Lock Inference Proven Correct

Dave Cunningham[1], Sophia Drossopoulou[1], and Susan Eisenbach[1]

Imperial College London {dc04,sd,sue}@doc.ic.ac.uk

Abstract. With the introduction of multi-core CPUs, multi-threaded programming is becoming significantly more popular. Unfortunately, it is difficult for programmers to ensure their code is correct because current languages are too low-level.
Atomic sections are a recent language primitive that expose a higher level interface to programmers. Thus they make concurrent programming more straightforward. Atomic sections can be compiled using transactional memory or lock inference, but ensuring correctness and good performance is a challenge. Transactional memory has problems with IO and contention, whereas lock inference algorithms are often too imprecise which translates to a loss of parallelism at runtime.
We define a lock inference algorithm that has good precision. We give the operational semantics of a model OO language, and define a notion of correctness for our algorithm. We then prove correctness using Isabelle/HOL.

1 Introduction

Programmers increasingly need to write multi-threaded programs to make full use of the available hardware. When writing multi-threaded code, programmers typically use the same data-structures and algorithms as sequential code. However, many simple *sequential assumptions* that allow local reasoning about program behaviour no longer hold in a concurrent setting. For instance, one can no-longer assume that the value stored in a variable is the value last written by the local thread. Writing code that is robust enough to remain correct, despite these weak semantics, is a mammoth task compared to writing sequential code where the assumptions always hold.

In order to stay productive, programmers often try to make operations *atomic* [15], using locks. If a block of code is atomic, a programmer can once again make sequential assumptions and reason locally about the behaviour of his code. Unfortunately, locks are extremely unforgiving. Small errors can cause the silent loss of atomicity (an *atomicity violation*). Even if the locking code is sufficient for atomicity, there are extra constraints that must be met to avoid *deadlock*. Finally, even if programmers understand these details, they can have great difficulty when writing large programs that have many complicated thread interactions.

Programmers create large programs using encapsulation. This allows them to concentrate on small parts of the program without worrying about the rest. However, in a concurrent scenario, it is not possible to lock successfully without having intimate knowledge of the behaviour of any called functions. If the internal behaviour of a function changes, it may be necessary to update locking code in distant parts of the program, to reflect these changes. Thus programmers lose encapsulation and consequently have to reason about much more than just the local code and state. The loss of encapsulation results in the perception of concurrent programming as prohibitively difficult.

The difficulty of using locks has led to the introduction of a new language primitive - the *atomic section*. Using this primitive, programmers need only designate an arbitrary block as an atomic section and the implementation does any global reasoning and instrumentation required so that sequential assumptions hold. Thus, programmers can make these assumptions, without having to write complex locking code, and without losing the benefits of encapsulation.

The problem moves from the application programmer to the language implementation, which must output code without atomicity violations, and without sacrificing parallel performance. Whatever mechanisms are used by the implementation, they must be used carefully. Any emergent behaviours such as deadlocks must be prevented or otherwise not exposed to the programmer, who should be able to use atomic sections free from implementation-specific constraints.

Current implementations of atomic sections use either transactional memory [1, 17] or lock inference [11, 9, 5, 3]. Transactional memory relies on being able to rollback blocks of code whose atomicity has been violated, but in general this means it cannot allow I/O or system calls in atomic sections. This restriction cannot in general be hidden from the programmer. Lock inference does not have this problem, but relies on precise static approximation of program behaviour. The more precise the inference, the more threads are allowed to execute in parallel. Programmers should not have to design their code so that the lock inference can easily understand it.

Our lock inference algorithm [5] is designed to give good precision. Whereas previous work relies on pointer analysis to statically model program behaviours, we use a more direct approach that has more in common with how programmers infer locks manually. However since our approach does not make such extensive use of proven technology, its correctness is brought into question. The contribution of this paper is a notion of correctness for our analysis, and an account of our experience proving it in the Isabelle proof assistant. In Section 2 we give examples showing how our analysis works. In Section 3 we formalise a model Object-Oriented language, our lock inference algorithm, the notion of correctness that binds them together, and describe our experience with Isabelle. In Section 4 we compare to related work and we conclude with Section 5.

