University of London Imperial College of Science, Technology and Medicine Department of Computing

Locking Atomic Sections

David Cunningham

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing of the University of London and the Diploma of Imperial College, April 2010

Abstract

Despite the growth of multi-core technology, a large subset of programmers still have neither the tools nor the language features to write concurrent programs effectively. When many programmers are simultaneously working on large, complex code bases with unknown bugs and insufficient documentation, they are unlikely to be very productive unless the programming language affords them the power to create abstractions and modularise their code. Applications are formed by composing independent modules. However, concurrent algorithms do not compose well. Using classical language features such as locks and compare-and-swap operations, one either has to rewrite the algorithm from scratch or expose its internal implementation to the client.

Atomic sections have simple semantics and allow programmers to compose concurrent algorithms. However, atomic sections are typically implemented using transactions. Thus, they lack performance and are restrictive to programmers. Here, we explore the implementation of atomic sections using locks instead of transactions.

Firstly, we give a type system that uses universe type annotations (a form of ownership types) to verify that programmer-supplied locks are sufficient to give race safety and implement programmer-denoted atomic sections. We prove that the system prevents races during execution, and give some extensions including a type system for checking that atomic sections have been implemented correctly.

Secondly, we develop a program analysis that facilitates implementing atomic sections (denoted by the programmer) with automatically-inserted locks. There is no annotation overhead since everything is inferred. Since the implementation takes complete responsibility for the locking protocol used, the programmer does not need to know that locks are being used internally. This means we must ensure the program does not deadlock, for which we use runtime deadlock detection and rollback. Our program analysis builds on related work by having improved accuracy, a nice characterisation of recursive object structures, and a machine-checked proof of correctness.

Contents

1	Introduction						
	1.1	Lock Checking	11				
	1.2	Lock Inference	16				
	1.3	Outline of Thesis	20				
2	Bac	Background					
	2.1	The Joy of Sequential Programming	21				
	2.2	Parallel Paralysis	24				
	2.3	Locking The State Away	25				
	2.4	Declarative Multi-Threaded Encapsulation	33				
	2.5	A Generalised Lock Discipline	34				
	2.6	Granularity	37				
	2.7	Deadlock Detection	38				
	2.8	Chapter Summary	40				
3	Loc	k Checking	42				
	3.1	Introduction	42				
	3.2	An Example of Universe Types and Race Safety	43				
	3.3	Related Work	47				
	3.4	Formal Preliminaries	50				
		3.4.1 Syntax and Semantics	51				

		3.4.2	Encodings
	3.5	Unive	rse Type System
	3.6	Race S	Safety
		3.6.1	Static Types for Race Safety
		3.6.2	Run-time Type System
		3.6.3	Typing Algorithm
	3.7	Implei	mentation Issues
		3.7.1	Distinguishing Reads and Writes
		3.7.2	Single Object Locking
		3.7.3	Atomicity
	3.8	Chapt	er Summary
1	Loc	k Infor	once 90
T	4 1	Introd	uction 90
	4.2	Relate	de Work 91
	1.2	4 2 1	Transactions 91
		422	Lock Inference 05
		4.2.2	Comparison 08
	13	Patha	raph Analysis 101
	4.0	Forms	lism 105
	1.1	1 / 1 1	Suptay and Somantics
		4.4.1	Analysis Transition Functions
		4.4.2	Soundpose 100
		4.4.5	Agging meaning to path graphs
		4.4.4	Assigning meaning to path graphs
		4.4.0	
		4.4.0	Multiple Inreads
	4 5	4.4.7	Algorithm
	4.5	Inserti	$\ln g \perp ocks$

		4.5.1	Inferring Locks from a Path Graph	. 115
		4.5.2	Iteration and Granularity	. 117
		4.5.3	Deadlock	. 118
		4.5.4	Parole	. 119
	4.6	Additi	onal language features	. 124
		4.6.1	Arrays	. 124
		4.6.2	Casts	. 127
		4.6.3	Additional Control Flow	. 128
		4.6.4	Splitting the Atom	. 128
	4.7	Case S	δ tudy	. 132
		4.7.1	AOLserver	. 132
		4.7.2	Porting AOLserver	. 133
		4.7.3	Experiment	. 135
		4.7.4	Results	. 137
		4.7.5	Evaluation and Future Directions	. 142
	4.8	Chapt	er Summary	. 148
5	Con	clusio	ns and Further Work	150
	5.1	Summ	ary	. 150
	5.2	Evalua	ation	. 152
		5.2.1	Expressiveness	. 152
		5.2.2	Scalability	. 154
		5.2.3	Performance	. 156
	5.3	Future	e Case Studies	. 157
	5.4	Final	Words	. 157
\mathbf{A}	Pro	ofs of 3	Race Safety	159
В	\mathbf{Cor}	rectne	ss of Path Graph Analysis	204

List of Figures

1.1	Example program showing heap hierarchy structure	14
1.2	Example program showing what locks implement each atomic section	18
2.1	Example set implementation using locks for synchronisation \ldots \ldots \ldots	27
2.2	Fragment of code showing an incorrect attempt at synchronisation \ldots .	29
2.3	Fragment of code showing another incorrect attempt at synchronisation .	30
2.4	External synchronisation for avoiding deadlock	31
2.5	External synchronisation for avoiding interference	31
2.6	Using the $\verb+atomic+$ annotation to specify the encapsulation we need	35
3.1	Example program showing heap hierarchy structure	45
3.2	Example code in the systems of Flanagan et al (left) and Boyapati et al	
	(right)	49
3.3	Source program definition	51
3.4	Run-time state and syntax	52
3.5	Small step operational semantics	52
3.6	Example code to demonstrate the semantics of method call	53
3.7	Static universe type system	57
3.8	Universe composition and decomposition	58
3.9	Run-time universe type system	59
3.10	Static race safety type system	64

3.11	Example	67
3.12	Run-time race safety type system	69
3.13	Definition of <i>Reachable</i>	69
3.14	Pseudocode Typing Algorithm	78
3.15	Single object locking	82
3.16	Static atomicity type system	89
4.1	Example of an atomic section where aliasing is a concern.	102
4.2	Example of an atomic section with an iteration over objects	102
4.3	Path graphs representing the accesses of the previous examples	104
4.4	Syntax and Semantics of Execution Model	106
4.5	The analysis	108
4.6	Pseudocode Analysis Algorithm	114
4.7	The same path graph after DFA transformation	116
4.8	Result of path graph analysis applied to early release	120
4.9	Implementing wait/notify with preempt	129
4.10	CFG of an atomic section that calls wait()	131
4.11	Top-level diagram of tclvar.c data structures	135
4.12	Locks inferred by our analysis to the AOLserver tclvar.c fragment	138
4.13	The GetObjCmd function written in our toy language	139
4.14	The LockArray function, also in class ServPtr	140

Statement of Originality

Chapters 1, 2, and 5 are my own work, except the 'student' example in Chapter 1 is Susan Eisenbach's invention. Chapter 3 is formed from an unpublished workshop paper [17], and is my own work except some parts of 3.6.1, which were heavily-reworked by Sophia Drossopoulou, and another appearance of the 'student' example from Chapter 1. David Flynn helped me find some real-world examples of race conditions. Peter Müller, Werner Dietl, and Tristan Allwood provided comments and suggestions to earlier drafts of the paper. The formalism and proof is my own work. Chapter 4 is a fusion of two published papers [19, 18] and some additional work by myself. It benefits from the comments and suggestions of Khilan Gudka and two sets of anonymous reviewers. The toy language used to facilitate the case study was designed and implemented by Khilan Gudka. All chapters benefited from criticism and suggestions from Sophia Drossopoulou and Susan Eisenbach.

Acknowledgements

I have been privileged to work with my supervisors Sophia Drossopoulou and Susan Eisenbach. They have dispensed an enormous amount of valuable advice and guided me through many trials during my 4 years as their student. In particular from Sophia I have learnt the importance of conciseness and simplicity when presenting formalisms; it is not mere cosmetics as I at one point claimed! Thanks to Susan's good judgement I always felt comfortable on the PhD program and it was always clear what objectives I needed to pursue. My supervisors have been very accommodating of me, and I have always been grateful for their deep knowledge and insight.

I must thank the Engineering and Physical Sciences Research Council who funded me for 3 years under the Doctoral Training Account program.

I greatly enjoyed working with Khilan Gudka, whose enthusiasm and hard work inspired me to work hard myself. Solving hard problems on one's own can become depressing. But discussing and solving the same problems with another who shares your enthusiasm is a most positive and rewarding experience that I would be glad to repeat with him at any time.

I would like to thank Peter Müller, Werner Dietl, Cormac Flanagan, and Chandrasekhar Boyapati for responding to my emails about universe types, ownership types and race safety. Also I am grateful to the SLURP group, particularly Tristan Allwood who gave me many last-minute corrections on my paper submissions. SLURP is a diverse group of people with varying tastes in programming languages and this has led to many stimulating discussions. Vijay Saraswat also deserves thanks for giving me consideration as I was writing my thesis at the same time as working at IBM on the X10 project.

Since this thesis contains elements of type theory I ought not to forget the positive influence I received from Steffen van Bakel, during my first months at Imperial College. He essentially introduced me to type theory, and I remember his course even today. Also not to be forgotten is Herbert Wiklicky, who acquainted me with another major part of this thesis, program analysis, and also abstract interpretation, which has given me a powerful mental framework for understanding program analyses. Finally, whilst supervising my MSc project, Sophia Drossopoulou patiently taught me how to reason about type systems for object-oriented languages. My continued study in the field owes a lot to this initial experience.

My long-term girlfriend Katherine, who I met in the first few weeks of my PhD, has always been supportive and helpful even though I have spent many hours (often at the wrong time of day) stuck in the glow of the computer screen. Her warmth, loyalty, and sense of fun have given me many years of happiness. Finally, I owe many thanks to my parents whose sacrifices gave me the best schooling available as well as lending me much financial support during my time in university.

Chapter 1

Introduction

It is commonly known [80] that future execution speed improvements in typical consumer grade computers will come from adding more independent processing elements, rather than increasing the clock rate of individual cores. This is due to physical limitations that are being approached in microprocessor design [10]. As such, software will increasingly need to be multi-threaded in order to make full use of the available hardware. Computationally intensive software that remains single-threaded will not perform as well as competing multi-threaded software that is able to to exploit multi-core. Conventionally, the majority of programmers use threads and locks to write multi-threaded programs. Unfortunately, programmer-inserted locks do not scale to the kinds of massively large programs that are now common place.

When programming multiple threads using locks, traditional mechanisms and software engineering methodologies no longer provide programmers with the encapsulation and modularity they need to write large programs. Encapsulation and modularity are essential when many programmers are working on the same program and thus the knowledge of the system is distributed. Further factors, e.g., bugs and undocumented code, help create an environment where no single person has comprehensive knowledge of the system. Separating unrelated pieces of code into modules and hiding implementation details within these modules would normally allow software to be written in such an environment, but when locks are involved this is no longer the case. This problem has prompted the search for more tractable language support for concurrency control.

Atomicity [60] is a property a block of code is said to have when its behaviour is unaffected by other threads. The term derives from the idea that other threads are not aware of the internal steps of computation, only the complete undivided effect of the block of code. We will sometimes say a block of code is *atomic* if it has the property of atomicity. It is easy to reason about the behaviour of atomic code in a multi-threaded program since the effect of other threads can be disregarded and the problem is reduced to that of a single-threaded program. We consider the core activity of lock-based programming to be the search for a set of *lock annotations* that ensures the atomicity of a given block of code. This core activity is hard and mistakes lead to bugs that are very hard to reproduce and locate. However, if we know the programmer's high-level intent, in terms of atomicity, we are in a position where we can provide them support.

There are a number of methods where blocks of code that are intended to be atomic are annotated as such. One possible use for these annotations is to check whether programmer-supplied lock annotations are sufficient to ensure the intended atomicity, otherwise providing useful error messages. Another approach is to enforce the atomicity automatically, without input from the programmer. This thesis attempts to advance the state of the art in both approaches and we will now discuss them individually.

1.1 Lock Checking

Lock checking is where programmer-supplied locks are checked to make sure they are sufficient to ensure a block of code is atomic. Usually lock checking tools require blocks of code to be annotated as **atomic** when that is the intention, as well as requiring the programmer to use locks to engineer the atomicity. Such systems often use novel type annotations that allow the programmer to express relationships between objects and locks that are invariant at run-time. The relationship defines the lock that should be taken to

CHAPTER 1. INTRODUCTION

protect a given object. For example, a linked list with all its nodes may be protected by a given lock stored in the encapsulating linked list object. The type annotations and lock insertions are checked by the compiler to ensure they are correct and mutually consistent, which then implies the atomicity has been successfully attained.

Previous work [31, 11] uses ownership type annotations [14, 72] (or "guard" annotations that strongly resemble ownership) to establish an invariant relationship between objects. If locks are just special kinds of objects then such relationships can be used to statically define which lock protects an object referenced by a particular variable. The system thus knows which locks to take to protect a block of code just by observing the types i.e., without precisely knowing the objects that will be accessed. This requires object types to have ownership annotations as well as the atomic annotations and the locking code itself. Additionally, classes can have ownership parameters that allow the expression of reusable data structures employing complex ownership structures, albeit at the expense of heavier type annotations.

Universe types [67, 24] are a simple yet powerful form of ownership types, used in the JML tools [58]. When compared with other ownership type systems, universe types let programmers succinctly specify the topological relationship between objects using just a few keywords. As such, the owner of an object is implicitly understood by the type system, and implicitly stored by the run-time environment. Thus, programmers need not explicitly declare them and the annotation burden is lessened.

We developed a type system for *race safety* using universe types to partition the heap. Race safety is a slightly weaker property than atomicity and we ensure it by requiring all accesses to be protected by lock acquisitions. We prove the system is sound and provides race safety. We then show how the system can also ensure atomicity with a small extension, which additionally constrains the nesting of lock acquisitions. As is standard, we treat objects sharing the same owner as guarded by the same lock. At run-time we associate this lock with the owner. All objects have an implicit reference to their owner.

Universe types also allow references to objects whose owner is unknown through the annotation any [25]. This is not supported by [31, 11]. Initially, the presence of any was a challenge for us, but turned out to increase the expressiveness of our language. The any annotation allows the expression of data structures that contain objects with various owners. Use of such data structures does not require us to compromise the design of other data structures in our program. This improves upon [11, 31], where in particular one sometimes has to alter the design of unrelated data structures so that they take their lock once for each access in an iteration. Iterating over the unrelated structures can be atomic in our system but could not be made atomic in [11, 31]. We will give a detailed comparison later (§3.3).

When the type does not indicate the owner of an object, we use paths as an alternative mechanism to guarantee correct synchronisation. Paths are sequences of field accesses starting from a variable, e.g., $\mathbf{x.f.g}$ and can be arbitrarily long. We use an effects system where these paths need not be comprised entirely from final variables/fields as would be required in previous work [11, 31]. Programmers would find this requirement restrictive. Related work also required locks to have coarse granularity even if only a single object was accessed. It was straightforward to extend our system with single object locks.

Figure 1.1 gives an example of a program that is protected by our type system. There is a pair of linked list implementations, **Dept** and **Hall**, and two iterations over the objects in the linked list, **releaseMarks** and **cleanRooms**. We will discuss this example in detail later (§3). For now, notice the **rep**, **peer**, and **any** universe type annotations on object reference types. These are used to specify the ownership structure that will be maintained by the program at run-time. Objects that are encapsulated by the current object are denoted with **rep**, and if an object is at the same level of encapsulation then it is a **peer**. We have also given a snapshot of the heap that shows objects of the various classes, the references between them (arrows), and also the ownership relationships, which are

```
1
     class Student {
            int mark;
 \mathbf{2}
 3
            boolean roomClean
 4
    }
 5
 6
     class Dept {
            rep DeptStudentNode first;
 7
            void releaseMarks () { ... }
 8
    }
 9
10
    class DeptStudentNode { // Closed list
11
12
            peer Student s;
13
            peer DeptStudentNode next;
14
    }
15
    class Hall {
16
17
            rep HallStudentNode first;
18
            void cleanRooms () { ... }
    }
19
20
21
    class HallStudentNode { // Open list
22
            any Student s;
            peer HallStudentNode next;
23
    }
24
25
26
    void releaseMarks () {
27
            sync (this) {
28
                    rep DeptStudentNode i = this.first;
29
                    sync (i) {
                           while (i!=null) {
30
                                   i.s.mark = ...;
31
                                   i = i.next;
32
            }
33
    }
                    }
                           }
34
35
    void cleanRooms () {
36
            sync (this) {
                    rep HallStudentNode i = this.first;
37
                    sync (i) {
38
39
                           while (i!=null) {
40
                                   sync (i.s) {
41
                                           i.s.roomClean = true;
42
                                   }
43
                                   i = i.next;
44
    }
            }
                    }
                           }
                                                              (8) Hall
       (1) Dept
      (2) DeptStudentNode
                               (3)
                                   Student
                                                       (9)
                                                           HallStudentNode
                                             ┥
      (4) DeptStudentNode
                               (5)
                                   Student
                                                       (10)
                                                           HallStudentNode
                                                       (11) HallStudentNode
                   (6) Dept
                                  Student
                              (7)
```

Figure 1.1: Example program showing heap hierarchy structure

denoted by nested boxes. This ownership structure is used by the type system to ensure the programmer-supplied locks given by the **sync** statements are sufficient to eliminate race conditions.

Referring to the heap diagram, the linked list comprised of HallStudentNode objects at addresses (9), (10), and (11) have the same owner, the Hall at (8). Thus their next field defined at line 23 has type peer. On the other hand their s field defined at line 22 is used to store references to students with various owners and thus its type annotation must be any. We call this an *open* list.

The linked list of DeptStudentNode objects at (2), and (4) is similar but its s field references students owned by the same object 1 as the nodes themselves. Thus the annotation used at line 12 is peer, the same as its next field. Since the students are more contained, we call this a *closed* list. We will formalise these type annotations and revisit open and closed lists later (§3).

The types therefore reflect the run-time ownership hierarchy, and we use this to check the locks. Our sync statement is used by the programmer to specify where locks should be acquired and released. It is similar to the Java synchronized block except that our synchronized block is designed to work with the object hierarchy. To protect an object, one should sync on any object that has the same owner. At line 30 we iterate over a closed list of students such as illustrated by the objects (2), (3), (4), and (5), setting their marks to an unspecified quantity. Since this is a closed list, we know from the soundness of the type system that all these students are owned by the same object as the first node. Thus it is sufficient for race safety to lock just the first node for the duration of the loop, which we do at line 30.

To ensure the full property of atomicity, one also needs to constrain the nesting of sync blocks. For instance, the body of releaseMarks is atomic whereas cleanRooms is not. The reason is that the additional sync within the loop at line 41 is executed many times during the loop and at run-time the locks are not properly nested. The consequence

of this is that other threads can observe a state where only some of the students rooms have been cleaned. In real code, this could lead to a race condition. This is not the case with **releaseMarks**. As far as other threads can observe, the marks are released instantaneously with no opportunity to observe some students' marks but not others.

In later sections we will give in full detail the universe type system we use. We will then give full details of the the race safety type system that uses the universe type system including proofs of correctness. Finally we show how to extend the type system to constrain the nesting of locks in order to ensure the full property of atomicity.

1.2 Lock Inference

When programs only contain blocks that are annotated as atomic, without explicit locking, we call them *atomic sections*. Ensuring that atomic sections obey the atomicity property is the responsibility of the compiler and run-time system and transparent to the programmer. This considerably reduces the burden of multi-threaded programming. This transparency allows many different implementation strategies. Although atomic sections allow the programmer to pretend a block is executed in the absence of other threads, such an implementation would have poor performance. Allowing non-interfering threads to execute in parallel with the atomic section is therefore a transparent optimisation. Current implementations of atomic sections do this using either transactional memory [1, 66] or lock inference [49, 41, 13]. The more efficient implementations of transactions and all lock inference implementations, including ours, prohibit the access of shared objects outside of atomic sections.

Transactional memory rolls back the state of an atomic section, to the entry state of the block, when the state is detected to be corrupted by interference. This means I/O or system calls cannot be allowed in atomic sections, a restriction that cannot in general be hidden from the programmer. When a state is rolled back, the cycles that contributed to that state are wasted. There is further waste because the machinery that detects interference and facilitates the rollback incurs considerable run-time overhead.

Lock inference is a compile-time technique that inserts locking code into the program to ensure the atomicity of the atomic sections. Lock inference does not restrict I/O or require any run-time mechanisms (except the locks themselves), but since it relies on static approximation of program behaviour, it sometimes lacks precision. The more precise the inference, the more threads are allowed to execute in parallel.

Whatever mechanisms are used by the implementation, they must be designed carefully. Any emergent behaviours such as deadlocks or livelocks must be prevented or otherwise not exposed to the programmer, who should be able to use atomic sections free from implementation-specific constraints. Programmers should not have to design their code so that the lock inference can easily understand it, nor should they be forced to obey the restrictions of transactional memory.

Our lock inference algorithm 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 decide which locks to insert manually. We also use a run-time deadlock detection technique that permits using locks of a finer granularity. We give a formalisation of our algorithm and prove it correct.

We use a data-flow analysis, at link or JIT time, to infer the object accesses performed by each atomic section. This analysis needs to traverse any code that might be invoked by the block in question, so the whole program is needed. When the analysis terminates, we know, at each program point, the set of objects that are accessed from that point until the end of the atomic section. The inferred accesses then need to be translated into locks. We believe our representation of accesses is novel, and the most precise to date. As a simplification all objects after construction are shared.

We try and use one lock per object, or *instance* locks, where possible, so that the parallelism can scale with the data. Sometimes code can access a statically unbounded number of objects. This happens during iterations over objects, and when we approximate

```
1
    class HashEntry {
       string key; Object val; HashEntry next;
2
       HashEntry (string key) { this.key = key; }
3
4
       HashEntry findKey (string key) {
           if (this key==key) {
5
6
               return this;
7
           } else {
8
               if (next==null) { return null; }
9
               else { return next.findKey(key); }
10
   } } }
11
    class HashTable {
12
13
       HashEntry[] buckets;
       HashTable() { buckets = new HashEntry[100]; }
14
15
       int index(string key) {
16
           int hash = key.hash % buckets.length;
           if (hash<0) { hash = hash + buckets.length; }</pre>
17
18
           return hash;
19
       }
20
       HashEntry createHashEntry(string key) {
          HashEntry entry = new HashEntry(key);
21
22
           int index = index(key);
23
           entry.next = buckets[index];
24
           buckets[index] = entry;
25
           return entry;
26
        }
27
       HashEntry findHashEntry(string key) {
28
           HashEntry entry = buckets[index(key)];
29
           if (entry==null) { return null; }
30
           return entry.findKey(key);
31
   }
       }
32
33
    class Client {
        string name; HashTable allClients; Client interlocutor;
34
35
        Client (HashTable allClients, string name) {
36
           this.allClients = allClients; this.name = name;
           atomic { //locks: {allClients, !allClients.buckets}
37
38
               HashEntry e = allClients.createHashEntry(name);
39
               e.val = this;
40
           }
41
           run();
42
       }
        string read() { return ""; }
43
44
       void accept(Client source, string msg) {
45
           atomic { //omitted
               print "<"+source.name+">u--->u<"+name+">u"+msg;
46
47
       }
}
48
       void run() {
49
           while (true) {
50
               string msg = read();
51
               if (msg=="connect") {
                  string name = read();
52
                  atomic { //locks: {HashEntry}
53
54
                      HashEntry e = allClients.findHashEntry(name);
55
                      interlocutor = (Client) e.val;
56
               } }
57
               if (msg=="send") {
58
                   string cargo = read();
59
                  atomic { //locks: {this, interlocutor}
60
                      interlocutor.accept(this,cargo);
               }
61
                  }
62
               if (msg=="disconnect") {
63
                  atomic { //locks: {!this}
64
                      interlocutor = null;
   65
```

Figure 1.2: Example program showing what locks implement each atomic section

an array index expression. In such cases, we use the type of the accessed objects to take a *multilock* which guards all instances of that type and sub-types. The semantics of multilocks require that if one thread has taken the multilock on an object, any other threads attempting to lock a subordinate instance of that multilock will be blocked until the multilock is released. We also distinguish between read/write accesses, and we use read/write locks to allow multiple parallel reads. We need re-entrant locks in case objects happen to be aliased at run-time, causing the same lock to be taken twice.

The code in Figure 1.2 is for an instant messaging system, where a client can send a stream of messages to another client, by name, through a central server. The Client constructor registers a new client in a centralised hash table of clients. This must be an atomic operation in order that the uninitialised value of the hash entry e.val is not visible to other threads. Our system infers two locks – the hash table itself (for reading), and the array of buckets inside the hash table (for writing). Although the hash entry is modified, it is a newly constructed object and thus cannot be seen by other threads. We give the inferred locks as comments in the code, where ! preceding the lock denotes a write lock.

The atomic section starting on line (53) iterates (HashEntry.findKey is recursive) through a list of hash entries, and thus the analysis has to lock (for reading) the multilock that subsumes every hash entry. Atomic sections starting on lines (59) and (63) simply access a pair of clients and a single client object, and the locking reflects this. The atomic section starting on line (45) is only ever called from within another atomic section starting on line (59), so does not have any locking code inserted into it. If it were also called from a *pre-emptive* context (i.e. from outside an atomic section), we would have a problem inserting locking code into it, because this code would also be executed by the atomic section starting on line (59). We solve this problem by duplicating functions if they are called from both atomic and pre-emptive contexts.

Our program analysis gives us information about what locks should be held at every

program point in the atomic section. This means we have enough information to release locks straight after the last access of any objects they guard. Releasing locks early reduces contention, at no extra cost.

1.3 Outline of Thesis

In Chapter 2 we give some background. We describe locks, and we recall common practice when using them to solve concurrency problems. Using a running example, we discuss the interplay between modularity, encapsulation, and locks. We then show how atomicity can be used to effectively convey the notion of encapsulation in the context of multiple threads. We then survey a selection of related work that uses the notion of atomicity to enhance programming language technology. We give more detailed descriptions of lock checking, lock inference, and transactional memory.

In Chapter 3 we introduce our lock checking type system, which uses universe types to partition the heap. This is initially presented as a type system that prevents race conditions, but later extended to ensure atomicity. The extension is only to ensure locks are nested correctly. We also give an extension that allows locking both single objects and groups of objects, with appropriate exclusion. We prove soundness and race safety of the type system.

In Chapter 4 we give a program analysis that can be used to automatically insert locks in order to ensure atomicity of a given atomic section. We prove soundness of the analysis mechanically. We give the lock insertion scheme and explain how we handle deadlock, early release of locks, and some more advanced language features like arrays, casts, and message passing. We finally do a case study on some real code. We discuss the quality of the inferred locks and how they can be further improved.

Finally we present concluding remarks in Chapter 5. Appendix A contains the detailed proofs from Chapter 3, whereas Appendix B lists the source code for the Isabelle/Proof-General proof from Chapter 4.

Chapter 2

Background

This chapter gives background that is common to lock checking (§3) and lock inference (§4). We will defer discussion of existing work that is specific to lock checking and lock inference to the appropriate later chapter. We discuss encapsulation, modularity, information hiding, locks, locking discipline, and atomicity. We advocate the importance of these concepts to motivate their further exploration and development.

2.1 The Joy of Sequential Programming

Programming single-threaded software is comparatively easy. Through the evolution of programming languages and methodologies, many programmers can now work together and create large software systems. Of critical importance to supporting this power is the *encapsulation* of program behaviour within abstract objects and operations, together with the ability to intuitively *compose* such abstractions to create higher level abstractions. In this section we will study this phenomenon.

For example, although variables are fundamentally limited to primitive types such as integer, one can compose these primitives types into structures, and continue composing such structures indefinitely to create arbitrarily high-level types. Such types can often be manipulated opaquely – without needing to know their internal layout. Equally, although fundamental program behaviour is limited to simple assembly instructions such as ADD or JMP, one can compose these operations into functions, and continue composing such functions to create arbitrarily high-level operations. Such operations can be invoked opaquely – without needing to know how their behaviour is internally implemented. In many ways, the act of programming is fundamentally an act of composition – creating new functionality by defining high-level abstractions in terms of more primitive components.

The ease of programming is heavily affected by the complexity of the system being developed. As software grows, and the number of potential interactions between different aspects of the software increases exponentially, it is easy for complexity to get out of hand and slow down development. The value of encapsulation is that it keeps the complexity under control. Using encapsulation, we can create high-level compositions that are simple to compose further, despite being internally complex. We do this by *hiding* internal state and presenting just a simple interface. By hiding internal state, we guarantee no interference from unrelated parts of the software, thus getting back control of complexity. In principle this allows us to compose indefinitely, creating arbitrarily complex functionality while keeping the code at a manageable level of complexity.

For example, consider a Set object that encapsulates its internal representation but exposes the operations insert and delete. Let us consider this a low-level abstraction, and use it to create something more high-level. Consider a computer game where players are divided between teams *red* and *blue*. Players are allowed to change teams to keep the game balanced. The system might have an object TeamManager that represents the teams and exports the operations getRedTeam, getBlueTeam, and switchPlayerTeam. The implementations of the first two operations each return a reference to the respective underlying Set, whereas switchPlayerTeam inserts the specified player into its new team and then deletes it from its old team.

It would be reasonable to base the design of the rest of the system on the assumption that a player is in exactly one team. Note that although switchPlayerTeam breaks this

CHAPTER 2. BACKGROUND

invariant temporarily (when the player is in both teams), it is restored by the time the operation completes. The programmers developing the rest of the system do not need to know how the switchPlayerTeam operation is implemented internally i.e., they do not need to know that the invariant is ever broken. Dividing the code into modules and encapsulating module implementations has empowered these programmers with the ability to *reason locally* about the system. Not only can they write correct code without having to research the precise behaviour of switchPlayerTeam, but the code they write is robust to changes to that implementation. One such change could be to reverse the order of the Set operations, i.e., deleting, then inserting the player. This would cause a different temporary state where the player is in neither team, but the rest of the system will be oblivious and behave correctly.

A key factor that contributes to the effectiveness of modularity and encapsulation is that the code implementing the rest of the system is executed before or after, but never during the execution of switchPlayerTeam. Thus the intermediate state of the system during the execution of switchPlayerTeam can be as broken as required, e.g., allowing the player to be in both teams at once, or in no team at all. The intermediate state will be hidden: The rest of the system will not be allowed to observe the state until the operation has finished, whereupon the consistency is restored. Thus the programmers writing the rest of the system do not need to concern themselves with this broken state.

The benefits of hiding intermediate state from the rest of the system may seem obvious, and probably many programmers take it for granted. However, we will now see that when we introduce concurrency, the whole situation falls apart. Traditional notions of encapsulation and modularity only hide intermediate states from other operations by *the same thread*. They do not hide intermediate states from *other threads*. Therefore even in the presence of good software engineering practice, we no longer have the desirable properties that make working on large systems tractable. Suddenly, the correctness of a given part of the system depends on intimate implementation details of every other part of the system. If conventional modularity and encapsulation is not sufficient, perhaps we can add something new; something that will hide our intermediate state from other threads as well. This thesis is about how we can implement programming language features that allow us to keep the benefits of modularity and encapsulation even when we move to a multi-threaded system.

2.2 Parallel Paralysis

If we have multiple threads of execution in our red/blue teams example (§2.1), we have to consider the possibility that one thread will be executing switchPlayerTeam while another thread is examining the contents of the team sets. Now the second thread can observe the intermediate state of switchPlayerTeam i.e., observe that a player is in both teams. This is called *interference*. It is an unwanted thread-to-thread interaction that completely bypasses our careful use of modularity and encapsulation.

One solution is to write the rest of the system such that it is robust against interference. For instance, the system could detect if a player is in both teams, and assume it is in the red team. So the intimate implementation details of switchPlayerTeam force the whole of the rest of the system to be aware that a player can sometimes be in two teams at once. This means the code becomes more complicated, and the programmer needs complete system knowledge to write correct code. It also means the implementation of switchPlayerTeam cannot be changed, e.g., to delete the player from its current team first, thus leaving a state where a player is in neither team, without the rest of the system needing to be updated to reflect this change. When multiple programmers work together on large software systems, this is not feasible.

The interference described above is just one problem that may occur. The presence of another thread invalidates many of the assumptions that make writing single-threaded programs more tractable. Suddenly it becomes very difficult to write programs at all, since we have to be careful that any intermediate state used by an operation is not hazardous to some other thread. We also must be careful that any state we have recently observed has not been changed by some other thread since we observed it. Programming in these conditions is possible, using tricks like the solution in the above paragraph. Lock-free [23, 64, 83] and wait-free [45] algorithms depend heavily on such tricks to allow threads to operate on the same shared memory. These algorithms are often used in specialist situations, because they usually have exceptional performance, but they are not easy to write and they are hard to scale [81]. For the typical software engineer working on a large project without full knowledge of the system, an easier solution is to restrict execution so that we can get back some typically single-threaded guarantees about the state of the system.

