being lost when incorrectly removing an edge.

You might be wondering, what if there is another field edge for `.g` $\neq$ `.f` transitioning out of `n`? Figure 3.11 shows an example.

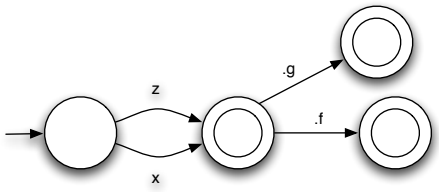


Figure 3.11: An example paths graph with two field edges transitioning out of a node.

Then, if we remove the edge $s_0 \xrightarrow{x} n'$, we would lose other paths that start with `x`. Actually, two field transitions from the same node is not possible during the data flow stage. This is because at most one field edge is created by a statement, and those created by distinct statements will have distinct source nodes (see transfer function for load statements).

Note that the output of the data flow analysis is a NFA and will therefore not be minimal - paths will be represented using more states and transitions than necessary. However when we convert it to a DFA and minimise it, then the scenario in Figure 3.11 will be possible.

Object creation statement An object creation statement `x = new C` allocates a new instance of class `C` and makes `x` point to it. Figure 3.12 gives a graphical illustration.

Based on the transfer function for copy statements, it would seem that paths starting with `x` need to be rewritten to start with `new C`. `x` definitely needs to be replaced with something otherwise paths may refer to different objects before and after the assignment. However, the object created by `new` is not reachable before this statement. Therefore, there is no way of referring to these objects there. So, what do we do about these accesses? What if `x.f` is accessed further down and it points to a shared object? It turns out that it doesnt matter.

One of the properties of atomic sections is that threads are not allowed to communicate within them. Therefore, objects created inside an atomic section are not 'seen' by other threads until it finishes executing. This implies that if an object is reachable from such a new object, an assignment (copy, load, etc) must have been made inside the atomic section to enable this.

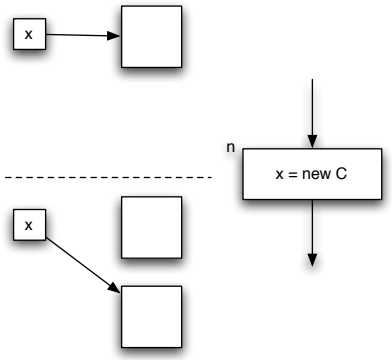


Figure 3.12: `new` statement.

Hence, based on the transfer functions for these types of assignments, objects referred to by paths starting with `x` after the statement, are already covered by other paths. This is because of how assignments essentially rewrite paths to what is on the right hand side. For example, assume that we have the control flow graph in Figure 3.13.

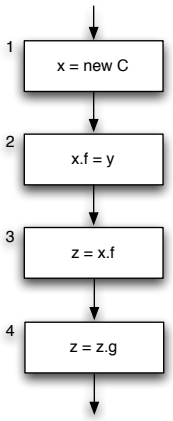


Figure 3.13: An example involving `new`.

The rule for load statements generates $s_0 \xrightarrow{z} 4$ at 4, which is translated to $s_0 \xrightarrow{x} 3 \xrightarrow{.f} 4$ at 3. Now there is an access to an object reachable from the new object assigned to `x` (because we have `x = new C` higher up). If `x.f` is not null, then by the fact that there is no inter-thread communication, the object it points to must have been assigned to in the current atomic section. This is exactly the case as we have `x.f = y` at 2. Hence, $s_0 \xrightarrow{x} 3 \xrightarrow{.f} 4$ is translated to $s_0 \xrightarrow{y} 4$ by its transfer function.

This means we can ignore paths starting with `x` (we do not need to translate them). The kill and gen sets are:

$\text{kill}_n(\text{G}) = \{\ s_0 \xrightarrow{x} n' \mid s_0 \xrightarrow{x} n' \text{ is in G } \}$
$\text{gen}_n(\text{G}) = \{\ \}$

Method call statement Method calls `m(`$\mathtt{x}_0$`,`$\mathtt{x}_1$`,...,`$\mathtt{x}_n$`)`, invoke a method on receiver object $\mathtt{x}_0$ with the given arguments $\mathtt{x}_1$,...,$\mathtt{x}_{n-1}$ and store the return value in $\mathtt{x}_n$. We make the assump-

tion that x_0,...,x_{n-1} are fresh variables not appearing below the call. This can be achieved by introducing temporary variables.

The treatment of method calls in a backwards analysis is shown in Figure 3.14.

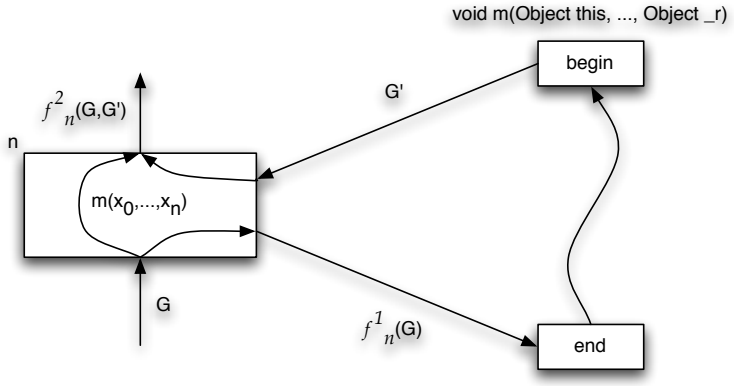


Figure 3.14: Interprocedural analysis in a backwards data flow analysis.

The arrows denote data flow, not control flow. We transform the incoming graph G as appropriate for entering the method (from the bottom), push the graph through the method and finally recombine the resulting graph G' at the top of the method with G. The first transformation is done by f^1_n and the recombining is done by f^2_n.

Method calls cause a problem because store statements in the called method can potentially affect paths in the calling method. Figure 3.15 shows an example.

```
class A {
    void m(A this) {
        this.f = y; }
}

void main() {
    // assume x.f points to some object
    m(x);
    // x.f may now point to a different object
}
```

Figure 3.15: Methods can change the object pointed to by a path.

It is for this reason that the paths graph in the calling method has to be passed into the called method. On the other hand, assignments to variables are not affected because they are stored on the stack and are local to the method. Therefore, assignments to variables in the called method do not affect paths in the calling method. This means the graph does not need to be modified. To keep track of which variables should be transformed in the called method, variable edges store the calling context they were generated in. For example, consider the program in Figure 3.16. In this example, $\xrightarrow{x}$ generated in `main` will not be transformed by `x = y` in `m` because the context when in `m` is `[1]` (assuming 1 is the label of the node calling `m`), while `x` is generated in the context `[]`.

However, the result variable is a special case because assigning to it *does* change its value in the calling method. Therefore, the calling contexts stored with all result variable edges are

```
class A {
     void m(A y) {
       A x;
       x = y; }
}

void main() {
     m(x);
     x.f...;
}
```

Figure 3.16: An example where a path `x` passed into a method `m` is not affected by the assignment to it in `m`.

extended to allow transformations by copy, load and new statements in the called method.

Finally, variables passed as arguments need to be renamed to their corresponding parameters (as this is what they are referred to in the called method). This will only affect the result variable, as all others are fresh and do not appear below the call. We use `_r` for the result parameter. This renaming is performed by f_n^1:

$$\begin{aligned} f_n^1(G) = \quad & \{\ n' \xrightarrow{.f} n'' \mid n' \xrightarrow{.f} n'' \text{ is in G } \} \\ & \cup \{\ s_0 \xrightarrow{x} n' \mid s_0 \xrightarrow{x} n' \text{ is in G and } \texttt{x} \text{ is not the result variable } \} \\ & \cup \{\ s_0 \xrightarrow{\_r} n' \mid s_0 \xrightarrow{y} n' \text{ is in G and y is the result variable } \} \end{aligned}$$

When combining G and G', we rename parameters back. All other variables in G' can be thrown away as they are not accessible in the caller. Finally, result variable edges in G can be discarded because the result variable will have been assigned to in the called method.

$$\begin{aligned} f_n^2(G, G') = \quad & \{\ s_0 \xrightarrow{x} n' \mid s_0 \xrightarrow{x} n' \text{ is in G and x is not the result variable } \} \\ & \cup \{\ n' \xrightarrow{.f} n'' \mid n' \xrightarrow{.f} n'' \text{ is in G } \} \\ & \cup \{\ s_0 \xrightarrow{a} n' \mid s_0 \xrightarrow{p} n' \text{ is in G' and p is a param. and a is the corresponding arg.} \} \\ & \cup \{\ n' \xrightarrow{.f} n'' \mid n' \xrightarrow{.f} n'' \text{ is in G' } \} \end{aligned}$$

3.3.2.3 Arrays

Array accesses require a new type of transition $n' \xrightarrow{[i]} n''$ called an *array lookup transition*. The key challenge with arrays is being able to distinguish between array elements, as this is crucial for being able to lock them individually. However, we leave this as further work. Instead, we are conservative: two array locations a[i] and a[j] may or may not be the same, we dont know. We now give the transfer functions.

Array load statement An array load statement `x = a[y]` assigns to `x` the value of the element in array `a` as given by `y`. This is similar to load statements except `a`'s elements may be of primitive type. This is okay because `x` will also be of primitive type (otherwise the program will not type check), so there will be no paths starting with `x`. Hence, we can ignore this case. Note, we are also accessing the array object `a`. The kill and gen sets are:

$\text{kill}_n(\text{G}) = \{\ s_0 \xrightarrow{x} n' \mid s_0 \xrightarrow{x} n' \text{ is in G } \}$

$\text{gen}_n(\text{G}) = \{\ s_0 \xrightarrow{a} n\ \} \cup \{\ n \xrightarrow{[y]} n' \mid s_0 \xrightarrow{x} n' \text{ is in G}\ \}$

Array store statement An array store statement `a[x] = y` assigns to the element given by `x` in array `a` the value of `y`. Again, this may or may not be an object.

If the array's elements are of primitive type, no transformation is required because this statement does not change the objects pointed to by paths. If instead they are of object type, we treat array stores similarly to store statements by considering all array accesses that have compatible types as potential aliases. That is, a[i] and b[j] may be aliases if a and b are of the same type or if the type of a is a subtype of the type of b or vice versa. This gives the following kill and gen sets:

$\text{kill}_n(\text{G}) = \{\ \}$
$\text{gen}_n(\text{G}) = \{\ s_0 \xrightarrow{a} n\ \} \cup \{\ s_0 \xrightarrow{x} n'' \mid n' \xrightarrow{[y]} n''$ and a is not an array of primitives and type of paths up to n' is compatible with type of a's elements $\}$

3.4 Summary

In this chapter we have presented a novel data flow analysis that computes a finite state automata describing path expressions for objects accessed in an atomic section. Our analysis can competently handle language features such as loops (see Appendix B), method calls, array accesses and inheritance. The advantage of our approach is that it provides a compile-time representation for objects accessed at run-time without sacrificing object granularity.