2 Approach

We use a two phase locking protocol [8]. We derive a set of locks that we acquire before the atomic section and release afterwards. Our inference therefore takes an atomic section as input, which we call a *program*. We henceforth assume that atomic sections have already been converted into Control Flow Graphs (CFGs) and are therefore ready for program analysis. For simplicity, we assume

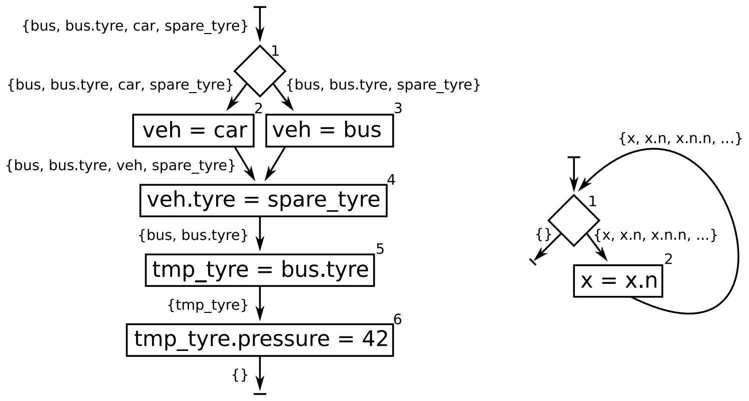

Fig. 1. Two example atomic sections

that atomic blocks are never nested[1]. We use a runtime mechanism that detects when a thread's lock acquisition would cause a deadlock [14] and rolls back all the lock acquisitions of that thread. Since lock acquisitions are together at the beginning of the atomic section, no transaction log is required to facilitate the roll back. We also assume everything accessed from an atomic section is shared between threads, and everything shared between threads is only accessed from atomic sections. We will discuss sharing more in section 5.

Consider the atomic section in Fig. 1. We use a backwards 'may' analysis to infer a set of locks at each edge. We assume every object has a lock that protects it, as in Java. Starting at node 6, we first infer a lock to protect the access of the object `tmp_tyre`. This propagates towards the beginning of the atomic section, where lock acquisition code is inserted. However, we have to translate locks as they are propogated to account for the statements they pass through. E.g. at node 5, `bus.tyre` is assigned to `tmp_tyre`, so acquiring the lock on `tmp_tyre` before node 5 does not help us ensure atomicity since it is not the object accessed at node 6. However, the correct object is held in `bus.tyre` so we can lock that instead. Also at node 5, a lock is inferred to protect the access of `bus`.

At node 4, we infer a lock to protect the `veh` access, but we also need to lock `spare_tyre`. This is because `veh` and `bus` may be aliases, and thus it may have been `spare_tyre` that was pressurised. In general we do not know, so we approximate. In this case we include both the `bus.tyre` lock (not aliased case) and the `spare_tyre` lock (aliased case).

In both branches of the conditional, there is a copy statement. At node 3, we translate `veh` to `bus` but since the set already contains `bus`, we effectively lose `veh`. Node 2 is similar. Since we do not know which branch will be chosen at node 1, we take the union of the two branches to form the final result. It is this set of locks that we hold for the duration of the atomic section.

[1] At runtime, one can set a flag that disables the inferred locks of inner atomic sections.

The right hand side of Fig. 1 is an atomic section that iterates over objects. Here, the algorithm as described above would not terminate. To force the analysis to terminate, we use a nondeterministic finite automaton (NFA) [12] at each edge, instead of the set of locks. When they contain a cycle, NFAs can finitely represent the unbounded set of objects accessed by such iterations. We call such NFAs *path graphs*.