2.3 Locking The State Away

Locks [57, 70] are a programming language primitive that allow us to restrict parallel execution in convenient way. Like a traffic light, the idea is to prevent some thread from proceeding past some critical point in the program until it is safe to do so. It does not matter whether the implementation of this idea involves the thread in question being unscheduled by the operating system, looping on the spot, or even continuing with a benign execution¹. The effect is that the useful activity of a thread is suspended until the *system*, i.e., every thread, is ready for it to proceed.

Threads are suspended both to prevent them invalidating an assumption held by some other thread in the system, and also to prevent them from executing when their own assumptions are invalidated. Thus each thread is *synchronised* with all the other threads: Threads are not allowed to observe each others' intermediate states, but as long as they do unrelated work, they may proceed in parallel.

To achieve such synchronisation with locks, the language offers primitives to create a $lock^2$, acquire a lock, or release a lock [38, 52]. If a thread tries to acquire a lock that

¹For example, it does something else in the mean time, or it computes a result that is then discarded

²Programs written in languages without garbage collection also need code to destroy locks.

has already been acquired by another thread, it is suspended until the lock is released. Locks are dynamic, their number is unbounded and can vary at run-time. However, each code block that is wrapped in an acquire/release of a particular lock instance can assume that other threads will not be executing code wrapped in an acquire/release of the same lock instance. Implementations that break the state can prevent the execution of any other code that cares about such breakage, and vice versa. Usually the programmer will associate a lock with each instance of the data structure holding the state in question. Before working with the data structure, code can get a reference to the lock and acquire it, releasing it when the work is done. Since the threads exclude each other's access to the data structure instance, this is called *mutual exclusion*, and the locks are often called *mutual exclusion locks*, or just *mutexes*. Locks allow us to extend the traditional notion of encapsulation to a multi-threaded context, because they hide the intermediate state from other threads.

In the case where a thread acquires the same lock twice without first releasing it, we will assume locks are *reentrant*, and the second acquire will have no effect other than to increase the number of releases required before the lock is available to other threads. Reentrant locks can thus be nested arbitrarily, but still provide mutual exclusion at the level of the outermost nest.

In simple situations, locks work well. Until now we have not discussed the implementation of the Set class used in the red/blue team example (§2.1). Unless the language provides sets as primitive types, they must be implemented in terms of more basic abstractions. For simplicity, let us assume that Set is implemented as in fig. 2.1. Note the type Lock and the variable 1, which holds the lock instance, the keyword lock, which acquires its lock parameter, and the keyword unlock which releases its lock parameter.

Without the locking code, interference is possible. Despite the fact that the Set implementation is modular and the details of its operations are encapsulated behind the insert and delete methods, two concurrent insert operations may result in a Set

```
class Set {
       protected final Lock l = new Lock();
       protected ArrayList list = new ArrayList();
       public void insert (Object o) {
              lock(1);
              if (!list.contains(o)) list.add(o);
              unlock(1);
       public void delete (Object o) {
              lock(1);
              int index = list.indexOf(o);
              if (index!=-1) list.remove(index);
              unlock(1);
       }
       public boolean contains (Object o) {
              lock(1);
              int index = list.indexOf(o);
              unlock(1);
              return index!=-1;
       }
}
```

Figure 2.1: Example set implementation using locks for synchronisation

containing a duplicate item that needs two calls to **delete** before it is fully removed. Two concurrent **delete** calls can result in the wrong element being removed from the **Set** as well as the correct one. Calling **delete** and **insert** concurrently can result in the wrong element being deleted. Other combinations are also possible. This example supports our earlier claim, that the number of possible interactions between different aspects of the code grows much faster than the program itself grows. However, due to the mechanisms discussed, using locks in the manner shown eliminates all of these possibilities and successfully hides the internal details of all the operations. The **Set** as presented is thus called *thread-safe*.

Let us review the cost, benefits, and risks of using locks to make classes thread-safe. The amount of code has increased but the new lines are idiomatic, so the cost is minimal. There is some additional state at run-time, but locks tend to be very small (often just one machine word) so this is usually negligible. There is also overhead in the lock and unlock operations but this is also quite small [7].

Using locks has successfully solved the multi-threaded problem for this class, but there are hazards that must be avoided. These are the mistakes we could have made in Set: Firstly, we could have forgotten to insert one or more lock acquisitions or releases. There are many problems that can be caused by this. Releasing a lock without first taking it usually raises an exception with most lock implementations, but forgetting to release a lock is more serious, as it prevents further lock acquisitions and causes the system to *deadlock*. This can usually be debugged by examining the state of the system to find out which lock is causing the problem, and tracking the operations on that lock. Languages like $C++^3$ and Java even have lexically scoped locks that ensure lock and unlock operations are always balanced.

Forgetting, or more likely not recognising the need for, both the lock acquisition and release operations is a more common problem, and of course will allow the aforementioned interference to occur. Debugging such situations is hard because interference tends to cause symptoms that do not directly imply their cause. Even if lock acquisitions and releases are correctly inserted, there is always the danger that the wrong lock might be used. In which case the synchronisation will not affect the right threads at the right time, and the hard-to-debug interference will creep back into the system.

So far, the application of locks to solve concurrency problems has been idiomatic. Let us now see if we can use locks to solve the concurrency issues with switchPlayerTeam. The code in fig. 2.2 attempts to use locks in a similar manner to fig. 2.1 to make the class thread-safe. While this will prevent two switchPlayerTeam calls from executing simultaneously, there is an additional problem, one which we did not face with Set. Because the rest of the code can acquire references to the Set objects through the getRed and getBlue methods, it is possible for them to access the sets without the TeamManager lock and thus encounter a state where a player is in both teams. Although examining a Set causes a lock to be acquired, this is a different lock to the lock acquired in switchPlayerTeam.

³For example, with boost::recursive_mutex::scoped_lock [6].

```
class TeamManager {
    protected final Lock l = new Lock();
    protected final Set red = new Set();
    protected final Set blue = new Set();
    public Set getRed () { return red; }
    public Set getBlue () { return blue; }
    public void switchPlayerTeam (Player player, Set from, Set to) {
        lock(1);
        from.delete(player);
        to.insert(player);
        unlock(1);
    }
}
```

Figure 2.2: Fragment of code showing an incorrect attempt at synchronisation

Locking 1 within the two getter methods will not help, since the lock will be released before the sets are examined.

Clearly we lack synchronisation, so let us try inserting locking code wherever else in the system the sets are accessed. Taking the TeamManager's lock would suffice, but we would need to export the lock instance from the class via some kind of public getLock method so that the rest of the system can acquire/release it. This means we need to remember which team manager the Set is associated with, and this may not be possible if we pass the Set to some library that is not aware of the TeamManager class. It is not good for modularity if code that uses the Set object to have to obey some discipline involving the unrelated TeamManager object. Another solution is to move all the setusing code into the TeamManager, and avoid references to the sets escaping into other parts of the system. However, this may hurt modularity, by having too much code in the TeamManager class.

A final solution is to abandon the TeamManager lock and exclusively use the Set locks for synchronisation. This means that the locking internal to Set is sufficient for the basic insert, delete, and contains operations. However, the switchPlayerTeam method becomes more complicated, as we have to acquire both teams' locks. Figure 2.3 assumes that the

```
public void switchPlayerTeam (Player player, Set from, Set to) {
    lock(from.getLock());
    lock(to.getLock());
    from.delete(player);
    to.insert(player);
    unlock(from.getLock());
    unlock(to.getLock());
}
```

Figure 2.3: Fragment of code showing another incorrect attempt at synchronisation

Set class provides access to its internal lock with a getLock method. This implementation shows some promise, as the locks should exclude any code that touches one of the Set objects. However, there is a more serious problem lurking in this design.

As we have discussed, threads acquire and release various locks as they execute. At any time, a lock will be acquired by at most one thread. Also, when acquiring a lock that is currently acquired by another thread, a thread must first wait for it to be released. A thread can therefore be waiting for another thread. The other thread may be waiting for a further thread, and so on. If this chain forms a *wait cycle* then all the threads in the cycle will wait for each other and the system will stop functioning. This is another form of deadlock, and it can happen with the code in fig. 2.3: Consider two players, a and b, and two threads concurrently calling switchPlayerTeam(a,red,blue) and switchPlayerTeam(b,blue,red) on the same TeamManager instance. It is possible for both threads to have acquired the lock on the first line, but be unable to execute the second line because they are waiting for each other.

In order to control such situations, the programmer needs to be careful about the order in which locks are acquired. If both threads had acquired the red team Set's lock first, there would have been no deadlock. However, this is not easy to achieve. When we write the code, we do not know what sets will be referenced by the from and to parameters, so we do not know in what order we should place the first two lines of switchPlayerTeam. In this case we could decide at run-time by comparing from and to using some global

```
public void someMethod () {
    lock (red.getLock());
    switchPlayerTeam(player,blue,red);
    unlock(red.getLock());
}
```

Figure 2.4: External synchronisation for avoiding deadlock

```
public void swapPlayers () {
    lock (red.getLock());
    lock (blue.getLock());
    switchPlayerTeam(player1,blue,red);
    switchPlayerTeam(player2,red,blue);
    unlock(blue.getLock());
    unlock(red.getLock());
}
```

Figure 2.5: External synchronisation for avoiding interference

ordering e.g., the address of the structures in memory. However, there may be other constraints on the ordering of locks, e.g., the second **Set** may have been contained within the first **Set**. In such a case it would be necessary to lock the outer **Set** first, in order to safely extract the inner **Set**.

A final option is to acquire the locks externally from switchPlayerTeam in the second thread, where the order within switchPlayerTeam is blue followed by red. Suppose the second thread executes someMethod from fig. 2.4. We have added code to lock the red team outside of the call to switchPlayerTeam. Since the lock is reentrant, this effectively disables the red team's lock acquire within the switchPlayerTeam and the effective lock order becomes red, followed by blue. Now both threads are taking the locks in the same order, and no deadlock occurs.

Consider fig. 2.5, which swaps the teams of a pair of players, keeping the teams balanced. Without synchronisation, other threads could see a state where both players were on the red team, which would clearly be unbalanced and unfair. This method is composed of two calls to switchPlayerTeam, but the synchronisation within switchPlayerTeam is not sufficient to stop the intermediate state between the two calls from being visible. Hence we have added more synchronisation externally, being careful to keep the red and blue order consistent with the rest of the program.

In both cases, we have required unrelated code to know the implementation details of **switchPlayerTeam**. Although locks have helped us encapsulate the intermediate states of our operations, we cannot encapsulate the locks themselves. This presents a software engineering problem.

We have introduced locks as a means of preventing interference. In some cases they work well. However, for the switchPlayerTeam implementation there were a number of subtleties that needed to be overcome. It is interesting to discuss the software engineering cost of these workarounds. Most of the solutions involved doing something external to the switchPlayerTeam implementation. Sometimes this was a reasonable requirement, like locking Set objects wherever they are accessed. On the other hand, the solution to avoid deadlock in fig. 2.3, by taking the red lock external to switchPlayerTeam, exposed the fact that switchPlayerTeam accessed the red and blue team sets. If we were to modify switchPlayerTeam so that it accessed the Player, we would have to adjust the rest of the program to take into account any additional lock acquisitions, to ensure the lock order is globally consistent.

Similarly, to avoid interference in fig. 2.5 we rely on the fact that switchPlayerTeam only accesses the red and blue team. If it were changed to also access the Player object, we would have to acquire the Player's lock for the duration of the two calls. Otherwise, other threads that happen to access the Player object will be able to see intermediate state between the two calls to switchPlayerTeam where the Player lock is released. Thus we lose the benefits of encapsulation, the code is less robust, and scaling up the software becomes intractable.

Locks initially showed great promise. They were able to provide encapsulation of state for multi-threaded programs by synchronising threads. However, they eventually betrayed us, because details of the locks themselves leaked out of the code into higher levels of abstraction. This unfortunately causes almost as much of a problem as the concurrency issues they are meant to solve.

2.4 Declarative Multi-Threaded Encapsulation

Previously, we have been treating locks as a mechanism. We have identified potential interference and used locks in ad-hoc ways to eliminate that interference. We have not discussed policy. If locks allow the extension of encapsulation to concurrency, by hiding state from other threads, we would like to reason about this directly, at the higher level of program design at which encapsulation is usually discussed. We would like a more declarative concept that we can use to express the encapsulation that we want instead of thinking about how to use locks to get it.

Atomicity (§1) is exactly the declarative concept we need. If a block of code is atomic then other threads cannot observe the intermediate states that arise during the execution of the block. Using atomicity, programmers can at least express the encapsulation they need, without worrying about all the details that achieving this atomicity might involve. Atomicity has already received positive attention from programmers [82]. So far, encapsulation boundaries have coincided with function boundaries, but this is only because our examples have been deliberately simple. Functions are often used for implementing encapsulation, but they are also have other uses, such as factoring out duplicate code, implementing callback structures, and for splitting up code to avoid excessive indentation. Atomicity seems to have a more specific utility: It just provides encapsulation. Additionally, even if a function body is a sensible unit of encapsulation, if it is only called from one place within the same class, a programmer may decide to inline it to keep the code simple. For these reasons, it seems that considering atomicity as a property of arbitrary blocks of code is more sensible than restricting our attention to just function bodies.

Often, the location of encapsulation boundaries is idiomatic, e.g., the operations of

concurrent data structures. However in general, even for single-threaded programming, choosing what and where to encapsulate is one of the most difficult and important aspects of software engineering. Encapsulation is something about which programmers are already aware, and so we suggest that programmers are already mentally prepared for using atomicity as an encapsulation tool when writing multi-threaded software.

Figure 2.6 shows where the encapsulation boundaries belong in the earlier examples (§2.3). We have omitted any mechanisms that might be used to implement the atomicity, the given code is just a specification. Atomicity is particularly suited to composition. If a block is atomic, this covers all the state and code within the block, down to the most fundamental machine operations. In the switchPlayerTeam method we need to compose the two set operations into one operation, and this operation should be atomic. By simply marking the block we need to be atomic, we can easily express this. The difficulties we had in the previous section are thus only due to getting the locking mechanisms to do what we want, they are not fundamental to the activity of concurrent programming.

2.5 A Generalised Lock Discipline

There are general disciplines that, if globally adhered to, can tell us where to insert locks to achieve atomicity. We focus on one particular discipline, *two-phase* locking [29], as it is simple and general-purpose. These are the rules a programmer must follow when applying the two-phase locking discipline to achieve atomicity:

- 1. One must identify the objects in the system that are *thread-local*, and those that are *shared*, i.e., accessed by more than one thread. Objects are either thread-local or shared.
- 2. One must choose a lock to protect each shared object. The same lock can be used to protect more than one object. The lock must exist throughout the lifetime of the objects it protects.

```
class Set {
       protected ArrayList list = new ArrayList();
       public void insert (Object o) {
               atomic {
                      if (!list.contains(o)) list.add(o);
               }
       }
       public void delete (Object o) {
               atomic {
                      int index = list.indexOf(o);
                      if (index!=-1) list.remove(index);
               }
       }
       public boolean contains (Object o) {
               atomic {
                      int index = list.indexOf(o);
                      return index!=-1;
               }
       }
}
class TeamManager {
       // ...
       public void switchPlayerTeam (Player player, Set from, Set to) {
               atomic {
                      from.delete(player);
                      to.insert(player);
               }
       }
}
class SomeClass {
       // ...
       public void swapPlayers () {
               atomic {
                      switchPlayerTeam(player,blue,red);
                      switchPlayerTeam(player,red,blue);
               }
       }
}
```

Figure 2.6: Using the atomic annotation to specify the encapsulation we need.

- 3. One must decide what blocks of code should be atomic.
- 4. For each such block of code, one must determine the shared objects that are accessed by the block and by any methods it calls (including any late binding). An object access consists of a read or write of a non-final field of the object. The locks that protect these objects must be held whilst the accesses occur.
- 5. All the acquisitions must precede all the releases in the block. This is the origin of the name two-phase. The block can be divided into an acquisition phase and a release phase.
- The order of lock acquisitions must obey some global order to avoid deadlocks as described in (§2.3).

Just as disciplined programming can help avoid some of the pitfalls of dynamic memory management, using a locking discipline can help avoid race conditions and deadlocks. Although it can be proven that programs successfully applying this discipline are free from interference and deadlocks, it is still very easy to make mistakes when applying the discipline and thus still suffer these problems. This motivates both lock checking and lock inference.

Although there are other ways that locking can be used to implement atomicity, such as hand-over-hand locking [57], we assert that these can only be correctly applied in very special circumstances. For example, hand-over-hand locking, which can be used to traverse an object graph in a thread-safe manner while allowing other traversals in parallel, is not correct unless all concurrent traversals proceed through the objects in the same order. We consider specialist synchronisation to be too low-level to be a useful abstraction for domain experts, best left to the design of specific high-performance data structures such as Java's concurrent collections.

Having said this, there is still a lot of flexibility within the two-phase method. For example, the programmer can choose which subset of objects should be shared between
threads, and can also choose the guarding structure i.e., choose which locks should guard each object. This makes it applicable in a wide range of situations.

2.6 Granularity

The relationship between locks and data is flexible, and is characterised with the notion of granularity [39]. One can associate either many or few objects with each lock. The advantage of protecting few objects with each lock is that when this lock is acquired, fewer threads are excluded. Thus, *fine* granularity promotes more parallelism. However, this means that more locks may need to be taken to ensure atomicity for a given set of accesses, i.e., a given block of code. At the other end of the scale, is the use of one lock for all objects, in which case only this single lock needs to be taken to ensure atomicity for any block of code, but the chance of blocking an unrelated thread is much higher. This is the most *coarse* granularity possible.

When the number of locks is not bounded by the size of the program, e.g., when each object has a single lock that protects its fields, we say that that *instance locks* are used. Alternatively, if the number of locks can be bound by the size of the program, e.g., associating locks with syntactic entities such as classes or construction sites in the code, we call these *static locks*. Using a single global lock for all objects is another example of static locks.

Static locks are usually coarser than instance locks and allow less parallelism. On the other hand, their bounded nature makes it easier to support them in static type systems and program analyses.

Even with instance locks, the granularity can be adjusted to reduce the number of locks that need to be taken. Many static tools need to conservatively abstract the set of objects accessed when analysing an iteration over an object structure. They usually achieve this by requiring all the objects touched to be guarded by the same lock, thus avoiding undecidability problems with lists and other unbounded object structures that are extremely common in object-oriented programming.

In order to mitigate the lost parallelism, when tools require a coarser granularity, multi-granularity locks can be used. The idea is that a coarse granularity is used for loops and other places in the code where an approximation needs to be made, but elsewhere in the code, a fine granularity can be used. Typically locks are created as members of a group of locks. Individual locks can be acquired and released as normal, but there are additional operations for locking the entire group. This is usually implemented with a centralised counter, rather than naively iterating over the entire group [39].

2.7 Deadlock Detection

The term *deadlock* is used for a variety of states. It is generally applied to a system that has stopped making useful progress because of some adverse condition. For instance, when we first spoke of deadlock (§2.2), it was caused by forgetting to release a lock. This kind of error can easily be prevented, e.g., with lexically scoped locks as seen in Java and C++. Here, we are more interested in the deadlock that occurs when locks form a cycle. Consider two threads that have each acquired a different lock, but are now both blocked trying to acquire the other thread's lock. Henceforth when we refer to deadlock, we are referring only to this cyclic lock problem.

We first considered deadlock as a bug that must be avoided by the programmer (§2.2). However when the system gets into such a state, it is possible to detect by searching for a cycle in the *waits-for* graph formed by threads, locks, and lock acquisitions. This is firstly useful as a debugging aid. When deadlock occurs, one can pause the program and run a tool that scans the waits-for graph and reports any cycles. The programmer then has enough information to tackle the cause of the problem. The Java Hotspot virtual machine has this feature [65]. Because a lock implementation will usually store the set of threads waiting on a lock, so that one of them can be woken upon the lock's release, the waits-for graph is implicitly present in the program state and just needs to be traversed. In later chapters, we will want something more from deadlock detection. We will essentially want lock acquisition to fail (dynamically) when it would cause a deadlock. This is a harder problem because we do not want to negatively affect performance in the (common) non-deadlocking case. Locks are sometimes used in very tight loops so we have very little room for overhead. We need to scan for a cycle on every failed lock acquisition. If there are the same or fewer active threads than the number of CPUs, this extra computation would occur on hardware that would otherwise be idle. However, electricity would still be wasted and it is not uncommon to have more threads than CPUs when designing a system to work on different architectures with different numbers of CPUs.

Another solution is to use timeouts when acquiring locks [56]. This often reports false deadlocks in the case where locks are held by another thread for a long time, and one has to wait for the timeout before getting a result. The timeouts have to be carefully chosen to match the application and the hardware running it. Luckily, there is an alternative solution in the form of Dreadlocks [56]. Dreadlocks work by storing the set of waits-for threads in each lock. This is called a *digest*. Initially the digest is empty but in the time spent waiting for another thread to release its lock, the digest accumulates more threads until it discovers itself, at which point the deadlock status is reported. The representation is more compact and easier to interpret than a waits-for graph, and the various threads co-operate to distribute the information around the graph.

Further performance improvements come from the representation of the digest (a set). For small numbers of threads a bit field can be used, but even for large numbers one can use a Bloom Filter [9], a conservative approximation of a set that uses hashes. When using a Bloom filter, hash collisions can mean that the set appears to contain elements that were not put into it, which will manifest as false deadlocks. However, for many applications this is acceptable as the false positives are rare and only result in a tiny loss of performance that is more than outweighed by the benefits of using a more efficient set representation.

2.8 Chapter Summary

We can get a good understanding of the software engineering cost of adding locks to a program by considering the cost of applying the two phase discipline discussed in the last section $(\S 2.5)$

- When determining the shared objects accessed by the operation in question, one has to look deep into the methods called by the operation. For swapPlayers (fig. 2.5) this means studying the implementation of TeamManager and even Set to see what objects they access. When combined with inheritance, this becomes very difficult since we are not able to extend classes and override methods without changing the code everywhere the overridden method may be called. It also clearly breaks encapsulation since the set of objects accessed by a method depends on how the method is implemented.
- When avoiding deadlocks, we must know what locks are acquired by any methods called by the operation in question, so that we can make sure that we keep the acquisitions consistent with the global ordering. For example in fig. 2.3 we had to acquire the red team's lock external to switchPlayerTeam in order to prevent a possible deadlock. When we add late binding, the problems are compounded. Again, this breaks encapsulation since the locks internal to a method are part of its implementation, and should be liable to change without notice.

For simple examples such as the **Set** class, where we wanted to encapsulate the internal intermediate states, locks worked well. However when we wanted to build more complicated systems formed by composing simpler abstractions, we ran into several problems when using locks. Although we can use locks to encapsulate the internal intermediate states, we have to break encapsulation to do so. This makes locks hard to use in a large program. When we also consider the unforgiving errors we get when locks are incorrectly applied, it is clear that programmers face serious challenges using locks in large projects.

In this chapter we showed how encapsulation has helped us scale up our software. We also showed how multi-threaded code can often violate this encapsulation. We showed how locks can be used to solve this problem, but correctly applying them often requires the breaking of encapsulation in different ways. We suggested that the fundamental property in which programmers should be interested is atomicity. We showed how locks can be used to implement atomicity, and went into detail about some of the techniques and trade-offs that can be helpful when doing so.

We conclude by suggesting that imperative programmers fear taking advantage of parallel hardware because locks introduce significant new software engineering problems. Programming concurrency with lower level constructs is even harder [81]. Mistakes lead to deadlocks or interference. Both are hard to debug since good test coverage is difficult when threads interleave non-deterministically. In the rest of this thesis we will give lock checking and lock inference techniques that are designed to mitigate these problems as much as possible.

Chapter 3

Lock Checking

3.1 Introduction

A race condition, or more concisely, just a race, is a kind of interference that can occur in concurrent programs when two threads are not properly synchronised, and thus can simultaneously access the same object. We saw an example of this in (§2.2) when two threads simultaneously accessed the same **Set** object. This can then lead to corruption of data structures, and eventual software failure. To date, many well-known pieces of software have fallen foul of race conditions, often long after their initial development, sometimes leading to denial-of-service attacks or other security problems [48, 79, 4, 54, 62, 21]. Programmers typically attempt to avoid race conditions through disciplined programming [57].

We give a type system that eliminates races by enforcing a particular locking discipline. We would like also to enforce atomicity, but a block of code has to be race-free before it can be atomic. Later, we will extend the race safety type system so that it can also enforce the atomicity of blocks marked as **atomic** in the code. This is a small extension that adds some simple restrictions on the nesting of **sync** blocks. Even without this extension the system is still useful because just knowing that a program is race-free is helpful for a programmer. The programmer can then manually impose additional disciplines to get atomicity.

For race safety, we simply require object accesses to occur within a synchronised block of an appropriate lock. For simplicity, we assume all objects are shared. Each object is thus guarded by a lock. For storage purposes, locks are associated within objects (not necessarily the object they protect) so the guarded-by relationship is an object-object relationship. We use instance locks, so the number of locks is statically unbounded, and the guarded by relationship is challenging to reason about statically. Ownership type systems face a similar problem – to statically reason about object-object relationships as noted by Boyapati et al. [12]. As briefly discussed earlier ($\S1.1$), we use universe types to solve this problem. The programmer inserts universe annotations into their code, and also inserts sync blocks. These are then together checked for mutual consistency.

3.2 An Example of Universe Types and Race Safety

The run-time state of an object-oriented program consists of a graph of objects linked by field references. In an ownership system, each object is owned by another object. The owner of an object does not change during the object's lifetime. The ownership relation describes a tree structure whose root is null. This tree structure can be used to represent the encapsulation inherent in the design of a program [14, 72]. As such, it can also be used to represent the locking discipline used.

In a universe type system, reference types consist of a class and a keyword that indicates the topological relationship. We call the keyword an *ownership type qualifier*; it is one of **rep**, **peer**, and **any**.

When universe types are used in a program, they only have meaning when considered relative to some *observer* object. For field type annotations, the observer is the object containing the field. For method parameter and local variable annotations, the observer is the receiver of the method. When the observer has the same owner as a particular object, the object is a **peer** of that observer. If instead, the observer is the owner of the object,

CHAPTER 3. LOCK CHECKING

the object is a **rep** of that observer. Any object can be **any**, regardless of its observer. Thus **any** gives no information about ownership, which forms a sub-type relation that we will formalise shortly. In earlier work [67], **any** was called **readonly** since field assignments through such references were not allowed. We do not impose this restriction.

We now return to our earlier example (fig. 1.1) in more detail. For convenience, we reproduce it in fig. 3.1. Each object has an address, e.g., (1), and a class name, e.g., **Dept**. Owned objects are drawn in the box of their owners; the tree is represented by the nesting of the boxes, (1) owns (2-5), and (8) owns (9-11). From observer (1), (3) has type **rep Student**¹, but from observer (2), the (3) has type **peer Student**. Thus, the type of (3) is relative to the observer. Also, from observer (8) the object (3) has type **any Student**, and the object (1) has type **peer Dept**².

The source code shows the universe type annotations being used in a program. The program is about students who live in halls of residences and who belong to departments. There is no correlation between the department a student is in, and their hall. An execution of this code could give rise to the heap in the diagram. For example, class **Dept** has field **first** of type **rep DeptStudentNode**, which, in the diagram corresponds to the reference from (1) to (2). On the other hand, HallStudentNode has field **s** of type **any Student**, which, in the diagram corresponds to the reference from (9) to (3). Thus, through **any**, students (owned by their respective departments), can be accessed also from a separate domain, namely their halls of residence.

We now discuss the use of the tree-hierarchy imposed by the universe types to avoid races: We require that the run-time system records the owner of an object (which does not change). We associate a lock with each object, with objects guarded by their owner's lock rather than their own. Any accesses to a field of an object, for example e'.f or e'.f = ..., must be within a sync *e* block where *e* evaluates to a peer of e'. This is in contrast to Java's synchronized, which just locks *e* and not all of its peers. One can also

¹It trivially also has type any Student.

²The reference from (8) to (3) is illegal in systems enforcing owners-as-dominators [14, 72] but is legal in universe types, which, instead, enforce owners-as-modifiers.

```
1
    class Student {
\mathbf{2}
            int mark;
3
            boolean roomClean
4
    }
5
6
    class Dept {
7
            rep DeptStudentNode first;
            void releaseMarks () { ... }
8
    }
9
10
    class DeptStudentNode { // Closed list
11
12
            peer Student s;
            peer DeptStudentNode next;
13
14
    }
15
    class Hall {
16
            rep HallStudentNode first;
17
18
            void cleanRooms () { ... }
    }
19
20
21
    class HallStudentNode { // Open list
22
            any Student s;
23
            peer HallStudentNode next;
24
    }
25
26
    void releaseMarks () {
27
            sync (this) {
28
                    rep DeptStudentNode i = this.first;
29
                    sync (i) {
                           while (i!=null) {
30
31
                                   i.s.mark = ...;
                                   i = i.next;
32
            }
                   }
33
    }
                           }
34
35
    void cleanRooms () {
36
            sync (this) {
37
                    rep HallStudentNode i = this.first;
38
                    sync (i) {
39
                           while (i!=null) {
40
                                   sync (i.s) {
41
                                           i.s.roomClean = true;
42
                                   }
43
                                   i = i.next;
                    }
44
   }
            }
                           }
                                                                              (8) Hall
                       (1) Dept
                      (2) DeptStudentNode
                                               (3)
                                                   Student
                                                                       (9)
                                                                           HallStudentNode
                                                             I€
                      (4) DeptStudentNode
                                               (5)
                                                   Student
                                                                       (10)
                                                                           HallStudentNode
                                                                       (11) HallStudentNode
                                   (6) Dept
                                             (7)
                                                  Student
```

Figure 3.1: Example program showing heap hierarchy structure

think of sync *e* as locking the object owning *e*. We propose that our semantics of sync should replace synchronized. Nested boxes are disjoint; code that accesses both must take both locks.

Consider the body of releaseMarks. Since we are adhering to the above rule, the field access of this.first (line 28) is enclosed within sync (this) (line 27). More interesting is the body of the while loop, where a statically unknown number of field accesses through i.next (line 32) is correctly synchronised by acquiring a *single* lock, sync (i), before the loop (line 29). Even though i will point to different objects at each iteration, the synchronisation is correct, because the field next is peer and thus the type system ensures all these objects will have the same owner. The same is true when we access the student (line 31).

We needed to tackle the challenge of avoiding races when the owner of the accessed object is unknown, i.e., when the object has type **any** C for some class C. (Note that the owner is only statically verifiable when the ownership type qualifier is not **any**.) In such a case, any accesses of the form p.f or p.f = ..., where p must be a path³, must be within a sync p block, and the block may not assign to any of the fields appearing in p. This ensures that the object locked is the same as the object accessed, and thus the owners are trivially the same.

The difference between the body of cleanRooms in fig. 3.1 and releaseMarks is that in the former, HallStudentNode has an any pointer to Student. Thus, the student is not necessarily a peer of the node i; therefore, when we access i.s (line 41) the sync (i) (line 38) is no longer sufficient. We must lock the owner of the student i.s and this is possible through the "fresh" sync (i.s) (line 40) even though i.s is any. We must be sure however that the body of the sync (i.s) block does not write to the field s, otherwise the type system would reject our program.

Note that in **releaseMarks**, students will receive their marks *atomically* (there is never a state visible where a subset of students have their marks) but this is not the case

³A path is a sequence of field accesses starting from a parameter or this.

for the cleaning of rooms. A student may notice their room has been cleaned whereas another student's room has not. Recall the earlier definitions of open and closed list $(\S1.1)$. In general, we must lock individual elements when iterating through an open list. This is not necessary for a closed list.

3.3 Related Work

Not all the previous lock checking work uses ownership or guard annotations to specify the locking discipline as a relationship between objects. Some work [8] uses a finite set of programmer-supplied region names, and specifes the locking discipline as a relationship between objects and regions. This can reduce the annotation burden since regions are easier to infer than ownership types. However, it has the disadvantage that the set of locks is finite, and thus the program does not scale as well to many threads (instance locks are not possible). It is also possible to detect race conditions through program analysis using points-to sets, which are similar to regions [68].