Chapter 4

Inferring locks from object accesses

In the previous chapter, we described an algorithm for inferring the objects being accessed inside an atomic section. Our algorithm produces a finite state automaton describing path expressions for these objects. We call this a paths graph.

Now that we know which objects are being accessed, the next step is to determine the locks that protect them. This is ultimately what lock inference is about.

In object-oriented languages, the compile-time path for an object identifies its unique lock at run-time. Hence, the set of paths described by our paths graph are the locks that need to be acquired. For example, consider the graph in Figure 4.1.

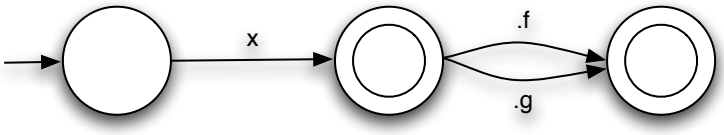


Figure 4.1: Example paths graph.

The set of paths described is {`x, x.f, x.g`} and the corresponding locks are {`x, x.f, x.g`}, i.e. they are equivalent. This is fine when there are no loops, but when there are, this set is infinite. For example, the graph in Figure 3.1 describes the set {`n, n.next, n.next.next, ...`}. This poses a problem because it is infeasible to lock all paths in such a set. One proposed solution is to instead lock 'something' that the objects have in common. We lock their classes.

4.1 Multi-locks

This requires the provision of locks that subsume other locks. We call these subsuming locks *multi-locks* and normal locks *single-locks*. The semantics are that acquiring a multi lock has the effect of acquiring all single-locks it subsumes. A multi-lock cannot be acquired if any of its single-locks have been acquired and a single-lock cannot be acquired if its subsuming multi-lock has been acquired. We make the following assumptions:

- Multi-locks are not nested
- Single-locks are subsumed by at most one multi-lock

Figure 4.2 illustrates this organisation.

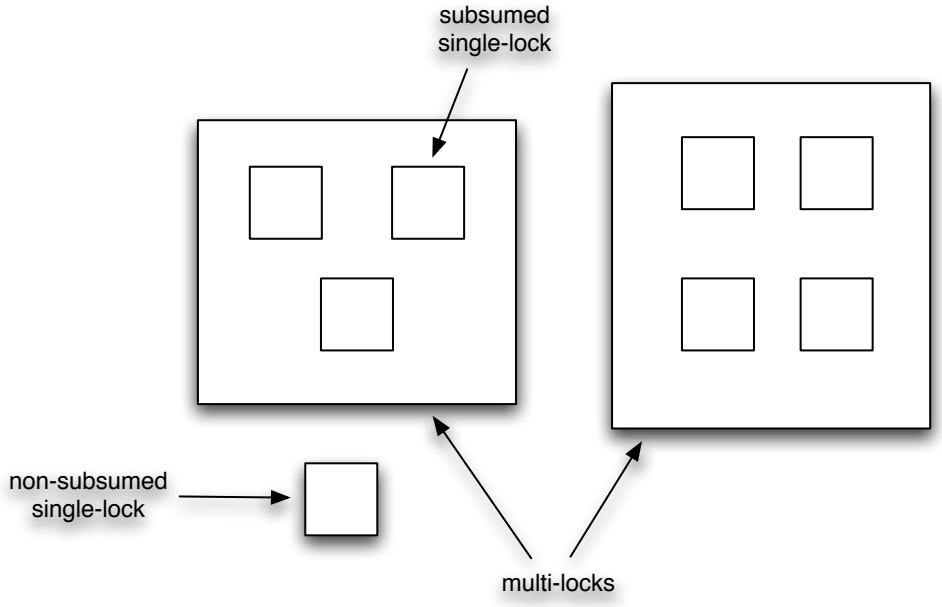


Figure 4.2: Multi-locks and single-locks.

```
class A {
    A f, g, h, i;
}
```

Figure 4.3: Class A.

4.2 Combining multi-locks and single-locks

In general, paths up to a cycle in the graph describe a finite set and can therefore be locked individually. The locks to protect them are the paths themselves. However, locks for the remaining unbounded set of paths are the multi-locks that subsume them—their class types in our case. We associate a multi-lock with each class that subsumes all single-locks protecting its instances. For example, assume class `A` in Figure 4.3 exists and that there is a variable `x` of type `A`. Consider the paths graph in Figure 4.4. Paths up to the loop can be locked individually, requiring the locks {`x, x.f, x.g`}. Remaining paths are of the form, `x.f.g.h`, `x.f.g.h.i`, `x.f.g.h.h.i`, etc. These are all of type `A` so we can protect them by simply acquiring the multi-lock on class `A`. So now we have a finite set of locks protecting a potentially unbounded number of objects. The final set is {`x, x.f, x.g, A`}.

4.3 Array accesses

A path involving an array lookup such as `a[i].f` poses a problem because we cannot accurately determine what the index `i` is. This is because our paths graph algorithm does not keep track of this information. Hence, the value of `i` at the time of the access may be completely different than at the top of the atomic section (we leave this as future work). This is a problem because we cannot be sure whether we have locked the object being accessed. We overcome this by acquiring a lock that subsumes all array elements—their class type again. This updates our

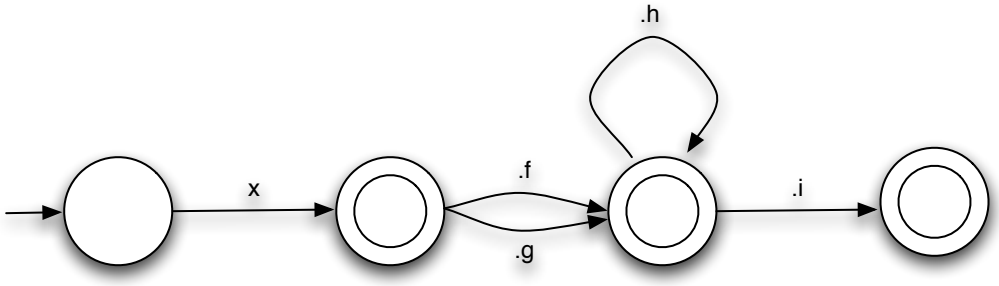


Figure 4.4: A paths graph with loops.

algorithm as follows:

- Locks for paths up to a loop **or array access** are the paths themselves
- Locks for the remaining paths are the multi-locks that subsume them—their class types

As an example, consider the paths graph in Figure 4.5. The path up to the array access is just `x`. The remaining paths are `x[i]` and `x[i].f`. Assuming that the array's elements are of type `A` (where `A` is defined in Figure 4.3), both paths are of type `A`. So, we can protect them by acquiring the multi-lock on `A`. The final set of locks is {`x, A`}.

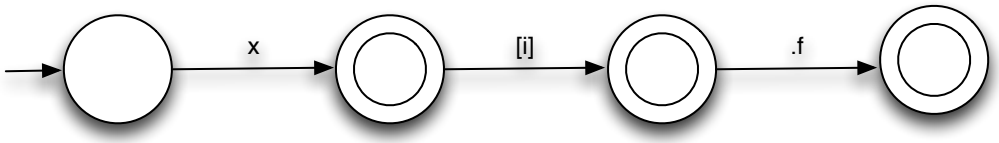


Figure 4.5: A paths graph with an array access.

4.4 Inheritance

Inheritance permits a variable of type `A` to point to instances of any class that is either equal to or a subclass of `A`. This means that in the example of Figure 4.4, although the compile-time type of paths is `A`, the run-time type could also be any subclass of it. However, a class's multi-lock only protects its instances. Therefore, we have to acquire multi-locks on all subclasses too. This updates our algorithm as follows:

- Locks for paths up to a loop or array access are the paths themselves
- Locks for the remaining paths are the multi-locks that subsume them—their compile-time class types **and all subclass types**.

4.5 Our algorithm

We use a non-standard data flow analysis to calculate for each state s, the locks protecting all paths from s_0 to s. This is represented as a pair `(Ps,Cs)`, where `Ps` is the set of single-locks and

`Cs` the set of multi-locks. We make the assumption that a class's name identifies its multi-lock in the same way that an object's path identifies its single-lock.

Initially these sets are empty. We start from the start state and propagate through the graph until they stabilise. The final set of locks is obtained by combining the results of all states. Note if a class `A` is to be locked, there is no point locking paths subsumed by it. Figure 4.6 gives pseudo-code for obtaining the final set of locks.

Final set of classes to lock (multi-locks):

```
classesToLock = {  }
for each state s in the paths graph {
    (Ps,Cs) = s.locks;
    classesToLock.addAll(Cs); }
```

Final set of paths to lock (single-locks):

```
pathsToLock = { }
for each state s in the paths graph {
    (Ps,Cs) = s.locks;
    for each path p in Ps {
        if(classOf(p) is not in classesToLock)
            pathsToLock.add(p); } }

locks = pathsToLock + classesToLock
```

Figure 4.6: Pseudo-code for obtaining the final set of locks.

4.5.1 Calculating entry information

In a standard data flow analysis, edges are unlabelled and data from predecessors is combined at the entry to a node using set union or intersection. However, paths graphs have labelled edges, so we instead first extend data from predecessors to take into account the field, variable or array index labelling each edge before taking the set union. For example, consider the paths graph in Figure 4.7.

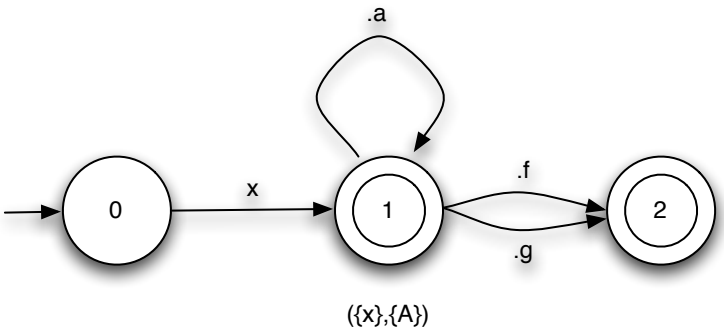


Figure 4.7: Calculating entry information in our lock inference algorithm.

Furthermore, assume the locks calculated so far at state 1 are `({x},{A})`. That is, the single-lock for `x` and the multi-lock for class `A`. When propagating this to state 2, we extend it taking

```
entryPs = {}
entryCs = {}
for(each predecessor pred of s) {

    (Ps,Cs) = pred.locks;
    for(each transition t from pred to s) {

        if t is a variable transition with variable v
            add (v, {s}) to entryPs

        else if t is a field transition with field f {
            // extend paths
            for each (p,visitedStates) pair in Ps {
                add (p.f, {s} ++ visitedStates) to entryPs }
            // extend classes
            for each class C in Cs {
                add typeAndAllSubtypesOf(field f in C) to entryCs } }

        else if t is an array lookup transition {
            // extend classes
            for each path p from start state to pred
                add typeAndAllSubtypesOf(p) to entryCs } } }

entry(n) = (entryPs, entryCs);
```

Figure 4.8: Pseudo-code for calculating entry information.

into account the fields on the edges: if we have to lock `x` at state 1, then because of the `.f` and `.g` transitions we have to lock `x.f` and `x.g` at 2. Furthermore, if we have to lock `A` at 1, then we must lock the types of `A.f` and `A.g` at 2, which again is `A`: Therefore the locks at the entry to 2 are `({x.f,x.g},{A})`:

To detect loops, we keep track of the states visited. Each element in `Ps` is now a pair: `(p,visitedStates)` where `p` is a path. Figure 4.8 gives the pseudo-code for calculating entry sets.