There are many subtle mechanisms at work in our approach – e.g. the translation of locks across assignments, the handling of aliasing, and the use of NFAs to force termination. One could be forgiven for not immediately having confidence in the correctness of our algorithm. Fortunately, we can prove its correctness, as we will shortly demonstrate.

3 The Formal System

We will now give a syntax and semantics for a small Java-like language, the program analysis transition functions over this language, and then prove that the given transition functions infer correct locking information. **Notation:** $A \rightharpoonup B$ is the type of a partial map, $[a \mapsto b, c \mapsto d]$ is a partial map that maps a to b and c to d. We use _ to indicate an anonymous variable. We denote the empty sequence with ε and use . to prepend values onto sequences. Sometimes, for readability, we use commas instead of logical conjunctions ($\wedge$).

3.1 Syntax and Semantics

We analyse atomic sections independently, which we refer to as *Programs*. We assume programs have already been converted to a control flow graph (CFG) representation, where function calls are handled using bounded callstrings to approximate recursion at a fixed depth [18].

We let x, y, z range over local stack variables, f, g range over fields. Every CFG node has a unique id n chosen from some countable set $Node$. Thus our program P is defined in Fig. 2. In order, the statements are *copy assignment*, *object construction*, *heap load*, *heap store*, and *condition*. Every statement has a given successor n which is where execution proceeds after that statement, except the condition $\langle n; n' \rangle$ which non-deterministically chooses to continue execution from either n or n'. If a node has the successor n where $P(n)$ is undefined then the atomic section terminates. The right-hand program in Fig. 1 is therefore $P = [1 \mapsto \langle 3; 2 \rangle, 2 \mapsto [x = x.n; 1]]$, note that $P(3)$ is undefined.

We now give a model of the accesses incurred by an execution of a program P (we are not interested in the resulting heap or stack). This model is abstract, but not static. We have a judgement $P \vdash h, \sigma, n \rightsquigarrow A$. The intuition is that the sequence of actions A are performed by an execution of P, from the initial heap and stack h, σ and from the initial CFG node n. To represent non-terminating executions, we allow the execution to cease at any point. The sequence A may thus be shorter than a completed execution. However, our correctness theorem generalises over A, so it covers complete as well as incomplete executions. Note

$$\begin{array}{rcl} P \in \mathit{Program} &=& \mathit{Node} \rightharpoonup \mathit{Statement} \\ st \in \mathit{Statement} &::=& [x = y; n] \mid [x = \texttt{new}; n] \mid [x = y.f; n] \mid [x.f = y; n] \mid \langle n; n' \rangle \end{array}$$

$$\begin{array}{rcl} a \in \mathit{Addr} &=& \mathbb{N} \\ v \in \mathit{Value} &::=& a \mid \texttt{null} \\ h \in \mathit{Heap} &=& \mathit{Addr} \rightharpoonup \mathit{Object} \\ \mathit{Object} &=& \mathit{Field} \rightarrow \mathit{Value} \end{array} \qquad \begin{array}{r} f, g \in \mathit{Field} \\ \sigma \in \mathit{Stack} = \mathit{Var} \rightharpoonup \mathit{Value} \\ x, y, z \in \mathit{Var} \\ \alpha \in \mathit{Action} = a \mid \tau \end{array}$$

$$\frac{}{P \vdash h, \sigma, n \rightsquigarrow \varepsilon}\ (\textsc{Stop}) \qquad \frac{P(n) = [x = y; n'] \qquad P \vdash h, \sigma[x \mapsto \sigma(y)], n' \rightsquigarrow A}{P \vdash h, \sigma, n \rightsquigarrow \tau.A}\ (\textsc{Copy})$$