Of the work that does use instance locks, the first to exclude race conditions from object-oriented programs using a static type system was formalised using the concurrent object calculus [30]. This idea was subsequently refined to more concrete models of objectoriented languages [31, 35], which included parameterised classes to allow the instantiation of data structures with different locking policies. Later, the technique was extended to also enforce atomicity [36] using the two-phase discipline. There was also a dynamic approach [32] that checked conformance to the discipline at run-time. The annotation burden in these systems was significant, so later attempts were made to infer some of the annotations, first for race safety [37] and then for atomicity [34]. These papers are variations on the same approach: The programmer supplies guard annotations in their classes; the guard annotations were a form of ownership types; this is supported through their use of final expressions and parameters, although individual fields were owned instead of entire objects. Similar results were also obtained using ownership types directly [11, 12]. One interesting difference in that case was that objects ownership was transitive. The ownership hierarchy was a forest instead of just a single tree, and individual trees were guarded by a lock each. Locking an object in our system does not lock the whole tree as in [53, 11, 12]. We lock only the immediate level and further lock acquisitions are required if deeper objects are accessed. With more locks, we reduce contention and let more threads execute in parallel.

There is a substantial difference between our work and that just discussed; we allow paths of non-final field types (in fact our formalism does not have the **final** type qualifier) whereas the previous work requires paths to be constructed from only final field dereferences. The price we pay for forsaking this restriction is that we must find another way to ensure that the meaning of paths is not affected by the side-effects of the body of the **sync** block. For this we use a system of effects, and for this to work we require the effect of overriding methods to be restricted to that of the method they override. This is actually less of a restriction than forcing variables and fields to be final, since that prevents writing to those fields anywhere in the code. A combination of our approach and that of [34, 12] would be less restrictive.

Effects are also used to prove preservation of properties of ownership type system in [78]. A concept similar to universes was studied in conjunction with synchronisation in [53]. This was mainly for the purpose of verifying object invariants rather than absence of race conditions. Objects can "change hands" over time, therefore their owners are not constant at run-time. Also, there is no concept of **peer**.

Another difference is our use of any. In [34], an open list (§1.1) of students can be written if we design the student so that it has a final field that stores the owner. In other words, we create a class that can be referenced by a variable whose type does not specify an owner such as s of HallStudentNode. However, this change is global to the program so every other reference (e.g., the field s of DeptStudentNode) must use the

```
// We have to design Student like this:
                                               // We have to design student like this:
class Student {
                                               class Student<x> {
  final Object owner;
                                                  /* fields here */
                                               ł
   /* fields guardedby this.owner */
3
// Therefore, the HallStudentNode looks like: // Therefore, HallStudentNode looks like:
                                               class HallStudentNode<x> {
class HallStudentNode<x> {
  Student s guardedby x;
                                                  Student<self> s;
  HallStudentNode<x> next guardedby x;
                                                  HallStudentNode<x> next;
}
                                                7
// Hall locks its students like so:
                                               // Hall locks its students like so:
class Hall<x> { // Open list
                                               class Hall<x> { // Open list
  HallStudentNode<this> first guardedby x;
                                                  HallStudentNode<this> first;
  void cleanRooms () {
                                                   void cleanRooms () {
     sync (x) { //protects fields of this
                                                     sync (x) {
        HallStudentNode<this> i=this.first;
                                                        HallStudentNode<this> i=this.first;
        sync (this) { //protets nodes
                                                        sync (this) {
          while (i) {
                                                          while (i) {
             final Student s_ = i.s;
                                                             final Student<self> s_=i.s;
             sync (s_.owner) {
                                                             sync (s_) {
               s_.roomClean = true;
                                                                s_.roomClean = true;
             7
                                                             3
             i = i.next;
                                                             i = i.next;
          }
                                                          }
    }
}
                                                       }
                                                    }
  }
                                                  }
}
                                               }
                                               \ensuremath{//} Since the students we reference have type
// But a Student is a Student, so design
// DeptStudentNode in the same manner:
                                               // Student<self>, this field must be the same:
class DeptStudentNode<x> {
                                                class DeptStudentNode<x> {
  Student s guardedby x;
                                                  Student<self> s;
  DeptStudentNode<x> next guardedby x;
                                                  DeptStudentNode<x> next;
3
                                               7
                                                // But we do not own the student, so we must
// And thus we have to lock each student
                                               // lock each student individually.
class Dept<x> { // must be open too!
                                                class Dept<x> { // must be open too!
  DeptStudentNode<this> first guardedby x;
                                                  DeptStudentNode<this> first guardedby x;
  void releaseMarks () {
                                                  void releaseMarks () {
     sync (x) {
                                                     sync (x) {
        DeptStudentNode<this> i=this.first;
                                                        DeptStudentNode<this> i=this.first;
        sync (this) {
                                                        sync (this) {
           while (i) {
                                                           while (i) {
             final Student s_ = i.s;
                                                             final Student<self> s =i.s:
              sync (s_.owner) {
                                                              sync (s_) {
               s_.mark = ...;
                                                                s_.mark = ...;
                                                             }
             7
    }
}
}
             i = i.next;
                                                             i = i.next;
                                                          }
                                                       }
                                                    }
  }
                                                  }
}
                                               }
```

Figure 3.2: Example code in the systems of Flanagan et al (left) and Boyapati et al (right)

same type (that does not specify an owner). This means that we cannot make a closed list of students, because the owner of the student is no longer indicated by its type. The only solution is to use open lists everywhere, which have the undesirable property that we cannot lock all the elements of the list at once, we have to acquire the same lock once for each student. The implication of this is that iterating through the list cannot be atomic (as in our releaseMarks).

The type system of [12] is even more restrictive, as a closed list implementation can only contain self-owned objects. This means that objects contained in an open list also can only be owned by the root. The code for both solutions is given in fig. 3.2.

Our work complements these approaches by discussing a different kind of ownership type system (universes) and its application to race safety. Although it is interesting to see how static race safety can be achieved using universes, our major contributions are increased expressiveness and greater concurrency.

3.4 Formal Preliminaries

In this section we will give the syntax and semantics of our model language. We will also give our universe type system, which is a prerequisite for the next section, where we will discuss the actual race safety type system. universe types are introduced in [67], and given a type theoretic presentation in [16]. We use ideas from [67] but with some differences: We decided that owners-as-modifiers, while useful for verification, are not needed for type soundness and race safety. Our type system allows field assignments through **any** objects as long as the heap remains well-formed. Therefore our type system is more permissive and could be restricted to also require owners-as-modifiers.

For sequences, we use the notation \overline{e} in the style of [51], and sometimes the notation $e_{1..n}$, both of which are distinct from the undecorated e. The *i*th element of $e_{1..n}$ is e_i and of \overline{e} is $\overline{e}\downarrow_i$. We use similar notation when we access the *i*th element of a tuple: $(a, b, c)\downarrow_2 = b$. We use an underscore _ to represent a variable whose value can be

Figure 3.3: Source program definition

arbitrary. To denote that a particular construct, e.g., new c, occurs within an expression e, we sometimes write new $c \in e$. \mathcal{P} is the powerset.

3.4.1 Syntax and Semantics

Programs are defined in Fig. 3.3, and consist of the three functions \mathcal{M} , \mathcal{MBody} , and \mathcal{F} , which define the method signatures, method bodies, and field types of each class in the program, together with (\leq_c) , which gives the inheritance relationship between classes. Note that all types t are annotated with an ownership type qualifier u that can be one of the three keywords rep, any, peer, or self, which is the type of this and thus a specialisation of peer. It prevents the type system losing type information during local member access. We use spawn e to start a new thread to execute e, and sync e e' to acquire the lock that guards the object e while we execute the expression e'. We give the run-time syntax in Fig. 3.4 and use a small step semantics.

We define (\leq_u) as $\operatorname{self} \leq_u \operatorname{peer} \leq_u \operatorname{any}$ and $\operatorname{rep} \leq_u \operatorname{any}$. We define the sub-type relation (\leq) as $u \ c \leq u' \ c' \iff u \leq_u u' \land c \leq_c c'$. The program state consists of the heap h and a sequence of expressions \overline{e} . It is reduced with respect to a base stack frame σ according to the rules in fig. 3.5. The (INTERLEAVE) rule uses the single-threaded semantics. The base stack frame contains the value of this and \mathbf{x} . Single-threaded execution steps are decorated with actions, ranged over by β . If a step accesses an address a, then its action is a, otherwise its action is τ . These actions do not affect the execution,

$$\begin{array}{rclcrcl} s \in State & : & RunExpr \times Heap \\ h \in Heap & : & Addr \to Object \\ & Object & : & (Val \times Id^c \times (Id^f \to Val)) & // \text{ owner, class, fields} \\ a \in Addr & : & \mathbb{N} \\ v, w \in Val & ::= & a \mid \text{null} \\ \sigma \in Stack & ::= & (a, v) & \sigma(\texttt{this}) = \sigma \downarrow_1, \sigma(\texttt{x}) = \sigma \downarrow_2 \\ \beta \in Actions & ::= & a \mid \tau \\ e \in RunExpr & ::= & v \mid \texttt{this} \mid \texttt{x} \mid \texttt{new} \ t \mid e.f \mid e.f = e \mid e.m(e) \mid (t) \ e \mid \texttt{spawn} \ e \\ & \mid & \texttt{sync}_e \ e \mid \texttt{synce}_e \ w \ e \mid \texttt{frame} \ \sigma \ e \\ E[\cdot] & ::= & E[\cdot].f \mid E[\cdot].f = e \mid v.f = E[\cdot] \mid E[\cdot].m(e) \mid v.m(E[\cdot]) \\ & \mid & (t) \ E[\cdot] \mid \texttt{synce}_e \ E[\cdot] \ e \mid \texttt{synced}_e \ w \ E[\cdot] \end{array}$$

Figure 3.4: Run-time state and syntax

$$\begin{array}{c} e_{i} = C[\operatorname{spawn} e'] \\ \hline \sigma \vdash \mathbf{x}, h \stackrel{\tau}{\rightsquigarrow} \sigma(\mathbf{x}), h \end{array} (VAR) \qquad \begin{array}{c} e_{i} = C[\operatorname{spawn} e'] \\ \hline e_{n+1} = \operatorname{frame} Active(\sigma, C[\cdot]) \ e' \\ \hline \sigma \vdash e_{1..n}, h \stackrel{(i,\tau)}{\leadsto} e_{1..i-1} \ C[\operatorname{null}] \ e_{i+1..n+1}, h \end{array} \\ \\ \hline \begin{array}{c} h, \sigma \vdash v : t \\ \hline \sigma \vdash (t) \ v, h \stackrel{\tau}{\rightsquigarrow} v, h \end{array} (CAST) \qquad \begin{array}{c} \sigma \vdash e_{i}, h \stackrel{\beta}{\rightsquigarrow} e'_{i}, h' \\ \hline \sigma \vdash e_{1..n}, h \stackrel{(i,\beta)}{\leadsto} e_{1..i-1} \ e'_{i} \ e_{i+1..n}, h' \end{array} (INTERLEAVE) \end{array}$$

$$\frac{\sigma \vdash e, h \stackrel{\beta}{\rightsquigarrow} e', h'}{\sigma \vdash E[e], h \stackrel{\beta}{\rightsquigarrow} E[e'], h'} (CTX) \qquad \frac{e_i = C[\texttt{synced}_{e'} w v]}{\sigma \vdash e_{1..n}, h \stackrel{(i,\tau)}{\rightsquigarrow} e_{1..i-1} C[v] e_{i+1..n}, h} (UNLOCK)$$

$$\frac{h' = h[a\downarrow_{3}(f) \mapsto v]}{\sigma \vdash a.f = v, h \stackrel{a}{\rightsquigarrow} v, h'} (ASSIGN) \qquad \frac{e_{i} = C[sync_{e'} \ a \ e] \qquad w = h(a)\downarrow_{1}}{\varphi j \in \{1..n\} : \ Locked(e_{j}, w) \Longrightarrow i = j} (LOCK)}{\sigma \vdash e_{1..n}, h \stackrel{(i,\tau)}{\leadsto} e_{1..i-1} \ e'' \ e_{i+1..n}, h}$$

$$\frac{\sigma \vdash a.f, h \stackrel{a}{\rightsquigarrow} h(a)\downarrow_{3}(f), h}{\sigma \vdash \text{frame } \sigma' v, h \stackrel{\tau}{\rightsquigarrow} v, h} (\text{FRAME2}) = \frac{\sigma' \vdash e, h \stackrel{\beta}{\rightsquigarrow} e', h'}{\sigma \vdash \text{frame } \sigma' e, h \stackrel{\beta}{\rightsquigarrow} \text{frame } \sigma' e', h'} (\text{FRAME1})$$

$$\frac{\sigma' \vdash e, h \rightsquigarrow e', h'}{\sigma \vdash \texttt{frame } \sigma' \ e, h \stackrel{\beta}{\rightsquigarrow} \texttt{frame } \sigma' \ e', h'} (\texttt{FRAME1})$$

$$\begin{array}{c} h(a) \text{ undefined} \\ h' = h[a \mapsto (_, c, \lambda f. \texttt{null})] \\ \underline{h', \sigma \vdash a: u}_{} \\ \hline \sigma \vdash \texttt{new} \; u \; c, h \stackrel{\tau}{\leadsto} a, h' \end{array} (\text{New})$$

$$\frac{e = S(\mathcal{MBody}(h(a)\downarrow_2, m))}{\sigma \vdash a.m(v), h \stackrel{\tau}{\leadsto} \texttt{frame}(a, v) \ e, h} (CALL)$$

$$\begin{array}{rcccc} S & : & SrcExpr \rightarrow RunExpr\\ S(\texttt{sync}\ e_1\ e_2) & = & \texttt{sync}_{S(e_1)}\ S(e_1)\ S(e_2)\\ S(Con(e_1 \dots e_n)) & = & Con(S(e_1) \dots S(e_n)) & (\text{for all other } Con \in SrcExpr) \end{array}$$

Figure 3.5: Small step operational semantics

Figure 3.6: Example code to demonstrate the semantics of method call

they are merely to help us formalise race conditions. At the multi-threaded level, each step may introduce at most one more thread, stopped threads are never eliminated from the system, and we wrap actions with the index of the thread that caused them.

Method calls are modelled by substituting the call construct with the method body in question combined with the stack frame, which records the receiver and parameter. The **frame** σ *e* construct marks the boundaries between the different calling contexts in the run-time expression, and holds the new stack σ , which is used to execute the method body *e*.

Consider the code in fig. 3.6. If we assume a is the address of an A object on the heap, and likewise b is the address of a B object, let us execute a.m(b) with respect to an arbitrary base stack frame. The execution will not change the heap, so will only describe the evolution of the run-time expression. We underline new code to make the changes easier to follow. Firstly the call to m is substituted with the body of m:

$$frame (a, b) (x.foo(null).bar(null))$$
(CALL)

Immediately this introduces a new stack frame that holds the observer and argument (a, b) for the invoked code. So, when we evaluate **x** within this code, we refer to the stack frame (a, b) instead of the base stack frame:

frame
$$(a, b)$$
 (b.foo(null).bar(null)) (VAR)

Now we have another method call, so we invoke the method body for *foo*:

$$frame (a, b) ((frame (b, null) this).bar(null))$$
(CALL)

Now we have code from two invoked methods in the thread, and consequently two nested stack frames. As before, we use the innermost stack frame to reduce this:

$$\texttt{frame} (a, b) ((\texttt{frame} (b, \texttt{null}) b).bar(\texttt{null})) \tag{VAR}$$

Now there is nothing left to evaluate within the stack frame, we can return the value and destroy the frame:

frame
$$(a, b)$$
 (b.bar(null)) (FRAME2)
frame (a, b) (frame (b, null) null) (CALL)

frame
$$(a, b)$$
 null (FRAME2)

<u>null</u>

The sync construct of the run-time language has an extra expression subscript when compared to the source language sync. The (CALL) semantics rule translates the invoked method body through the S substitution, which uses a copy of the lock expression for the the extra subscript expression. The subscript does not affect the behaviour of the program, it is just instrumentation that was needed for proving race safety. We demonstrate the lifecycle of a sync construct with an example: The source expression sync e e'' is translated by S into the run-time expression $sync_e e' e''$. Initially e = e' but as the expression reduces, the e' will reduce until it reaches an address a. Then, the lock that guards a (i.e., its owner), w will be taken. The subscript persists under this $synced_e w e''$ expression until e'' terminates and the lock released. The subscript expression e will always remain as a record of the initial locking expression, even after the main body has started executing.

The semantics rules (CAST) and (NEW) use the run-time universe type system to constrain their behaviour (e.g., in (NEW), we use the type system to determine the owner), this makes the proofs simpler. In the case of **new any** c, the owner is arbitrary, i.e., the semantics is non-deterministic. This is perhaps unusual but it does not affect our result,

so we did not complicate the type system by explicitly disallowing it. The syntax allows for the expression **new self** c but this will always result in a stuck execution⁴. In practice we could disallow the use of **self** and **any** when constructing new objects, in either the syntax or the static type system. As one would expect, the semantics deterministically chooses the owner of the new object in the case of **peer** and **rep**.

For interleaved execution we use the context $C[\bullet]$, which extends the evaluation context syntax $E[\bullet]$ to add the stack frame construct:

$$C[\bullet] ::= \dots \mid \texttt{frame } \sigma \ C[\bullet]$$

Rule (LOCK) represents the locking of an object, effectively rewriting a thread as follows:

$$C[\texttt{sync}_{e'} \ a \ e] \rightsquigarrow C[\texttt{synced}_{e'} \ w \ e]$$

provided that no other thread has locked that object $(Locked(e_j, w) \Rightarrow i = j)$. Note that it is $w = h(a)\downarrow_1$, the owner of the object, that is actually locked. This is because we are effectively locking all the objects owned by w, not just the object at address a.

The predicate Locked(e, w) determines whether the thread e has the lock on object w: It holds whenever the construct synced w is a subexpression of e.

In rule (SPAWN), $Active(\sigma, e)$ provides the σ' from the innermost frame σ' within the thread e according to the context rules for $C[\bullet]$. If there is no such frame σ' , it returns σ .

3.4.2 Encodings

Our model language is small but we can consider additional features indirectly since they can be encoded. Conditional statements, numbers, arithmetic, and Booleans can be encoded as in the Object Calculus [2]. Sequential composition e; e' can be encoded using method call e.m(e') assuming a method m that returns its parameter is added to the class of e. Multiple method arguments can be encoded by passing a single newly

⁴It would need to reduce to an unused address that is equal to $\sigma(\texttt{this})$, but this address is already used so reduction is not possible.

constructed object with the arguments assigned to its fields. Iteration can be encoded with tail recursion.

3.5 Universe Type System

The universe type system is given in Fig. 3.7. The judgement $\Gamma \vdash e : t$ gives the universe type t of an expression e with respect to an environment Γ . As there is only one method parameter, this environment is simply a pair containing the types of **this** and **x**.

Universe annotations have meaning only with respect to an observer as discussed in (§3.2). The type annotations in field and method signatures are meant with respect to the object that contains them. When we are typing method bodies, the annotations within are considered with respect to the object **this**, whatever value **this** might have at run-time. Thus the type returned by the type system is also meant in respect to **this**. The ownership type qualifier **self** (a specialisation of **peer**) is used for the parameter **this**, e.g.

(self Dept, _) ⊢ this.first.s : rep Student (self Hall, _) ⊢ this.first.s : any Student

There is one aspect of this type system that deserves detailed discussion because we use it later. The purpose of $u \triangleright u'$ is to determine the type that best describes an object that is "twice removed" from the observer by references of type u and u'. In other words, we have two subsequent references and two respective types, and we want to know the type of the most distant object from the observer. (\triangleright) is defined in Fig. 3.8. The object (1) in fig. 3.1 observes the object (8) to be peer. (8) considers (9) to be rep, so (1) considers (9) to be peer \triangleright rep = any.

We use (\mathbb{P}) to 'translate' a type u' from one observer to another, where the old observer is u with respect to the new observer. This is useful for class member lookups where the type of the member is from the perspective of the object that contains it, but

$$\begin{array}{l} \forall c' \geq c \ : \ \mathcal{F}(c',f) = t \Longrightarrow \mathcal{F}(c,f) = t \\ \forall c' \geq c \ : \ \mathcal{M}(c',m) = t'_r \ m(t'_x) \Longrightarrow \mathcal{M}(c,m) = t_r \ m(t_x) \\ (\text{where } t_r \leq t'_r, t_x \geq t'_x) \ (\text{WFCLASS}) \\ \hline \mathcal{M}(c,m) = t_r \ m(t_x) \implies (\texttt{self } c,t_x) \vdash \mathcal{MBody}(c,m) : t_r \\ \hline \vdash c \end{array}$$

A program is well-formed iff $\forall c : \vdash c$

Figure 3.7: Static universe type system

			u'		
	$u \Vdash u'$	self	peer	rep	any
	self	self	peer	rep	any
	peer	peer	peer	any	any
u	rep	rep	\mathtt{rep}	any	any
	any	any	any	any	any
			u^{*}	/	
	$u \bowtie u'$	self	u^{*} peer	′ rep	any
	$u \vDash u'$ self	self self	u peer peer	' rep rep	any any
	$\frac{u \triangleright u'}{\texttt{self}}$ peer	self self peer	u peer peer peer	' rep rep any	any any any
u	$\frac{u \vDash u'}{\texttt{self}}$ peer rep	self self peer any	peer peer peer any	' rep any peer	any any any any

We extend to types by defining: $u \triangleright (u' c) = (u \triangleright u') c$ and $u \triangleright (u' c) = (u \triangleright u') c$

Figure 3.8: Universe composition and decomposition

→ ____u ⊳ u'

we want a type from the caller's perspective.

The opposite of (\triangleright) is (\triangleright) , which we use to translate a type to another observer. The object (1) observes the objects (2,3) to be **rep**, however object (2) considers (3) to be **rep** \triangleright **rep** = **peer**.

Note that the (SPAWN) rule matches any type t and does not require any particular type of the subexpression e, it just requires it to be well-typed in the same environment. This is because our semantics evaluates **spawn** e to **null** and thus the type rule is similar to (NULL).

Finally, we require classes to be well-formed. The types of method bodies must agree with their signatures. Note that when typing a method body, we use **self** in the type of **this**. This is consistent with our notion of observer for method bodies as described above. We also require consistency between field and method signatures in subclasses.

To prove soundness of this system, we need a type system for run-time expressions. This type system is capable of typing addresses using the owner of the current **this** object, which is stored in the heap, and typing variables \mathbf{x} and **this** using the values in the stack σ . The judgement is $h, \sigma \vdash e: t$, it is given in Fig. 3.9.

In order to later prove race safety, we need to establish the soundness of the underlying universe type system. We will now give a series of lemmas and the ultimate soundness theorem we require.

$$\begin{array}{c} h, \sigma \vdash \sigma(\texttt{this}) : t \\ h, \sigma \vdash \texttt{this} : t \end{array} (THIS) \qquad \begin{array}{c} h, \sigma \vdash \sigma(\texttt{x}) : t \\ h, \sigma \vdash \texttt{x} : t \end{array} (VAR) \qquad \begin{array}{c} h, \sigma \vdash e : u \ c \\ \mathcal{F}(c, f) = t \\ h, \sigma \vdash e. f : u \triangleright t \end{array} (FIELD) \\ \hline h, \sigma \vdash e. f : u \triangleright t \end{array} (FIELD) \\ \hline h, \sigma \vdash e. f : u \triangleright t \end{array} (FIELD) \\ \hline h, \sigma \vdash e. f : u \triangleright t \\ \mathcal{F}(c, f) = u \triangleright t \\ h, \sigma \vdash e. f : u \\ \mathcal{F}(c, f) = u \triangleright t \\ h, \sigma \vdash e. f = e' : t \end{array} (ASSIGN) \\ \hline h, \sigma \vdash e. f = e' : t \\ \hline h, \sigma \vdash e. t \\ h, \sigma \vdash e. t \\ h, \sigma \vdash sync_{e''} \ e' e : t \end{array} (SYNC) \qquad \begin{array}{c} h, \sigma \vdash e : t' \\ h, \sigma \vdash f = t \\ h, \sigma \vdash e : t \\ h, \sigma \vdash sync_{e''} \ e' e : t \end{array} (SYNC) \qquad \begin{array}{c} h(a)\downarrow_2 = c \\ h, \sigma \vdash a : u \ c \\ \hline h, \sigma \vdash e : t \\ h, \sigma \vdash f = e' : t \\ \hline h, \sigma \vdash e : t \\ h, \sigma \vdash sync_{e''} \ e' e : t \end{array} (SYNCD) \qquad \begin{array}{c} h(a)\downarrow_2 = c \\ h, \sigma \vdash a : u \ c \\ \hline h, \sigma \vdash e : t \\ h, \sigma \vdash f = \sigma' \ c : u \ c \\ \hline h, \sigma \vdash e : t \\ h, \sigma \vdash f = \sigma' \ c : u \ c \\ \hline h, \sigma \vdash e : t \\ h, \sigma \vdash f = \sigma' \ c : u \ c \\ \hline h, \sigma \vdash e : t \\ \hline h, \sigma \vdash e : t \ c \\ \hline h, \sigma \vdash e : t \\ \hline h \iff t = t \\ \hline h \iff t = t \\ \hline h \underset{t}{t} t \\ \hline t = t \\ t = t \\ t = t \\ \hline t = t \\ t = t \\ t = t \\ \hline t = t \\ t =$$

Figure 3.9: Run-time universe type system

Lemma 3.5.1 The (\mathbb{P}) operator composes types:

$$\left. \begin{array}{l} a,w \vdash a',w':u\\ a',w' \vdash a'',w'':u' \end{array} \right\} \Longrightarrow a,w \vdash a'',w'':u \vDash u'$$

Proof: Case analysis of u and u'.

Lemma 3.5.2 The (\triangleright) operator decomposes types:

$$\left. \begin{array}{l} a,w\vdash a',w':u\\ a,w\vdash a'',w'':u' \end{array} \right\} \Longrightarrow a',w'\vdash a'',w'':u\vDash u'$$

Proof: Case analysis.

The ownership type qualifier self has a special purpose – when calling local methods and accessing local fields, we want the type of such accesses to be exactly the annotation u given in the class. Note that self $\triangleright u = u$. If we were to use peer instead of self as the type of this, then we would lose information in the case where u = rep and the type system would be unnecessarily restrictive.

Firstly we guarantee that at all times after an object is constructed, both its class and its owner remain constant. As a corollary, execution will not affect the universe type judgement of another expression. We present a "substitution" lemma (although there is no conventional substitution here since we are using a stack to hold the arguments). We require h and σ to be consistent with Γ , but the expression e is the same on both sides up to the S translation used to record the original lock expression into the **sync** construct. S is defined at the bottom of fig. 3.5. Finally we show soundness of single-threaded and multi-threaded execution. These lemmas and theorems are presented below: Lemma 3.5.3 Ownership and class membership are constant:

$$\begin{array}{c} h(a) = (v, c, _) \\ _ \vdash _, h \rightsquigarrow _, h' \end{array} \right\} \Longrightarrow h'(a) = (v, c, _)$$

Proof: Induction over structure of reduction.

Lemma 3.5.4 The run-time types of expressions are preserved over the execution of other expressions.

$$\left. \begin{array}{c} h,\sigma \vdash e:t \\ _\vdash_,h \rightsquigarrow_,h' \end{array} \right\} \Longrightarrow \ h',\sigma \vdash e:t$$

Proof: Induction over the structure of $h, \sigma \vdash e : t$.

Lemma 3.5.5 Static type safety implies run-time type safety with respect to a suitable stack.

$$\left. \begin{array}{l} \Gamma \vdash e:t \\ h, \sigma \vdash \mathtt{x}: \Gamma(\mathtt{x}) \\ h, \sigma \vdash \mathtt{this}: \Gamma(\mathtt{this}) \end{array} \right\} \Longrightarrow h, \sigma \vdash S(e):t$$

Proof: Induction over the structure of $\Gamma \vdash e: t$.

Theorem 3.5.6 Run-time types and heap well-formedness are preserved over execution.

$$\left. \begin{array}{c} \vdash h \\ h, \sigma \vdash e: t \\ \sigma \vdash e, h \rightsquigarrow e', h' \end{array} \right\} \Longrightarrow \begin{array}{c} \vdash h' \\ h', \sigma \vdash e': t \end{array}$$

Proof: Induction over the structure of $h, \sigma \vdash e : t$.

Theorem 3.5.7 The well-typedness of all threads in a system is preserved over a step of multithreaded execution.

$$\left. \begin{array}{c} \vdash h \\ h, \sigma \vdash e_{1..n} : t_{1..n} \\ \sigma \vdash e_{1..n}, h \rightsquigarrow e'_{1..m}, h' \end{array} \right\} \Longrightarrow \begin{array}{c} \vdash h' \\ h', \sigma \vdash e'_{1..m} : t_{1..m} \end{array}$$

Proof: Case analysis of $\sigma \vdash e_{1..n}, h \rightsquigarrow e'_{1..m}, h'$.

In the above theorem, m is either n or n + 1. New threads can initially have any type, but must maintain this type as they execute.

In summary, we have formalised and stated soundness of a fairly typical universe type system, as can be found in other papers [16]. As such we fall short of formally proving correctness of the above theorems and lemmas here, as this has been done before. However, we do use these lemmas in our later work on race safety.

We have added multi-threading to the model, but we assert this does not affect correctness: The heap remains well-formed at every step and threads cannot write to the stacks of other threads. Now we will describe how we use universes to help prevent races.

3.6 Race Safety

3.6.1 Static Types for Race Safety

In Fig. 3.10 we give a type system that requires correct synchronisation and thus guarantees race safety. The system uses paths and locks, as defined below. In the static system we do not use paths containing addresses a. These are used later in the dynamic system.

 $p ::= \texttt{this} \mid \texttt{x} \mid a \mid p.f$

 $l \in Lock ::= p \mid \texttt{rep} \mid \texttt{peer} \mid \texttt{self}$

The judgement $\mathbb{L}, \Gamma \vdash e : F$ denotes that the expression e is race free if all locks l in the synchronisation set \mathbb{L} have been acquired for the duration of its execution. The set F is the effect of e, i.e., the set of fields that e may write to as it executes. We use an annotation on methods to allow us to handle method calls. These annotations are represented with the function $\mathcal{E}ff$, which returns pairs of sets of synchronisation sets and sets of fields:

$$\mathcal{E}\!f\!f: (Id^c \times Id^m) \to (\ \mathcal{P}(\mathcal{P}(Lock)) \times \mathcal{P}(Id^f) \)$$

Well-typed expressions do not overwrite fields appearing in their synchronisation set (this is ensured using F), and \mathbb{L} is sufficiently large to guarantee for any path p in \mathbb{L} , evaluation of p only touches locked objects.

Lemma 3.6.1 The effects of well-typed expressions do not undermine their locks.

 $\mathbb{L}, \Gamma \vdash e : F \implies \mathbb{L} \ \# \ F \ \land \ \forall p \in \mathbb{L} \ : \ \mathbb{L}, \Gamma \vdash p : _$

Proof: induction on the derivation of $\mathbb{L}, \Gamma \vdash e : F$.

We say that an expression is *internally synchronised* if it can be typed with an empty synchronisation set, otherwise it is *externally synchronised*.

We now discuss the type system in greater detail. Suppose we have $\mathbb{L}, \Gamma \vdash e : _$. The synchronisation set and effect of variables and constants are empty, cf.,(NULL), (VAR), (THIS). This also holds for (NEW), as object creation does not interact with other threads. A cast does not require more locks or produce more effects than its sub-term, cf.,(CAST). Spawning requires the new thread to be internally synchronised, and therefore requires its sub-term to have an empty synchronisation set. Since the sub-term is executed in a new thread, its effect is of no interest to the current thread, therefore the whole expression has empty effect, cf.,(SPAWN).

The (SUB) rule is a form of subsumption as it increases the effect and synchronisation set, provided that none of the fields in the new effects F appear in any of the paths of the new synchronisation set \mathbb{L} , thus preserving lemma 3.6.1.

The (FIELD) and (ASSIGN) rules are similar. They calculate the lock l that guards the object access in question, using the guarded by judgement $\Gamma \vdash_{gb} e : l$. This lock must be acquired before the execution of the access in order to guarantee race safety, therefore l is included in the synchronisation set \mathbb{L} . The judgement finds the owner via the ownership type qualifier u provided that $u \neq any$ (UNIV), or uses the path p when e is such a path

$$\mathbb{L} \# F \Longleftrightarrow \forall f \in F, p \in \mathbb{L} : f \notin p$$

$$\begin{array}{l} \forall c' \geq c \ : \ F' = \mathcal{E}f\!\!f(c',m) \downarrow_2 \Longrightarrow \mathcal{E}f\!\!f(c,m) \downarrow_2 \subseteq F', \\ \mathbb{L}' \in \mathcal{E}f\!\!f(c',m) \downarrow_1 \Longrightarrow \exists \mathbb{L} \in \mathcal{E}f\!\!f(c,m) \downarrow_1 \ : \ \mathbb{L} \subseteq \mathbb{L}' \\ \mathcal{M}(c,m) = t_r \ m(t_x), \mathbb{L} \in \mathcal{E}f\!\!f(c,m) \downarrow_1 \Longrightarrow \\ \mathbb{L}, (\texttt{self} \ c,t_x) \vdash \mathcal{M}\mathcal{B}\textit{ody}(c,m) : \mathcal{E}f\!\!f(c,m) \downarrow_2 \\ \hline \vdash c \end{array}$$
 (WFCLASS)

Figure 3.10: Static race safety type system

(PATH). If both rules are applicable, then the locks obtained will be syntactically different (e.g. self and this), but will indicate the same owner.⁵

The (SYNC) rule calculates the synchronisation set for the expression sync e e' by removing the lock that guards the object accessed by e from the synchronisation set of e'.