4.5.2 Transfer function

Each state has the same transfer function, which detects and removes cycles and adds the corresponding class types to `Cs`. Note there can only be one cycle because we remove it immediately. We present the transfer function procedurally in Figure 4.9 as it is clearer this way. Here is an example of removing the cyclic component as performed by the algorithm: Suppose we have `({x.f.g.h},{2,3,2,1})` in `Ps`. Then, removing the cycle $2 \rightarrow 3 \rightarrow 2$ results in: `({x.f},{2,1})`. (Note, visited states are appended to the front).

4.6 Summary

In this chapter we have provided an algorithm for mapping inferred path expressions calculated in the previous chapter to the locks that protect them. This is essentially the problem of lock inference. We reap the benefits of inferring path expressions by being able to infer fine-grained

```
{ entryPs, entryCs } = entry(n);
exitPs = { }
exitCs = Cs
for each (p,visitedStates) in entryPs {\\

    if visitedStates contains a cycle {
      add typeAndAllSubtypesOf(p) to exitCs;
        remove cyclic component from visitedStates and p to give (p,visitedStates)
        add (p,visitedStates) to exitPs;

    } else {
        add (p,visitedStates) to exitPs; } }

locks = (exitPs, exitCs);
```

Figure 4.9: Transfer function for our lock inference algorithm.

locks, as a path determines an object's unique lock at run-time. Our algorithm also has the advantage that it can infer locks for accesses made inside loops, where the number of objects accessed is unknown. For this, we lock the type of these objects. To support both coarse-grained and fine-grained locking simultaneously, we use multi-locks and single-locks.

Chapter 5

Preventing deadlock

In the previous two chapters, we inferred which objects are being accessed in the atomic section and calculated the locks that need to be acquired to protect them. We are now at a stage where we have almost everything we need to ensure atomicity for the atomic section. There is just one complication left.

As mentioned in the Background chapter, a well-known problem associated with locks is deadlock (see §2.2.2.4). This describes the state of a system where two or more threads are "stuck" waiting for each other to release locks. Why is this of significance? Well, it is quite possible that the locks we insert could cause the user's program to deadlock. This is a serious issue because the programmer has no control over the inferred locking policy and relies on the implementation to "get it right".

Existing approaches guarantee the absence of deadlock at compile-time by always acquiring locks in the same order [32] and if such an ordering cannot be found they typically coarsen the granularity [41]. However, this has a number of negative implications:

- Ordering the locks at compile-time is only possible if there is a finite number of locks. Given that the number of objects existing at run-time may be unbounded, this means that each lock may protect a large number of objects resulting in a coarse granularity.
- Deadlock is rare, yet the reduced parallelism caused by coarse locks will be experienced in all executions of the program.

Another option is to detect deadlock when it occurs and take the necessary measures to recover from it. The advantage of this is that the granularity of locking does not suffer and we can still guarantee that the atomic section will not deadlock. Recovery will typically entail forcing one of the threads to give up its locks and reacquire them. This is okay for us because we acquire all locks at the top of the atomic section before executing it, so no updates need undoing.

Deadlock is typically detected at run-time by looking for cycles in a graph whose nodes represent threads and which has an edge T1 $\rightarrow$ T2 if thread T1 is waiting for thread T2 to release a lock. This graph is called a *waits-for* graph. However, maintaining an explicit waits-for graph can be expensive both in space and time. More-so given that we now also have multi-locks.

Normally, a thread can only wait on one other thread because locks have up to one holder at a time. However, with multi-locks this is not the case. Consider the scenario given in Figure 5.1. Here, S1 and S2 are single-locks subsumed by multi-lock M1. They are held by threads T2 and T3 respectively and T1 wants to acquire M1. A multi-lock can only be acquired if it is not held by another thread and no subsumed locks have been acquired. Therefore, in this example T1 must wait for T2 and T3 to release their locks. This corresponds to the following two edges in

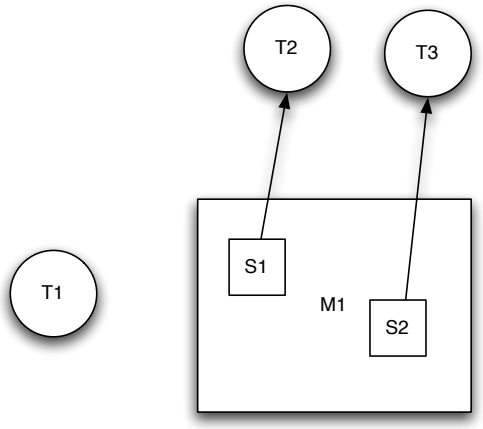


Figure 5.1: An example scenario involving multi-locks and single-locks.

the waits-for graph: T1 → T2 and T1 → T3, both from T1. In general, this can result in the waits-for graph having a large number of edges.

5.1 Our approach

Therefore, we present an alternative approach not yet considered that uses information already present in a multi-lock/single-lock implementation to detect deadlock. Namely:

- Current holder of a lock
- Single locks subsumed by a multi-lock
- A single-lock's subsuming multi-lock
- Which lock a thread is waiting to acquire

For example, our algorithm can detect deadlock in the example scenario of Figure 5.2 without maintaining an explicit waits-for graph. In this example, T1 holds M2 and T2 holds S2 and S4. T1 is waiting to acquire M1 and T2 is waiting to acquire M2. T1 can only acquire M1 once T2 has released S2. But T2 is currently waiting for M2, which is being held by T1. So, T1 is waiting for T2 and T2 is waiting for T1: deadlock!

Deadlock is detected by finding a cycle. Figure 5.3 gives pseudo-code for the algorithm (searching for a cycle starts from a lock).

5.2 Need a bit more for progress

We use our algorithm to check if blocking on a lock could lead to deadlock. If it could, we prevent the block. However, just preventing deadlock like this is not sufficient for progress. For example, in the scenario of Figure 5.2, simply removing the edge from T1 to M1 does not allow T2 to continue and eventually release it. T1 additionally needs to release M2. Hence, in addition to preventing the block, the requesting thread must release all held locks and retry to acquire them.

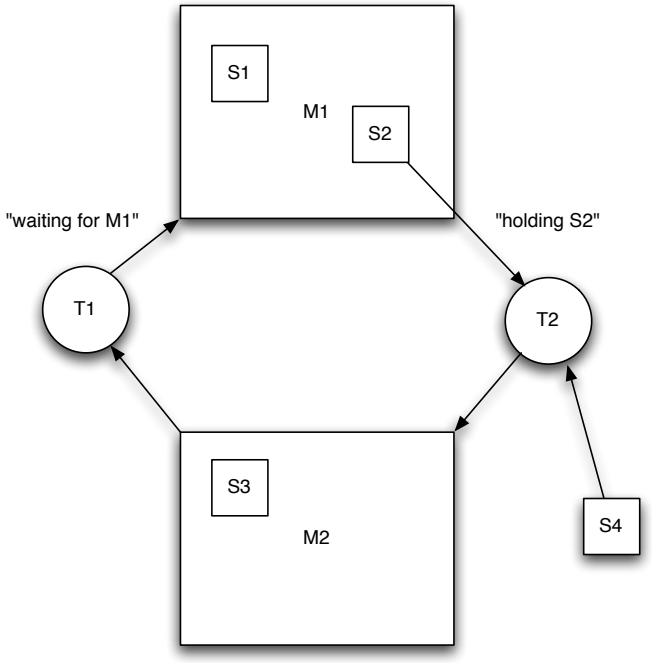


Figure 5.2: Example of deadlock.

5.3 Summary

In this chapter we provided the last component in our lock inference algorithm. In particular, we described an approach for preventing deadlock. This is extremely important because deadlock is a well-known problem associated with locks that can cause two or more threads to hang. We must ensure that our implementation does not cause a user's program to deadlock because they have no control over the inferred locking policy and just expect the implementation to "get it right". While existing work takes the necessary measures to guarantee freedom from deadlock at compile-time, this can often lead to coarser locks. We therefore take the approach of delaying to deal with deadlock until run-time. This permits fine-grained locks to be inferred during the lock inference stage of the previous chapter. The other advantage of our approach is that it does not require explicitly maintaining a waits-for graph, as we use the relationships that already exist between multi-locks, singe-locks and threads.

Multi-locks:

```
boolean cycleExists() {
    if already visited lock
      return true // cycle
    else if lock is held by thread t {
        // traverse edge to next lock (bypassing the thread)
        l = lock waited on by t
        return true if cycle from l }
    else if number of subsumed locks is > 0 {
        // traverse edges from each subsumed lock
        for each subsumed lock l {
            if l has been locked
                return true if cycle from l } }

    // no cycle accessible from this lock
    return false; }
```

Single-locks:

```
boolean cycleExists() {
    If already visited this lock
        return true // cycle
    else if this lock is held by thread t {
        // traverse edge to next lock (by passing thread)
        l = lock waited on by t
        return true if cycle from l }
    else if subsuming multi−lock m is locked
        return m.cycleExists();

    return false; }
```

Figure 5.3: Pseudo-code for detecting deadlock.

Chapter 6

Prototype implementation

We have implemented our lock inference and deadlock prevention algorithms of chapters 3 - 5 together with a prototype object-oriented language called *Single-step*. Figure 6.1 shows an overview.

We generate an abstract syntax tree (AST) representing the program and transform it at each stage. The type checker and simplifier perform pre-processing required for the analysis. The simplifier transforms the program as described in §3.1. We perform lock inference on each atomic section separately. The interpreter executes the program by walking the AST. There is support for multiple threads and a simple round-robin scheduler interleaves their operations. The run-time maintains a virtual heap.

6.1 Language features

Our language supports:

- Threads (created using `spawn { }`)
- Inheritance: casting, testing run-time type of an object, dynamic method binding, method and field overriding
- Arrays
- While loops, if statements, methods, recursion, overloading
- Integers, booleans, `this`, `null`

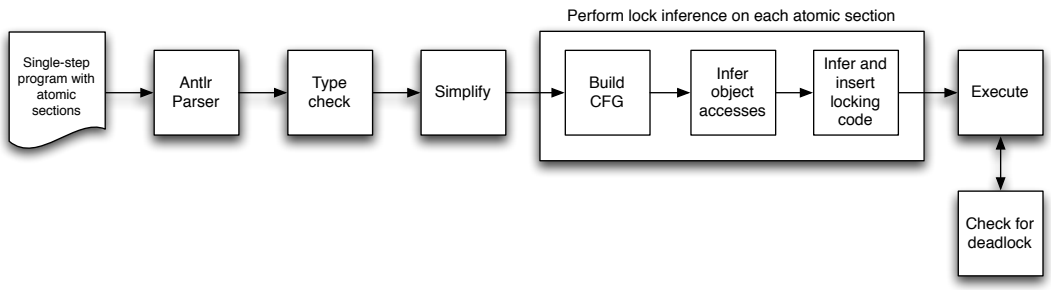


Figure 6.1: An overview of our prototype implementation.