$$\frac{P(n) = \langle n'; n'' \rangle \qquad P \vdash h, \sigma, n' \rightsquigarrow A \quad \vee \quad P \vdash h, \sigma, n'' \rightsquigarrow A}{P \vdash h, \sigma, n \rightsquigarrow \tau.A}\ (\textsc{Cond}) \qquad \frac{P(n) = [x.f = y; n'] \qquad a = \sigma(x) \qquad P \vdash h[(a, f) \mapsto \sigma(y)], \sigma, n' \rightsquigarrow A}{P \vdash h, \sigma, n \rightsquigarrow a.A}\ (\textsc{Store})$$

$$\frac{P(n) = [x = \texttt{new}; n'] \qquad a \notin \mathit{dom}(h) \qquad P \vdash h[a \mapsto \lambda f.\texttt{null}], \sigma[x \mapsto a], n' \rightsquigarrow A}{P \vdash h, \sigma, n \rightsquigarrow \tau.(A[a \mapsto \tau])}\ (\textsc{New}) \qquad \frac{P(n) = [x = y.f; n'] \qquad a = \sigma(y) \qquad P \vdash h, \sigma[x \mapsto h(a, f)], n' \rightsquigarrow A}{P \vdash h, \sigma, n \rightsquigarrow a.A}\ (\textsc{Load})$$

Fig. 2. Syntax and Semantics of Execution Model

that while the language does not allow assignment of `null`, the runtime uses `null` as a default field value, and allows `null` to be stored on the stack. Assignment of `null` can thus be encoded by reading an uninitialised field.

We can consider the execution of the above example P in the heap $h = [1 \mapsto (n \mapsto 2), 2 \mapsto (n \mapsto 3), 3 \mapsto (n \mapsto 3)]$ and the stack $\sigma = [x \mapsto 1]$. The heap is undefined at addresses other than $1, 2, 3$, and by abuse of notation, fields other than n are `null`. The execution would normally not terminate because the "list" contains a cycle. However, the judgement $P \vdash h, \sigma, 1 \rightsquigarrow 1.2.3.\varepsilon$ holds regardless. It is also true that $P \vdash h, \sigma, 1 \rightsquigarrow 1.2.\varepsilon$ and in fact $\forall P, h, \sigma, n : P \vdash h, \sigma, n \rightsquigarrow \varepsilon$.

Note that we do not record accesses of objects that are constructed by P, due to the substitution in (New). This is because the locks that we infer will ensure that the new object remains thread-local until the end of the atomic section, so we do not have to infer a lock for constructed objects.

3.2 Analysis Transition Functions

To infer locks that make P execute atomically, we use a backwards 'may' analysis to infer a static approximation of the set of objects, the *path graph*, accessed by P. This is in contrast to the complete set of possible A such that $P \vdash h, \sigma, n \rightsquigarrow A$, which cannot be known statically.

Our representation of P is a control flow graph (CFG). At each CFG edge we accumulate a path graph, which is a special kind of nondeterministic finite automaton where every state is an exit state. A path graph is a finite representation of a potentially infinite set of locks, e.g. for the iteration example in

Fig. 1, we do not infer the infinite set of locks $\{x, x.n, x.n.n, \ldots\}$, rather the finite path graph $\{x \to 2, 2 \to^n 2\}$. First we will give a formal definition of path graphs, then we give the formal transition functions that show how path graphs are pushed around the CFG as the analysis reaches its fixed point. The definitions are in Fig. 3. We lock a path graph using multi-granularity locks. We first remove nodes involved in cycles, locking all objects that have the same type as those nodes. We then take the set of paths through the path graph and lock them in prefix order, ignoring any objects whose types are already locked.