Because methods in our system are not necessarily internally synchronised, we extend their signatures through $\mathcal{E}ff$, whose shape is defined in Fig. 3.10, which returns a set of synchronisation sets, and a set of fields to which the method body may assign. The (CALL) rule thus requires that these locks and assignments are included in the resulting synchronisation set and effects F. We use a *set* of synchronisation sets, rather than a single synchronisation set because there may be more than one correct way to synchronise a method call. E.g. a method with body this.f.s = x might have $\mathcal{E}ff$ as follows: ⁶

 $(\{\{\texttt{this},\texttt{this}.f\},\{\texttt{this},\texttt{rep}\},\{\texttt{self},\texttt{this}.f\},\{\texttt{self},\texttt{rep}\}\},\{\texttt{s}\}).$

The synchronisation sets expressed in $\mathcal{E}ff$ are given from the perspective of the target of the method call, so they need to be translated into the perspective of the receiver before being used. This is done through the operator \mathbb{P} , defined in Fig. 3.10.

Well-formed classes, cf., (WFCLASS), requires, in addition to the requirements imposed for universe type soundness, that: Firstly, if a method m existed in a superclass, then the superclass's synchronisation sets and effects should be larger than those in the subclass. Secondly, each of synchronisation sets \mathbb{L} in $\mathcal{E}ff(c,m)\downarrow_1$ should be sufficient for correct synchronisation of the body of m.

Programs will not exhibit race conditions if they are well-typed in both the race safety type system and the universe type system. The derivation trees for the two systems need not correspond. We could have presented the intersection of the type systems as a single type system but since the universe type system can stand on its own, we chose to present

⁵Obviously, if neither rule is applicable the expression is type incorrect.

⁶The synchronisation sets will grow with the number of accesses in a method body. Therefore, in practice we would need a better syntax that scales more favourably, e.g. this|self, this.first|rep. This is outside the scope of this thesis.

the race safety type system alone. Also, there is only limited communication between the systems – we use the universe type judgement in the logic of the guard judgement and to get the class c in (CALL). One can imagine how the race safety type system might be "plugged" into other universe or ownership type systems, e.g. ones that consider generics.

Examples

We now discuss the application of our type rules with some examples. We first consider the body of releaseMarks in fig. 3.1. We have an environment Γ_1 where $\Gamma_1(\text{this}) =$ self Dept. Because our tiny language does not include local variables, we will consider i as a field in class Dept of type rep DeptStudentNode, and mentally map each appearance of i in fig. 3.1, onto this.i. Thus,

 $\emptyset, \Gamma_1 \vdash \text{sync (this)} \{ \text{this.i} = \text{this.first}; \\ \text{sync (this.i)} \{ \text{this.i.s.mark} = \dots; \\ \text{this.i} = \text{this.i.next} \} \} : \{ i, \text{mark} \}$

On the other hand, in fig. 3.1, line 41, we obtain the following with an environment Γ_2 where $\Gamma_2(\texttt{this}) = \texttt{self Hall}$:

Because this.i.s has type any Student, the type system can only use the guard rule (PATH). The type system accepts the above synchronised block because we are not assigning to the fields i or s within the block. However, the following expression would be type incorrect in Γ_2 , and thus we are forced to lock at every loop iteration.

sync(this.i) {... this.i = this.i.next }

The method badCode() in Fig. 3.11 accesses and synchronises this.getFirst() (line (11)). This is not a path, but has type rep DeptStudentNode so the type system can use (UNIV) rule to accept the code. The sync (this.first2) block (line (14)) fails type checking because the path being locked is any and also comprises a field first2 which

```
1
   class BrokenDept extends Dept {
 2
       any Student first2;
       any DeptStudentNode getFirst2() {
 3
 4
           sync (this) { return this.first2; }
 5
       ŀ
 6
       rep DeptStudentNode getFirst() {
 7
           sync (this) { return this.first; }
 8
       }
 9
       void badCode() {
10
           sync (this) {
11
               sync (this.getFirst()) {
12
                  this.getFirst().s = NULL;
13
               }
14
               sync (this.first2) { // FAIL
15
                  this.first2 = this.first2.next;
16
                  this.first2.s = NULL;
17
               }
18
               sync (???) {
                  this.getFirst2().s = NULL; //FAIL
19
20 }
       } }
               }
```

Figure 3.11: Example

is assigned during the synchronised block. The final access (line (19)) is not a path (due to containing a method call) and has type **any**, so no amount of synchronisation will persuade the type system to accept it.

Finally, we give examples of method calls. Assume a method clean()⁷ in class Student such that:

$$\mathfrak{E}\!f\!f(\mathtt{Student}, \mathtt{clean}) = (\{\{\mathtt{self}\}\}, \{\mathtt{cleanRoom}\})$$

Then, in class Dept, it holds that $\Gamma_1 \vdash \texttt{this.first.s}$: rep Student. Thus, by application of (CALL), we obtain:

$$\{rep\}, \Gamma_1 \vdash this.first.s.clean() : \{cleanRoom\}$$

In class Hall, $\Gamma_2 \vdash$ this.first.s : any Student. Here, this.first.s.clean() causes a type error. On the other hand, with a method makeBed(), where:

 $\mathcal{E}ff(\texttt{Student}, \texttt{makeBed}) = (\{\{\texttt{self}\}, \{\texttt{this}\}\}, \{\texttt{cleanRoom}\})$

⁷For simplicity, we ignore method parameters.

We would obtain

 $\mathbb{L}, \Gamma_2 \vdash \texttt{this.first.s.makeBed()} : \{\texttt{cleanRoom}\}$ (where $\mathbb{L} = \{\texttt{this,this.first,this.first.s}\}$)

3.6.2 Run-time Type System

As is standard, we give a run-time type system in order to prove soundness and race safety (presented in Fig. 3.12). We type run-time expressions e according to a heap hand stack σ . The judgement has the shape $\mathbb{L}, h, \sigma \vdash e : F$. The meanings of \mathbb{L} and Fare unchanged. The function $h(\sigma, p)$ calculates the path p in the given heap and stack to retrieve a value in a finite number of steps bounded by the size of p. If this process attempts to dereference null, we define it to return null. In Fig. 3.10 we used (PATH) and (Univ) to derive locks from expressions. We needed to extend this functionality to derive locks from partially executed expressions so we added the rule (VAL) and replaced (PATH) by the rules (VAR) and (FIELD). (UNIV) was changed to use the run-time universe type system, which understands partially executed expressions. (CALL) needed us to extend (\triangleright) to translate locks in the context of partially executed targets and arguments.

The type system is lifted to the sequence of threads that ultimately comprises our model of the run-time state by (THREADS). We require all the threads to be internally synchronised and also that no two threads have the same lock. The shape of the judgement is $h, \sigma \vdash \overline{e}$.

We use the predicate Virgin(e) to note that e has not yet been executed i.e., contains no addresses, synced or frame constructs. Reachable(e) (Fig.3.13) denotes that the subterms of e have been executed in the right order, e.g., Reachable(a.f = y.f) but $\neg Reachable(y.f = a.f)$. We extend this to sequences of expressions $Reachable(\overline{e})$ if all the expressions are reachable.

Because of the instrumentation of **sync** with a subscript that records the initial lock expression, we need to use the same substitution as used in the semantics rule (CALL)

Figure	3.12	Run-time	race	safety	type	system
				•/		•/

Reachable(e)	\Leftarrow	$e \in \{\mathtt{x}, \mathtt{this}, v, \mathtt{new} \; t\}$
Reachable(e.f)	$\Leftarrow\!\!=$	Reachable(e)
Reachable((t)e)	$\Leftarrow\!\!=$	Reachable(e)
$Reachable(e_1.f = e_2)$	\Leftarrow	$Reachable(e_1) \wedge Virgin(e_2)$
Reachable(v.f = e)	\Leftarrow	Reachable(e)
$Reachable(e_1.m(e_2))$	$\Leftarrow\!\!=$	$Reachable(e_1) \wedge Virgin(e_2)$
Reachable(v.m(e))	\Leftarrow	Reachable(e)
$Reachable(\texttt{spawn}\ e)$	\Leftarrow	Virgin(e)
$Reachable(\texttt{sync}_{e_1} \ e_2 \ e_3)$	\Leftarrow	$Virgin(e_1) \land Reachable(e_2) \land Virgin(e_3)$
$Reachable(synced_{e_1} w e_2)$	\Leftarrow	$Virgin(e_1) \land Reachable(e_2)$
$Reachable(\texttt{frame } \sigma e)$	$\Leftarrow\!\!=$	Reachable(e)

Figure 3.13: Definition of *Reachable*

when defining the following substitution lemma:

Lemma 3.6.2 Static race safety implies run-time race safety.

$$\left. \begin{array}{c} \mathbb{L}, \Gamma \vdash e : F \\ h, \sigma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \\ h, \sigma \vdash \mathtt{this} : \Gamma(\mathtt{this}) \end{array} \right\} \Longrightarrow \begin{array}{c} \mathbb{L}, h, \sigma \vdash S(e) : F \\ Virgin(S(e)) \end{array}$$

Proof: Induction over derivation of $\mathbb{L}, \Gamma \vdash e: F$

The following lemmas were needed to support the proof of soundness for the race safety type system. Firstly, we can prove that the execution of a path yields another path with no more fields than the original path, does not change the heap, and the new path resolves to the same value as the original path. This was necessary to show that the execution of expressions does not change the lock that guards it.

Lemma 3.6.3 Path resolution is preserved over execution.

$$\left.\begin{array}{l} h(\sigma,p)=v\\ \sigma\vdash p,h\rightsquigarrow e,h'\end{array}\right\} \Longrightarrow \begin{array}{l} e=p',h=h'\\ h'(\sigma,p')=v\\ \forall f\not\in p\ :\ f\not\in p' \end{array}$$

Proof: induction over $\sigma \vdash p, h \rightsquigarrow e, h'$.

A given expression should be guarded by the same lock no matter what state of execution the expression has reached. We require heap well-formedness because we need the universe type judgements within the guard logic to be preserved over the execution of e.

Lemma 3.6.4 Guards are preserved over execution.

$$\left. \begin{array}{l} h, \sigma \vdash_{gb} e: l \\ \vdash h \\ \sigma \vdash e, h \rightsquigarrow e', h' \end{array} \right\} \Longrightarrow h', \sigma \vdash_{gb} e': l$$

Proof: Case analysis of $h, \sigma \vdash_{gb} e : l$.

If the heap has changed enough since the lock p was taken that p resolves to a different object, then the lock l = p no longer guards the same objects. This lemma establishes that the heap changes are not sufficient to cause this, as long as the effect F of the execution does not contain a field used by p.

Lemma 3.6.5 Path resolution is preserved over execution of other expressions.

$$\begin{array}{c} h(\sigma, p) = v \\ \sigma' \vdash e, h \rightsquigarrow e', h' \\ _, h, \sigma' \vdash e : F \\ \{p\} \ \# \ F \end{array} \end{array} \right\} \Longrightarrow h'(\sigma, p) = v$$

Proof: Induction over steps of resolution.

The next lemma helps to prove that the guard of an expression should be unaffected by the execution of other expressions. We require the reducing expression will not interfere with any paths that the guard might be using, in order to invoke lemma 3.6.5. Lemma 3.6.6 Guards are preserved over the execution of other expressions

$$\left. \begin{array}{l} h, \sigma \vdash_{gb} e : l \\ \sigma' \vdash e', h \rightsquigarrow _, h' \\ _, h, \sigma \vdash e' : F' \\ \{l\} \ \# \ F' \end{array} \right\} \Longrightarrow h', \sigma \vdash_{gb} e : l$$

Proof: Induction over $h, \sigma \vdash_{gb} e : l$.

The next lemma complements lemma 3.6.6. It uses the fact that if Virgin(e), only (FIELD) and (VAR) are used in the derivation of $h, \sigma \vdash_{gb} e : p$. This means the derivation uses neither h nor σ when l = p. Lemma 3.6.7 Virgin guards are preserved over the execution of other expressions.

$$\left. \begin{array}{c} h, \sigma \vdash_{gb} e : l \\ Virgin(e) \\ _ \vdash _, h \rightsquigarrow _, h' \end{array} \right\} \Longrightarrow h', \sigma \vdash_{gb} e : l$$

Proof: Induction over $h, \sigma \vdash_{gb} e: l$.

The following lemma is used to show that if e is part of an expression, and e has not yet started executing, then its race safety is unaffected by any other part of the expression that does happen to be executing.

Lemma 3.6.8 Types of virgin expressions are preserved over the execution of other expressions.

$$\mathbb{L}, h, \sigma \vdash e : F \\ Virgin(e) \\ _ \vdash _, h \rightsquigarrow _, h'$$

$$\left\} \Longrightarrow \mathbb{L}, h', \sigma \vdash e : F \\$$

Proof: Induction over $\mathbb{L}, h, \sigma \vdash e : F$.

This lemma requires that the locks taken by the executing thread e' are disjoint from the locks \mathbb{L} guarding the path p. This means that the modifications made by e' cannot affect the resolution of the path. This lemma complements lemma 3.6.5.
Lemma 3.6.9 Path resolution is preserved over the execution of other expressions when locks do not collide.

$$\begin{split} h(\sigma,p) &= v \\ \sigma' \vdash e', h \rightsquigarrow _, h' \\ \emptyset, h, \sigma' \vdash e' : _ \\ \mathbb{L}, h, \sigma \vdash p : _ \\ \{h(a)\downarrow_1 | h, \sigma \vdash_{gb} a : l, l \in \mathbb{L}\} \cap \{w | Locked(e',w)\} = \emptyset \end{split} \right\} \Longrightarrow h'(\sigma,p) = v \end{split}$$

Proof: Induction over $\mathbb{L}, h, \sigma \vdash e : F$.

We can use the above lemma to show that under similar conditions, the guarded by judgement is unaffected by the execution of another thread.

Lemma 3.6.10 Guards are preserved over the execution of other expressions when locks do not collide.

$$\begin{array}{l} h, \sigma \vdash_{gb} e : l \\ l \in Path \Longrightarrow \mathbb{L}, h, \sigma \vdash l : _ \\ \sigma' \vdash e', h \rightsquigarrow _, h' \\ \emptyset, h, \sigma' \vdash e' : _ \\ \{h(a)\downarrow_1 | h, \sigma \vdash_{gb} a : l, l \in \mathbb{L}\} \cap \{w | Locked(e', w)\} = \emptyset \end{array} \right\} \Longrightarrow h', \sigma \vdash_{gb} e :$$

Proof: Induction over $\mathbb{L}, h, \sigma \vdash e : F$.

Now we can show that that the execution of one thread e' will not interfere with the typing of another thread e, assuming the locks simultaneously held by both threads are disjoint and the locks \mathbb{L} required by e have not been taken by e'.

l

Lemma 3.6.11 Types are preserved over the execution of other expressions when locks do not collide.

$$\begin{split} \mathbb{L}, h, \sigma \vdash e : F \\ \sigma' \vdash e', h \rightsquigarrow _, h' \\ \emptyset, h, \sigma' \vdash e' : _ \\ Reachable(e) \\ \forall w : \neg (Locked(e, w) \land Locked(e', w)) \\ \{h(a) \downarrow_1 | h, \sigma \vdash_{gb} a : l, l \in \mathbb{L}\} \cap \{w | Locked(e', w)\} = \emptyset \end{split} \right\} \Longrightarrow \mathbb{L}, h', \sigma \vdash e : F \\ \end{split}$$

Proof: Induction over $\mathbb{L}, h, \sigma \vdash e : F$.

Using the substitution lemma in the case of method calls, it is now possible to prove the race safety type system is sound. Firstly we state soundness for single-threaded execution. Note that we require the heap to be well-formed. This is necessary so that field accesses yield objects of the correct owner.

Lemma 3.6.12 The type of a thread is preserved over the execution of that thread.

$$\begin{array}{c} Reachable(e) \\ \vdash h \\ h, \sigma \vdash e:t \\ \mathbb{L}, h, \sigma \vdash e:F \\ \sigma \vdash e, h \rightsquigarrow e', h' \end{array} \end{array} \Longrightarrow \begin{array}{c} \mathbb{L}, h', \sigma \vdash e':F \\ Reachable(e') \\ Reachable(e') \end{array}$$

`

Proof: Induction over derivation of $\mathbb{L}, h, \sigma \vdash e : F$

Now, we extend the above result to multiple threads:

Theorem 3.6.13 Well-typedness of the system is preserved over execution.

 $\left. \begin{array}{c} \vdash h \\ h, \sigma \vdash \overline{e} : \overline{t} \\ h, \sigma \vdash \overline{e} \\ \sigma \vdash \overline{e}, h \rightsquigarrow \overline{e}', h' \\ Reachable(\overline{e}) \end{array} \right\} \Longrightarrow \begin{array}{c} h', \sigma \vdash \overline{e}' \\ Reachable(\overline{e}') \end{array}$

Proof: Case analysis of $\sigma \vdash \overline{e}, h \rightsquigarrow \overline{e}', h'$

The race safety type system has been shown to be sound, so we will now work towards a theorem of race safety, i.e. that well-typed programs will not exhibit race conditions. Firstly the following lemma states that objects are only accessed if the appropriate lock has been acquired by the thread in question. Note the use of the action a to denote an access of address a by the execution step.

Lemma 3.6.14 Objects are only accessed while their owners are locked.

$$\left. \begin{array}{c} \mathbb{L}, h, \sigma \vdash e: _, \\ \sigma \vdash e, h \stackrel{a}{\rightsquigarrow} _, _ \end{array} \right\} \Longrightarrow \begin{array}{c} (\exists l \in \mathbb{L} : h, \sigma \vdash_{gb} a: l) \lor \\ Locked(e, h(a)\downarrow_1) \end{array}$$

Proof: Induction over structure of $\mathbb{L}, h, \sigma \vdash e : _$.

The multi-threaded case follows. We are dealing with entire threads here (as opposed to sub-terms), thus there is no context. Therefore, unlike the above lemma, we do not mention any set of locks \mathbb{L} taken by the current context.

Theorem 3.6.15 Objects are only accessed while their owners are locked by the corresponding thread.

$$\left. \begin{array}{c} h, \sigma \vdash e_{1..n} \\ \sigma \vdash e_{1..n}, h \xrightarrow{(i,a)} _, _ \end{array} \right\} \Longrightarrow \ Locked(e_i, h(a)\downarrow_1)$$

Proof: Case analysis of $\sigma \vdash e_{1..n}, h \stackrel{(i,a)}{\leadsto} _, _$

For race conditions, we use the definition from [30], where the state of a multi-threaded system exhibits an *instantaneous race condition* if the semantics allows two possible execution steps of different threads that both access the same object. The required non-determinism is provided by the (INTERLEAVE) rule.

We prove that no well-typed state can ever have an instantaneous race condition, and by theorem 3.6.13, all intermediate states of execution will be free from instantaneous race conditions. We show, for an arbitrary well-typed run-time state that if two possible execution steps can access the same object, then those steps must be steps of the same thread:

Theorem 3.6.16 Race safety

$$\left. \begin{array}{c} h, \sigma \vdash \overline{e} \\ \sigma \vdash \overline{e}, h \stackrel{(i,a)}{\leadsto} _, _ \\ \sigma \vdash \overline{e}, h \stackrel{(j,a)}{\leadsto} _, _ \end{array} \right\} \Longrightarrow i = j$$

١

We prove race safety as follows:

(1)
$$h, \sigma \vdash \overline{e}$$

(2) $\sigma \vdash \overline{e}, h \xrightarrow{(i,a)} _, _$
(3) $\sigma \vdash \overline{e}, h \xrightarrow{(j,a)} _, _$
let $w = h(a)\downarrow_1$ (4)
(1) + (2) + (4) + thm3.6.15 \Rightarrow locked(e_i, w) (5)
(1) + (3) + (4) + thm3.6.15 \Rightarrow locked(e_j, w) (6)

$$(1) + (\text{Threads}) \Rightarrow$$

$$\forall w', i, j \in \{1..n\}.$$

$$locked(e_i, w') \land locked(e_j, w') \Rightarrow i = j \quad (7)$$

$$(5) + (6) + (7) \Rightarrow i = j \qquad \Box$$

3.6.3 Typing Algorithm

The rules in fig. 3.10 give constraints that help us to understand which programs are welltyped and which should be rejected, however they do not tell us how to automatically check conformance in a compiler. To address this, we now give an algorithm that will check if a program is well-typed.

The problem can be decomposed into checking classes independently. This is done with the programmer annotations already known, i.e., with a given $\mathcal{E}ff$, and \mathcal{M} . We also know the class hierarchy, i.e., the \leq relation. We use these as global variables in the pseudocode below, as they are constant for a particular program.

For each class we must first check the first two lines of (WFCLASS) but this not difficult given that $\mathcal{E}ff$, \mathcal{M} and \leq are finite and available. We also check that the annotations are valid, by checking that each of the provided \mathbb{L} do not clash with the provided F for the method, using the # operator defined earlier. It is necessary to check that all the paths in each \mathbb{L} are protected by locks in the same \mathbb{L} . The algorithm to do this is a particular instance of the main typing algorithm which is used to check the method body.

The method body is checked against all synchronisation sets in the method annotation, i.e., $\mathcal{E}ff(c,m)\downarrow_1$, to make sure that each is on its own a sufficient set of locks for protecting the execution of the method. We use a syntax directed recursive function called **check**, given in fig. 3.14. This function either returns the smallest set F for the given code, or raises an error. We ensure the returned F is contained within the set that is annotated for the method. The syntax **try** ... **end** catches any errors while evaluating its body and silently ignores them. This allows us to search more than one way of typing a sub expression, and ignoring possibilities that yield a deeper error.

The function $locks(\Gamma, e)$ finds all locks l such that $\Gamma \vdash_{gb} e : l$, or the empty set if none can be found. This is straightforward to define if we assume a function $universe(\Gamma, e)$, so we omit this pseudocode. We are assuming the code has already passed a universe type checking system, and the function universe exists to query information from this

```
function <u>check</u> (\mathbb{L}, \Gamma, e)
      let F = \emptyset
      case e in
            this: return {\cal F}
            x: return F
            null: return F
            \verb"new c: return F"
             (t)e: return \underline{check}(\mathbb{L},\Gamma,e)
             spawn e:
                   \underline{\mathtt{check}}(\mathbb{L},\Gamma,e)
                   return F
             e.f:
                   F = \underline{check}(\mathbb{L}, \Gamma, e)
                   for each 1 in \underline{locks}(\Gamma, e)
                       if l in \mathbb L return F
                   end
                   error
             e.f = e':
                   F.include(\underline{check}(\mathbb{L},\Gamma,e))
                   F.include(\underline{check}(\mathbb{L},\Gamma,e'))
                   F.include(f)
                   foreach 1 in locks(\Gamma, e)
                       if 1 in L return F
                   end
                   error
             e.m(e'):
                  let u, c = \underline{universe}(\Gamma, e)
                   \texttt{F.include}(\overline{\textit{E\!f\!f}(c,m)}{\downarrow_2})
                   F.include(\underline{check}(\mathbb{L},\Gamma,e))
                   F.include(\overline{\underline{check}}(\mathbb{L},\Gamma,e'))
                   foreach \mathbb{L}' in \mathcal{E}\!\!f\!f(c,m)\!\!\downarrow_1
                          try
                                if \underline{\text{translate}}(u, e, e', \mathbb{L}') \subseteq \mathbb{L} return F
                          {\tt end}
                   end
                   error
             sync e e':
                   F.include(\underline{check}(\mathbb{L},\Gamma,e))
                   foreach 1 in \underline{locks}(\Gamma, e)
                         try
                                F.include(\underline{check}(\mathbb{L} \cup \{l\}, \Gamma, e'))
                                return F
                         end
                   end
                   error
      {\tt end}
end
```

Figure 3.14: Pseudocode Typing Algorithm

previous pass. It returns the type of the expression e, either peer, rep, or, as a last resort, any⁸, as well as the class c. The function universe is also used in fig. 3.14 in the case of method calls.

Finally, we need a function translate (u, e, e', \mathbb{L}) that is defined similarly to the \triangleright operator defined in fig. 3.10. It will return the set of locks \mathbb{L} translated to the calling context. If \triangleright is undefined for a given input then translate will raise an error for the same input. We use the try statement to iterate through all of the annotated synchronisation sets of a method when it is called, searching for one that can be translated into the calling context. This is because, if possible, methods should be annotated with a synchronisation set for when they are called through an any target, as well as a different synchronisation set for when they are called through a non-any target. Type checking will therefore pass if there is at least one way of protecting the body of the called method using the locks that have been acquired by the caller.

3.7 Implementation Issues

For implementing our type system, we have considered ways to achieve good performance within our semantics.

The constraints on extending classes may seem severe, particularly the requirement that an overriding method cannot have more effects F than the original method. However, if we forsake separate compilation, we can infer the effect F of each method, and thus we do not need to restrict inheritance. We believe the constraints on the synchronisation set required for a call to be race safe are not so severe because most functions will be internally synchronised, whereas one cannot hide field assignments F within a function. In general separate compilation does not mix well with concurrency since concurrency is concerned with the effect of the rest of the program. Separate complication requires specification at module boundaries, and a behaviour specification is quite hard to write

⁸A typing algorithm for the universe type system is beyond the scope of this thesis.

and maintain. In our work such a specification is represented by the sets \mathbb{L} and F.

In practice, we could use more accurate techniques of alias detection, e.g., using a points-to analysis, to refine the type system's judgement about whether a field assignment affects a path p in \mathbb{L} . This would allow us to reject fewer correct programs. We could easily distinguish between identically named fields in different classes by prepending the class name to all fields.

It may be useful to use the method of [12] for preventing deadlock. We can require the programmer to write additional type annotations to firstly divide the heap into a statically bounded set of regions [61] and secondly specify a partial order over these regions. Types are therefore augmented with a region identifier, which specifies the region where the owner of the object lies (locks are associated with the owner). The type system would check that the specified order has no cycles, and that assignments do not let variables of a certain region reference objects of other regions.

To actually prevent deadlocks, the type system has to ensure that locks are taken (i.e. sync blocks are nested) in the order specified. Locks can statically be given numeric values. Method signatures can be annotated with the lowest lock that they take. Thus each call can be checked to make sure its context has not already locked a lock with greater value than any lock the method will acquire.

Since we require an implicit owner field in all objects, for casts and synchronisation of **any** expressions, there may be unnecessary memory overhead⁹. If this becomes an issue, we propose introducing new types to lay alongside **rep**, **peer**, and **self**, which would have identical semantics except that they may not be cast to **any**. Objects of these types would not require an owner field since the owner would always be statically known. We anticipate that programmers would use these types only when memory usage was a problem. Static analysis could also be used to infer those objects that are never cast to **any**, and optimise away the owner field.

Other type systems [8, 37, 35, 11, 12] have extra features such as thread local storage

 $^{^{9}}$ In some cases, previous work required the programmer to use an explicit owner field, see fig. 3.2

CHAPTER 3. LOCK CHECKING

and final variables. We did not formalise them, as although they are useful in practice, they are well-understood and can be easily added to an implementation.

3.7.1 Distinguishing Reads and Writes

We consider two simultaneous accesses of the same object by different threads to be a race condition, but this need only be so if one of the accesses is a write. Distinguishing between reads and writes would allow more liberal synchronisation. We now show how this can be done with an extension of our formalism.

We need to distinguish between read and write accesses, i.e. distinguish between field lookups and field assignments. We also need read/write locks, i.e. we need a pair of constructs like $\operatorname{sync}^R(...)$ and $\operatorname{sync}^W(...)$ that attempt to acquire the read/write lock on an object, respectively. The semantics would allow multiple threads to have the read lock at any one time, so long as no other thread was writing. This would be reflected in the rule (THREADS), which would ensure that if a one thread is writing and another is reading the same object then both threads must be the same. These locks have already been implemented in Java, and are straight-forward to add to the type system.

3.7.2 Single Object Locking

While we need to lock all objects owned by an object when iterating through nodes as in fig. 3.1 (line 30), we do not need to lock all the peers of that object if we do not use it for iteration. E.g. we do not need to lock the peers of this when we access this.first. We'd like to give the programmer the choice of when to lock an object and all its peers (as is currently available with sync) and when to lock only a single object. Then, it would be possible for two threads to execute in parallel the releaseMarks method of the two departments in the diagram of fig. 3.1. We extend the type system and semantics as shown in Fig. 3.15.

Programmers use syncobj when they only want to lock the object in question e.g. fig. 3.1 (line 40), and the old sync construct if they want to lock that object and all of its

$$\begin{array}{c} \mathbb{L}, \Gamma \vdash e: F \\ \Gamma \vdash_{gb} e: l = (*, _) \\ \mathbb{L} \cup \{l\}, \Gamma \vdash e': F \\ \overline{\mathbb{L}}, \Gamma \vdash \text{sync } e: e': F \end{array} \\ (SYNC) \\ \hline \mathbb{L}, \Gamma \vdash \text{sync } e: e': F \end{array} \\ \begin{array}{c} \mathbb{L}, \Gamma \vdash \text{sync } e: l = (1, _) \\ \mathbb{L} \cup \{l\}, \Gamma \vdash e': F \\ \overline{\mathbb{L}}, \Gamma \vdash \text{sync } b: i = j \end{array} \\ (SYNC) \\ \hline \mathbb{L} \cup \{l\}, \Gamma \vdash e': F \\ \hline \mathbb{L}, \Gamma \vdash \text{sync } b: i = j \end{array} \\ (LOCKDOMAIN) \\ \hline \phi \vdash e_{1..n}, h^{(i,\tau)} e_{1..i-1} C[\text{synced}_{e'} w e] e_{i+1..n}, h \\ e_i = C[\text{syncobj}_{e'} a e] \\ \forall j \in \{1..n\} : Locked(e_j, h(a)\downarrow_1) \Longrightarrow i = j \\ (LOCKOBJ) \\ \hline \sigma \vdash e_{1..n}, h^{(i,\tau)} e_{1..i-1} C[\text{synced}_{bj}(e_j, a) \Longrightarrow i = j \end{array} \\ (LOCKOBJ) \\ \hline \sigma \vdash e_{1..n}, h^{(i,\tau)} e_{1..i-1} C[\text{syncedobj}_{e'} a e] e_{i+1..n}, h \\ \\ \text{Source syntax: Run-time syntax:} \\ e::= \dots \mid \text{syncobj } e e e::= \dots \mid \text{syncobj}_e e e \mid \text{syncedobj}_e a e \\ \forall i \in \{1..n\} : \emptyset, h, \sigma \vdash e_i : _ \\ \forall i, j \in \{1..n\}, h(a)\downarrow_1 = w : \\ Locked(e_i, w) \land Locked(Obj(e_j, a) \Longrightarrow i = j \\ Locked(e_i, w) \land Locked(Obj(e_j, a) \Longrightarrow i = j \\ Locked(e_i, w) \land Locked(Obj(e_j, a) \Longrightarrow i = j \\ Locked(b_j(e_i, a) \land Locked(Obj(e_j, a) \Longrightarrow i = j \\ Locked(b_j(e_i, a) \land Locked(Obj(e_j, a) \Longrightarrow i = j \\ Locked(b_i, w) \land Locked(Obj(e_j, a) \Longrightarrow i = j \\ Locked(b_j(e_i, a) \land Locked(Obj(e_j, a) \Longrightarrow i = j \\ Locked(b_j(e_i, a) \land Locked(Obj(e_j, a) \Longrightarrow i = j \\ Locked(b_j(e_i, a) \land Locked(Obj(e_j, a) \Longrightarrow i = j \\ Locked(Dbj(e_i, a) \land Locked(Obj(e_j, a) \Longrightarrow i = j \\ Locked(Dbj(e_i, a) \land Locked(Obj(e_j, a) \Longrightarrow i = j \\ Locked(Dbj(e_i, a) \land Locked(Obj(e_j, a) \Longrightarrow i = j \\ Locked(Dbj(e_i, a) \land Locked(Dbj(e_j, a) \Longrightarrow i = j \\ Locked(Dbj(e_i, a) \land Locked(Dbj(e_j, a) \Longrightarrow i = j \\ Locked(Dbj(e_i, a) \land Locked(Dbj(e_j, a) \Longrightarrow i = j \\ Locked(Dbj(e_i, a) \land Locked(Dbj(e_j, a) \Longrightarrow i = j \\ Locked(Dbj(e_i, a) \land Locked(Dbj(e_j, a) \Longrightarrow i = j \\ Locked(Dbj(e_i, a) \land Locked(Dbj(e_j, a) \Longrightarrow i = j \\ Locked(Dbj(e_i, a) \land Locked(Dbj(e_j, a) \Longrightarrow i = j \\ Locked(Dbj(e_i, a) \land Locked(Dbj(e_j, a) \Longrightarrow i = j \\ Locked(Dbj(e_i, a) \land Locked(Dbj(e_j, a) \Longrightarrow i = j \\ Locked(Dbj(e_i, a) \land Locked(Dbj(e_j, a) \Longrightarrow i = j \\ Locked(Dbj(e_i, a) \land Locked(Dbj(e_j, a) \Longrightarrow i = j \\ Locked(Dbj(e_i, a) \land Locked(Dbj(e_j, a) \Longrightarrow i = j \\ Locked(Dbj(e_i, a)$$



peers. A thread may not lock all the peers of an object if any of its peers has already been locked by another thread. Likewise, we do not let a thread lock a single object if another thread has locked all the peers of that object. We use the old predicate Locked(e, w) but we also add LockedObj(e, a), which holds when e contains $syncedobj_{e'} a e''$ for some e', e''.