- Locking paths (`lockpath p`), locking classes (`locktype T`), releasing all acquired locks (`unlockAll`).

6.2 Ensuring fairness between multi-locks and single-locks

Consider the example scenario given in Figure 6.2.

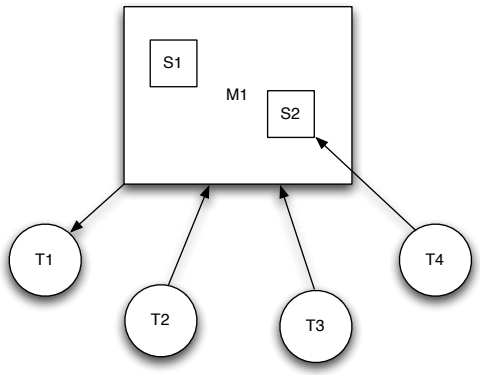


Figure 6.2: Problem of fairness between multi-locks and single-locks.

Here, T1 holds multi-lock M1, T2 and T3 are waiting to acquire it, and T4 is waiting for single-lock S2. When T1 releases M1, assume it is passed to the next thread waiting for it and only when there isn't one is S2 allowed to be acquired. However, this can lead to unfairness. Consider if M1 is passed to T2 and then T3 but just before T3 releases it another thread T5 requests to acquire it. Even though T4 came before T5, T5 will be given M1 making T4 wait longer. If this happens repeatedly, T4 may never be able to acquire S2 resulting in *starvation*.

We ensure fairness by alternating priorities such that if a multi-lock m is locked, then a request to lock one of its single-locks prevents m being passed to a waiting thread once it is unlocked (the single-locks are allowed to proceed instead), while if one or more single-locks are locked, then a request to lock their subsuming multi-lock prevents subsequent single-locks from being acquired (and the multi-lock is allowed to proceed instead once all its single-locks have been released).

6.3 Summary

In this chapter we briefly described the implementation of the algorithms in chapters 3 - 5 as part of a prototype language called Single-step. This provides a complete implementation of atomic sections.

Chapter 7

Evaluation

In this chapter we evaluate our proposed implementation for atomic sections. There are three components to our implementation:

- Inferring which objects are being accessed
- Mapping these objects to the locks protecting them
- Preventing deadlock

We first evaluate each of these in turn, before evaluating our implementation as a whole on a benchmark for transactional memory called STMBench7 [23].

7.1 Inferring object accesses

Our algorithm produces a finite state automaton describing path expressions for objects accessed. We call it a *paths graph*. It provides a number of advantages over existing work, notably:

- **It provides a compile-time representation for objects accessed at run-time without sacrificing granularity**

  In approaches that perform points-to analysis [44], all objects created at the same construction site are effectively considered the same object. Hence, they provide a very coarse approximation. However, path expressions resolve to the actual objects accessed at run-time. Therefore, field lookups, lock acquisition, etc, done using paths will be performed on the corresponding run-time object - we still have fine-grained objects.

- **It can infer accesses inside loops**

  This is a problem acknowledged in existing work with the suggestion to use regular expressions. However, nobody has come up with a general algorithm. In fact, we have found regular expressions to be inadequate. Our paths graph on the other hand is able to represent such accesses (see Appendix B).

- **It is simple and extensible**

  The algorithm is a simple data flow analysis with operations on the graph performed by each type of statement formulated as transfer functions. Therefore, it is also easy to extend for additional statements because all it requires is additional transfer functions.

However, its main limitation is due to it being a compile-time analysis. Until run-time, it is typically not known which execution paths will be taken. To ensure our algorithm produces an output that is correct for any arbitrary run, we have to be conservative and assume any path can be taken. This typically leads to more path expressions being inferred than will be accessed in a particular execution of the atomic section. Consider the example given in Figure 7.1.

```
if(b) {
  x = x.f;
  y = y.f;
} else {
  z = z.f;
}
```

Figure 7.1: Both branches of the if have to be considered at compile-time.

Here, `x` and `y` are accessed in the 'then' branch and `z` in the 'else' branch. At compile-time we don't know which will be taken, therefore we assume either can and infer `{x, y, z}`. Other limitations we have identified are now given.

7.1.1 Aliasing

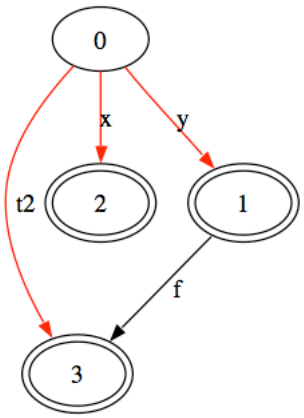
Recall, two variables are aliases if they point to the same object. This causes a problem when using path expressions because assigning to a field accessed using one path can change the objects accessed using other ones. Aliases are usually computed using a points-to analysis but this goes beyond the scope of the project. Instead, we conservatively assume that if we have `p.f` and `p'.f` (i.e. right-most fields are the same), then `p` and `p'` may be aliases.

This can again lead to more paths being inferred than actually need to be. Consider the example program in Figure 7.2(a).

```
class A {
    A f, g;
}

main {
    x = new A();
    y = new A();
    // x and y are not aliases
    atomic {
        x.f = t2;
        t1 = y.f.g; }}
```

(a)



(b)

Figure 7.2: We handle aliasing conservatively, which can lead to more paths being inferred than necessary.

Throughout this program, `x` and `y` are not aliases. However, our algorithm doesn't know this, so when updating the graph at `x.f = t2`, it thinks they could be and thus the object referred to by `y.f` could be affected by the assignment. As a result, it adds an unnecessary variable edge $\xrightarrow{t2}$. The resulting graph is shown in Figure 7.2(b).

7.1.2 Dynamic method binding

A feature present in most object-oriented languages is dynamic method binding that binds method calls to method bodies at run-time based on the receiver object's concrete class type. However, this means that at compile-time we do not know which method body will be executed for a given method call and thus have to again be conservative and assume any potential one may. For example, consider the program in Figure 7.3.

```
class A {
    A x;
    void m() { this.x; }
}

class B extends A {
    void m() { } // do nothing
}

void main() {
    atomic {
      A a = new B();
        a.m(); }}
```

Figure 7.3: With dynamic method binding, we don't know exactly which method will be called at compile-time.

In this example, the object pointed to by `a` is guaranteed to be an instance of class `B` when execution reaches the method call. However, because our analysis is conservative, it considers all possible targets. In particular, if `a` is of type `A` then the method executed would be `m()` in class `A`. Thus the paths inferred are {`a`} (because of the access in `A.m()`) whereas this set should really be { }.

7.1.3 Simplifying the program

The final limitation we have identified is due to simplifying the program before doing the analysis. Our analysis deals with simple loads and stores of the form `x = y.f` and `x.f = y` respectively. Consequently, we cannot eliminate paths of length longer than 2 from the graph even though it would be clearly sensible to do so. Consider the example program in Figure 7.4.

```
x.f.g = y;
//... access x.f.g below ...
```

(a)

```
t1 = x.f
t2.g = y
//... access x.f.g below ...
```

(b)

Figure 7.4: Our analysis cannot eliminate paths of length longer than 2 even if it is sensible to.

It is clear from the unsimplified program in Figure 7.4(a) that the object being assigned to is the same as the one accessed below, therefore all paths starting with `x.f.g` should be renamed to start with `y` (note, we still need to keep {`x, x.f`}). However, this is not clear in the simplified version in Figure 7.4(b) and so we keep paths starting with `x.f.g`.

As already explained, this lack of precision is mainly because our algorithm is run at compile-time. However, it can largely be overcome by incorporating additional analyses such as points-to.

We detail some of these in the Conclusions and Future Work chapter.

7.2 Inferring locks from objects

The granularity of the inferred locking policy will have a dramatic impact on the amount of concurrency permitted at run-time. Our approach and most other lock inference approaches manage locks on behalf of the programmer. Therefore, it is our responsibility to ensure locks are as fine-grained as possible.

We exploit the fact that in object-oriented languages, an object's path is used to refer to its fine-grained lock. Our approach has a number of appealing features:

- **We infer fine-grained locks when possible**

  Except for accesses inside loops and accesses involving array lookups, we infer fine-grained locks, that is, the lock protecting the object accessed at run-time. This permits significantly more concurrency over approaches such as [32] that protect all objects created at the same construction site with the same lock because accesses involving distinct objects are allowed to proceed in parallel.

- **We can lock accesses made inside loops**

  Loops present a significant challenge for lock inference, because one can only assume that the number of objects accessed is unbounded. If we used a single lock for each object, this would amount to an infinite number of locks. So, the challenge is: how to lock a potentially infinite number of objects with a finite number of locks? We achieve this by locking their classes - something that they have in common. The advantage of this is that it doesnt require annotations from the programmer [41] and also means that we can still offer fine-grained locking outside loops.

However, the biggest drawback of our approach is that the granularity of locking in the case of loops and arrays is too coarse. Locking a whole class is a bit too drastic and can really impede concurrency given how ubiquitously these programming constructs are used.

Inheritance makes things worse because a class's lock only protects its instances but the object referred to by a variable can be an instance of any subclass of its compile-time type too. This therefore requires having to lock the class hierarchy rooted at the compile-time class type of a variable. Figure 7.5 shows the most extreme example.

```
Object o = new Object();
atomic {
    while(o != null) {
        o = o.f; } }
```

Figure 7.5: Inheritance can result in locking all classes!

Our analysis will infer that class `Object` needs to be locked because of the potentially unbounded number of `Object` objects being accessed in the loop. Locking the `Object` class only protects its instances. However thanks to inheritance, the run-time types of these objects may be any subclass too. Therefore, we have no choice but to lock all subclasses of `Object`. If `Object` is the super class of all classes, like in Java, this means locking every single class! However, notice that `o` only ever points to an object of type `Object`, so we do really only need to lock `Object`.

Therefore, having additional information about the types of objects a path can point to reduces the need to be conservative. This technique is known as *concrete type inferencing* but is beyond the scope of the project.

7.3 Preventing deadlock

Freedom from deadlock is an important requirement for lock inference implementations, given that most commonly the programmer will have no control over the inferred locking policy. Our proposed approach recovers from deadlock when it occurs at run-time by revoking the requesting thread's locks and forcing it to retry. We avoid the need to buffer updates as in transactional memory by acquiring all locks before executing the atomic section.