$$\begin{aligned} \mathit{Edge} &::= x \to n \mid n \to^f n' \\ G \in \mathit{PathGraph} &= \mathbb{P}(\mathit{Edge}) \\ X \in \mathit{AnalysisState} &= \mathit{Node} \to \mathit{PathGraph} \end{aligned}$$

$$\begin{aligned} \mathit{acc} &: \mathit{Node} \to \mathit{Statement} \to \mathit{PathGraph} \\ \mathit{tr} &: \mathit{Node} \to \mathit{Statement} \to \mathit{PathGraph} \to \mathit{PathGraph} \end{aligned}$$

$$\begin{aligned} \mathit{acc}(n)[x = y; \_] &= \emptyset & \mathit{acc}(n)[x = y.f; \_] &= \{y \to n\} \\ \mathit{acc}(n)[x = \texttt{new}; \_] &= \emptyset & \mathit{acc}(n)[x.f = y; \_] &= \{x \to n\} \end{aligned}$$

$$\begin{aligned} \mathit{tr}(n)[x = y; \_](G) &= G \setminus \{x \to n' | x \to n' \in G\} \cup \{y \to n' | x \to n' \in G\} \\ \mathit{tr}(n)[x = \texttt{new}; \_](G) &= G \setminus \{x \to n' | x \to n' \in G\} \\ \mathit{tr}(n)[x = y.f; \_](G) &= G \setminus \{x \to n' | x \to n' \in G\} \cup \{n \to^f n' | x \to n' \in G\} \\ \mathit{tr}(n)[x.f = y; \_](G) &= G \setminus \{n' \to^f \_ | x \to n' \in G, \\ &\qquad\qquad (\nexists z \neq x : z \to n' \in G), \\ &\qquad\qquad (\nexists n''' : n''' \to_{-} n' \in G)\} \\ &\quad \cup \{y \to n' | \_ \to^f n' \in G\} \end{aligned}$$

Fig. 3. The analysis

The state of the analysis, X, stores a path graph at each CFG node, which represents the path graph at the edges pointing into that node. For conditional nodes $P(n) = \langle n'; n'' \rangle$, we simply have $X(n) = X(n') \cup X(n'')$, as is standard with backwards 'may' analyses. For all other nodes n, where $P(n) = [st; n']$, we calculate $X(n)$ as follows: $X(n) = \mathit{acc}(n)(st) \cup \mathit{tr}(n)(st)(X(n'))$. The access function acc provides the locks required to protect accesses performed by the local node n. The translation function tr translates path graphs from below n so that their meaning is preserved in spite of the changes to the heap and stack caused by n.

The access function adds locks to protect load and store statements, and otherwise adds nothing. The translation functions we will explain one at a time. Copy statements are handled simply by replacing $x \to n'$ with $y \to n'$ (for any n'). Construction is similar except it only removes edges. Accesses are 'lost' when they propagate through construction because the analysis realises that the object accessed is actually thread-local and therefore does not need to be locked. Loads are similar to copies, except that the $x \to n'$ edge gets replaced by a $n \rightarrow^f n'$ edge. This only makes sense if we can guarantee that an edge $y \rightarrow n$ exists in the new path graph. This is easily shown, however, since the access function adds precisely this edge. The case for store is (as one would expect) the most complicated. First, we can see that it adds an edge from y to any node in the path graph that might have been affected by the assignment to the f field. This is because we conservatively assume everything can be an alias of everything else. However, we know syntactically that x is an alias of x, so we can remove any $x.f$ accesses from the path graph. At node 4 of Fig. 1, we have `veh.tyre = spare_tyre`, and below we have $X(5) = \{bus \rightarrow 5, 5 \rightarrow^{tyre} 6\}$. We therefore add the edge $\{veh \rightarrow 4\}$ due to the access function *acc*. We also add $\{spare\_tyre \rightarrow 6\}$ due to the last part of the translation function *tr*. There is no $veh \rightarrow 5$ in $X(5)$, but even if there was, we would still not subtract $5 \rightarrow^{tyre} 6$ from $X(5)$ because $bus \rightarrow 5$ is present.

When the analysis reaches a fixed point, we know that the path graph $X(n)$ at every node n satisfies the constraints in Fig. 3. We denote this with $P \vdash X$.