We augment the lock syntax so the "all peers' locks" are now denoted with (*, u) or (*, p), whereas the single object locks are denoted with (1, p). We do not need (1, u), as a universe qualifier does not identify a single object. We updated the (UNIV) and (PATH) rules to wrap their result in (*, ...) and give a new guard rule that returns (1, p) if the expression is a path. The type rule for threads guarantees that no two threads have conflicting locks.

An efficient implementation might use a counter in the owner of an object to record how many of its child objects have been individually locked, rather than iterating through them all. Lock-free programming techniques can be used to ensure this has negligible performance cost compared to a standard mutex implementation. There has been a lot of research [7] into increasing the performance of lock operations in the most common cases.

We believe single object locking (in the context of static race safety checkers) is a novel idea. None of the prior work supports it. It is a small extension of our formalism, and as such we did not incorporate it into our proofs; however, in practice it should be simple to implement and allow more parallelism.

3.7.3 Atomicity

In some sense, the requirement for freedom from races is too weak: A program that does not type in our system, can be "corrected" by wrapping each access in a tiny synchronised block, thus converting all its race conditions to stale value errors. As described by Flanagan [36], there are program misbehaviours due to thread interactions that are not classed as race conditions by the standard definition.¹⁰ Ideally, we would like to identify atomicity violations [34, 35, 36] in programs, these include bugs such as stale value errors that we currently cannot detect. This can be done by requiring that a block is *atomic*, which means that the execution of such a block in the context of arbitrary threads will be equivalent to a serialised execution of that block with no interleaving of other threads.

We therefore suggest the programmer both annotates blocks with **atomic** when the block should be atomic, but also uses **sync** to achieve race safety and atomicity. Race safety can be checked using the type system already presented. We would extend this type system to recognise **atomic** and place additional restrictions on the locks found within. The annotation **atomic** does not affect the semantics, universe type system, or race safety type system. It only causes more programs to be rejected if it is used in a context where the enclosed **sync** blocks are not appropriate for atomicity.

A block is guaranteed to be atomic if the whole program has no races and if the block is *two-phase* [68]. Two-phase means that non-redundant sync blocks are always nested (never sequenced). Thus the following code is not two-phase: sync (x) {x.f = 1}; sync (y) {y.f = 2}. However, if we wrapped it in another sync (y) then the last sync block would become redundant and the whole block would be two-phase. It is natural to provide this two-phase checking as an extension to our race safety system that additionally requires non-redundant sync blocks to always be nested. We could present the whole type system again with the extra constraints, but for clarity we will instead give another type system that requires only the additional constraints needed for atomicity. We invoke this type system by modifying (WFCLASS).

We propose the additional judgement $\mathbb{A}, \mathbb{B}, \Gamma \vdash e : F$, we give the rules in Fig. 3.16. The sets \mathbb{A} and \mathbb{B} are subsets of *Lock*, and we use the set F of fields to prevent the locks in \mathbb{A} and \mathbb{B} being spoilt by assignments. \mathbb{A} contains the locks that must be taken in order to make e race-free and two-phase.¹¹ Therefore if e is two-phase and race-free then \mathbb{A}

¹⁰Some programmers use "race condition" to describe these errors as well.

¹¹Since \mathbb{A} is ignored if the program has no atomic sections, we still need the original type system and \mathbb{L} .

must be empty (modulo subsumption). \mathbb{B} is the set of locks required to render all of the synchronisation redundant. Therefore if $\mathbb{A}, \mathbb{B}, \Gamma \vdash e : F$ then $\mathbb{A} \subseteq \mathbb{B}$, and also $\mathbb{A} \# F$ and $\mathbb{B} \# F$.

Object accesses generate locks in \mathbb{A} and \mathbb{B} like they have previously in \mathbb{L} , in rules (Assign) and (Field). Locks are not eliminated from \mathbb{B} by (Sync).

The need to distinguish A and B arises from the cases where expressions have more than one subexpression, e.g., assignment, synchronisation, and method call. This can be demonstrated in the simpler case of sequential execution i.e., e; e', if one were trivially added to the model. For the expression e; e' to be two-phase, it is not sufficient to take the union of the locks required to make e two-phase and e' two-phase. Instead we need to take either the locks required so that e is two-phase and the locks of e' are all redundant, or vice versa. For example, sync (x) {x.f = 1} is two-phase, and so is sync (y) {y.f = 2}; but their sequential execution, i.e., sync (x) {x.f = 1}; sync (y) {y.f = 2}, is not. Instead, we should either take x or y, therefore sync (x) {sync (x) {x.f = 1}; sync (y) {y.f = 2}. Such a rule for (;) is illustrated below:

$$\mathbb{A}, \mathbb{B}, \Gamma \vdash e : F \quad \mathbb{A}', \mathbb{B}', \Gamma \vdash e' : F$$
$$\mathbb{A}'' \in \{\mathbb{A} \cup \mathbb{B}', \mathbb{A}' \cup \mathbb{B}\} \qquad (Seq)$$
$$\mathbb{A}'', \mathbb{B} \cup \mathbb{B}', \Gamma \vdash e; e' : F$$

The idiom $\mathbb{A}'' \in \{\mathbb{A} \cup \mathbb{B}', \mathbb{A}' \cup \mathbb{B}\}$ is used where one sub-term is executed after another, and is used in (Assign), (Sync), and also in (CALL), which needs to consider the body of the method and thus involves three sets.

(CALL) also requires additional method signature annotations, which we represent by adding a third component to $\mathcal{E}ff$. This is the same as the previous $\mathcal{E}ff$ but also gives an extra set of tuples (\mathbb{A}, \mathbb{B}) that we access with $\mathcal{E}ff(c, m)\downarrow_3$. This gives a selection of possible sets of locks required for a method to be atomic (\mathbb{A}) and the locks required for all the method's internal synchronisation to become redundant (\mathbb{B}) . $\mathcal{E}ff$ is supplied for each method by the programmer as before (we leave inference as further work). The full system is presented in 3.16.

If a new thread is spawned part-way through an atomic block e, it could take a lock not yet taken by e, and see state that should not have been visible until after e had completed. Therefore, the execution was not equivalent to a serial execution and thus not atomic. We prevent this by requiring the lock $\perp \in \mathbb{B}$ in the rule **spawn**. Such a lock is impossible to acquire, and thus **spawn** can occur only once in an atomic block, in the centre of the nesting of **sync** blocks.

Comparison with Flanagan et al.

The approach here is very similar to previous work [36, 35] but there are some differences in presentation which we will now discuss. The most obvious difference is our use of \mathbb{A} and \mathbb{B} . The previous work [36] had the notion of an 'atomicity', either left mover, right mover, both mover, atomic, or \top . These relate directly to reduction [59] and indicate how the action commutes with arbitrary actions in other threads. Atomicities were explicitly ordered and the ordering was used in the type system rules to require, e.g., that the expression be at least atomic. Operations were defined to join, compose, and take the iterative closure of atomicities, and these were used in the rules for branching, sequential composition, and loops respectively. Our approach was to use a similar approach to the race safety type system, i.e., to use sets. The type system gives the set of locks \mathbb{A} required for an expression to be atomic, and also the set of locks \mathbb{B} needed for it to be a both mover. This also meant we needed to define no extra operations, we just used basic set operations like union and membership in our rules.

We believe our type system benefits from this difference, especially when compared with more recent work [35]. The original work abstracted from synchronisation, requiring accesses to be marked in the input source code to indicate whether they were appropriately synchronised. This later work, like ours, used the type system to indicate what the atomicity of an expression would be, in the context of arbitrary synchronisation. In other words it was able to derive what kind of synchronisation would be needed for a particular expression to have a particular atomicity. The atomicities were extended with conditional operations, so could only have meaning when evaluated with a specific set of locks. This meant the presentation became quite a lot more complicated than our two sets A and \mathbb{B} .

Another side-effect of our two-set approach is that we cannot say a spawn statement is atomic, as in [35]. We can only determine the locks required for it to be atomic, and the locks required for it to be a both mover. To ensure it is atomic we put the impossible lock \perp in the set \mathbb{B} , leaving \mathbb{A} empty, which has the effect of defining spawn to be always atomic (never a both mover). If we were to distinguish between the locks required for an expression to be a left mover and right mover, we could let spawn be a left mover. This would allow the spawning of any number of threads (instead of just a single thread) after the acquisition of locks in an atomic section, but would complicate our presentation of the type system for what seems to be a minor benefit.

The presence of F and # in our system solves the same problem as the requirement in previous work [35] that synchronisation expressions be final. This is exactly the same distinction as we noted previously when discussing race safety (§3.3). We use the effect system instead of requiring our locking expressions to be final paths (paths starting from a final variable and using only final fields). As before, a hybrid approach could be used, where the programmer could use an empty set F if the path in the locking expression was final.

3.8 Chapter Summary

We wanted to use universe types for race safety because we believe the annotations are simpler and thus a more programmer-friendly experience than the related work, which uses more conventional ownership types. We have given a language, semantics, and race safety type system. We have proved that our system prevents races (§A). We took the approach of defining a minimal system and then giving a number of straightforward extensions that add features to the type system and allow more parallelism in programs. These extensions can distinguish between read/writes, prevent deadlocks, verify atomicity, and allow locks to be taken at the granularity of single objects.

Ownership types are important for good race safety type systems because they allow the type system to understand the extent of heap accesses in while loops and recursive functions. The ownership types in our system are different to those in the related work. Our system has the qualifier **any**, which can be used to implement open data structures without constraining the ownership hierarchy and thus the synchronisation of the rest of the program.

We found that our type system required fewer annotations than previous work [31, 11] and that the cases of accessing differently owned objects, it could understand more programs. We also used a system of effects, that we do not believe has been tried before, which lets the programmer use non-final paths. Another advantage of using universes is that they have already been implemented in JML[58]. In the future we would like to extend and refine our system to include generics using the techniques discussed in [25]. We have shown how the type system could be extended to also ensure not just race safety but also atomicity. However in the next chapter we propose a different approach to ensuring atomicity that requires much less input from the programmer.

$$\begin{array}{c} \underbrace{-, _, \Gamma \vdash e : _}_{\emptyset, \{\bot\}, \Gamma \vdash \text{spawn } e : \emptyset} (\text{SPAWN}) & \underbrace{e \in \{\text{null}, \textbf{x}, \text{this}, \text{new } t\}}_{\emptyset, \emptyset, \Gamma \vdash e : \emptyset} (\text{TRIV}) \\ \hline \\ \underbrace{\emptyset, \mathbb{B}, \Gamma \vdash e : F}_{\emptyset, \mathbb{B}, \Gamma \vdash a \text{tomic } e : F} (A \text{TOMIC}) & \underbrace{A, \mathbb{B}, \Gamma \vdash e : F \quad \Gamma \vdash_{gb} e : l}_{A', \mathbb{U}', \Gamma \vdash e' : F \quad l \in \mathbb{B}'} \\ A' \subseteq A, \mathbb{B}' \subseteq \mathbb{B}, & A \subseteq \mathbb{B}, F' \subseteq F, (\text{SUB}) \\ \hline A, \mathbb{B}, \Gamma \vdash e : F & A \in \mathbb{B}' \\ \hline A, \mathbb{B}, \Gamma \vdash e : F & A, \mathbb{B}, \Gamma \vdash e : F & A', \mathbb{B}'' = \mathbb{B} \cup \mathbb{B}' \quad A'' \in \{A \cup \mathbb{B}', A' \cup \mathbb{B}\} \\ \hline A, \mathbb{B}, \Gamma \vdash e : F & A', \mathbb{B}', \Gamma \vdash e' : F \quad l \in A'', \mathbb{B}'' \quad f \in F \\ \hline A, \mathbb{B}, \Gamma \vdash e : F & A', \mathbb{B}, \Gamma \vdash e : F & A', \mathbb{B}, \Gamma \vdash e : F \\ \hline A, \mathbb{B}, \Gamma \vdash e : F & A_{\mathbb{A}}, \mathbb{B}, \Gamma \vdash e : F & A_{\mathbb{A}}, \mathbb{B}, \Gamma \vdash e : F \\ \hline A, \mathbb{B}, \Gamma \vdash e : F & A_{\mathbb{A}}, \mathbb{B}, \Gamma \vdash e : F & A_{\mathbb{A}}, \mathbb{B}, \Gamma \vdash e : F \\ \hline A, \mathbb{B}, \Gamma \vdash e : F & A_{\mathbb{A}} = (u, e_{\mathbb{A}} e_{\mathbb{A}}) \otimes \mathbb{B}_{\mathbb{A}} \quad \mathcal{E}f(c, m) \downarrow_{\mathbb{A}} \subseteq F \\ \hline A, \mathbb{B}, \Gamma \vdash e : F & A_{\mathbb{A}} = (u, e_{\mathbb{A}} e_{\mathbb{A}}) \otimes \mathbb{B}_{\mathbb{A}} \quad \mathcal{E}f(c, m) \downarrow_{\mathbb{A}} \subseteq F \\ \hline A, \mathbb{B}, \Gamma \vdash e : F & A_{\mathbb{A}} = (u, e_{\mathbb{A}} e_{\mathbb{A}}) \otimes \mathbb{B}_{\mathbb{A}} \quad \mathbb{B}_{\mathbb{A}} \cup \mathbb{A}_{\mathbb{A}} \cup \mathbb{B}_{\mathbb{A}} \cup \mathbb{A}_{\mathbb{A}} \cup \mathbb{B}_{\mathbb{A}} \cup \mathbb{A}_{\mathbb{A}} \cup \mathbb{B}_{\mathbb{A}} \cup \mathbb{B}_{\mathbb{A}} \cup \mathbb{B}_{\mathbb{A}} \cup \mathbb{B}_{\mathbb{A}} \cup \mathbb{A}_{\mathbb{A}} \cup \mathbb{A}_{\mathbb{A}} \cup \mathbb{A}_{\mathbb{A}}$$

 ${\rm Eff}: (Id^c \times Id^m) \to (\ {\mathcal P}({\rm Lock})) \times {\mathcal P}(Id^f) \times {\mathcal P}({\rm Lock}) \times {\mathcal P}({\rm Lock})) \)$



Chapter 4

Lock Inference

4.1 Introduction

We have just seen how programmers can implement atomicity manually, using locks. We have given a type system that can be used to show that marked blocks of code are successfully made atomic by the programmer's locks. However, there is an alternative approach that has many advantages. The system can take the programmer's specification, in terms of the same **atomic** annotations used for lock checking, and implement the atomicity itself. In such a system, there are no programmer-supplied locks, and marking a block with **atomic** changes the behaviour of the program by eliminating interactions from other threads. We are thus changing the role of **atomic** to a language feature in its own right, and we call this feature the *atomic section*.

In the past, languages have evolved by offering higher level abstractions to programmers and taking care of the low-level details transparently in the compiler and run-time. Garbage collection and goto-less programming are examples of this philosophy. Both offer semantics that are quite distant to that provided by the hardware, and fill the gap using a combination of compile-time and run-time techniques. Atomic sections continue this philosophy in the context of concurrency.

The semantics of the atomic section can be understood by imagining that all the other

threads are suspended while the atomic section is executed. This clearly prevents all of the problems described earlier (§2.2), and keeps intermediate states well-encapsulated. Of course, implementations can use more efficient approaches that allow threads to proceed in the background to some extent, as long as their presence does not change the observable behaviour of the system as a whole.

While atomic sections are attractive for programmers, they put a great burden of correctness and performance upon the language implementation. In this chapter we will study the problem of extending current object-oriented programming languages to support atomic sections in an efficient manner. We hope to show that the language implementation is the correct place to handle the problem. We will also discuss some concessions in the high level semantics that are needed to build efficient implementations, and whether or not such concessions are acceptable in large software products.

4.2 Related Work

Naive atomic section implementations can just stop every other thread when an atomic section is entered. We call this a *stop the world* implementation. However, more efficient implementations need to allow threads to proceed when they are not *accessing the same memory* as the atomic section. On a shared memory architecture with n independent cores, the latter implementation could be as much as n times faster than the naive implementation. Thus understanding memory access is a key problem for all atomic section implementations. There are currently two main techniques for understanding memory accesses, and they divide all the existing work into two categories, which we will discuss separately, and then compare.

4.2.1 Transactions

The first implementations of atomic sections used *transactional memory* or more simply *transactions*. We also say such implementations are *transactional*. Transactions re-

CHAPTER 4. LOCK INFERENCE

execute code when interference is detected, to avoid its ill effects. This is a form of optimistic concurrency control that is inspired by database implementations [22].

There are two components that characterise all transactional implementations. Firstly, problems caused by the unpredictable interactions with other threads must be detected at run-time. What constitutes a problem depends on the transactional memory implementation in question. We discussed these kinds of problems earlier ($\S2.2$), but we do not avoid them as we would with locks ($\S2.3$). Secondly, if problems are detected, the transaction is *aborted* and the state is *rolled back* to the beginning of the atomic section for another attempt. If no problems occur by the time the atomic section has finished executing, then the transaction *commits* and the execution proceeds onwards.

The need to support rolling back prohibits the use of certain system calls, where the program interacts with other processes or systems that are outside the domain of the language implementation. We will henceforth call these calls I/O. Note that it is sometimes (but not always) possible to refactor I/O out of the atomic section, either manually or automatically. Sometimes I/O can itself be rolled back, using a cancelling call. However in general I/O must be restricted in atomic sections. This is one example of a restriction to the natural semantics of atomic sections that has been considered to facilitate a more efficient implementation. These semantic restrictions can be complemented with a type system that gives early warning when programs do not conform. In this case, the type system would prevent certain operations being performed in an atomic block or functions called from an atomic block. Alternatively, a stop-the-world implementation could be used for these atomic sections, however the performance would obviously be very poor.

Transactional implementations differ in what problems they can detect, how this is done, how they prevent problems that they cannot detect, and how they roll back. Detection and rollback are related since both require some kind of record of what accesses have occurred. The original proposal [47] used an extension of standard hardware cache coherency protocols [75] to keep a thread-local copy of the state of the atomic section, isolating its effects from the rest of the system. The cache coherency protocol would then detect *conflict* (another thread accessing the same memory) through the same mechanism it uses to coordinate cache lines between multiple cores. The local state copy is simply discarded to roll back the atomic section, otherwise it is written back to main memory to commit the atomic section. This implementation prevented nothing, detected everything, and rolled back using cache lines.

A later software-based proposal [77] was similar, storing the original value and reverting on abort. Accesses are tracked with in-memory data structures implemented with compile-time instrumentation. Other proposals followed [42, 43] that were similar in concept but differed in the implementation.

The aforementioned implementations have the property that accesses are understood whether they are inside an atomic section or not. This means that an atomic section will be rolled back if it conflicts with another thread's access irrespective of whether this other thread was also in an atomic section. Transactional semantics that support this are called *strong*. An access outside of an atomic section behaves like a normal primitive operation (read/write), i.e., like a tiny atomic section. This is desirable because it allows nontransactional code, i.e., code whose accesses are not tracked at run-time, to coexist with transactional code. Strong implementations usually detect everything, prevent nothing, and roll back by discarding a thread-local version of the state.

More recent implementations, initially proposed by Harris et al. [44], have not used a thread-local cache of the state. Instead they have updated the shared memory directly. This is faster in the common case where interference is rare. Such implementations use version numbers to detect interference, and restore the original value in order to roll back.

An interesting feature of such implementations is that they use "pessimistic writes"; they acquire a lock before each write access and release the locks when the transaction commits. This is needed to allow direct shared memory updates. One side-effect of this is that shared memory accesses must only occur in atomic sections, another restriction

CHAPTER 4. LOCK INFERENCE

on the semantics of the atomic section, and one that can also be enforced using a type system. Such a type system would distinguish between thread-local and shared memory and raise an error if shared memory accesses occur outside of an atomic section [1, 66]. In contrast to the strong semantics previously discussed, this is called *weak* semantics.

Deadlocks can occur if pessimistic writes are not globally ordered, but can be detected at run-time, by looking for cycles (§2.7) or by using timeouts, and the transaction rolled back. Thus deadlocks are not visible to the programmer. In this transactional implementation, concurrent writes are prevented, concurrent reads are allowed, interference due to a concurrent read and write access is detected, and so are deadlocks due to out-of-order writes.

It is also possible to protect read accesses by acquiring locks. This means the transactional code follows the two-phase discipline (§2.5). Interference never occurs, but rolling back is still needed if deadlock occurs. Thus interference is prevented, but deadlock must be detected and instrumentation of accesses is still needed to facilitate rolling back.

We have discussed hardware and software implementations of transactional memory. The overheads associated with the instrumentation only affect software implementations, but hardware implementations are limited by the size of the cache and thus cannot handle atomic sections with many accesses. They also have problems with long (as in time) atomic sections since page faults, interrupts, context switches, or anything else that causes the cache to be reset will prematurely abort the transaction. The latter problem takes away many of the encapsulation benefits provided by the atomic section, since the programmer has to know how many accesses are performed by all the functions he calls. Consequently, hybrid approaches have been proposed [20] that use hardware transactional memory for small atomic sections, otherwise drop down to software transactional memory. This also allows programs that use transactional memory to be portable i.e., executed on architectures with and without hardware support, although faster if support is available.

CHAPTER 4. LOCK INFERENCE

Finally, a manifestation of the locks concept of granularity (§2.6) is visible with transactional implementations, since accesses can either be measured at word level (fine granularity) or at the level of objects (coarse granularity). In the latter case, accesses of different fields in the same object are considered to conflict (even though they do not) whereas in the former case no conflict would be detected. When transactions use locks, the earlier notion of granularity can be directly applied, although most implementations use exactly one lock per object or field since it gives the most parallelism and there does not seem to be any advantage in using a coarser granularity. Nevertheless, this shows that granularity is a more general concept in concurrency control than we originally suggested.

4.2.2 Lock Inference

Another way of implementing atomic sections is to use locks according to the two-phase discipline. The problem with locks is that they are hard for programmers to use correctly. If the implementation takes on this responsibility, transparently to the programmer, then this problem evaporates. However, inserting correct locks is not easy, because it requires the compiler to predict what accesses code will perform, a classically difficult problem. All lock inference approaches restrict the semantics of atomic sections so that no shared accesses can be performed out of an atomic section. However, as mentioned previously (§4.2.1), type systems can be used to help the programmer comply. It is also possible to add tiny atomic sections around accesses that are outside of an atomic section, should a strong semantics be needed.

The first and least extensive attempt at inferring locks was Flanagan et al's extension [33] to previous lock checking work (§3.3). This was not an attempt to infer locks completely, but more to fill in a few programmer omissions. However, there have been many approaches that do attempt to infer locks completely, and they differ quite widely.

Points-to sets [3] are used for alias analysis in compilers but can also be used for lock inference [84, 49]. Objects are characterised by the place in the code where they are created. This is a syntactic notion that is very amenable to program analysis as it is bounded in the cases of loops and recursion. We can use this object characterisation for accesses as well, by associating a lock with each creation site. The lock in question then protects all the objects created there. Objects are typically assigned from one variable to the next in the code; this "data flow" [71] is approximately understood by the program analysis. Ultimately each variable ends up associated with the set of object creation sites that (conservatively) represents the objects that it may reference. Then inferring what locks must be acquired to protect an atomic section is as simple as inspecting the pointsto information associated with the variables being accessed in the atomic section. This approach is attractive because it is only a small extension to an extensive body of pointer analysis research and implementation. Unfortunately it cannot make use of instance locks so granularity tends to be poor. The number of locks is bounded by the number of object creation sites and thus the size of the program.

Meanwhile, a different technique was used in Autolocker [63]. Both instance and static locks were allowed, specified by the programmer with guard annotations. These were similar to those used in the lock checking work (§3.3), although less powerful since iterations over objects at instance-level granularity cannot be expressed due to the lack of any ownership-like features. Assignment was restricted to avoid non-termination in the static lock inference phase. More recently, custom alias analyses have been used to allow instance locks without annotations, falling back to static locks if aliasing is uncertain [28, 41], but the choice of instance/static locks was still on a per-object basis.

Only recently, in our published work [19, 18] and independently by Cherem et al. [13], have multi-granularity locks been used to allow the instance/static distinction on a peratomic-section basis, without any annotations. These recent works are also the first to use translation techniques to handle assignment rather than restricting it or falling back to static locks. By considering assignment more accurately we only need to use static locks when programs iterate over objects, and with ownership types we could avoid them altogether (future work).

One key difference between our approach and Cherem et al. is that they force the termination of their algorithm in such iterations with a simple static bound, whereas we represent cycles accurately with a non-deterministic finite automaton (NFA). This technical difference should result in a better approximation of accesses and thus a more precise application of locks. It also creates future opportunities for removing static locking entirely. The program analysis used by Khedker et al. [55] was very similar to ours in its use of NFA-like representations, but there they are used to statically accelerate garbage collection.

We believe our approach is the only one that prevents deadlock with a dynamic mechanism (§2.7) as transactions do. When deadlock is detected the locks are released and reacquired. We avoid the full roll back of transactions by acquiring all of the locks together at the entry to the atomic section. Thus there is nothing other than lock acquisitions to undo. The other approaches attempt to statically order lock acquisitions, falling back to static locks if this is not possible. Our work has been a constant struggle to preserve instance locks wherever possible.

One contribution of Cherem et al. [13] is a framework for specifying and combining lock inference approaches. Although they distinguished between dereferencing and object offset (i.e., in the style of the C language), whereas we just use fields (in the style of Java), 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.

Finally, a very similar technique [69] to lock inference has been used to implement futures in a Java-like language. Futures are a much simpler concurrency primitive than atomic sections, but their efficient implementation presents many of the same problems. The futures mechanism spawns a thread to compute a short background task and then joins it to receive the result of the class slightly later in the program. Basic synchronisation is needed to stop the two threads interfering with each other. The cited paper used pointsto information to infer what accesses may be performed by both threads in order to insert appropriate synchronisation.

4.2.3 Comparison

On the surface, lock inference and transactions are very different methods of implementing the semantics of atomic sections. However they share some similarities, and in other ways they complement each other.

Both approaches have proposed restricting shared memory accesses outside atomic sections. Both approaches have made use of run-time deadlock detection (§2.7). Both approaches synchronise by holding back a problematic thread: Transactional implementations do this by forcing the thread to return to an earlier state, whereas inferred locks do not allow the thread to proceed from this earlier state in the first place. Both approaches have seen proposals for field-level and object-level granularity of accesses. However there is still a fundamental difference between them:

Lock inference tries to predict what accesses may occur. Because this is an undecidable problem, it must be approximated. This approximation will introduce inaccuracies and this will manifest as over-conservative locking and a loss of parallelism. Transactions avoid the problems of static inference by measuring accesses at run-time. They therefore have perfect accuracy when compared to lock inference. However, the price they pay is that they miss the opportunity for preventing errors and thus need to occasionally roll back the state to hide these errors. This means accesses must be instrumented and there is a major performance penalty. Hardware transactional memory can avoid the performance penalty but is only practical if transactions can be unbounded, which means falling back to software instrumentation if the platform-specific limits are exceeded, and thus falling back to software overheads.

CHAPTER 4. LOCK INFERENCE

This leaves us with an interesting trade-off. With transactions, one suffers a significant *linear* performance cost i.e., the performance of a single thread is decreased. With lock inference, this is not the case (aside from the small impact of the locks themselves). However, it is possible to be over-conservative and acquire unnecessary locks, blocking non-interfering threads and hurting *parallel* performance.

While this is the clearest performance difference, there are other complications that do not form such a neat picture. When a pair of transactions collide, one of them rolls back. Not only is the rolling back a waste of cycles, but so was the computation of the now-discarded state. On a system with more threads than cores (a likely situation when software has been designed for a range of architectures) these cycles could have been put to better use executing another thread. Even if the core would otherwise be idle, unnecessary cycles will waste power¹ and also can crowd the memory bus. To avoid these wasted cycles, some transactional implementations use *back off* techniques where threads sleep for a short while after a transaction aborts before retrying. However, sleeping for too long hurts performance as well, and it's often not clear for how long a thread should sleep. Different algorithms will perform differently with different back off strategies, and some implementations [46] allow the user to select a particular one. This adds complexity and exposes the fact that an atomic section is implemented with transactions.

There are also differences in the amount of compiler technology needed. The instrumentation required for software transactional memory is relatively easy to insert. Lock inference needs to know what objects will be accessed, *without* executing the code in question, so a full static inference is always required. If implemented in compilers, both lock inference and transactions need access to the whole program in order to handle functions that may be invoked from atomic sections. Consequently, native calls are a problem for both approaches. Note that there has been some work to extend transactions into native code by analysing and translating this code [85] and similar techniques could be used for

¹Power is an important consideration not just in mobile devices but in servers and supercomputers, where performance is often measured per watt. Domestic consumers are also becoming increasingly aware of their power consumption.

CHAPTER 4. LOCK INFERENCE

lock inference. Plugins can be handled using JIT technology in both cases i.e., analysing / instrumenting the code at run-time as it is linked.

Both approaches require concessions from the simple semantics of atomic sections. Transactions cannot roll back I/O whereas lock inference cannot easily understand reflection. In both cases this can be solved by not allowing the problematic feature to be used in atomic sections, perhaps enforced using a type system. It is not clear how debilitating is the restriction on reflection, but we expect it to be more reasonable that restricting I/O, which is used much more heavily in software.

It's also worth comparing lock inference to programmer-supplied locks. Ideally we would like to infer locks that are at least as good as well-written programmer-supplied locks. However the software engineering benefits of using atomic sections may be enough to persuade people to accept slightly inferior locks. The problem of granularity when using locks is already faced by programmers. Depending on the contention in the application, it may not be necessary to use the finest granularity imaginable (as provided by transactions). For this reason it may be that some future lock inference will allow enough granularity for typical applications while still not approaching the granularity of transactions.

However, the state of the art of lock inference leaves much to be desired in terms of granularity. The system we present in this thesis is an attempt to shrink the gap between lock inference and its two main competitors – transactions and manually inserted locks. We do this by shrinking the error in the static analysis, thus allow finer-grained locking to be used. We therefore shrink the expected parallel performance cost of implementing atomic sections with lock inference while still avoiding many of the pitfalls of transactions. In the rest of this chapter we will describe our approach.

4.3 Pathgraph Analysis

As before, when we were checking if locks were sufficient for atomicity (§3.7.3), we use a two phase locking protocol (§2.5). We infer the accesses of an arbitrary block of code, and derive a set of locks that we acquire before the atomic section and release when they are no longer needed, often at the end of the atomic section. For simplicity, we assume that atomic sections are never nested². We also assume everything accessed from an atomic section is shared between threads, and everything shared between threads is only accessed from atomic sections. The first restriction could be relaxed in a real implementation, assuming the thread locality of objects is known. This would leave us with a weak semantics (§4.2.1) as implemented by most software transactional memories.

Our lock inference analysis therefore takes an atomic section as input, which we call a *program*. We henceforth assume that programs have already been converted into control flow graphs (CFGs) and are therefore ready for program analysis. We use a run-time mechanism (§2.7) that detects when a thread's lock acquisition would cause a deadlock and rolls back all the lock acquisitions of that thread. Since lock acquisitions always appear together at the beginning of the program, only lock acquisitions need to be rolled back and thus no transaction log is required.

In this section, we present an analysis that infers what accesses occur in an atomic section. We will prove the analysis correct. However this is only the first part of the story. Once we have the accesses, we will discuss in a later section (§4.5) how to infer locks from them.

Consider program in fig. 4.1. We use a backwards 'may' analysis [71] to infer a set of accesses at each edge. These accesses are the accesses that may occur during the execution from this edge onwards, but their meaning is defined in terms of the heap at the edge where they occur. Starting at node 6, we first infer the access of the object tmp_tyre. This propagates towards the entry point of the program. Ultimately, the set

 $^{^{2}}$ At run-time, one can set a flag that disables the inferred locks of inner atomic sections, or alternatively compile two versions of the code, i.e., with and without locking instrumentation.