Our approach has a number of advantages over existing work:

- **We don't require a finite number of locks or have to coarsen the locking granularity**

  Existing approaches typically avoid deadlock by acquiring locks in some order. However, this requires there being a finite number of locks. Given that the number of objects at run-time can be potentially unbounded, the result is that locks are coarse. Furthermore, if such an ordering cannot be found then the locking policy is typically coarsened. However, this leads to reduced parallelism even though the occurrence of deadlock is rare.

- **We use locks of differing granularities**

  Our approach can detect deadlock even when a combination of multi- and single-locks have been acquired/being waited on. Having locks of differing granularities is extremely important for lock inference as it enables balancing performance with protecting accesses inside loops.

- **We don't require an explicit waits-for graph to be maintained**

  The typical approach to detecting deadlock is to maintain a waits-for graph. However, this can be expensive at run-time. Instead, we utilise the links between multi-locks, single-locks and threads which already exist. Therefore, our approach requires less memory.

The main limitation of our approach is that it performs a completely fresh search for cycles on every lock operation. If the number of threads and locks is large, this can impose a large performance penalty. In hindsight, a waits-for graph may be slightly better in some situations as it only records arcs that correspond to 'waiting' whereas we also traverse arcs that correspond to 'acquired' (see Figure 5.2), although we don't have to maintain a separate data structure. An incremental approach to deadlock detection has been proposed by [44]. We again revisit this in the Conclusions and Future Work chapter.

Furthermore, recovering from deadlock at run-time requires that locks be revoked (otherwise progress cannot be guaranteed). To ensure this does not break atomicity, we are confined to acquiring all locks at the start of the atomic section. This can adversely affect the amount of parallelism that is allowed because a thread must wait until all required locks are available. At present, we dont see any alternative, as deadlock prevention at run-time will almost definitely require revoking locks.

7.4 Atomic sections for software

Having evaluated each component of this project, we now evaluate performance as a whole.

7.4.1 Performance

Performance is a very important factor for any implementation of atomic sections, given programmers typically want their software to run as fast as possible. However, while ideally we would like to quantitatively compare our implementation against competing techniques such as transactional memory, this would be beyond the scope of the project. Furthermore, our use of a prototype language means that we must abstract away the notion of time, as it is meaningless in this context.

To evaluate the performance aspect of this project as a whole, we compare our approach to coarse-grained locking (a single global lock protecting each block of code that would be placed inside an atomic section). The reason for this is because this is a comparable level of simplicity that the programmer can expect.

We use the following metrics:

- Number of computation steps taken by a thread to complete an operation. This can be thought of like the number of CPU cycles on a single-core single-CPU machine.
- Number of threads (logically) executing in parallel at the same time. This reveals information about the amount of concurrency in the system.

7.4.2 STMBench7 benchmark

STMBench7 [23] is a new benchmark for evaluating transactional memory implementations. It comprises of a large and complex data structure consisting of a set of graphs and indexes intended to be suggestive of many complex applications, such as CAD/CAM. It also provides a large number of operations to model a wide range of workloads and concurrency patterns. Figure 7.6 shows a portion of its data structure. At the top-most level, it consists of a number of *module* objects, each of which has an associated *manual*. Each module object contains a tree of assemblies. Non-leaf nodes of this tree are called *complex assemblies* and leaves are *base assemblies*. Each complex assembly can have any number of children and they are stored in a set. A base assembly contains several *composite parts*. A composite part has a *document* and links to a graph of *atomic parts* connected via *connection objects*. There is a many-to-many mapping from base assemblies to composite parts. The set of all composite parts for a module is called the *design library*. Each element in the tree contains links to its parents allowing bottom-up as well as top-down traversal.

STMBench7 provides forty-five operations covering the following four categories:

- **Long traversals:** go through all assemblies and/or all atomic parts. Some of them update documents or atomic parts.
- **Short traversals:** traverse the structure via a randomly chosen path, starting from a module, a document or an atomic part.
- **Short operations:** choose some object (or a few objects) in the structure and perform an operation on the objects or its local neighbourhood.

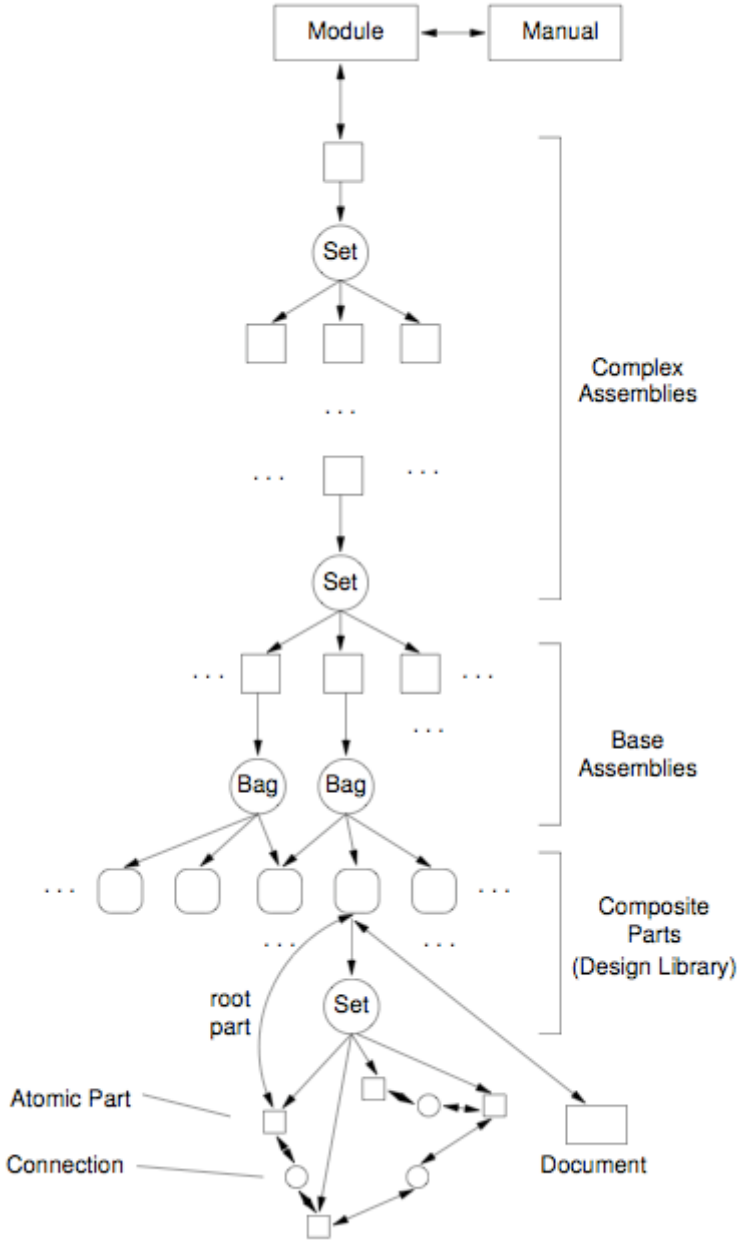


Figure 7.6: Main datastructure in the STMBench7 benchmark.

- **Structure modification operations:** create or delete elements of the structure or links between elements (randomly).

One thing to note is that STMBench7 assumes each operation is to be executed atomically. For evaluation of our work, this means enclosing each one within `atomic {}` or acquire and release operations for the single global lock we are comparing against.

7.4.2.1 Implementing STMBench7 in Single-step

STMBench7 is written in Java and uses some features that our prototype language Single-step does not support such as generics [22]. We therefore had to re-implement the data-structure and a subset of the operations in our prototype language.

7.4.2.2 Operations

Given the time-scale involved, it was not possible to consider all of the operations in the benchmark. Instead we implemented the following nine:

- **Short Traversal 1 (ST1):** traverse the structure top-down, from the module to an atomic part, via a random path and return the sum of the attributes `x` and `y` of the visited atomic part. The traversal fails if a base assembly with no descendant composite parts is visited.
- **Short Traversal 3 (ST3):** traverse the structure bottom-up from a randomly chosen atomic part to the root complex assembly. Each complex assembly is visited at most once. This operation returns the number of complex assemblies visited.
- **Short Operation 1 (OP1):** randomly choose 10 atomic parts and perform a read-only operation one each of them. Return the number of atomic parts whose `x` attribute is strictly greater ($>$) than its `y` attribute.
- **Short Operation 6 (OP6):** choose a random complex assembly and perform a read-only operation on all its sibling complex assemblies. Return the number of complex assemblies processed.
- **Short Operation 7 (OP7):** choose a random base assembly and perform a read-only operation on all its sibling base assemblies. Return the number of base assemblies processed.
- **Structural Modification 1 (SM1):** create a composite part, with its corresponding document and a graph of atomic parts and add it to the design library. Fail if the maximum number of composite parts has been reached.
- **Structural Modification 2 (SM2):** delete a randomly chosen composite part together with its corresponding document and the graph of descendant atomic parts.
- **Structural Modification 3 (SM3):** Create a link between a base assembly and composite part (chosen at random).
- **Structural Modification 4 (SM4):** Choose a random base assembly then delete a randomly chosen link between the base assembly and some composite part.

7.4.2.3 Experimental setting

All experiments were performed on an Apple MacBook Pro with Intel Core 2 Duo 2.33GHz CPU and 2GB of RAM. The operating system was Mac OS X Tiger 10.4.9 running Java J2SE build 1.5.0_07-164.

7.4.2.4 Important assumptions

Our prototype interpreter simulates multi-threading by interleaving the operations of different threads using a simple round-robin scheduler. The behaviour is as if you were executing a multi-threaded program on a single-core single-CPU machine.

In the results we present, we use the metric "number of computation steps" to give some abstract notion of time. This can lead to misleading results because for example, three threads executing for 200 steps each is equivalent to one thread executing for 600 steps (because of interleaving their operations). Please take this into account when looking at our readings.

7.4.2.5 Accessing distinct parts of the tree

We first look at the short operations (OP1, OP6, OP7) that each access distinct parts of the tree (atomic parts, complex assemblies and base assemblies respectively). Given that they access disjoint data, they should be allowed to execute in parallel. We spawned three threads, one for each short operation and measured the number of threads active at each computation step. Figure 7.7 shows the graph we obtained.