3.3 Soundness

We want our inferred path graph at the initial edge $X(n)$ to represent at least the addresses accessed by the program as it executes. For this we need a *concretisation* function γ that interprets $X(n)$ in a given stack and heap to reveal which addresses it statically represents. We overload this function to also extract the addresses from a sequence of actions A (i.e. ignoring duplicate addresses and τ actions). We can state the theorem we want to prove:

Theorem 1. *Soundness:*

$$\left.\begin{array}{l} P \vdash h, \sigma, n \rightsquigarrow A \\ P \vdash X \end{array}\right\} \Longrightarrow \gamma(A) \subseteq \gamma(h, \sigma, X(n))$$

Proof: Induction over length of A.

This intuitively says that whatever may be accessed by an execution beginning from n, these accesses will be represented by the path graph at that node in the fixed point of the analysis. It remains to see how to define γ in the case of path graphs.

3.4 Assigning meaning to path graphs

We now consider an arbitrary path graph G and an *assignment* φ which maps each node in this path graph to a set of addresses. We will define γ by flattening an appropriate φ, i.e. just keeping the set of addresses mapped by φ and forgetting at which node they occur.

Definition 1. *Valid assignments:*

$$\begin{array}{rl} h, \sigma \vdash G : \varphi \Longleftrightarrow & (\forall x \mapsto n \in G : \sigma(x) \in \varphi(n)) \ \wedge \\ & (\forall n \rightarrow^f n' \in G : \{h(a, f) | a \in \varphi(n)\} \subseteq \varphi(n')) \end{array}$$

The intuition is that if the path graph contains the edge $x \to n$ then we want $a_x = \sigma(x)$ to be present in φ at n. However, if the stack is undefined at that variable, or if it is `null` then we ignore it. If the stack contains a valid address a_x for x, and G also contains $n \to^f n'$ then we want $h(a_x, f)$ to be present at n', unless that address is not defined on the heap[2] or the field contains `null`. We want addresses to flow around the path graph, initially with stack lookups, and then using the heap to follow field edges and find more addresses. Even if the path graph contains a cycle, such as with our linked list example, then the set of addresses involved can remain finite since the heap is finite. This is a purely theoretical mechanism to allow us to realise a path graph in a given stack and heap. At run-time we will use multi-granularity locks to effectively lock many more addresses than φ. To formally represent the flowing around the path graph, we give a judgement $h, \sigma \vdash G : \varphi$, and we let $\gamma(h, \sigma, G)$ be the flattened minimal φ that satisfies $h, \sigma \vdash G : \varphi$. We say that an assignment is *valid* in the context of some h, σ, G if it satisfies this judgement.

Note that there will likely be many valid φ for a given h, σ, G. In particular, $\forall h, \sigma, G : h, \sigma \vdash G : \varphi_{max}$ where $\varphi_{max} = \lambda n.Addr$. There will, however, be one minimal φ for a particular h, σ, G. We can define a partial ordering over assignments by lifting $\subseteq$ point-wise: $\varphi_1 \sqsubseteq \varphi_2 \iff \forall n.\varphi_1(n) \subseteq \varphi_2(n)$. We also let $\varphi_1 \sqcap \varphi_2 = \lambda n.\varphi_1 \cap \varphi_2$, i.e. the point-wise intersection of the two assignments.

Theorem 2. *Valid assignments join to make valid assignments:*

$$\left.\begin{array}{l} h, \sigma \vdash G : \varphi_1 \\ h, \sigma \vdash G : \varphi_2 \end{array}\right\} \Longrightarrow h, \sigma \vdash G : (\varphi_1 \sqcap \varphi_2)$$

Proof: Follows from the definitions.

If we define the minimal assignment:

$$\Phi(h, \sigma, G) = \bigsqcap \{\varphi | h, \sigma \vdash G : \varphi\}$$

Using the above theorem, we know that $h, \sigma \vdash G : \Phi(h, \sigma, G)$. Clearly, there cannot be any other valid $\varphi \sqsubset \Phi(h, \sigma, G)$. Now we can finish our notion of correctness by defining $\gamma(h, \sigma, G) = flatten(\Phi(h, \sigma, G))$.