Figure 4.1: Example of an atomic section where aliasing is a concern.



Figure 4.2: Example of an atomic section with an iteration over objects.

of accesses that accumulates at the entry point will be used to derive the inserted locking code. For now we assume that there exists a mapping such that every object has a lock that protects it and this lock can be discovered, either at compile time, or at run-time via the object's address (e.g., as in Java). Thus we can reduce the problem to determining the objects accessed by a program.

The analysis has to translate accesses as they are propagated to account for the statements they pass through. For example, 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, we add the access of bus.

At node 4, we add the veh access, but we also need to include spare_tyre. This is because veh and bus may be aliases, and thus it may have been spare_tyre that was pressurised. We could use an alias analysis to help here. However, if the aliasing is still uncertain, we must conservatively approximate. In this case we include both the bus.tyre access (not aliased case) and the spare_tyre access (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 what branch will be chosen at node 1, we take the union of the two branches to form the final set of accesses. We then map these accesses to locks that we acquire for the duration of the atomic section.

The program in Figure 4.2 is an atomic section that iterates over objects, e.g., the nodes of a linked list. Here, the algorithm as described above would not terminate as the sets of accesses would grow infinitely. To force the analysis to terminate, we redesign our algorithm to use a non-deterministic finite automaton (NFA) [50] at each edge, to represent the set of accesses. We call these NFAs *path graphs*. They can represent a possibly-infinite set of *paths*. A path is a sequence of field accesses starting from a stack variable, and can be used to characterise statically an object access in terms of the



Figure 4.3: Path graphs representing the accesses of the previous examples

syntax of the program being analysed. We used paths to describe the accesses in these two examples and also in our lock checking work (§3.6).

A finite path graph can still represent an infinite number of accesses, if it contains a cycle. The path graphs that represent the analysis state at the entry of the fig. 4.1 and fig. 4.2 are given on the left and right of Figure 4.3 respectively.

The path graph, like all NFAs, has a start node and can be interpreted as a set of locks by reading along the arrows. Unlike more general NFAs, every node of a path graph is an 'exit node' so the set of represented paths is prefix-complete. This makes sense since in the code we cannot access an object unless it is either bound to a variable before the atomic section began, or can be retrieved through another object, in which case this preceding access would also be present in the path graph.

Path graphs are a very rich source of information about future object accesses. They encode not only what is accessed but also at which node it was accessed. We can also apply classical automata theory to them to refine this information. The location where an access occurs can be used to determine if the access was a read or a write, and to look up type/points-to information, both useful when inserting locking code. We will later describe how to interpret path graphs to derive the locking code.

Knowing the location of the accesses in the program is the secret to making path graphs understand iteration. If a new access is found that occurs at the same place in the code as an existing access (this only happens when we have iteration), these two accesses are joined together in the representation, which forms the cycle in the graph. We do this by using the CFG nodes to form the set of nodes in the path graph e.g., all the **x.n.n.n..** accesses occur at CFG node 2. In general, the node reached by following an arrow in a the path graph is the node of the CFG where the access occured.

In the next section we formalise the concept of path graphs and give our analysis transition functions. We can then prove the result of the analysis is sound.

4.4 Formalism

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:** $A \rightarrow B$ 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.

4.4.1 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 [71].

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. 4.4. In order, the statements are *copy assignment*, *object construction*, *heap load*, *heap store*, and *condition*. Every statement has a given successor n where execution proceeds after that statement, except the condition $\langle n; n' \rangle$ that 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 program in fig. 4.2 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

$P \in Program = Node \rightarrow Statement$	
$st \in Statement ::= [x=y;n] \mid [x=\texttt{new};n] \mid$	$[x = y.f; n] \mid [x.f = y; n] \mid \langle n; n' \rangle$
$\begin{array}{llllllllllllllllllllllllllllllllllll$	$\begin{array}{llllllllllllllllllllllllllllllllllll$
$\frac{\text{Stop}}{P \vdash h, \sigma, n \rightsquigarrow \varepsilon}$	$\begin{array}{c} P(n) = [x = y; n'] \text{Copy} \\ \hline P \vdash h, \sigma[x \mapsto \sigma(y)], n' \rightsquigarrow A \\ \hline P \vdash h, \sigma, n \rightsquigarrow \tau.A \end{array}$
$ \begin{array}{ccc} P(n) = \langle n'; n'' \rangle & \text{COND} \\ P \vdash h, \sigma, n' \rightsquigarrow A & \lor & P \vdash h, \sigma, n'' \rightsquigarrow A \\ \hline P \vdash h, \sigma, n \rightsquigarrow \tau.A \end{array} $	$P(n) = [x.f = y; n'] \qquad \text{STORE}$ $a = \sigma(x)$ $P \vdash h[(a, f) \mapsto \sigma(y)], \sigma, n' \rightsquigarrow A$
$P(n) = [x = \texttt{new}; n'] \qquad \text{NEW}$ $a \notin dom(h)$ $P \vdash h[a \mapsto \lambda f.\texttt{null}], \sigma[x \mapsto a], n' \rightsquigarrow A$	$P \vdash h, \sigma, n \rightsquigarrow a.A$ $P(n) = [x = y.f; n'] \text{LOAD}$ $a = \sigma(y)$ $P \vdash h, \sigma[x \mapsto h(a, f)], n' \rightsquigarrow A$
$P \vdash h, \sigma, n \rightsquigarrow \tau.(A[a \mapsto \tau])$	$P \vdash h, \sigma, n \rightsquigarrow a.A$

Figure 4.4: Syntax and Semantics of Execution Model

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. Thus, we reason about partially complete executions, which are truncated after an arbitrary and unbounded amount of execution. 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 that while the language does not allow assignment of null, the run-time 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.

4.4.2 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. 4.2, we do not infer the infinite set of locks $\{x, x.n, x.n.n, \ldots\}$, rather the finite path graph $\{x \rightarrow 2, 2 \xrightarrow{n} 2\}$. 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. 4.5.

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: $X(n) = acc(n)(st) \cup tr(n)(st)(X(n'))$. 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 $\begin{array}{rcl} Edge & ::= & x \to n \mid n \stackrel{f}{\to} n' \\ G \in PathGraph & = & \mathcal{P}(Edge) \\ X \in AnalysisState & = & Node \to PathGraph \\ acc & : & Node \to Statement \to PathGraph \\ tr & : & Node \to Statement \to PathGraph \to PathGraph \\ acc(n)[x = y; _] & = & \emptyset & acc(n)[x = y.f; _] & = & \{y \to n\} \\ acc(n)[x = \mathsf{new}; _] & = & \emptyset & acc(n)[x.f = y; _] & = & \{x \to n\} \\ tr(n)[x = \mathsf{new}; _](G) & = & G \setminus \{x \to n' \mid x \to n' \in G\} \cup \{y \to n' \mid x \to n' \in G\} \\ tr(n)[x = \mathsf{new}; _](G) & = & G \setminus \{x \to n' \mid x \to n' \in G\} \\ tr(n)[x = y.f; _](G) & = & G \setminus \{x \to n' \mid x \to n' \in G\} \\ tr(n)[x = y.f; _](G) & = & G \setminus \{x \to n' \mid x \to n' \in G\} \cup \{n \stackrel{f}{\to} n' \mid x \to n' \in G\} \\ tr(n)[x.f = y; _](G) & = & G \setminus \{n' \stackrel{f}{\to} _ \mid x \to n' \in G\} \\ (\nexists n''' : n''' \Rightarrow n' \in G), \\ (\nexists n''' : n''' \Rightarrow n' \in G)\} \\ \cup \{y \to n' \mid _ \stackrel{f}{\to} n' \in G\} \end{array}$

Figure 4.5: The analysis

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 $n \stackrel{f}{\to} n'$ edge. This only makes sense if we can guarantee that an edge $y \to n$ 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. 4.1, we have veh.tyre = spare_tyre, and below we have
$X(5) = \{bus \to 5, 5 \xrightarrow{tyre} 6\}$. We therefore add the edge $\{veh \to 4\}$ due to the access function *acc*. We also add $\{spare_tyre \to 6\}$ due to the last part of the translation function *tr*. There is no $veh \to 5$ in X(5), but even if there was, we would still not subtract $5 \xrightarrow{tyre} 6$ from X(5) because $bus \to 5$ 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. 4.5. We denote this with $P \vdash X$.

4.4.3 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 what 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 4.4.1 Soundness:

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

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.

4.4.4 Assigning meaning to path graphs

We now consider an arbitrary path graph G and an assignment φ that 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 what node they occur. **Definition 4.4.2** Flattening of assignments: $flatten(\varphi) = \{a | \exists n : a \in \varphi(n)\}$

Definition 4.4.3 Valid assignments:

$$\begin{split} h, \sigma \vdash G : \varphi &\iff (\forall x \mapsto n \in G : \sigma(x) \in \varphi(n)) \land \\ (\forall n \xrightarrow{f} n' \in G : \{h(a, f) | a \in \varphi(n)\} \subseteq \varphi(n')) \end{split}$$

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 $n \xrightarrow{f} n'$ then we want $h(a_x, f)$ to be present at n', unless that address is not defined on the heap³ 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 flow 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 pointwise: $\varphi_1 \sqsubseteq \varphi_2 \iff \forall n. \varphi_1(n) \subseteq \varphi_2(n)$. 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 4.4.4 Valid assignments join to make valid assignments:

³Although this cannot happen in a language like Java, for simplicity our formalism permits initial stacks/heaps to contain undefined addresses.

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

Proof: Follows from the definitions.

If we define the minimal assignment:

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

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 $\gamma(h, \sigma, G) = flatten(\Phi(h, \sigma, G)).$

4.4.5 Proof

We have proved correctness in Isabelle/HOL using Proofgeneral [5]. The file is 800 lines long, takes 30 seconds to process on a 3GHz P4. It is listed in an appendix (§B) and can also be downloaded [15]. 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 4.4.1 and all auxiliary lemmas were proved automatically. Theorem 1 was a long proof but often the final stages of each case were automatic. In 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 fig. 4.4. 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.

4.4.6 Multiple Threads

Our formalism is not threaded. Theorem 4.4.1 only states that the execution of the atomic section, as a single thread, is conservatively approximated by the result of the analysis. However, in the context of additional threads, one might be worried that thread interactions could cause a different set of addresses to be accessed from those that were protected by the inferred locking code. In other words, although the meaning of the path graph was translated during the analysis to account for changes in the state caused by the local thread, what about changes to the state that were caused by other threads?

We have not formalised multiple threads because it would have substantially increased the size of the formalism and proof. However, we can argue in the form of a proof sketch that this kind of situation cannot occur. We want to show that threads only access objects that they have locked.

First we make a number of simplifications. We consider only two threads that are both executing atomic sections (it does not matter if the two atomic sections are in fact the same atomic section). Recall that one of our initial assumptions was that objects shared between threads are only accessed from within atomic sections, so any heap mutation caused by a thread outside of an atomic section must be at a thread-local object that is not touched by the other thread. This argument could also be generalised to more than two threads.

Let us consider the state where both threads have completed the lock acquisition phase and are about to start executing the body of the atomic section. We will be assuming the existence of some small step semantics, where multi-threaded execution is modelled by non-deterministic interleaving, and an arbitrary thread completes one step in each step of the system. We can now proceed via induction on the number of steps that the system has taken. We are trying to show that all accesses that have occurred up to and including the current step were protected by an appropriate lock.

The first access of an atomic section must have been through a stack variable, which

is thread local and thus its meaning cannot have changed since it was used in the lock acquisition code. Thus the first access was correctly protected. Subsequent accesses can also be via stack variables and the same argument can be used for their correctness, except when the variable in question was assigned to since the locks were acquired (e.g., tmp = x.f; tmp.g = 42). In this case, the lock that protects the tmp access was not acquired by evaluating tmp but through x.f. How do we know that the value stored in x.f was not changed between the lock acquisition phase and the tmp = x.f assignment in the atomic section body? Recall that path graphs are prefix complete since every node is an exit node. This means in this case (and all other cases like it) the lock x has been held consistently during the lock of x.f and the tmp = x.f access later on. Since the other thread has only been accessing objects for which it has acquired appropriate locks (induction hypothesis) we can thus be sure that it has not affected the field x.f and thus the correct lock is held at the tmp.g = 42 access. This completes the proof sketch.

To prove this formally would require not just a small step semantics but also a notion of interleaving and execution of the system as a whole. The semantics would begin to look more like that of the lock checking chapter (§3.4). The reason we did not pursue this more formally is that the effort would have outweighed the result. Essentially what we have verified is the two phase locking discipline itself, not some detail of our lock inference. Since two phase locking has been used successfully for many years both in theory and practice, it would be extremely surprising if it were not immune to the original problem. Our semantics and proof are considerably more concise because we stuck to a big step semantics and considered only a single thread, which we consider to be a bigger advantage than the completeness of lifting the result to multiple threads.

4.4.7 Algorithm

The functions in fig. 4.5 give the core of the program analysis, but we have not yet given a complete algorithm. In fig. 4.6 is an algorithm which uses tr and acc from fig. 4.5

```
function analyse (P)
    let wl = dom(P)
    let X = [ ] // empty associative array
    foreach n in dom(P) do
         X[n] = acc(n)
    end
    while wl \neq \emptyset do
         let wl' = wl
         let wl = \emptyset
         foreach n in wl' do
              let old = X[n]
               case P(n) in
                   \langle n'; n'' \rangle: X[n].include(X[n'] \cup X[n''])
                   [\underline{\phantom{x}};n']\colon \mathbf{X}[n] \texttt{ include}(tr(n)(P(n))(\mathbf{X}[n']))
               end
              if X[n] \neq \text{old wl.include}(prev(P, n))
         end
    {\tt end}
    return X
end
```

Figure 4.6: Pseudocode Analysis Algorithm

to return an X from a given P such that $P \vdash X$. The syntax is the same as used in the typing algorithm from the previous chapter fig. 3.14. The state of the analysis X is implemented as an associative array that is initialised to the path graphs given by the *acc* function, i.e., the local accesses performed by each node. It is populated, as the iteration proceeds, using a modified Worklist algorithm [71], and the *tr* function, which defines how the path graphs are translated by each node. Path graphs are sets, so we use standard set operations on them. X contains a path graph for each node n, and we accumulate into this path graph using the **include** method. The function prev(P, n) returns the set of predecessor nodes for a given node n in P. We assume this has been precalculated or is otherwise already known. We use dom(P) to mean the domain of P.

The *prev* function is used to update the worklist. For every node n that is processed, if the iteration adds more information to X, then the predecessors of n are considered in the next iteration, in order to flow this information further. When X becomes stable, the algorithm terminates.

When we implemented this algorithm we discovered a lot of time was spent pushing path graphs through tr. However for a given tr call there were only a few additional edges in the input path graph since the last iteration, so only these needed to be processed. By pushing only these deltas through tr there was a speedup of several orders of magnitude. We leave a more refined implementation as further work.

4.5 Inserting Locks

In this section we describe in detail our approach for inserting locking code into an atomic section, based on the result of the analysis previously described. We also show how the analysis supports readers/writers and early unlocking.

4.5.1 Inferring Locks from a Path Graph

The analysis outputs a path graph, which is essentially an NFA where every node is an exit node.

To convert a path graph to lock acquisition code, we must identify and break the cycles. A cycle implies that an unbounded number of accesses occured and thus they must be treated specially if we intend to insert only a finite amout of locking code. In fig. 4.3 the left hand side has no cycles, and the right hand would be treated by removing the .n edge. A node is *involved with* a cycle if it is part of the cycle or if it is reachable from a node that was involved with a cycle. If we iterate through a linked list then the accesses of the nodes of the list will be involved with a cycle in the path graph, but also any objects stored in the list (its cargo) will be involved with the cycle too and need special treatment. Since we do not statically know which of the cargo objects was accessed, we can only assume they all were, so the special treatment extends beyond the cycle itself.

We treat all nodes that are involved in a cycle as follows: We attain the type of the object accessed at that node (from the program CFG), which defines a multiple granularity lock (§2.6) that protects all accesses of that type. This allows us to lock the infinite number of accesses represented by the path graph with a finite quantity of locking code. The choice of type system makes a big difference to the granularity of the locks,



Figure 4.7: The same path graph after DFA transformation

as we will discuss later (§4.5.2). A path graph with no cycles can be depth-first-searched to extract a finite set of paths and thus a finite amount of lock acquisition code can be inserted. Paths whose locks are subsumed by the multiple granularity locks derived from the cycles can be omitted, assuming the multiple granularity locks were. The locks must be acquired in prefix order, and are remembered in variables, to be released at the end of the atomic section.

We wanted to use read/write locks, i.e. locks that allow multiple threads to have the read lock so long as no thread has the write lock. The path graphs encode where in the CFG the access will occur, and we can use this to determine the kind of access (load/store = read/write respectively).

If we followed this approach with the path graph on the left of fig. 4.3, we would lock bus twice, a waste of cycles. This is because the path graph is an NFA rather than a deterministic finite automaton (DFA). There is a standard algorithm that converts NFAs to DFAs which we can use to eliminate multiple locks of the same thing. Figure 4.7 shows the left-hand path graph from fig. 4.3 would look after conversion to DFA.

In the process of conversion to a DFA, many path graph nodes can be squashed into a single node, e.g., nodes 4 and 5 in fig. 4.1. This means if we want to get information from the node, e.g., the type of the object accessed or its read/write status, we must approximate the information from the set of nodes that were compressed. Since the access in 4 was a write, and in 5, a read, we would take the write lock on **bus** in order to protect both. Similarly one can join types by taking the most specific super-type.

CHAPTER 4. LOCK INFERENCE

Conversion from NFA to DFA eliminates locks that are syntactically equal, but the same lock can still be locked twice due to aliasing. Thus we still need to use reentrant locks to handle the cases where e.g., **car** and **bus** are in fact the same instance.

Consider $atomic \{ y = x.f ; y.g = 10 \}$, for which we would lock x for reading and x.f for writing. If x was null, then the atomic section would, by the semantics of the language, throw a NullPointerException (NPE). More importantly, so would our locking code. In general, throwing an exception from the locking code instead of in the atomic section would not preserve the semantics of the atomic section, as there may be side-effects from before the NPE that we must allow to happen. We need to either check for null before locking, or catch the NPE. If x is null we do not lock it, or x.f.

4.5.2 Iteration and Granularity

Lock inference is a conservative static analysis and thus it is unavoidable that we infer that an unbounded number of objects may be accessed in an iteration through the heap. Our approach so far has been to represent these iterations as accurately as we can in the analysis, using path graphs that introduce cycles only where necessary and otherwise keep the set of accesses finite. However, when we insert locking code we need to decide how we should lock these infinite sets of accesses.

As hinted earlier, we can attain the type of the infinitely many objects from the path graph, and use the type to lock all instances of these objects. There is a finite number of types, since this is bounded by the size of the program. If the type system has subtyping then we have to lock instances of any sub-types too.

One way to do this in a Java-like language is to use the class types to define the locking. If one iterates over a list of Node objects then we lock the Node class and any subclasses. We can use multiple granularity locks (§2.6) to minimise the impact of this strategy on the overall granularity of the system. The benefit of using the standard class types are that standard programs (with atomic sections) can be accepted with no additional type annotations. However the granularity of the class locks is very poor, e.g., it is not possible to iterate over two independent lists in parallel because the Node lock would be taken in both cases.

Our earlier lock checking type system did not have this problem as it used a notion of ownership to define the locking for such iterations. If we were to use an ownership type system here, we could lock the owner of the objects involve in the iteration. This could in principle allow such cases as parallel iterations over independent lists. We will discuss this more with other future directions in the final chapter.

4.5.3 Deadlock

Existing approaches guarantee the absence of deadlock at compile-time by always acquiring locks in the same order, and if such an ordering cannot be found they typically coarsen the granularity. If we use object types to define our multiple granularity locks then these are static, thus can be statically ordered (e.g., alphabetically). On the other hand, our instance locks cannot be statically ordered. We want to avoid coarsening the granularity so we detect deadlocks at run-time (§2.7) and roll back the lock acquisition phase. Although this sounds like a transaction, there are no side effects to roll back, so no transaction log is required. This is because we acquire all locks at the beginning of an atomic section. We therefore can roll back these acquisitions without suffering the overhead of maintaining a transaction log.

Although some approaches acquire locks as they're needed, our deadlock rollback requires us to take all locks at the beginning of the atomic section. We expect deadlock to be rarer than transaction collision, because the lock acquisition phase is much shorter (in terms of execution time) than a whole atomic section. Since two threads can only deadlock when they are both acquiring locks, this should mean that the chance of deadlock is much less.

Deadlock rollback also offers the possibility of high-priority threads forcing low-priority

threads to give up their locks if they are stalled in a lock-acquisition phase, a solution to what is generally called *priority inversion*. If implemented, this would be similar to the contention management sometimes used with transactions. However it does not prevent priority inversion caused by low priority threads that spend a long time executing in atomic section bodies. Interrupting such threads would require arbitrary code to be rolled back and we would thus need to use transactional memory.

4.5.4 Parole

If an atomic section has finished accessing an object, but still has a lot of computation left to complete, it can release the lock early to allow other threads to proceed in parallel with this latter part of the atomic section. Another way of saying this, is that the latter part of the atomic section is locked with a finer granularity. We can also demote write locks to read locks, which allows other threads to proceed if they only need to read the data, and multilocks to instance locks, which liberates threads that also only need to lock individual objects. This is particularly important in the context of conservative analysis because it means we can get rid of our over-conservative lock acquisitions as soon as we have executed enough of an atomic section to know for sure what objects will be accessed during the rest of the atomic section.

Recall that our program analysis infers accesses at each edge of the CFG, and that these are the accesses that may occur during the execution from that edge onwards. Thus if we take a node in the CFG, and subtract the accesses from before and after that node, we can determine if there are any locks that are only needed for the node in question and can thus be released after the node has executed. For example, a list node may be discovered through iteration, and then written to. Before the iteration, the analysis has no idea which node is written so approximates the access with the (sometimes very large) set of same-typed accesses, which is locked with a multilock. After the iteration, this widening from single lock to multilock has not occurred so the edge contains a single



Figure 4.8: Result of path graph analysis applied to early release

write lock to represent the access. Between the iteration and the access, we can thus demote the multilock to an instance lock. This would liberate other threads that are stalled at the acquisition of some other instance lock of the same type.

Previously (§4.5.1), we described how to infer locks from a path graph. We converted the NFA path graph at the entry of the atomic section into DFA form to avoid unnecessarily acquiring the same path twice. We then used multilocks to represent accesses affected by cycles and turned the rest of the graph into paths to form the locking code. We will now perform this process at every CFG edge instead of just the entry edge. The example in fig. 4.8 is given in source code and CFG form with edges labelled. We give the locks calculated by this process, at each edge of the CFG, in the first column of the table. An exclamation mark represents a write lock. All of the locks in this example are paths because the atomic section has no cycles, but the subtraction method is independent of the representation, e.g., it also works for class types. We will now discuss the example and what code should be inserted to facilitate early release.

At the entry of the atomic section (edge 1), we take the locks $\{!x, !y\}$ using the

deadlock detection/rollback strategy described previously (§4.5.3). Once this is successful, execution will follow one of the branches. One branch (4, 5) will cause us to eventually access both !x and !y, and the other (2, 3), only !x, as given in the rows for edges edges 2 and 4 in the table. The analysis records that at edge 2, only the lock !x is required, so by subtracting the set 2 from set 1 we can calculate locks to release, as listed in the *Early releases* column. If the set of locks increases, e.g., from edge 2 to edge 3, we have to acquire the extra lock in order for a later release to be well-balanced, we list this in the *Late reacquires* column. Such locks have already been acquired by the thread, reacquiring them just increases the re-entrant counter. There is no risk of deadlock. We always acquire before release, to make sure we do not let the re-entrant counter reach zero, so the locking remains two-phase. To summarise, the subtraction technique will sometimes yield lock acquisitions as well as releases, when the set of accesses increases from one edge to another.

At an assignment, where locks are translated from one variable to another, there will always be a release of one lock and an acquire of another lock. However, these are just due to the renaming of an edge in the path graph, so are redundant. We can remove these redundant actions by observing that after an assignment, the left and right hand side will alias each other. Thus a lock of !x and an unlock of !y after the assignment $\mathbf{x} =$ \mathbf{y} are redundant and can be removed. In fig. 4.8 this is denoted with strike-through text. Since redundant locking code occurs at every assignment, this technique can significantly reduce the amount of inserted locking code, and thus avoid wasting cycles. Note that if x and y are known to be aliased, then so are x.f and y.f so the elimination of redundant locking code can extend to paths of arbitrary length.

The remaining acquires and releases are not redundant: Unlocking !y at edge 2 allows other threads to proceed in parallel, as it is now certain that the thread will not write to (or read) y. Releasing x and !t at edges 6 and 7 respectively allows the atomic section to terminate with all locks released. This means we do not need to insert code that records the acquired locks so that they can be released explicitly at the exit of the atomic section. Acquiring x and releasing !x at edge 5 is a demotion. It allows other threads to read in parallel, as this thread no longer needs to hold the write lock on x.

The spurious lock of x at edge 3 is necessary because at this time x and x.f are aliases, and the releases at edges 6 and 7 will serve to release the same lock twice. Therefore, in the branch we must acquire x to balance the forthcoming releases.

There is a complication concerning nodes such as x = x.f. Consider the very simple case of an atomic section that contains only this node. We would acquire x before the atomic section and release x after the node. However, these locks do not balance because x resolves to different objects in each case. To solve this problem, we compile such statements to tmp = x.f; x = tmp which allows a release of x before the meaning of x is changed.

A similar problem occurs around field updates e.g., $x \cdot f = y$. While the stack does not change, the heap does, and this can change the objects to which paths resolve. If locking code is inserted after such a statement, and the paths involved in such locking code contain the field being assigned, then the resolution can be performed before the assignment. The resulting object can be cached in a local variable, which is simply read after the assignment, ensuring the correct lock is acquired/released.

Objects that are created in an atomic section should begin life with their lock acquired. This is important if we are releasing locks early. Consider the following atomic section: $x = \mathbf{new}; y.f = x; x.g = 42$. If we are releasing locks early, we would release the lock on y after the middle statement. However this would allow us to leak the new object to another thread, which could then access it and race with the remainder of the atomic section. This is not a problem if we defer all lock releases until the end of the atomic section, since the newly constructed object cannot escape. With early release it can be solved by constructing new objects with their lock already acquired. The implementation actually already does this, since the analysis fixed point contains the path x after the construction but not before it. Hence, this lock is acquired late in the normal fashion.

The code inserted among the statements of the atomic section differs from the code used at the entry of the atomic section in that it does not have to check for deadlock and rollback. It does have the problem of avoiding null exceptions, however. The solution is the same, we test if the path is null before acquiring a lock on it. If a path resolves to null then the access it represents will never occur, so no lock needs to be acquired. As before, any analysis that informs us with certainty whether a path is null or non-null can help us avoid doing these checks at run-time.

Sowing unlocking code throughout the invoked functions of an atomic section causes problems when one of the functions is called from more than one context. A number of solutions present themselves: Aggressively inlining or duplicating functions is simple and minimises contention, but will increase the size of the program and may cause performance problems such as instruction cache misses. It should be possible to release locks not during the call but after it has returned, but a reference would have to be kept to the objects in question in case re-assignment renders them inaccessible (this is a similar problem/solution to the x = x.f and x = y.f cases discussed above). Alternatively, we could acquire a given lock once for each access (rather than once for all accesses as we currently do), and release it after each access. This could be achieved by omitting the NFA to DFA conversion. It would involve more overhead but this could pay off when compared to aggressive inlining. We chose to use the first technique as it was the simplest to implement, but there are various options we have yet to explore that may have different performance implications.

In conclusion, we were pleasantly surprised that we needed no extra analysis mechanisms to release our locks early. Our path graph analysis turned out to be powerful enough to encode read/write information and early release information in its basic form. There are some caveats but they occur in the later stages of the compilation.

4.6 Additional language features

4.6.1 Arrays

We can add support for Java style arrays by adding two further CFG node types, x = y[i]and x[i] = y. These are similar to x = y f and y = x f but the offset is not statically known. In the semantics, arrays can be objects with integer-named fields. The semantics of array access only differs in that the index of the access is taken from a variable rather than being statically known.

In the analysis, we need to add support for these two new nodes, in the form of *acc* and *tr* functions fig. 4.5. Recall that *acc* defines the accesses performed at a particular CFG node. For an array access, the *acc* function will behave the same as an object access, since we are aiming for a Java-like language where arrays are objects. We have already chosen to use one lock to protect all the fields of an object, so we also abide by this philosophy when locking arrays, i.e., one lock protects all the elements of the array. As such we define array accesses to generate edges just like field accesses do:

$$\begin{aligned} & acc(n)[x=y[i];_] &= \{y \to n\} \\ & acc(n)[x[i]=y;_] &= \{x \to n\} \end{aligned}$$

Recall that the tr function translates the path graph through the CFG node in order to preserve its meaning despite the changes to the state caused by said CFG node. Let us consider code like tmp = x[i]; tmp.g = 42 where an object obtained from an array is later accessed. To do this we need a way of obtaining the object and locking it at the beginning of the atomic section. We need to encode array indexes into the path graphs, and support them appropriately in the lock inference phase. It seems natural to support array accesses at *statically known* indexes in the same spirit that we support field accesses. To this end we extend the path graph edges to include an array access edge $n \xrightarrow{[v]} n'$ where v is an integer literal. The number of integer literals is bound by the size of the program so we do not lose decidability. We can then support translating through such statements (where the index v is a literal) as follows:

$$\begin{split} tr(n)[x = y[v]; _](G) &= G \setminus \{x \to n' | x \to n' \in G\} \cup \{n \xrightarrow{[v]} n' | x \to n' \in G\} \\ tr(n)[x[v] = y; _](G) &= G \setminus \{n' \xrightarrow{[v]} _ | x \to n' \in G, \\ (\nexists z \neq x : z \to n' \in G), \\ (\nexists n''' : n''' \xrightarrow{=} n' \in G)\} \\ \cup \{y \to n' | _ \xrightarrow{[v]} n' \in G\} \end{split}$$

This is exactly the same as for field access, and the explanations that were given then will also apply now. However the real utility of arrays in a programming language comes from being able to use computed indexes, i.e., where the index i is a variable on the stack. Let us allow the edge $n \xrightarrow{[i]} n'$ in our path graphs. Now, we must consider the possibility that arr[i] and arr[j] are aliases, indeed also that arr[i] and arr[10] are aliases, so a modification is needed to the above rule.

$$\begin{aligned} tr(n)[x = y[v]; _](G) &= \text{ as above} \\ tr(n)[x[v] = y; _](G) &= \text{ as above} \cup \{y \to n' | \exists i._ \stackrel{[i]}{\to} n' \in G\} \\ tr(n)[x = y[i]; _](G) &= G \setminus \{x \to n' | x \to n' \in G\} \cup \{n \stackrel{[i]}{\to} n' | x \to n' \in G\} \\ tr(n)[x[i] = y; _](G) &= G \setminus \{n' \stackrel{[i]}{\to} _ | x \to n' \in G, \\ (\nexists z \neq x : z \to n' \in G), \\ (\nexists n''' : n''' \xrightarrow{} n' \in G)\} \\ \cup \{y \to n' | _ \stackrel{[i]}{\to} n' \in G\} \\ \cup \{y \to n' | \exists v._ \stackrel{[v]}{\to} n' \in G\} \end{aligned}$$

Since we have now introduced variables that contain integers into our analysis, we have to consider statements such as i = i + 1. Previously we could ignore such statements because they do not affect later object accesses, but that is no longer the case, e.g., consider i = i + 1; tmp = x[i]; tmp.g = 42. We could imagine extending the concept of the path graph yet further to include edges like $n \xrightarrow{[i+1]} n'$. However now we are edging closer to the pit of undecidability since the path graphs are no-longer bound by the size of the program. No matter how elaborately we support arithmetic, we will at some point need to widen the analysis to get back decidability. We can do this by adding a wildcard edge $n \xrightarrow{[*]} n'$, which means every object in the array was accessed.

$$\begin{aligned} tr(n)[i = _ + _; _](G) &= G \setminus \{n \xrightarrow{[i]} n' | n \xrightarrow{[i]} n' \in G\} \cup \{n \xrightarrow{[i]} n' | n \xrightarrow{[i]} n' \in G\} \\ tr(n)[x[v] = y; _](G) &= \text{ as above} \cup \{y \to n' | _ \xrightarrow{[i]} n' \in G\} \\ tr(n)[x[i] = y; _](G) &= \text{ as above} \cup \{y \to n' | _ \xrightarrow{[i]} n' \in G\} \end{aligned}$$

With all these new edges in the path graph we need to revisit our lock inference procedure. We detect nodes involved in a cycle, as before. However we consider any node reachable from a wildcard $n \xrightarrow{[*]} n'$ edge to be involved in a cycle. We derive multilocks from the types of such nodes, and remove them from the graph. This yields a tree, which we can easily turn into paths. There may be array access edges of the form $n \xrightarrow{[v]} n'$ and $n \xrightarrow{[i]} n'$ left in the path graph, but not any of the form $n \xrightarrow{[*]} n'$. This means we can have paths with array accesses in them, e.g., x.y[10].f or x.y[z].f. We can resolve these at the beginning of the atomic section, just like with field accesses, but we have to avoid the ArrayIndexOutOfBoundsException. We can do this by either catching the exception or testing the bounds ourselves before indexing into the array. If the index is outside the bounds, there is no need to take a lock, since no access can occur. This is no different to our treatment of the null value when locking paths, as we discussed earlier.