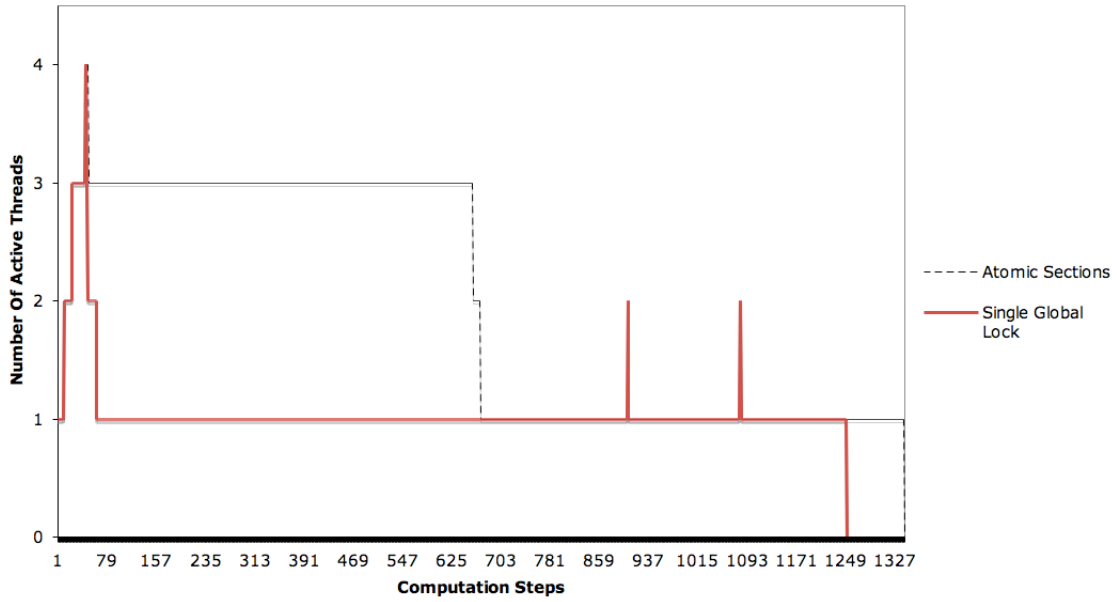


Figure 7.7: Short operations.

The initial thread initialises the data structure before the three operation threads are spawned. Our graph shows readings from just after this initialisation period. In the case of the single global lock implementation, the OP1 operation is executed between steps 64-906. The red spike indicates that the thread has released the lock and the next waiting thread has been resumed. The OP6 operation is executed between 907 and 1083 and finally OP7 between 1084 and 1253.

Furthermore, the initial spike shows the three threads being spawned (there are four including the initial thread). After the threads have spawned and are about to do their operations, we quickly see that with our atomic sections they are able to execute concurrently, while with the single global lock they can't. Between steps 25 and 658 all three threads execute with our implementation after which OP6 and OP7 finish and only OP1 is left. Note that it may appear that OP1 takes a lot longer to execute but this is because we are not assuming real concurrency, only interleaving. Hence, while it was executing concurrently, it could get less done in the same time than if it was executing on its own. The gap at the right between where the version using the global lock finishes and the version using our atomic sections finishes shows that there is a slight run-time overhead with atomic sections. We cant really quantify this because it can be due to inefficiencies in the interpreter.

Figure 7.8 shows the number of computation steps from spawning each thread till completion of its operation using atomic sections and the single global lock respectively. The values in the graph include the steps spent blocked by each thread (as this counts towards how long it takes to complete the operation). As expected, because each thread is serialised in the case of the single global lock, they take a lot longer to complete their operations.

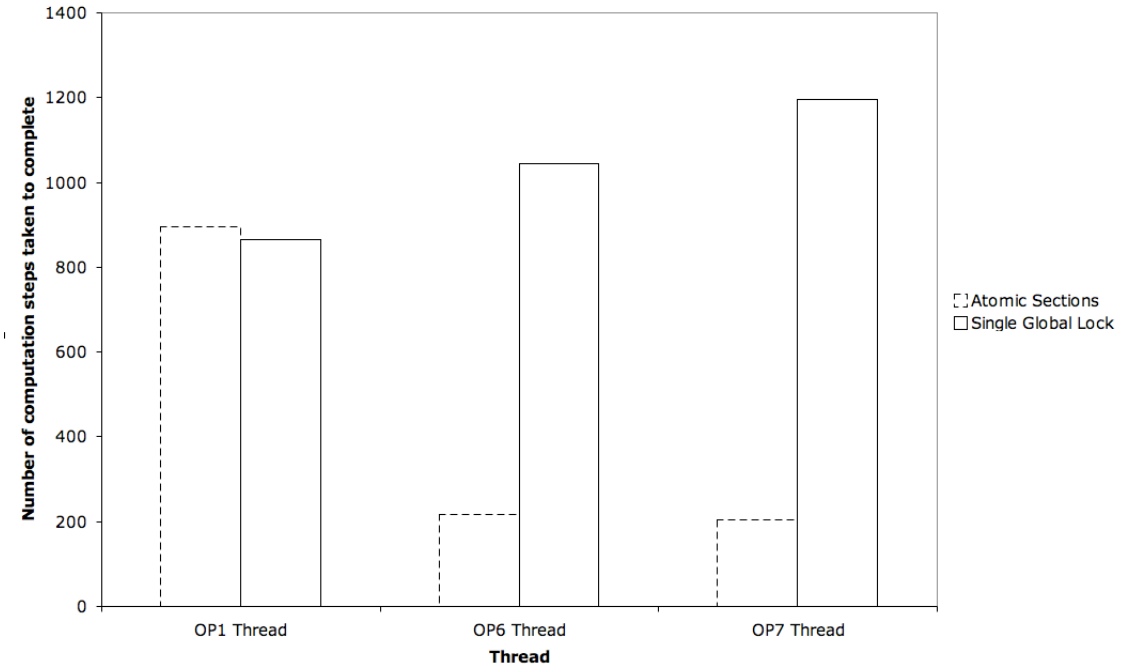


Figure 7.8: Short operations.

7.4.2.6 Traversing the tree

We now consider the two short traversal operations ST1 and ST2 that traverse the entire data structure top-down and bottom-up respectively. Figure 7.9 shows the number of active threads at each computation step (we show readings just after the data structure has been initialised and the initial spike shows the two threads being spawned). After the initial thread has spawned these threads, it terminates. Again we measured the number of active threads at each computation step. Figure 7.9 shows our results.

The graph shows that our atomic sections do not achieve any additional parallelism over the single global lock. This is because both operations iterate through the nodes in a while

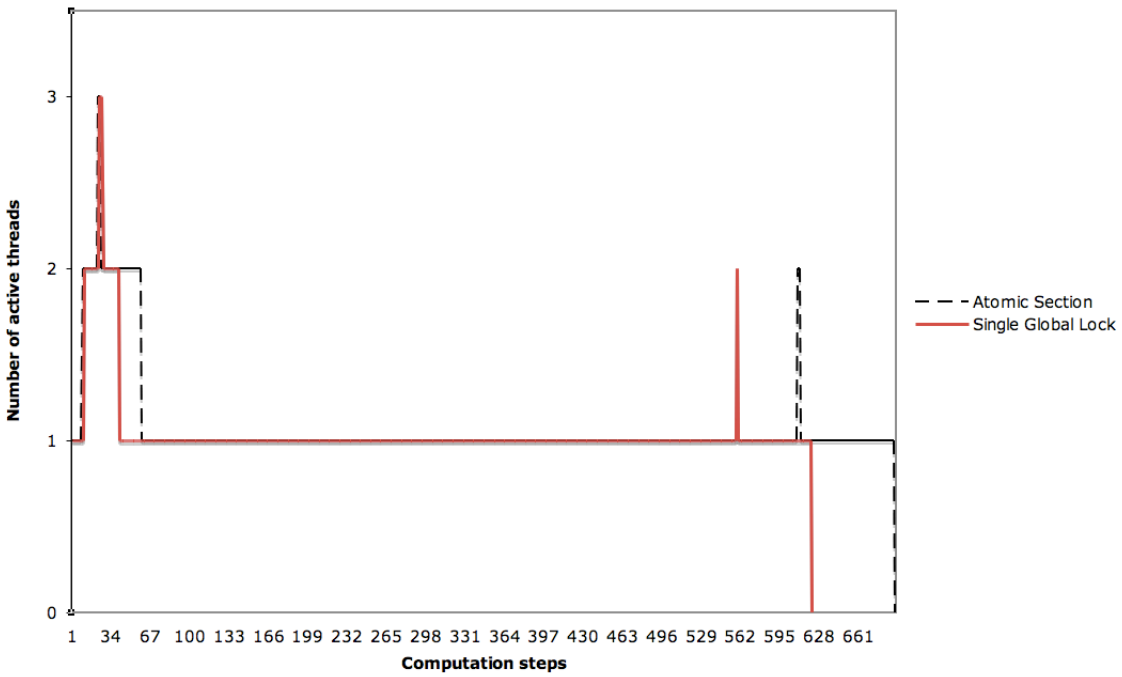


Figure 7.9: Short traversals.

loop resulting in the `Assembly` super class lock being inferred (in both cases). As a result, both operations are serialised. Figure 7.10 is a graph comparing the number of computation steps taken by each thread to complete its operation in both approaches.

The difference between the numbers of computation steps between the approaches in this graph is partly due to locking code inserted by our algorithm. However, as mentioned previously, it is hard to quantify such differences.

7.4.2.7 Modifying the tree

The final four operations that we implemented are the structural modification operations (SM1, SM2, SM3 and SM4) that modify the data structure. However, these operations require access to a global object that keeps track of counts and indexes. As a result, they all require the lock on this object, so they cannot execute in parallel. One solution would be to restructure the application but this would in some ways defeat the purpose of atomic sections. We omit graphs.

7.5 Summary

In this chapter, we first evaluated the three main components of our implementation of atomic sections: identifying object accesses, inferring locks from these accesses and preventing deadlock. We highlighted their key advantages with respect to the current state-of-the-art and also described the limitations of our approaches. Performance evaluations are very limited with this type of project given that we are using a prototype language. However, preliminary results using a benchmark called STMBench7 show that our algorithm can achieve more parallelism than using a single global lock. The main weakness of our approach is that the locks inferred for array accesses and accesses inside loops are too coarse. There are a number of techniques we can use to significantly improve this, which we consider in the Conclusions and Future Work

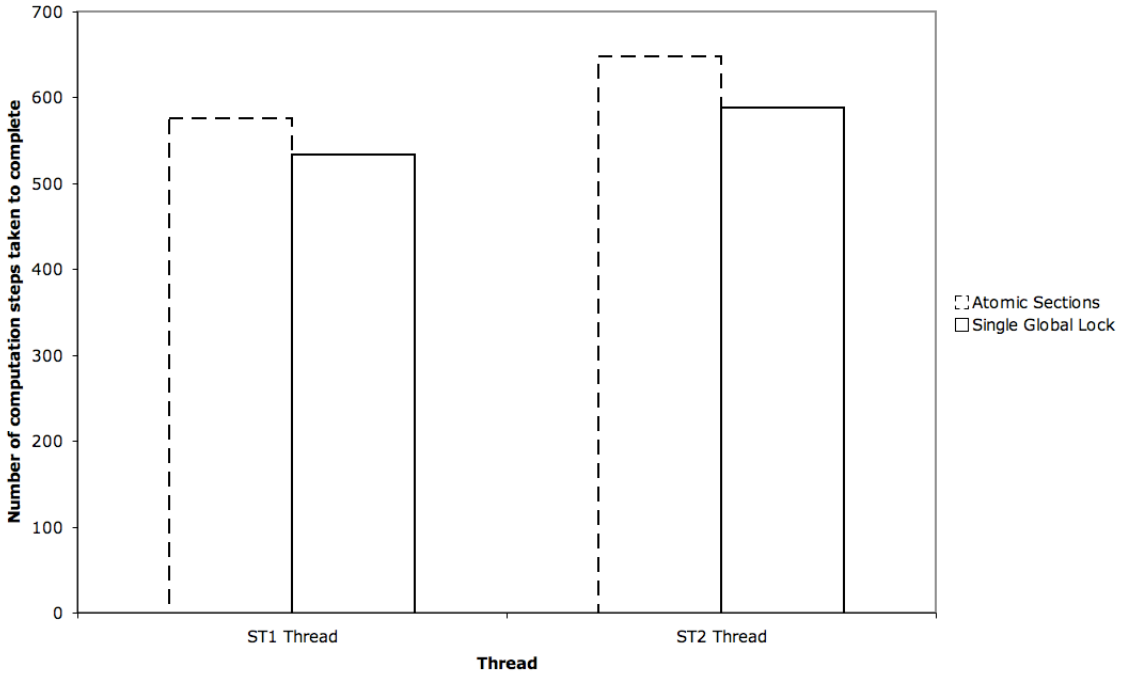


Figure 7.10: Short traversals.

chapter.