3.5 Proof

We have proved correctness in Isabelle/HOL using Proofgeneral [2]. The file is 800 lines long, takes 30 seconds to process on a 3GHz P4, and is available online [6]. Aside from basic notation, explicit quantifiers, and explicit handling of the cases where partial maps do not contain a mapping from a particular value, the Isabelle/HOL formalism is identical to the one considered here.

Theorem 2 and many auxiliary lemmas were proved automatically. Theorem 1 was a long proof but often the final stages of each case were automatic. In

[2] Although this cannot happen in a language like Java, for simplicity our formalism permits initial stacks/heaps to contain undefined addresses.

particular, the extra details that are required in a proof assistant (but usually omitted in a hand-written proof) can usually be handled automatically at the beginning or end of the proof. We originally proved correctness with a slightly different formalism that had an extra parameter in the execution judgement to accumulate the constructed objects and needed only primitive recursion on A in the (New) rule. We later converted this to the form given in the paper. The conversion required us to manually intervene in the proof, but in all cases except (New) this was very easy, needing only the removal of any references to the extra parameter. Our overall experience with Isabelle was positive, and we enjoy having greater confidence in the correctness of our proof.

4 Related Work

As our work is an implementation of atomic sections we compare it first with the more popular atomic section implementation technique of transactional memory and then with other lock inference techniques.

Transactional memory [10, 1, 17] has trouble with I/O, which presents no problem with lock inference approaches. Conversely, transactions (being dynamic) can handle reflection easily, whereas lock inference has to be conservative. Particularly, plugin systems can cause problems for lock inference, but we expect that JIT techniques would be a solution. Transactional memory needs compiler support to instrument code with logging mechanisms, but lock inference requires much more compiler support in the form of complex analyses. Conversely, transactions use a lot of runtime mechanisms to detect conflict and rollback, whereas lock inference only needs locks at runtime. In terms of performance, transactions can waste cycles rolling back, but have perfect granularity. Lock inference must use conservative approximation and thus will always have worse granularity than transactional memory. However, the granularity of lock inference is still reasonable, and can be very good if ownership types are available [4].

Lock inference algorithms have become more precise over time. Initially they used only points-to sets to statically characterise objects with very coarse granularity [19, 11], or if they did have better precision, they restricted assignment and required annotations [16]. More recently, custom alias analyses have been used to allow instance locks without annotations, falling back to static locks if aliasing is uncertain [7, 9], but the choice of instance/static locks was still on a per-object basis. Only recently [5, 3] have multi-granularity locks been used to allow the instance/static distinction on a per-atomic-section basis. Also, only recently have analyses begun to use translation techniques to handle assignment [5, 3] without coarsening the lock granularity. One key difference between our approach and [3] is that they force termination with a simple static bound, whereas we represent cycles accurately within an NFA. The program analysis used by Khedker et al. [13] was very similar to ours, but there it is used to infer object liveness for the purpose of accelerating garbage collection. We believe our approach is the only one that prevents deadlock with a dynamic mechanism. The other approaches attempt to statically order lock acquisitions, falling back to static locks if this is not possible.

One contribution of Cherem et al. [3] is a framework for specifying and combining lock inference approaches. Although they distinguished between dereferencing and object offset, whereas we just use fields, we believe our NFA approach can be represented in this framework. However, it is less useful because our approach is monolithic, supporting all the features we need without needing to be combined with other analyses. Another contribution of the above is a notion of correctness that is intuitively similar to ours, but specialised for their approach.

5 Conclusion

We have proved that our lock inference infers at least the objects accessed by the body of an atomic section. Since we use a two phase locking discipline, we therefore know that the atomic section executes atomically. It may not be clear why we used a single-threaded semantics; e.g., how can we be sure that other threads won't change the state so that the locked object was not the one accessed? We get this property for free from the locking discipline, since we lock any shared memory accessed.