The net effect of the above extensions to the path graph analysis and lock inference, is that if an atomic section contains an access of an object obtained from an array by means of a constant, or a variable whose value is assigned outside the atomic section, then we lock precisely the object accessed. If, however, the array index is computed inside the atomic section, we lock the type, paying a high cost in terms of granularity. On the positive side, the more arithmetic the analysis is taught to understand, the more precise locks it will infer to protect accesses of objects retrieved from arrays. But generally, it is best to perform array index arithmetic outside the atomic section if possible.

When doing early release, we were able to demote a write lock to a read lock when the analysis had stopped writing to an object but still had some reading to do. Similarly, we were able to demote a multilock to an instance lock if the atomic section was not going to iterate over an object structure but still had some individual objects to access. In the case of array indexing using a computed index, we may have to take a multilock as we would not statically know what element will be accessed. However, once the array index has been computed, the path graph would not contain the $n \xrightarrow{[*]} n'$ edge, and therefore lock inference would not yield a multilock, and consequently we could demote the multilock to an instance lock at this point, letting other threads proceed in parallel.

In conclusion, we can modify the analysis to handle arrays with much the same trade-offs as with objects. There may be some conservative approximation needed in the case where the programmer accesses objects obtained from an array using an index that was computed inside the atomic section, however early release can limit the granularity penalty.

4.6.2 Casts

The pathgraph analysis is agnostic to types. Indeed, our formalism and proof (§4.4) was not concerned with types, and modelled all objects as having the same (infinite) set of fields. It is only when we derive locks acquisition/release code from the inferred path graphs that we make use of typing information to choose multilocks (in the case of cycles and array accesses). As such, we can accommodate casts quite easily.

There is a recurring caveat, however: Consider $\texttt{atomic} \{ \texttt{y} = ((\texttt{A})\texttt{x}) \cdot \texttt{f} ; \texttt{y} \cdot \texttt{g} = 10 \}$, for which we acquire the locks x, $\texttt{x} \cdot \texttt{f}$. We must cast x to A before we can look in its field f. The type of x might be some super-type of A that does not have the f field we need to dereference. We must therefore either check that that the cast is possible, or catch the ClassCastException. We know the type of x and we know the type containing f because the path graph encodes the location in the CFG where the access occurred, and we can look up the type information of y at that point. This is the same mechanism we used to support read/write locks and also how we reduce cycles to multilocks. Points-to information, if available, could tell us that x is guaranteed to be a sufficiently precise type

at run-time, and thus could allow us to eliminate this extra check.

This solution to casting has the useful property that even if x is of type Object, we can cast it and access it inside the atomic section without ever having to lock the type Object. This means we do not lose precision through sub-typing, although we do have to handle the case where the downcast fails.

4.6.3 Additional Control Flow

We have presented an analysis that operates on general CFGs, so we can handle languages features such as **break**, **continue**, **return**, etc. through by adding more edges to the CFG. We can use the standard call strings technique [76] for extending our analysis to be inter-procedural, and conservatively approximate virtual dispatch by creating edges to all functions allowed by the type system. As usual, pointer analysis can help us statically resolve calls and thus determine a more precise call graph.

4.6.4 Splitting the Atom

Sometimes, it is desirable to turn off the atomicity of an atomic section for a while, perhaps to do some lengthy thread-local computation, or to communicate with other threads. While it is possible to refactor the code into two separate atomic sections, this can be challenging because atomic sections are lexically scoped. One may have to break conditionals, loops, and even functions between the two halves of the atomic section. In summary, one may have to make severe structural changes to the code. This motivates language support.

The most natural way to think about the extents of an atomic section is to think of the two points during the execution of the thread that define the duration of the atomicity. This sometimes does not correspond perfectly to lexical scoping. By analogy, sometimes it is hard to express locking patterns with Java's lexically scoped synchronized blocks, and one has to use the lock and unlock methods of a j.u.c lock [26]. However lexical scoping is still preferable because in the majority of cases where the locking duration does

```
class IntList {
       IntList next;
       int cargo;
       IntList (IntList next, int cargo) {
               this.next = next;
               this.cargo = cargo;
       7
}
class ConditionVariable {
       IntList waiters;
       void wait() {
               atomic {
                      waiters = new IntList(waiters,threadid);
                      preempt { park; }
       }
              }
       void notify() {
              atomic {
                      unpark waiters.cargo;
                      waiters = waiters.next;
       }
              }
       void notifyAll() {
              atomic {
                      while (waiters!=null) { notify(); }
}
       }
              }
```

Figure 4.9: Implementing wait/notify with preempt.

correspond to a lexical scope, it protects the programmer from accidentally forgetting to release a lock. This can happen quite easily, for instance when an exception is thrown, and a finally clause is needed.

To allow us greater expressive power without losing the benefits of lexical scoping, we can use a **preempt** section, which when placed inside an atomic section, releases/reacquires locks to break the atomic section into two distinct parts. We have used this construct to implement, e.g., message passing semantics on top of atomic sections.

We show how the **preempt** block can be used to implement the wait/notify of a condition variable in Figure 4.9. The IntList is a simple list of primitive integers. It is used to store the thread identifiers of threads waiting for a particular condition variable. The keyword threadid gets the id of the current thread. The unpark and park keywords allow suspending and resuming of threads, by thread id. The notify function pops an arbitrary thread from the list and unparks it. The notifyAll function calls notify until there are no more threads in the list. The wait function parks the calling thread after adding it to the list. We use atomic sections to protect the state of the waiters list, but we do not want to hold the lock whilst the thread is waiting, since this would prevent

another thread from notifying the condition variable (a form of deadlock). We therefore use the preempt block to allow other threads to proceed while the thread parks, splitting the atomic section.

Why did we not just end the atomic section before the park? It is quite desirable to call wait when inside an atomic section. The park would, in this case, occur during an atomic section and result in deadlock.

Park/unpark are fundamental constructs used to implement synchronisation primitives such as locks, condition variables, and countdown latches. In Java they are exposed through the park() and unpark() methods in the Thread class. A thread may suspend itself with the park call, which means it will yield and will not be scheduled again by the operating system until some other thread specifically targets it with the unpark call. Park/unpark are different to condition variables because they are used on specific threads, whereas condition variables are intended to synchronise the whole system, suspending whatever thread is necessary to achieve this aim. More importantly, park/unpark use a single element queue, which means that if the unpark statement occurs before the park statement, it is consumed and the thread is not suspended. This is different to the semantics of condition variables, where if the notify call occurs before the wait call, it is lost and the thread remains suspended. This distinction is important because it avoids a race condition in the code for wait.

The implementation of preempt sections uses the same program analysis as the implementation of atomic sections. A preempt section is represented in an atomic section's CFG by a *black hole* node that blocks the propagation of the path graph. This induces lock releases before the preempt, and acquisitions after it. We have to detect deadlock and backtrack when re-acquiring the locks after the preempt section, and also handle any null tests and casts that are required.

We finally give an example showing a number of features that we have discussed in this chapter. It shows the black hole node, which represents the preemp section, our treatment



```
ConditionVariable condVar;
atomic {
    sharedObject.cargo = 1;
    condVar.wait();
    sharedObject.cargo = 2;
}
```

The rest of the code is found in fig. 4.9.

Figure 4.10: CFG of an atomic section that calls wait().

of functions, read/write locks, and early release. Figure 4.10 is the CFG of an atomic section that called wait between a pair of object accesses. The object, sharedObject is an instance of a class like IntList from fig. 4.9. We have annotated the edges of the CFG with the fixed point locks, acquires, and releases. The grey circular node is the black hole node representing the preempt block in the wait implementation.

The fixed point of the analysis is in italic font, early lock release is in roman font, e.g., a release of the write lock on this is denoted U(!this). We have omitted redundant lock acquires and releases. The two this locks in the *far right* function correspond to the this variables in the wait function and the converter, which are distinct. In the implementation they are tagged with their function. The variables _t6 and _t13 are temporary variables introduced by the implementation to break down expressions when forming the CFG.

Preempt blocks seem to be useful in practice and have not received much attention from other researchers. They are not difficult to implement in our approach and it seems likely that they would be no harder to implement as part of other lock inference approaches or transactional memory. Generally, for a runtime technique where atomic sections are delimited by calls into a library, it is only necessary to end the current atomic section and begin a new atomic section at the boundaries of the preempt block. For a static approach using a CFG, one simply needs to consider a new exit and entry into the atomic section.

4.7 Case Study

To evaluate our lock inference strategy, we need to apply it to some real code. This will show how well the analysis scales to the kinds of atomic sections seen in practice. It also suggests whether the extra precision of the analysis is enough to make a practical difference. We can judge this by looking at the inferred locks to see what kind of contention we can expect at run-time.

4.7.1 AOLserver

Previous work [63, 28] has used the AOLserver [73] source code (which is available for public download) as a case study for atomic sections. AOLserver is a high performance HTTP server, written in C, which uses the Tcl scripting language to allow programmers to extend its functionality, e.g., to serve dynamic content. Although multiple Tcl interpreters can be running simultaneously, e.g., to serve multiple clients, they are essentially independent. The actual concurrency is handled with C code where a shared store is provided for communication between Tcl interpreters. Thus modules written in the Tcl scripting language can communicate via a native interface with this centralised shared memory database. The authors of the original work [63] kindly made their modified version of the source code available to us. The source code was annotated with atomic sections and a form of guard annotation (which we do not need). This particular code base is attractive firstly because real code bases that use atomic sections are rare, and also because the authors of [28] benchmarked their lock inference algorithm with these atomic sections. Their approach was very different to ours, and we were interested in comparing the scalability of their analysis with ours.

The analysis presented thus far in this chapter was implemented as a final year undergraduate project by Khilan Gudka [40]. We now use this implementation to demonstrate the analysis on real code. The implementation took the form of a compiler/interpreter for an extended subset of Java, with support for threads, atomic sections, primitive types, classes, reference types, arrays, inheritance, methods, overloading, sub-typing, dynamic binding, branches, loops, and return statements. Execution was implemented by rewriting the AST, so it does not make sense to measure run-time performance. However, one can measure the execution time of the analysis itself, and also study the inserted locks to see how fine-grained they are. The implementation is also capable of drawing, using GraphViz [27], the CFG for each atomic section annotated with the analysis fixed point and lock insertion information. We will not present these diagrams here because for large codes like AOLserver their layout is poor and they are very hard to interpret. However, they were still very useful for finding why the analysis inferred the locks that it did.

4.7.2 Porting AOLserver

Since AOLserver is written in C, and the implementation of our analysis was for a Javalike language, we had to translate the AOLserver code in order to process it. AOLserver is a large piece of software, so we chose one particular compilation unit, tclvar.c, which the previous work seemed to find most demanding. We analyse atomic sections independently, so we can judge the scalability of our analysis from a single compilation unit. The compilation unit we chose is the implementation of the shared memory database, described previously. We tried to reproduce the original code as closely as possible. Although it was written in C, the code had an object-oriented aesthetic and thus slipped quite easily into a Java-like language. We boxed the primitives when necessary to support passing by reference, and relocated global functions as methods of a convenient class. Because our analysis needs to follow function calls inside atomic sections, we also had to implement a small part of the Tcl API, on which the AOLserver code depended. Some of the Tcl functions we left as stubs or only partially implemented, but we were careful to reproduce any accesses the real Tcl API would perform.

The purpose of tclvar.c is to provide an interface into a simple shared memory database. The operations are called from Tcl code, via a native C interface. The structure of the database is a hash table of hash tables. The data is thus organised first by "module" then by "field", both of which are string keys. This is just to help script programmers avoid name clashes with other scripts when they use the database. Many of the operations available in the tclvar.c API therefore take both module and key, and do something with the value at that location, such as setting it, retrieving it, deleting it, incrementing it, etc. If the module or field does not exist, some operations create an empty hash table, and some raise an error, which returns back through the native interface and the user has to handle in their Tcl script.

The top-level structure of the software is sketched in fig. 4.11. The classes beginning with Tcl_ are written by us to emulate the behaviour of the Tcl API and were not part of the original AOLserver code. The dotted lines represent arrays, so the object structure is actually more like a tree.

The class ServPtr holds the context for the whole server. The Bucket class is just a wrapper around the Tcl_Hashtable. In the original code it contained a lock but that has been removed in our version which uses atomic sections instead. It is therefore now redundant but still present because we wanted to disturb the object graph as little



Figure 4.11: Top-level diagram of tclvar.c data structures.

as possible. The ServPtr class divides the first hash table into many hash tables, by hashing the module key (the same key used in the hash table, but a different hashing algorithm) to choose which of these level 1 hash tables to use. This was originally to allow more concurrency, since each lock (encapsulated in Bucket) would protect only a subset of key/value pairs in the first level of hash tables.

The first of the two hash tables uses the module name as a key, and an Array for the value (the value is actually cast to Object in the Tcl_HashTable implementation since we do not have generics. The Array class contains the data for a particular module, in the form of a hash table that maps field identifiers to values, both of which are StringPtr objects, which represent strings⁴.

4.7.3 Experiment

There are 11 atomic sections in tclvar.c. We will study a representative sample of 8, in order to see how the lock inference behaves. We refer to the atomic sections by the name of their enclosing function⁵. We also omit the prefix of the function names,

 $^{{}^{4}}$ In the toy language, strings are primitive values and must be boxed in order to be stored in a hash table.

⁵The other 3 atomic sections share the same function but are otherwise unremarkable.

NsTclNsv⁶. Each atomic section represents an operation that a website script may like to perform on the shared data. As they are designed to be ultimately called from Tcl code, the parameters take the form of an untyped Object[] called objv and the state of the interpreter, interp of type Tcl_Interp. In our version of the code, the latter is used to store the result of the operation, or the error, to be returned to the Tcl script.

- GetObjCmd: Return the string value at the given module and field. Returns error if module or field does not exist.
- SetObjCmd: Set the string value at the given module and field. Creates the module or field if necessary. Returns the new value.
- AppendObjCmd: Append the given strings to the value at the given module and field. Creates the module or field if necessary. Returns the new value.
- ExistsObjCmd: Tests if the given module and field exist.
- LAppendObjCmd: Append the given strings to the list at the given module and field (a list is a specially formatted string in Tcl). Creates the module and field if necessary. Returns the new value.
- IncrObjCmd: Adds 1 to the number (a number is a specially formatted string in Tcl) at the given module and field. Creates the module and field if necessary (yielding the number 1). Returns the new value.
- NamesObjCmd: Returns the list of module names matching a given pattern.
- UnsetObjCmd: Deletes a module or a field of a module, depending on the number of arguments given. Returns error if the module or field does not exist.

All of these functions except NamesObjCmd use a function called LockArray, so-called because it was used to acquire a lock that protected an Array object. With this locking

⁶The original name of the software was *Naviserver*.

code removed, the function's only task is to return an **Array** given a module name, and optionally (guided by a Boolean parameter) create the array if it doesn't already exist, rather than returning an error. There was also a corresponding **UnlockArray** function which we have removed altogether.

4.7.4 Results

Although our implementation was naive, using Java data structures to store the analysis state, we hoped to be able to analyse the AOLserver code fast enough to be practical for a real compiler. For each atomic section, we recorded how many nodes were present in the CFG, and how long it took to solve on a P4 3.2GHz CPU with 1GB of RAM, running the Java Hotspot VM v1.6.0. The results in [28] state 2399 seconds for tclvar.c. Our total time was approximately 8 seconds. Each atomic section consisted of an average of 250 CFG nodes. The solve time includes the time spent minimising the path graph and inferring a set of locks at each edge.

The code is quite torturous for a lock inference algorithm, as its object access patterns are highly unpredictable. It is hard to imagine a worse example to show the merits of lock inference but that is perhaps to be expected since we chose the compilation unit that the previous work found most demanding. The good news is that our analysis is much faster than [28], as it works directly on the CFG without a costly setup phase, and the solving phase is faster too. Their approach was to reduce the problem into SAT form, and the constraints were based around aliasing constraints. They typically gave a huge amount of work to the SAT solver. Our approach is more direct, operating straight on the CFG of the atomic section. Each atomic section is treated separately, so the time is linear in the number of atomic sections. Most of the time is spent cloning and garbage collecting, which suggests that a less wasteful design could get even better performance.

The locks inferred for each atomic section are given in fig. 4.12. Each atomic section has its own column. For reference, we give the full code for one of the atomic sections, and

	GetObjCmd		SetObjCmd		AppendObjCmd Ex		stsObjCmd	1	
	(this			(this)		(this)		(this)	ĺ
		(this.buckets)		(this.buckets)		(this.buckets)	(thi	s.buckets)	
				(objv)		(objv)		(objv)	1
			(interp))	(linterp)			1
		(!interp.res	(!interp.result)		sult)				
Array		R		R		R		\mathbf{R}	
Bucket		R		R		R	R		
StringPtr		R		R		W	R		
Tcl_HashEntry		[W]		W		W	[W]		
Tcl HashEntr	Tcl_HashEntry[]		W			W	iwi l		
Tcl_HashTable		W		W		W	[W]		
								1	
	LAppendObjCmd		IncrObjCmd		NamesObjCmd		UnsetObjCmd		
	(this)		(this)		(bucketPtr)		(this)		
	(this.buckets)		(this.buckets)		i(bucketPtr.arrays))	(this.buckets)	
	(objv)		(objv)		bucketPtr.arrays.bucket		kets	s (objv)	
	(!interp)		(interp)		(interp)			(interp)	
			(!interp.result)		(!result)			(!interp.result)	
				(countPtr)					
Array	\mathbf{R}		R					R	
Bucket	R		R					R	
StringPtr	W		R					R	
Tcl_HashEntry	W		W		R		W		
Tcl_HashEntry[]	W		W					W	
Tcl_HashTable	W		W					W	

Figure 4.12: Locks inferred by our analysis to the AOLserver tclvar.c fragment

the aforementioned LockArray function, in fig. 4.13 and fig. 4.14, which can be compared to the locks given in the first column of fig. 4.12. At the top of the column are given the paths that were locked, and at the bottom the types that were locked. Write path locks are denoted with an exclamation mark, and we distinguish between read and write locks of types using R or W respectively. The parentheses and square brackets will be explained shortly. We now explain what aspects of the code are responsible for causing these locks to be inferred.

We infer a type lock when we do not know which objects were touched (only their type). As discussed earlier, this is caused by either an iteration over an object graph or an array access through an unknown index. This case study is particularly torturous because both kinds occur: The hash table contains an array of entries, and the key is hashed to form an index into this array. Furthermore, at this index is a linked list that must be searched for the right key. For this reason we see a lot of accesses being guarded with type locks. However as expected with early unlocking, these big locks are released as soon as the specific object is known. We do not show explicitly where the early unlocking class ServPtr {

```
. . .
       int GetObjCmd (Tcl_Interp interp, Object[] objv) {
               if (objv.length != 3) {
                      interp.Tcl_WrongNumArgs(1, objv, "array_key");
                      return interp.TCL_ERROR;
               }
               Tcl_HashEntry hPtr;
               Object objv1 = objv[1];
               Object objv2 = objv[2];
               atomic {
                      Array arrayPtr = this.LockArray(interp, objv1, false);
                      if (arrayPtr == null) {
                             return interp TCL_ERROR;
                      }
                      StringPtr key = (StringPtr)objv2;
                      hPtr = arrayPtr.vars.Tcl_FindHashEntry(key.s);
                      if (hPtr != null) {
                              StringPtr val = (StringPtr)hPtr.val;
                              interp.Tcl_SetStringObj(interp.Tcl_GetObjResult(),val.s);
                      }
               }
               if (hPtr == null) {
                      string[] args = new string[2];
                      args[0] = "nousuchukey:";
args[1] = interp.Tcl_GetString(objv[2]);
                      interp.Tcl_AppendResult(args);
                      return interp.TCL_ERROR;
               }
               return interp.TCL_OK;
       }
}
```

Figure 4.13: The GetObjCmd function written in our toy language.

```
class ServPtr {
       Bucket[] buckets;
       Array LockArray(Tcl_Interp interp, Object arrayObj, bool create)
       {
              StringPtr arrayObj2 = (StringPtr)arrayObj;
              string array = arrayObj2.s;
              int hash = 0;
              int i = 0;
              while (i<array.length) {</pre>
                     hash = hash + (hash*2*2*2) + array[i];
                      i++;
              }
              hash = hash % this.buckets.length;
              Bucket bucketPtr = this.buckets[hash];
              Array arrayPtr;
              if (create) {
                      BoolPtr nu = new BoolPtr();
                      Tcl_HashEntry hPtr =
                         bucketPtr.arrays.Tcl_CreateHashEntry(array, nu);
                      if (!nu.b) {
                             arrayPtr = (Array)hPtr.val;
                     } else {
                             arrayPtr = new Array(bucketPtr,hPtr);
                             hPtr.val = arrayPtr;
                     }
              } else {
                      Tcl_HashEntry hPtr = bucketPtr.arrays.Tcl_FindHashEntry(array);
                      if (hPtr == null) {
                             if (interp != null) {
                                    string[] args = new string[2];
                                    args[0] = "no_such_array:";
                                    args[1] = array;
                                    interp.Tcl_AppendResult(args);
                             }
                             return null;
                     7
                      arrayPtr = (Array)hPtr.val;
              }
              return arrayPtr;
       }
       . . .
}
```

Figure 4.14: The LockArray function, also in class ServPtr.

occurs because it makes the code quite unpresentable.

Let us examine the atomic section in GetObjCmd and the locks inferred for it. The Bucket and Array classes are selected from the buckets array of ServPtr by the function LockArray using a hash on the module name, which is why they have type locks. There is also a hash of the module name for looking up the actual table corresponding to the module in the Array's hash table, so we see classes of the hash table implementation being locked by class. Finally the field name is hashed to find the hash entry in the module hash table to extract the desired value. This object is accessed (after being cast to a StringPtr) indirectly through a computed array index, hence the locking of the StringPtr class.

The GetObjCmd operation ought to be a read-only operation. However because the LockArray function is generalised, and can also modify the hash table if called with a parameter of true, the analysis conservatively infers write locks. This could be fixed using some combination of inlining, constant propagation, and dead code elimination before running our analysis. We were able to test this hypothesis by doing this optimisation manually. We added the function LockArrayFalse that did not have this Boolean parameter and omitted the true side of the branch. The effect on GetObjCmd was that the three write type locks turned into read type locks. We did the same transformation for the other atomic sections and the write locks that became read locks are denoted in fig. 4.12 with square brackets.

There are also some path locks, in fact all of which are redundant. The locks of this and this.buckets are redundant because in a real implementation these accesses are to immutable fields or array elements, and the analysis would therefore not infer any locks to protect them. The parentheses denote locks that are redundant, providing no safety and causing no contention. The interp variable is a pointer to the Tcl interpreter which is interpreting the Tcl script that requested the operation. As such it is thread local, so is also redundant. The interp.result access corresponds to setting the result string to be

CHAPTER 4. LOCK INFERENCE

received by the Tcl script. The result string is also thread local. Note that the omission of objv1 and objv2 is both deliberate and correct – as they are accessed after being cast to StringPtr, their accesses are protected by the type lock StringPtr. Although interp.result is also a StringPtr, its write lock is not subsumed by the read lock on StringPtr.

The NamesObjCmd operation is very different, because it only descends into the first hash table. As such, it does not use LockArray and therefore does not suffer from the problem discussed earlier, where LockArray is called with a false argument, but the analysis still infers locks for the true branch of the conditional within. Consequently the inferred locks of NamesObjCmd are much more accurate. The type lock on Tcl_HashEntry is still required, however, as we are iterating over the array within the hash table, and the hash entries within.

The other atomic sections are similar to GetObjCmd. There are some minor differences, such as extra redundant locks of thread local variables in e.g., IncrObjCmd. The extra type lock of StringPtr in the two append functions is due to their modifying the string within the StringPtr that was stored in the hash table. It would be possible to refactor the code to avoid this.

4.7.5 Evaluation and Future Directions

The analysis worked, and inferred locks in reasonable time given that this was not an optimised implementation. One critical question that remains is run-time performance. We cannot benchmark this code with this run-time since the language has only a naive interpreter and any results would be meaningless. Khilan Gudka is working on a real implementation, inferring locks at the JVM bytecode level. Thus, we expect getting real performance numbers to be practical in the near future. We can however hypothesise about performance already, given the results of the analysis.

There are two axes along which we can discuss performance. We can talk about serial

CHAPTER 4. LOCK INFERENCE

performance, which is impeded by the insertion of a lot of extra instructions. Perhaps more importantly we can talk about contention, the serialisation of atomic sections that may need to run in parallel. Contention is determined by the kinds of locks we insert.

Despite analysing atomic sections with hundreds of CFG nodes, we have only inferred a handful of locks. If we remove the locks protecting accesses of read-only and thread local state, the overhead should be very small indeed. The former is easy to do because final fields are well-understood. The latter would require a thread-local type system of some sort, which would be a direction for future research.

Removing redundant locks does not affect contention, because these locks were not introducing any actual synchronisation. In the case of final fields, this is because only read locks are ever taken. In the case of thread-local state, no other thread participates in the lock. It is the non-redundant locks that therefore define the contention.

We mentioned earlier how we were able to avoid over-conservative write locks in GetObjCmd by specialising the LockArray function in the case where a Boolean parameter was false, thus allowing the removal of dead code. Any removal of dead code has the potential to help this analysis infer fewer locks, so a real implementation should order the compiler passes to take advantage of this.

All this taken into account, our inferred locks have the property that the atomic sections GetObjCmd, ExistsObjCmd, and NamesObjCmd can run in parallel. However all of the other atomic sections will run in series.

Let us now compare the inferred locks with the locking strategy of the original code. Consider first that the original code was hand-written and thus could contain race conditions and deadlocks if the programmer did not invest significant care and attention. The locking strategy was that a lock was associated with each **Bucket** object. This lock protected all of the objects in the bucket, which includes the hash table that hold the modules who share a hash on their names, the hash entries within this hash table, the **Array** objects representing each module (which were stored in the hash table), the hash table encapsulated by the Array object and so on down the the individual StringPtr objects at the leaves of this tree. This meant that any two operations would run in parallel if their module names did not hash to the same thing. On the other hand, the lock used was not a read/write lock so reads of the same hash table would not be allowed to run in parallel.

Summarising these differences, we have more locks, we have read/write locks, in the case of two read operations on the same module name we permit more parallelism. But in the case of two writes on modules names that hash differently we have less parallelism. Let us not forget that unlike the hand-written code, our locks are guaranteed to be correct. But let us imagine what would need to be done to cause our analysis to infer locks that allow at least as much parallelism as the manually-written code found in AOLserver.

The most obvious improvement is to eliminate our type locks and use instance locks instead. How is this possible given the unpredictable and unbounded access patterns in these atomic sections? Let us look at each type lock in the GetObjCmd column of fig. 4.12 in turn.

The lock of the Bucket type is caused by the access of a bucket that was looked up in an array using an index computed by taking the hash of a string. The hand-written code solved this problem by computing the hash and finding the bucket before taking any locks. This works because the string is thread-local, and the bucket array was constant so needed no protection. The hash was calculated and the bucket found by the LockArray function. Of course since the original code stored the lock implementation within the bucket, there was no way it could take this lock before first locating the correct bucket.

In our transcription of this code, we begin our atomic section before the call to LockArray, thus have subsumed the hash and array lookup operations in the atomic section and have thus generated locks to protect these thread-local accesses. Such locks are not actually the source of the problem, they are redundant because system-wide only read locks are taken. The source of the problem is that now the bucket, which is later
accessed, is not available in a variable at the beginning of the atomic section. Its access exists in the path graph beyond an array access edge.

If we were to move the hash and array lookup out of the atomic section, would it remove the Bucket lock? Fortunately we do not need to perform this experiment because the NamesObjCmd atomic section already does this. Correspondingly we see no Bucket lock in the NamesObjCmd column but there is a lock of bucketPtr, which is the variable containing the precise bucket that will be accessed.

Can the programmer perform this optimisation manually by re-arranging code and moving array lookups out of the atomic section? In the case of LockArray this requires the atomic section to begin in a different function to where it ends. There would then need to be a pair of statements rather than a syntactic construct to delimit the atomic section. This is analogous to the difference between lock / unlock functions and Java's synchronized block.

Can we automatically hoist the hash calculation and array access out of the atomic section? Firstly, hoisting code out of the atomic section is only safe if the code does not perform any accesses of shared memory, so we would need some static notion of thread safety, e.g., a type system, in order to know when such an optimisation is possible. The original handwritten code happens to have this property but there was no static checking to ensure this was the case. In this case we have code that computes a hash of a string, therefore accessing the string, and also an array accesses. To ensure these accesses need no protection, a type system would give us the information that the string we are hashing and the array being indexed are not able to be modified by other threads. The array is constant in AOLserver, giving us this property, although this is not expressible in many type systems including Java. For the string, the language could have constant strings or the type system could establish that the string was thread-local, either of which would be acceptable within AOLserver. In general we can (if known) move any thread-local or constant memory accesses out of the atomic section, which would make the analysis infer fewer locks, give a better precision for the remaining accesses (as with this bucket example) and also reduce the amount of time spent holding locks.

In fact if we had the array support described earlier (§4.6) then we only need the programmer to manually hoist out the hash calculation. We would then lock the individual array element because the index would be known, i.e., bound to a variable, at the beginning of the atomic section where the locks were being taken. The array lookup would be statically known, and thus could be treated like a field lookup. Note that if the hash calculation were trivial and the arithmetic involved could be statically understood by our analysis, we would need no more machinery than the techniques discussed already in this chapter. However for a hash function this would be very unlikely, as the meaning of the arithmetic that defines a hash is very obscure.

We now know how to get rid of the Bucket lock but GetObjCmd has five more type locks that will still restrict concurrency. The Bucket instance contains a hash table that we then look up to get a Array object that contains a second hash table, which we also look up to get a string, which is then examined. These computed indexes lead to the locking of all five of the remaining type locks found in the first column of fig. 4.12. We do not know exactly which of these objects were accessed so we opted to lock all objects of the same types. Again, let us look to the original code to see how it avoids this problem. The lock in the Bucket instance does not just protect the bucket but also the other objects logically contained within. This includes all of the objects that may be accessed but is not so comprehensive as to include all the objects that happen to have the same type as the accessed objects. This is the source of the extra parallelism in the manually written code. The scope of the bucket lock is limited to one subtree of the whole structure, rooted at the particular bucket, and thus operations on different buckets may proceed in parallel. Again, this only works because the code obeys an unchecked property: The objects under the bucket must not be aliased from under another bucket, or there would be a race condition where two threads perform operations on different buckets but access the same state.

We would like to be assured that such a containment property holds before we exploit it automatically. In fact, we have already seen type systems that expresses containment in the form of universe types $(\S3)$ and other ownership type systems. Can we express the manual locking discipline in the form of our race safety type system? Firstly we would express the containment of all the state inside a Bucket by using a rep annotation on the arrays field of the bucket and **peer** annotations for the references beneath. This prevents the kinds of aliases just discussed because a **rep** of one bucket is distinct in the type system from a **rep** of another bucket. Now how could we modify our analysis to make use of this information? It is in fact just the lock inference stage rather than the path graph analysis that needs to be changed, and only slightly. We are still using the type of an access to protect it if the exact object is unknown. The difference is that if we have a universe type system we have a more precise type that represents a much smaller set of objects. In fact it represents exactly the set of objects we want the lock to protect. So it seems that if we were to augment the toy language with a universe type system or indeed any ownership type system then we could infer not just the same locks as the manually written code, but better locks admitting more parallelism. The extra parallelism comes from the read/write locks that would allow two operations on the same bucket to proceed in parallel if they are both reads, something that the original code lacked.

There is still a question of whether we should keep using the standard Java class types to define locking domains as well as using the universe types for the same purpose. In other words rather than just locking all objects owned by a particular object, should we additionally restrict the lock to those objects that are sub-types of a particular class? In the case of AOLserver this would not be very beneficial as we would just infer several locks to protect the objects under the bucket instead of one lock. This would add sequential overhead and in this case gives us no extra parallelism since most of the atomic sections in tclvar.c are operating on objects of the same set of types. Further exploration of real code is needed, but it seems like a better system would be to use only the ownership part of the type to define locking domains.