Chapter 8

Conclusions and Future Work

Atomicity is an important property for concurrent software, as it provides a stronger guarantee against errors caused by unanticipated thread interactions. However, concurrency control in general is tricky to get right because current techniques are too low-level and error-prone. Consequently, a new software abstraction is gaining popularity to take care of concurrency control and the enforcing of atomicity properties, called atomic sections.

The aim of this project was to explore lock inference for implementing atomic sections in object-oriented languages. Lock inference infers at compile-time the locks that need to be acquired for atomicity and inserts them transparently. As illustrated in the Background chapter, this technique offers a number of advantages over the currently popular approach of transactional memory, most notably being the ability to handle I/O and less run-time overhead. However, it is still in its infancy and there are a lot of challenges to be overcome.

We identified three main aspects of a lock inference algorithm:

- Inferring which objects are being accessed
- Mapping these objects to the locks protecting them
- Preventing deadlock

In this report, we have proposed possible solutions to each of these and implemented them in a prototype object-oriented language called Single-Step. We have also partially evaluated our approach using the STMBench7 benchmark.

8.1 Conclusions

We have found that it is possible to approximate object accesses at compile-time without sacrificing granularity by using path expressions. Our experiments show that this can lead to fine-grained locks which permit accesses to disjoint data to proceed in parallel. However, we also found that without additional information about a program, one has no choice but to be conservative. This can have a dramatic impact on how much concurrency the implementation allows with atomic sections being serially executed in the worst case. This shows that for a lock inference implementation to be appealing for use in real software, it will almost defininately require many optimisations. Thus the resulting algorithm will typically be quite complicated.

We have demonstrated that it is possible to support accesses inside while loops, however, this typically requires the provision of both coarse and fine locks so that the potentially unbounded number of accesses in a loop can be protected with a few locks while at the same-time fine-grained locks can be used for other objects. Furthermore, we have found that locking classes is

an extremely bad idea especially when inheritance is present. Without information about what types of concrete objects a path may point to, the algorithm will typically end up locking some portion of the class hierarchy, crippling concurrency.

We presented an algorithm for preventing deadlock at run-time that does not require keeping an explicit waits-for graph. We have not been able to accurately measure the performance implications of this due to our use of a prototype interpreter. However, these overheads will typically be nothing compared to the buffering and committing that occur in transactional memory. Furthermore, delaying deadlock detection till run-time means that the locking granularity does not need to be coarsened. The potential for increased parallelism as a result of this, may outset any overhead.

We now look towards possible future work for overcoming some of the limitations of this project and taking it further towards being an implementation of atomic sections for real-world software.

8.2 Future Work

8.2.1 Proving correctness

Our approach is only useful if it is correct. The two theorems that we would need to prove are:

- Our approach does not cause the user's program to deadlock.
- Soundness of our paths graph algorithm.

The first could be proved by modeling the locking mechanism and deadlock prevention algorithms together as a state machine with a separate state representing deadlock. Then, using a verification tool for concurrent systems such as LTSA [40] to check that we can never reach this deadlock state. The second property requires considering each each type of statement (e.g load statements, store statements) in turn and showing that the paths graph resulting from the transformation refers to at least the objects it did after the statement plus any new objects accessed in the statement.

8.2.2 Ownership types

A significant weakness of our approach is that the granularity of locking is too coarse when protecting potentially unbounded number of object accesses inside loops. At present we lock classes, however this protects significantly more objects than necessary and results in poor parallelism. Furthermore, inheritance makes things worse requiring that all subclasses be locked too. A more fine-grained solution is to lock another object that in some way subsumes the objects being accessed. One way of doing this is to introduce an *ownership relation* between objects such that objects can be owned by other objects. This relationship is typically one-to-many: each object can own many objects. For example, in a linked list implementation, the main `List` object may own its `Node` objects. Locking an owner object results in protecting all its owned objects.

Consequently, for protecting objects accessed inside while loops, we can instead lock their owners. This should drastically improve concurrency because owned sets of objects will typically be much much smaller than the set of instances of a class. For example, two traversals of distinct linked lists can proceed in parallel because the nodes will typically have different owners (first and second `List` object respectively). This technique would require a type system called *ownership types* [8, 6] and would be a very useful extension to our work.

8.2.3 Read/write locks

At present our locks prevent two threads from accessing the same object at the same time, that is they ensure *mutual exclusion*. However, it is perfectly fine for threads to perform reads at the same time because there is no interference caused. Consequently, another possible area of improvement is to instead use read/write locks that permit several threads to read from the object at the same time but only one thread to write to it. This should be easy to do given that we already distinguish between load statements (reads) and store statements (writes).

8.2.4 Optimised locking policies

A required restriction for ensuring atomicity is that the inferred locking policy must adhere to the two-phase locking protocol (see §2.2.2.1), that is it should not acquire a lock once a lock has been released. Our approach uses the most basic scheme of acquiring all locks at the start of the atomic section and releasing them at the end. However this has a number of disadvantages including:

- Atomic sections may have to wait a long time before they can start executing especially if they access a large number of objects.
- Locks are held until the end of the atomic section even if they are no longer required and could be released easier. This would allow other threads to acquire them.

There are a number of optimised alternatives to the basic scheme we employ that can afford more parallelism without sacrificing the ability to enforce atomicity. Another interesting extension then would be to consider such alternative schemes such as releasing locks when they are no longer required and acquiring locks lazily. Although the latter may conflict with preventing deadlock at run-time given that we can't undo updates already made.

8.2.5 Incremental deadlock detection

Recall that at present, our deadlock algorithm performs a fresh search for a cycle on each lock request. This is inefficient and another possible future area of work would be to look at ways of improving it. This is obviously important because deadlock detection at run-time incurs run-time overheads which should not be unsatisfactory. One piece of work [44] has proposed an algorithm for incremental detection that doesn't perform a fresh search each time, but instead does it progressively. It would be interesting to explore the use of such an algorithm in our implementation.

8.2.6 Implementing in an industry standard language

To be able to properly test the feasibility of lock inference as an implementation for atomic section, we would need to implement it in an industry standard language such as Java. This would enable us to try it out on large and complex real-world software. Furthermore, it would allow us to make useful comparisons with competing techniques such as transactional memory. However, real languages have a large number of language features, therefore it will be a significant challenge. We would start out with the language features currently supported in Single-step and then incrementally add support for additional features.

Bibliography

[1] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 207–222, 1999. 16

[2] C. S. Ananian and M. Rinard. Efficient object-based software transactions. In *Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, Oct 2005. In conjunction with OOPSLA'05. 16

[3] D. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 258–268, 1998. 16

[4] B. Chan and T. S. Abdelrahman. Run-time support for the automatic parallelization of java programs. *J. Supercomput.*, 28(1):91–117, 2004. 20, 29, 30

[5] D. Cunningham. Path inference for atomic sections; first year report. Available online at http://www.doc.ic.ac.uk/~dc04/1st_year_report.pdf, 2006. 16, 19, 20, 23, 29, 30, 34

[6] D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universe types for race safety. 2006. 21, 66

[7] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. *Proc. International Symposium on Distributed Computing*, 2006. 16

[8] W. Dietl and P. Muller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, 2005. 21, 66

[9] R. Ennals. Software transactional memory should not be obstruction-free. 2006. 16

[10] Everything2.com. Lock convoying, February 2006. Available online at http://everything2.com/index.pl?node_id=1786627 (accessed 25-December-2006). 11, 15

[11] P. Felber and M. Reiter. Advanced concurrency control in java. *Concurrency and Computation: Practice and Experience*, 14(4):261–285, 2002. 17

[12] C. Flanagan and S. Freund. Automatic Synchronization Correction. *Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005. 19

[13] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *SIGPLAN Not.*, 39(1):256–267, 2004. 10

[14] C. Flanagan, S. N. Freund, and M. Lifshin. Type inference for atomicity. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 47–58, New York, NY, USA, 2005. ACM Press. 10

[15] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349, New York, NY, USA, 2003. ACM Press. 10

[16] C. Flanagan and S. Qadeer. Types for atomicity. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 1–12, New York, NY, USA, 2003. ACM Press. 10

[17] Foldoc. Livelock, June 2007. Available online at http://foldoc.org/foldoc.cgi?livelock (accessed 19-June-2007). 11

[18] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. 16

[19] K. Fraser and T. Harris. Concurrent Programming without Locks. *Submitted for publication*, 2004. 16

[20] S. Freund and S. Qadeer. Checking concise specifications for multithreaded software. In *Workshop on Formal Techniques for Java-like Programs*, 2003. 10

[21] B. Goetz. Synchronization optimizations in mustang. *Java theory and practice (IBM developerWorks)*, October 2005. Available online at http://www-128.ibm.com/developerworks/java/library/j-jtp10185/ (accessed 30-12-2006). 17

[22] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005. 60

[23] R. Guerraoui, M. Kapałka, and J. Vitek. STMBench7: A benchmark for software transactional memory. In *Proceedings of the Second European Systems Conference EuroSys 2007*, pages 315–324. ACM, Mar. 2007. 13, 53, 58

[24] T. Harris. Design choices for language-based transactions. *University of Cambridge Computer Laboratory Tech. Rep., Aug*, 2003. 16

[25] T. Harris. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58(3):325–343, 2005. 16

[26] T. Harris and K. Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, 38(11):388–402, 2003. 15, 16

[27] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, 2005. 15, 16

[28] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing memory transactions. *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 14–25, 2006. 16

[29] J. Hatcliff, Robby, and M. B. Dwyer. Verifying atomicity specifications for concurrent object-oriented software using model-checking. In *VMCAI*, pages 175–190, 2004. 10

[30] M. Herlihy, J. Eliot, and B. Moss. Transactional Memory: Architectural Support For Lock-free Data Structures. *Computer Architecture, 1993. Proceedings of the 20th Annual International Symposium on*, pages 289–300, 1993. 15