We assume that all shared memory accesses occur in atomic blocks, a property that all the more efficient atomic section implementations require. In practice, both lock inference and transactional memory would greatly benefit from a type system to distinguish between thread-local and shared memory (e.g. [17, 1]), since thread-local state need not be protected by locks or logged by a transaction. Some may argue that such a type system would restrict programmers, but choosing between shared and thread local memory is already an important design decision that programmers often document with comments. The type system would enforce these comments by ensuring that local memory is never shared, and shared memory is always accessed within atomic sections.

Our analysis supports read/write locks for better precision, but we omitted this detail for simplicity. We could extend the proof to additionally require that the path graph node where an address is represented is the same as the CFG node where the access occurs during execution. Knowing the statement is sufficient to know what kind of access occurred there. The path graphs already store this information, but in the proof presented here we "flattened" this detail.

Our previous paper gave an implementation that released locks as early as possible, by calculating the difference between the CFG edges either side of each node. It would be good to prove that this is correct and we hope to do that soon. We are working on an implementation for Java, using Soot, that correctly handles features like arrays, functions, exceptions, constructors, finalizers, static initializers, and static members. Extending the proof to cover these constructs is also left as further work.

We believe that using ownership types would give us much better granularity when converting path graphs to actual locking code [4]. For instance, we could lock only the nodes that are iterated through, in a list, rather than every node in the system. We have avoided using ownership types until now because of the type annotations required. Inferring ownership type annotations would therefore be a useful subject for further work. We believe that with ownership types and knowledge of thread locality, we can infer locks with granularity equivalent or better than manual locking, and additionally with guaranteed correctness.

Acknowledgements We thank the anonymous reviewers for their helpful comments.

References

1. Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 63–74, New York, NY, USA, 2008. ACM.
2. David Aspinall. Proof general: A generic tool for proof development. In *TACAS '00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 38–42, London, UK, 2000. Springer-Verlag.
3. Sigmund Cherem, Trishul Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *Proceedings of the ACM Conference on Program Language Design and Implementation (PLDI 2008), to appear.*, Tucson, Arizona, June 2008.
4. Dave Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universe Types for Race Safety. In *VAMP 07*, pages 20–51, September 2007. `http://pubs.doc.ic.ac.uk/universes-races/`.
5. Dave Cunningham, Khilan Gudka, and Susan Eisenbach. Keep Off The Grass: Locking the Right Path for Atomicity. In *Compiler Construction 2008*, volume 4959 of *Lecture Notes in Computer Science*, pages 276–290, April 2008.
6. Dave Cunningham. Isabelle/HOL file containing correctness proof of lock inference, 2008. Available online at `http://www.doc.ic.ac.uk/~dc04/ftfjp08.thy`.
7. Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 291–296, New York, NY, USA, 2007. ACM Press.
8. K.P. Eswaran, J. Gray, R.A. Lorie, and I.L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM*, 19(11):624–633, 1976.
9. Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. Component-based lock allocation. In *PACT'07: Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, September 2007. To appear.
10. 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.
11. Michael Hicks, Jeffrey S. Foster, and Polyvios 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.

12. John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
13. Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *ACM Trans. Program. Lang. Syst.*, 30(1):1, 2007.
14. E. Koskinen and M. Herlihy. Dreadlocks: Efficient Deadlock Detection. In *Proceedings from the 20th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '08), Special track on Multicore*, June 2008. To appear.
15. David Lomet. Process structuring, synchronization, and recovery using atomic actions. *ACM SIGOPS Operating Systems Review*, 11(2):128–137, 1977.
16. B. McCloskey, F. Zhou, D. Gay, and E. Brewer. Autolocker: synchronization inference for atomic sections. *ACM SIGPLAN Notices*, 41(1):346–358, 2006.
17. Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 51–62, New York, NY, USA, 2008. ACM.
18. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.
19. Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. *SIGPLAN Not.*, 41(1):334–345, 2006.