[31] M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III. Software transactional memory for dynamic-sized data structures. *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, 2003. 16

[32] M. Hicks, J. S. Foster, and P. Pratikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006. 16, 18, 19, 20, 47, 56

[33] B. Hindman and D. Grossman. Strong Atomicity for Java Without Virtual-Machine Support. 2006. 16

[34] B. Hindman and D. Grossman. Atomicity via source-to-source translation. *Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 82–91, 2006. 16

[35] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. 31

[36] M. Jones. What really happened on mars rover pathfinder. *ACM Forum on Risks to the Public in Computers and Related Systems*, 19(49), December 1997. Available online at http://catless.ncl.ac.uk/Risks/19.49.html#subj1. 15

[37] D. Kalinsky and M. Barr. Priority inversion. *Embedded Systems Programming*, pages 55–56, April 2002. Available online at http://netrino.com/Publications/Glossary/PriorityInversion.html. 11, 14

[38] N. G. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *Computer*, 26(7):18–41, 1993. 15

[39] D. Lomet. Process structuring, synchronization, and recovery using atomic actions. *ACM SIGOPS Operating Systems Review*, 11(2):128–137, 1977. 11

[40] J. Magee and J. Kramer. *Concurrency: state models & Java programs*. John Wiley & Sons, Inc., New York, NY, USA, 1999. 66

[41] B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. *ACM SIGPLAN Notices*, 41(1):346–358, 2006. 16, 17, 19, 20, 21, 47, 56

[42] M. Moir. Transparent Support for Wait-Free Transactions. *Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 305–319, 1997. 16

[43] F. Nielson, H. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999. 24

[44] D. J. Pearce. *Some directed graph algorithms and their application to pointer analysis*. PhD thesis, Imperial College of Science, Technology and Medicine, February 2005. 18, 53, 57, 67

[45] K. Poulsen. Tracking the blackout bug. *Security Focus*, April 2004. Available online at http://www.securityfocus.com/news/8412. 15

[46] M. F. Ringenburg and D. Grossman. AtomCaml: first-class atomicity via rollback. *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 92–104, 2005. 16

[47] S. Rodger. *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett Publishers, Inc. USA, 2006. 31

[48] B. Saha, A. Adl-Tabatabai, R. Hudson, C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, 2006. 16

[49] W. Scherer III and M. Scott. Advanced contention management for dynamic software transactional memory. *Proceedings of the twenty-fourth annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, pages 240–248, 2005. 16

[50] A. Stoughton. *An Introduction to Formal Language Theory That Integrates Experimentation and Proof*. Draft, December 2006. 31

[51] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, March 2005. Available online at http://www.gotw.ca/publications/concurrency-ddj.htm. 15

[52] L. Wang and S. D. Stoller. Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.*, 32(2):93–110, 2006. 10

[53] A. Welc, A. Hosking, and S. Jagannathan. Transparently Reconciling Transactions with Locking for Java Synchronization. *European Conference on Object-Oriented Programming*, 2006. 16

[54] J. Whaley and M. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. *Proceedings of the 9th International Static Analysis Symposium*, pages 180–195, 2002. 32, 33

[55] Wikipedia. Lock convoy, December 2006. Available online at http://en.wikipedia.org/wiki/Lock_convoy (accessed 25-December-2006). 11, 15

[56] Wikipedia. Acid, 2007. Available online at http://en.wikipedia.org/wiki/ACID (accessed 7-June-2007). 17

Appendix A

Single-step toy language grammar

A.1 Declarations

program	→	(classDecl)* mainDecl **EOF**
classDecl	→	**CLASS IDENT** (**EXTENDS IDENT**)?
		LBRACE (methodDecl | instanceVariableDecl **SEMI**)* **RBRACE**
mainDecl	→	**MAIN** statementList
methodDecl	→	constructorMethodDecl
		| typeArr **IDENT LPAREN** paramsList **RPAREN** statementList
constructorMethodDecl	→	type **LPAREN** paramsList **RPAREN** statementList
paramsList	→	(param (**COMMA** param)*)?
param	→	typeArr **IDENT**
instanceVariableDecl	→	typeArr **IDENT**
variableDeclList	→	variableDecl (**COMMA** variableDecl)*
variableDecl	→	typeArr **IDENT** (**ASSIGN** expression)?
typeArr	→	type (**LBRACK RBRACK**)?
type	→	**INT** | **BOOL** | **IDENT** | **VOID**
builtInTypeArr	→	builtInType (**LBRACK RBRACK**)?
builtInType	→	**INT** | **BOOL**
classTypeArr	→	**IDENT** (**LBRACK RBRACK**)?
classType	→	**IDENT**
arrayDeclarator	→	**LBRACK** expression **RBRACK**

A.2 Statements

statementList	→	**LBRACE** (statement)* **RBRACE**
statement	→	statementList | expression SEMI | ifStatement | whileStatement
		| variableDecl SEMI | atomicStatement | returnStatement | spawnStatement
		| printStatement | lockStatement
ifStatement	→	**IF LPAREN** expression **RPAREN** statementList **ELSE** statementList
whileStatement	→	**WHILE LPAREN** expression **RPAREN** statementList
atomicStatement	→	**ATOMIC** statementList
returnStatement	→	**RETURN** (expression)? **SEMI**
spawnStatement	→	**SPAWN** (variableDeclList)? statementList
lockStatement	→	**LOCKP** path **SEMI** | **LOCKT IDENT SEMI** | **UNLOCKALL SEMI**
printStatement	→	**PRINT** expression **SEMI**

A.3 Expressions

expressionList	→	expression (**COMMA** expression)*
expression	→	assignExpr
assignExpr	→	path **ASSIGN** assignExpr | conditionalExpr
conditionalExpr	→	logicalOrExpr
logicalOrExpr	→	logicalAndExpr (**LOR** logicalAndExpr)*
logicalAndExpr	→	equalityExpr (**LAND** equalityExpr)*
equalityExpr	→	relationalExpr ((**NOT_EQUAL** | **EQUAL**) relationalExpr)*
relationalExpr	→	additiveExpr (((**LT** | **LTE** | **GT** | **GTE**) additiveExpr) | **INSTANCEOF** typeArr)?
additiveExpr	→	multiplicativeExpr ((**PLUS** | **MINUS**) multiplicativeExpr)*
multiplicativeExpr	→	unaryExpr ((**TIMES** | **DIVIDE** | **MOD**) unaryExpr)*
unaryExprNotPlusMinus	→	**LNOT** unaryExpr | **LPAREN** builtInTypeArr **RPAREN** unaryExpr | **LPAREN** classTypeArr **RPAREN** unaryExprNotPlusMinus | postFixExpr
unaryExpr	→	**INC** unaryExpr | **DEC** unaryExpr | **MINUS** unaryExpr | **PLUS** unaryExpr | unaryExprNotPlusMinus
postFixExpr	→	primaryExpr (**INC** | **DEC**)?
primaryExpr	→	pathOrMethod | newExpression | **TRUE** | **FALSE** | **NULL** | **INTEGER** | **LPAREN** expression **RPAREN** | **LOCKP** path | **LOCKT IDENT**
pathOrMethod	→	path (**LPAREN** (expressionList)? **RPAREN**)?
newExpression	→	**NEW** type ((**LPAREN** (expressionList)? **RPAREN**) | arrayDeclarator)
path	→	(**THIS** | (**IDENT** (arrayDeclarator)?)) (**DOT IDENT** (arrayDeclarator)?)*

A.4 Tokens

A.4.1 Keywords

ATOMIC	→	“atomic”
BOOL	→	“bool”
CLASS	→	“class”
ELSE	→	“else”
EXTENDS	→	“extends”
FALSE	→	“false”
IF	→	“if”
INSTANCEOF	→	“instanceof”
INT	→	“int”
LOCKP	→	“lockp”
LOCKT	→	“lockt”
MAIN	→	“main”
NEW	→	“new”
NULL	→	“null”
PRINT	→	“print”
RETURN	→	“return”
SPAWN	→	“spawn”

THIS	→	“this”
TRUE	→	“true”
UNLOCKALL	→	“unlockAll”
VOID	→	“void”
WHILE	→	“while”

A.4.2 Other

INTEGER	→	(‘0’..‘9’)+
IDENT	→	(‘a’..‘z’ | ‘A’..‘Z’) (‘a’..‘z’ | ‘A’..’Z’ | ‘0’..’9’)*
ESC	→	‘\\’ (‘r’ | ‘t’ | ‘n’ | ‘\\’ | ‘”’)
LPAREN	→	‘(’
RPAREN	→	‘)’
LBRACE	→	‘{’
RBRACE	→	‘}’
LBRACK	→	‘[’
RBRACK	→	‘]’
SEMI	→	‘;’
LOR	→	“||”
LAND	→	“&&”
LNOT	→	‘!’
EQUAL	→	“==”
NOT_EQUAL	→	“!=”
LT	→	‘<’
LTE	→	“<=”
GT	→	‘>’
GTE	→	“>=”
PLUS	→	‘+’
MINUS	→	‘-’
TIMES	→	‘*’
DIVIDE	→	‘/’
MOD	→	‘%’
INC	→	“++”
DEC	→	“- -”
DOT	→	‘.’
COMMA	→	‘,’
ASSIGN	→	‘=’

Appendix B

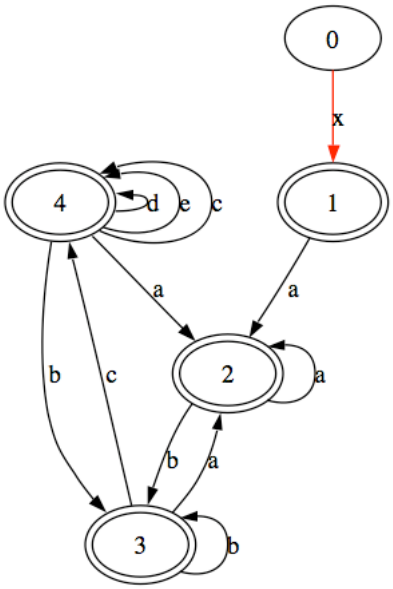
An example atomic section and its paths graph

Here we demonstrate that our paths graph algorithm can deal quite well with accesses inside loops. The example program in Figure B.1(a) has four levels of nesting consisting of three nested while loops an if statement at the innermost level. Each loop can either continue executing or return to its parent loop. The accesses can be represented using the regular expression: `x.(a*(b*(c*(d|e)*)*)*)*`. The paths graph generated using our algorithm is shown in Figure B.1(b):

```
class A {
  A a; A b; A c; A d; A e;
}

main {
  A x;
  atomic {
    while(b1) {
      x = x.a;
      while(b2) {
        x = x.b;
        while(b3) {
          x = x.c;
          while(b4) {
            if(b5) {
              x = x.d;
            } else {
              x = x.e; }}}}}}
```

(a)



(b)

Figure B.1: An example access involving nested loops and a conditional