If we have an ownership type system, we can also try and use it to define thread locality. Instead of saying an object is owned by some other object, we can have an object being owned by some thread-local owner instead. This would mean the object would not need to be locked, and would facilitate the optimisations previously mentioned. But in order to be correct, we would have to ensure these objects are not passed inadvertently to other threads. This is left as future work.

4.8 Chapter Summary

We introduced this chapter by suggesting that rather than the programmer adding their own locks, whose correctness are enforced automatically, it is better for the system to handle these details, leaving the programmer only to specify where the atomic sections should be. We have summarised some of the related work in this area and identified some areas that we suggested were lacking. These were the run-time performance of software transactional memory and its lack of support for I/O, and also the lack of precision of existing lock inference approaches. We then gave an analysis that is more precise, especially in the case of loops. We formalised this analysis and proved that the accesses inferred were a conservative approximation of those occurring at run-time. The formalisation and proof were developed in Isabelle.

We gave a scheme by which we use the knowledge of what accesses will occur to insert locks into the code that follow the two-phase protocol. We suggested using runtime deadlock detection to avoid coarsening the locking strategy. We then showed how the analysis could be extended to more high-level language features, particularly arrays, casts and message passing idioms. We demonstrated an implementation of the analysis by running it on code from a production web server, used in some of the previous work. We presented the inferred locks but were unable to measure run-time performance. We did some manual optimisations to improve the quality of the locks, by avoiding conservatism in the analysis. Finally, we made several suggestions about further optimisations that could bring the quality of the locks in line with and beyond that which were used by the programmers who originally wrote AOLserver.

Chapter 5

Conclusions and Further Work

Earlier (§2) we made the claim that the fundamental property in which programmers should be interested is atomicity. The techniques discussed were designed to help programmers get the atomicity they want, assuming they know when and where they want it. It is appropriate now to assess the extent to which we have moved the state of the art towards this objective, what remains to be done, and whether we can get there by building upon this work.

5.1 Summary

We asserted at an early stage (§2) that techniques that did not require having to roll back arbitrary code were likely to perform better at run-time. We thus chose to use the two-phase locking protocol. We left deadlock avoidance to the programmer with our lock checking work (§3) and later in our lock inference work (§4) we exploited the property that locks are acquired together to solve deadlock using a simple rollback strategy.

After requiring programmers to choose which blocks they wanted to be atomic. We then proposed either checking programmer-inserted locks for adherence to the two-phase discipline (§3), or automatically inserting two-phase locks (§4). We expected that automatically inserting locks would be harder than checking locks, and thus we would be able to make more progress doing the latter. However there were some surprises. Firstly, we can insert locks that are not lexically scoped, because the only benefit of lexical scoping is that it prevents the programmer forgetting an unlock statement. This gave us more flexibility to release locks early.

Secondly, we acquire all the locks together at the beginning of the atomic section, whereas if the programmer is in control they can put the locks wherever they like. This allowed us to solve the deadlock problem, which we could not do for the programmerinserted locks. We can avoid deadlock without coarsening the lock granularity, which is a property that no other lock inference approach (§4.2) seems to have.

Combining these advantages with the obvious advantage that it's more work for the programmer to insert locks themselves, makes the future look much brighter for lock inference than lock checking. However many of the techniques and lessons learnt from the universe types and race safety type systems (§3) are very relevant to lock inference. In fact the central role of paths and path graphs in our lock inference was inspired by the paths in our race safety type system used to reason about sync p e where p had type ANY. Ownership types of one form or another are likely to always play an important role in concurrency, as they allow us to express the relationship between objects in a static and precise way. We earlier (§4.7) advocated their use in the AOLserver code to allow inferring locks of a similar quality to those used in the original hand-written code.

We also suggested that ownership type systems such as universe types can also be used for expressing thread-locality, through the use of a special 'thread local' owner. This interacts nicely with ownership type parameters, or in the case of universe types, the **peer** modifier, to allow the programmer to build reusable data structures that can be deployed in both thread-local and shared memory contexts. However there would have to be additional static checking to make sure thread-local objects were not accessed from other threads. Ownership types and universe types do not differentiate between threads.

Looking forward, we think the best direction in which to proceed is to extend universe

types with notions of thread locality and use them as the base language for our lock inference work. We do not expect any surprises integrating universe types with lock inference but the thread-locality aspect is a major unknown, requiring careful study. Thread local types would have applications beyond our own work, for instance they may be useful as a general safety check, for accelerating transactions, or for allowing re-ordering of accesses within the constraints of a strong memory model.

5.2 Evaluation

Suppose we use our intuitions gained from our lock checking work to influence the direction our lock inference work should next take. Suppose we imagine that the AOLserver code is compiled, and its atomic sections are implemented with inferred locks of a similar or better quality to the original hand-written locks. Is this enough? Is such a fusion of proven techniques (§3) (§4) combined with a notion of thread locality suitable to form the heart of a modern concurrent object-oriented language?

There are three potential problems. We must make sure our language is expressive enough, i.e., does not reject too many programs, and does not have too much bureaucracy in the form of type annotations. We must also ensure that the compile time is reasonable, especially since compilation is more and more taking place in JIT compilers. Finally, runtime must be comparable to hand-written code, given the advantages of an automatic approach.

5.2.1 Expressiveness

The most interesting feature of our race safety type system, in terms of expressivenes, is the use of **any** to represent an unknown owner. This allowed programs to be written that the related work could not support. In addition there was no need to use final fields in paths, but in practice one can work around this by defining final local variables and evaluating the path before the **sync** block. In future we would drop this feature as the required effects system causes us to restrict inheritance or loses us separate compilation. However we believe there is a strong case for **any**, or at least some similar method of introducing subtyping into ownership. In principle we are more expressive as we allow the programmer to specify the inserted locks. However, there is no way to solve the problem of deadlock, as we did in the lock inference algorithm. So in practice the locks are still not expressive enough. The locks used internally in our lock inference work are much more expressive, but are hidden from the programmer.

Currently our lock inference analysis does not reject any programs in the toy language. It can handle whatever the programmer throws at it. In general though, program analyses have big problems with reflection, because it becomes very hard to statically discover even the type of what was accessed. With methods called via reflection, we have no idea what code will be executed. We have a number of options, however. We can fall back to a single global lock in these instances. We can try and use pointer analysis to discover what is actually accessed, although we cannot in general get back everything. Also it is possible that many of the uses of reflection could be satisfied by providing a more restricted language construct that we could hope to statically understand, e.g. the ability to iterate over the fields of an object. Taking all of this into account, and comparing to transactional memory, which prevents the use of I/O in atomic sections, we believe we have a reasonably expressive system.

In terms of bureaucracy, we expect that we will need some form of ownership types to make lock inference practical, and some extension to allow the specification and checking of thread locality, but that there will be no need for other annotations. So clearly these annotations must be either reasonably light-weight or inferred. Ownership inference is quite hard and we have no reason to believe that thread-locality inference is easy either, but what constitutes light-weight annotations? We believe programmers do not mind annotations if they are helping to document the code in some way or giving them some degree of control, and are not just banal. In other words are the annotations useful for programmers or do they just get in the way?

Thread locality is a very important part of the design of concurrent software, and many bugs are introduced by mistakenly thinking an object is thread-local when it is not. It is thus common for people to document which objects are shared and the locks that protect them, in order to help avoid such mistakes. By providing a system of annotations for these same concepts, it will not only help the programmer ensure the documentation is present and correct, but will also help to infer better locks. Also the ownership types, if used to specify which lock guards which object, give the programmer a great deal of control in terms of granularity (§2.6). This may be important for getting the desired level of performance out of the program. So in both cases we think these annotations are justified for a real language. Of course it remains to define an actual type system and try to use it to express some existing real code like we did with AOLserver.

5.2.2 Scalability

Correctness of concurrency, in particular lock inference, is very sensitive to the behaviour of code in non-local parts of the program. For this reason it does not mesh well with separate compilation where only the signature of code is known, not the code's complete behaviour. For this reason we must infer locks at link time where the whole program is known. In today's languages where plug-ins are allowed, some of the code may not even exist until the program has already started executing.

For example, the program might load a new class that extends an existing class, and we would then have to consider the possibility of polymorphic dispatch into the new methods from an existing atomic section. For this we would have to re-infer locks for that atomic section taking the new branch of the CFG into account. There is also a trend towards JIT compilers that have advantages in adaptive optimisation and allow the distribution of portable byte-code. These facts motivate an approach that can support inferring locks at run-time. It is therefore very important that the analysis itself can be implemented efficiently. There is one major hurdle to overcome, and this is perhaps the biggest weakness of all the work presented thus far. Since we have to analyse deep into functions, there is the possibility of the CFG of an atomic section growing very large. Even if there is a large amount of lazy initialisation code that does not get executed, it still has to be analysed and locks inserted. For instance the System.out.println() method conservatively calls thousands of methods, although it typically will not call these on the majority of invocations. Since a big CFG means lots of edges, and we have to do work at each edge, we expect the performance to be at least linear in the size of the CFG. Because more code implies usually more accesses, the amount of work per edge (in terms of translating the path graph) is likely to increase as well. This in itself implies our approach is not scalable in the sense that just blindly calling into a library in an atomic section can create problems.

How can we address this? One solution is to try and analyse functions independently and then combine the result. This allows re-use of the analysis result of a particular function in all the places where it is called. We expect this to help but there is still the possibility of path graphs becoming very large. Thus scalability would still become a problem. We would like some mechanism whereby subgraphs of the path graphs could be somehow abstracted or otherwise hidden, and manipulated opaquely. However it is not yet clear how this can be done. A final alternative is to ensure that code bloat is kept under control, perhaps by not using lazy initialisation and other tricks that, while efficient at run-time, result in a large static call-graph.

An interesting area of future work is the ability to incrementally analyse and infer locks for a particular atomic section. Just like one can invest progressively more work into analysing a block of code for general optimisation, so that the hottest code receives more attention, one could imagine the same being done for atomic sections. For example, each atomic section would initially use a giant global lock, but when executed frequently would be re-analysed, so that the most commonly executed atomic sections receive fine grain locks but they need not all be analysed to the full extent. This could help a lot with scalability.

5.2.3 Performance

Multi-threaded programming is used to get better performance, and even if there are surplus cores single-threaded performance is still important due to Amdahl's law. However we have not got any firm performance numbers that support the decisions we've made and this is a valid criticism of our work. This is because the toy language implementation was not suitable for doing performance analysis. A real implementation is a substantial piece of work in its own right. Khilan Gudka has made progress in this direction, developing a system for analysing and inserting locks into JVM bytecode.

Other than the inferred locks, which we can compare in terms of number and granularity to handwritten code, one potential source of performance degradation is our deadlock detection / rollback approach. However we expect this to not degrade performance significantly, a claim that we base on our experience that deadlock is actually rare at run-time, especially if the lock acquisitions are close together. Because deadlock can only occur when both threads are in their lock acquisition phases, the time period in which a deadlock is possible is very small. Increasing the number of threads, decreasing the number of locks in the system, increasing the number of locks acquired by a particular atomic section, and decreasing the amount of time spent in the body of the atomic section would all increase the probability of deadlock and might force a performance problem. It would be interesting to write micro-benchmarks to explore how hard it is to write code that triggers enough deadlocks to cause a significant performance degradation.

One interesting area of future work, to which we have given very little consideration, is the interaction of atomic sections implemented with lock inference with manuallysynchronised code. This might be useful for integrating libraries into an application, or for incrementally introducing atomic sections into an existing code base. A more compelling reason would be for specially hand-written high performance data structures, such as hash tables, trees, and linked lists. Taking advantage of these special cases one can write much faster code than allows more parallelism than the locks we can infer. While the operations on these data structures are atomic by themselves, it is sometimes necessary to use these operations within another atomic section, e.g., removing an object from a list and then modifying the object. It may be possible to infer better synchronisation, perhaps with help from the author of the data structure, than we would do using our usual lock inference.

5.3 Future Case Studies

There are some interesting questions about the potential usage of atomic sections and thread-locality in practice. It would be interesting to find out, for a selection of large real applications using shared memory atomic sections: How many atomic sections there are and how large they are. What percentage of code is executed from atomic sections? What percentage of running time is spent in atomic sections? What percentage of classes are shared compared to thread-local, and what percentage of actual instances. It would also be interesting to see if there is much difference between various applications in terms of these metrics. This kind of information would help us make decisions about performance and expressiveness when designing programming languages for concurrency.

5.4 Final Words

We have outlined a pair of novel approaches for implementing atomic sections by locking the appropriate objects for the duration of the atomic section. This may outperform the existing transactional implementations due to less instrumentation of code, and outperform existing lock inference approaches by providing more granularity. We also cause less inconvenience to the programmer since I/O is possible in an atomic section. Using universe types for race safety is novel and using field effect sets to protect paths from interference we had not seen before. We proved that the type system guaranteed race safety but left the extensions that required two-phase locking unproven, because we felt there were no new ideas.

Our path graph analysis for inferring accesses is novel and has considerable precision. We proved it correct using Isabelle. We gave a scheme whereby the accesses can be turned into locks. Although deadlock detection is not a new idea, we believe we are the first to apply it to lock inference. Through applying the inference to a real code base we discovered we are unable to make use of all the precision of the path graph analysis without some kind of ownership type system such as the universe type system we used for race safety. Also we would like a thread-local type system to further improve the quality of inferred locks.

In the past, programmers have typically been so scared by threads that even when writing servers that are processing many requests in parallel, they have opted for event loops [74] and other co-operative threading approaches that offer some of the advantages of threads, without any of the potential errors. However these allow no parallelism and we are approaching an era where CPU clock rates can no longer meet the computational demands of our software [80]. Programmers need new language features to replace locks and help them write sophisticated, multi-threaded programs. These features must have intuitive semantics, and an efficient parallel implementation. Moreover, programmers should be able to make use of encapsulation and composition, and program behaviour must remain intuitive when programs are scaled in this fashion. We hope that the work presented in this thesis helps to address these concerns.

Appendix A

Proofs of Race Safety

We use \underline{IH} as an abbreviation for "induction hypothesis". The symbol \boxdot indicates the end of a sub-proof, and \Box a whole theorem/lemma.

Lemma 3.6.1 The effects of well-typed expressions do not undermine their locks.

$\mathbb{L},$	Γ	F	e	: F	\Longrightarrow	$\mathbb{L}\#F$	\wedge	$\forall p$	\in	L	:	$\mathbb{L},$	Γ	F	p	:	_
---------------	---	---	---	-----	-------------------	-----------------	----------	-------------	-------	---	---	---------------	---	---	---	---	---

Let		$\mathbb{L},\Gamma\vdash e:f$	(1)
case(1) la	st wa	as (SUB)	(2)
(2)	\Rightarrow	$\mathbb{L}\#F, \forall p \in \mathbb{L}.\mathbb{L}, \Gamma \vdash p: _$	
case (1) la	st wa	as (VAR) (THIS) (NULL) (NEW) (SPAWN)	(2)
(2)	\Rightarrow	$\mathbb{L} = \emptyset$	(3)
		$F = \emptyset$	(4)
$(3)\!+\!(4)$	\Rightarrow	$\mathbb{L}\#F \ \land \ \forall p \in \mathbb{L} : \mathbb{L}, \Gamma \vdash p : _$	
case (1) la	st wa	as (FIELD)	(2)
(2)	\Rightarrow	e = e'.f	(3)
		$\mathbb{L}, \Gamma \vdash e' : F$	(4)
$(4)+\underline{\mathrm{IH}}$	\Rightarrow	$\mathbb{L} \# F \ \land \ \forall p \in \mathbb{L} : \mathbb{L}, \Gamma \vdash p : _$	
case(1) la	st wa	as (Assign)	(2)
(2)	\Rightarrow	e = e' := e''	(3)
. ,		$\mathbb{L}, \Gamma \vdash e' : F$	(4)
$(4)+\underline{\mathrm{IH}}$	\Rightarrow	$\mathbb{L} \# F \ \land \ \forall p \in \mathbb{L} : \mathbb{L}, \Gamma \vdash p : _$	~ /
case (1) la	st wa	as (Sync)	(2)
(2)	\Rightarrow	$e = \operatorname{sync} e' e''$	(3)
		$\mathbb{L}, \Gamma \vdash e' : F$	(4)
(4) + IH	\Rightarrow	$\mathbb{L}\#F \ \land \ \forall p \in \mathbb{L} : \mathbb{L}, \Gamma \vdash p : _$	
case (1) la	st wa	as (CALL)	(2)
(2)	\Rightarrow	e = e'.m(e'')	(3)
		$\mathbb{L}, \Gamma \vdash e' : F$	(4)
$(4)+\underline{\mathrm{IH}}$	\Rightarrow	$\mathbb{L} \# F \ \land \ \forall p \in \mathbb{L} : \mathbb{L}, \Gamma \vdash p : _$	~ /
case (1) la	st wa	as (CAST)	(2)
(2)	\Rightarrow	e = (t)e'	(3)
		$\mathbb{L},\Gamma \vdash e':F$	(4)
$(4)+\underline{\mathrm{IH}}$	\Rightarrow	$\mathbb{L}\#F \land \forall p \in \mathbb{L} : \mathbb{L}, \Gamma \vdash p : _$	~ /

 $\mathbb{L}, \Gamma \vdash e : F$ $\mathbb{L}, h, \sigma \vdash S(e) : F$ $h, \sigma \vdash \mathtt{x} : \Gamma(\mathtt{x})$ Virgin(S(e)) $h, \sigma \vdash \texttt{this} : \Gamma(\texttt{this})$ $\mathbb{L}, \Gamma \vdash e : F$ Let (1) $h, \sigma \vdash \mathtt{x} : \Gamma(\mathtt{x})$ (2) $h, \sigma \vdash \texttt{this} : \Gamma(\texttt{this})$ (3)case (1) last was (NULL) (VAR) (THIS) (NEW) (4) $\mathbb{L} = \emptyset$ (5)(4) \Rightarrow $F = \emptyset$ (6) $e \in \{x, \texttt{this}, \texttt{new}\ t, \texttt{null}\}$ (7)(7)Virgin(e)(8) \Rightarrow e = S(e)(9)Virgin(S(e))(8)+(9) \Rightarrow $\forall h, \sigma : \mathbb{L}, h, \sigma \vdash e : F$ (10)(5)+(6)+(7) \Rightarrow (10) + (9) $\mathbb{L}, h, \sigma \vdash S(e) : F$ \Rightarrow case (1) last was (CAST)(4)e = (t)e'(5)(4) \Rightarrow $\mathbb{L}, \Gamma \vdash e' : F$ (6) $\mathbb{L}, h, \sigma \vdash S(e') : F$ (6)+(2)+(3)+IH(7) \Rightarrow Virgin(S(e'))(8)S(e) = (t)S(e')(9)(5) \Rightarrow (9)+(7)+(CAST) $\Rightarrow \mathbb{L}, h, \sigma \vdash S(e) : F$ (9)+(8)Virgin(S(e)) \Rightarrow case (1) last was (SPAWN) (4) $e = \texttt{spawn} \ e'$ (4)(5) \Rightarrow $\mathbb{L} = \emptyset$ (6) $F = \emptyset$ (7) $\emptyset,\Gamma\vdash e':_$ (8)S(e) = spawn S(e')(9)(5) \Rightarrow $(8)+(2)+(3)+\underline{IH}$ $\emptyset, h, \sigma \vdash S(e') : \emptyset$ (10) \Rightarrow Virgin(S(e'))(11) $\mathbb{L}, h, \sigma \vdash S(e) : F$ (10)+(6)+(7)+(9)+(SPAWN) \Rightarrow (11)+(9)Virgin(S(e)) \Rightarrow case (1) last was (FIELD) (4)e = e'.f(4)(5) \Rightarrow $\mathbb{L}, \Gamma \vdash e' : F$ (6) $\Gamma \vdash_{gb} e' : l$ (7) $l \in \mathbb{L}$ (8)S(e').f = S(e)(5)(9) \Rightarrow $\mathbb{L}, \sigma \vdash S(e;) : F$ (6)+(2)+(3)+IH(10) \Rightarrow Virgin(e')(11)(7)+(2)+(3)+Lemma A29 $h, \sigma \vdash_{gb} S(e') : l$ (12) \Rightarrow $\mathbb{L}, h, \sigma \vdash S(e) : F$ (8)+(9)+(10)+12)+(FIELD) \Rightarrow (11)+(9) \Rightarrow Virgin(S(e))case (1) last was (ASSIGN)(4)Same as (FIELD) case (1) last was (SYNC) (4) $e = \operatorname{sync} e' e''$ (5)(4) \Rightarrow $\mathbb{L}, \Gamma \vdash e' : F$ (6) $\Gamma \vdash_{gb} e' : l$ (7)

Lemma 3.6.2 Static race safety implies run-time race safety.

$(9)+(6)+(2)+(3)+{ m IH}$	⇒	$l \in \mathbb{L}$ $\mathbb{L} \cup \{l\}, \Gamma \vdash e'' : F$ $\mathbb{L}, h, \sigma \vdash S(e') : F$ $\mathbb{L} \cup \{l\}, h, \sigma \vdash S(e'') : F$ Virgin(S(e'))	(8)(9)(10)(11)(12)(12)
	$\begin{array}{c} \uparrow \\ \uparrow $	$\begin{aligned} &Virgin(S(e''))\\ &\texttt{synced}_{S(e')} \ S(e') \ S(e'')\\ &h, \sigma \vdash_{gb} S(e') : l\\ &\mathbb{L}, h, \sigma \vdash S(e) : F\\ &Virgin(S(e)) \end{aligned}$	$(13) \\ (14) \\ (15)$
case (1) last was (SUB) (4)	⇒	$ \mathbb{L}, \Gamma \vdash e : F' \\ \mathbb{L}' \subseteq \mathbb{L} \\ F' \subseteq F \\ \mathbb{L} \# F \\ \forall p \in \mathbb{L} : \mathbb{L}, \Gamma \vdash p : $	$(4) \\ (5) \\ (6) \\ (7) \\ (8) \\ (9)$
$egin{array}{l} (5)+(2)+(3)+{ m IH}\ (9)+(2)+(3)+{ m IH}\ (11)+(6)+(7)+(8)+(12)+({ m Sub}) \end{array}$	$\begin{array}{c} \Rightarrow \\ \Rightarrow \\ \Rightarrow \\ \Rightarrow \end{array}$	$ \begin{array}{l} \mathbb{L}', h, \sigma \vdash S(e) : F' \\ Virgin(S(e)) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : \\ \mathbb{L}, h, \sigma \vdash S(e) : F \end{array} $	(10) (11)
case (1) last was (CALL) (4) $(6)+(2)+(3)+\underline{IH}$	\Rightarrow	$e = e'.m(e'')$ $\mathbb{L}, \Gamma \vdash e' : F$ $\mathbb{L}, \Gamma \vdash e'' : F$ $\Gamma \vdash e' : u c$ $\mathfrak{Eff}(c,m)\downarrow_2 \subseteq F$ $\mathbb{L}' \in \mathfrak{Eff}(c,m)\downarrow_1$ $(u,e',e'') \Vdash \mathbb{L}' \subseteq \mathbb{L}$ $\mathbb{L}, h, \sigma \vdash S(e') : F$ $Virgin(S(e'))$	$(4) \\ (5) \\ (6) \\ (7) \\ (8) \\ (9) \\ (10) \\ (11) \\ (12) \\ (12b) \\ (15) $
$(7)+(2)+(3)+\underline{IH}$ (8)+(2)+(3)+Lemma 3.5.5 (11)+(2)+(3)+Lemma A30 (5) (16)+(12)+(13)+(14)+(15)+(CALL) (12b)+(13b)+(16)	↑ ↑ ↑ ↑ ↑ ↑	$\mathbb{L}, h, \sigma \vdash S(e'') : F$ $Virgin(S(e''))$ $h, \sigma \vdash S(e') : u c$ $(h, \sigma, u, S(e'), S(e'')) \bowtie \mathbb{L}' \subseteq \mathbb{L}$ $S(e) = S(e').m(S(e''))$ $\mathbb{L}, h, \sigma \vdash S(e) : F$ $Virgin(S(e))$	(13)(13b)(14)(15)(16)

Lemma 3.6.3 Path resolution is preserved over execution.

$\begin{split} h(\sigma,p) &= v\\ \sigma \vdash p, h \rightsquigarrow e, h \end{split}$	$ \begin{array}{c} e = p', h = h' \\ h'(\sigma, p') = v \\ \forall f \notin p : f \notin p' \end{array} \end{array} $	
Let	$ \begin{aligned} h(\sigma,p) &= v \\ \sigma \vdash p, h \rightsquigarrow e, h' \end{aligned} $	(1) (2)
case (2) last was (Contradicts p	(CAST) (ASSIGN) (FRAME1) (FRAME2) (CALL) (NEW) being a path	(3)
case (2) last was ((VAR)	(3)
(3)	$\Rightarrow v = \sigma(\mathbf{x})$	(4) (5)
	e = v = p h = h'	(0) (6)
(4)+(5)+(6)	$\Rightarrow h'(\sigma, p') = v$	(0)
(5)	$\Rightarrow \forall f: f \notin p'$	(7)
(7)	$\Rightarrow \forall f \notin p : f \notin p'$	
case (2) last was ((THIS)	(3)
Same as (VAR	R)	()
case (2) last was ((Стх)	(3)
(3)	$\Rightarrow p = E[e']$	(4)
< ',	e = E[e'']	(5)
	$\sigma dash e', h \rightsquigarrow e'', h'$	(6)
(4)	$\Rightarrow e' = p''$	(7)
WLOG	$E[\bullet] = \bullet.f$	(8)
(8) + (7) + (4) + (1)	$\sigma \vdash p^*, n \rightsquigarrow e^*, n$ $\Rightarrow h(\sigma n'' f) = n$	(9)
(0)+(7)+(4)+(1) (10)	$\Rightarrow h(\sigma, p', J) = v$ $\Rightarrow h(\sigma, p'') = v'$	(10) (11)
(10)	$\Rightarrow h(0, p) = 0$ h(v' f) = v	(11) (12)
(11)+(9)+IH	$\Rightarrow e'' = p'''$	(12) (13)
	h = h'	()
	$h'(\sigma,p''')=v'$	(14)
	orall f otin p'' : f otin p'''	(15)
(5)+(13)+(8)	$\Rightarrow e = p''' \cdot f = p'$	(16)
(14)+(16)	$\Rightarrow h'(\sigma, p') = h(v'.f)$	(17)
(17)+(12) (4)+(5)+(8)	$\Rightarrow h^{*}(\sigma, p) = v$ $\Rightarrow m - r'' f$	(19)
(4)+(3)+(6)	$\Rightarrow p = p \ .j$ $f' \text{ such that } f' \notin p$	(10) (19)
(19)+(18)	$\Rightarrow f' \neq f$	(10) (20)
() ()	$f' \notin p''$	(21)
(21) + (15)	$\Rightarrow f' \notin p'''$	(22)
(22)+(20)+(1)	$16) \Rightarrow f' \notin p'$	(23)
$(19) \rightarrow (23)$	$\Rightarrow \forall f \notin p : f \notin p'$	
case (2) last was ((FIELD)	(3)
(3)	$\Rightarrow p = a.f$	(4)
	$e = h(a)\downarrow_3(f) = v'$	(5)
Lat	h = h'	(6)
Let $(7) + (5)$	$\begin{array}{l} p = v \\ \Rightarrow e - n' \end{array}$	(I)
(7) (7)	$ \Rightarrow b(\sigma, p') = v' $	(8)
(5) + (4)	$\Rightarrow h(\sigma, p) = v'$	(0)
(8) + (9) + (1)	$\Rightarrow h(\sigma, p') = v$	(10)
(10)+(6)	$\Rightarrow h'(\sigma, p') = v$	× /
(7)	$\Rightarrow \forall f \notin p : f \notin p'$	

163

Let	$egin{array}{l} h, \sigma dash_{gb} e: l \ \sigma dash e, h \leadsto e', h' \ dash h \end{array}$	
case (1) last was (UN (4)	$ \begin{array}{l} \mathbf{v}) \\ \Rightarrow l = u \\ h, \sigma \vdash e : u \ _ \end{array} $	(2 ({ ()
(6)+(3)+(2)+Lemma (5)+(8)+(7)+(UNIV $)$	$\begin{array}{rl} & u \neq \texttt{any} \\ 3.5.6 & \Rightarrow & h', \sigma \vdash e' : u \\ & \Rightarrow & h', \sigma \vdash_{gb} e' : l \end{array}$	(* (8
case (1) last was (VA)		(4
(4)	$\Rightarrow e = l = p$	
(5)+(6)+(2)	$p \in \{\mathbf{X}, \mathtt{TRIS}\}$ $\Rightarrow e' = \sigma(n)$	
(7)	$\Rightarrow e' = b'(p)$ $\Rightarrow e' = h'(\sigma, p)$	()
(8) + (VAL)	$\Rightarrow h', \sigma \vdash_{ab} e' : p$	
(9)+(5)	$\Rightarrow h', \sigma \vdash_{gb}^{\circ} e' : l$	
(4)	$\Rightarrow e = p.f$ $l = p'.f$ $h, \sigma \vdash_{gb} p : p'$ (Curv)	((((0)
(8)+(5)	$\Rightarrow \sigma \vdash n, h \rightsquigarrow n'', h'$	(8)
(0) + (0) (4) + (7)	$\Rightarrow \{p'\} \# F$	(10)
(7) + (9) + (3) + IH	$\Rightarrow \tilde{h'}, \sigma \vdash_{gb} p'' : p'$	(11)
(11)+(FIELD)	$\Rightarrow h', \sigma \vdash_{gb} p''.f : p'.f$	(12)
(12) + (10 + (6))	$\Rightarrow h', \sigma \vdash_{gb} e' : l$	
case (2) last was	(FIELD)	(8)
(8) + (5)	$\Rightarrow p = a$	(9)
	$e' = h(a)\downarrow_3(f)$ b' = b	(10)
$(7)+(9)+(V_{AL})$	$ \begin{array}{l} n = n \\ \Rightarrow h(\sigma \ n') = a \end{array} $	(11) (19)
(12) + (10)	$\Rightarrow h(\sigma, p', f) = e'$	(12)
(13) + (6)	$\Rightarrow h(\sigma, l) = e'$	(14)
(14) + (VAL)	$\Rightarrow h, \sigma \vdash_{gb} e' : l$	(15)
(15) + (11)	$\Rightarrow h', \sigma \vdash_{gb} e' : l$	
		Ē
case (1) last was (VA	.)	
(4)	$\Rightarrow e = v$	(
(5)+(2)	\Rightarrow contradiction (values ca	nnot reduce)

 $\begin{array}{c} h(\sigma,p) = v \\ \sigma' \vdash e, h \rightsquigarrow e', h' \\ _, h, \sigma' \vdash e : F \\ \{p\} \# F \end{array} \end{array} \right\} \Longrightarrow h'(\sigma,p) = v$

 Let

$$\begin{aligned} h(\sigma, p) &= v \tag{1} \\ \sigma' \vdash e, h \rightsquigarrow e', h' \tag{2}$$

$$\vdash e, h \rightsquigarrow e', h' \tag{2}$$

$$h \sigma' \vdash e \colon F \tag{3}$$

$$[p] \#F$$
 (4)

$$\begin{array}{rcl} (2)+(3)+\text{Lemma A32} & \Rightarrow & \forall a, f \notin F: h(a) \downarrow_3(f) = h'(a) \downarrow_3(f) \\ (5)+(4)+(1) & \Rightarrow & h'(\sigma,p) = v \end{array}$$

$$\left. \begin{array}{l} h, \sigma \vdash_{gb} e : l \\ \sigma' \vdash e', h \rightsquigarrow _, h' \\ _, h, \sigma \vdash e' : F' \\ \{l\} \# F' \end{array} \right\} \Longrightarrow h', \sigma \vdash_{gb} e : l$$

Let		$\begin{array}{l}h, \sigma \vdash_{gb} e: l\\\sigma' \vdash e' \ h \leadsto h'\end{array}$	(1) (2)
		$b \sigma \vdash e' \cdot F'$	(3)
		(l) = (l) + (l)	(4)
case (1) last was (UNIV)			(5)
(5)	\Rightarrow	l = u	(6)
		u eq any	(7)
		$h, \sigma \vdash e: u$	(8)
(8)+(2)+Lemma 3.5.4	\Rightarrow	$h', \sigma \vdash e : u$	(9)
(6)+(7)+(9)+(UNIV)	\Rightarrow	$h', \sigma \vdash_{gb} e: \overline{l}$	(-,
case (1) last was (VAL)			(5)
(5)	\Rightarrow	e = v	(6
× /		$h(\sigma, p) = v$	(7
		l = p	(8
(7)+(2)+(3)+(8)+(4)+Lemma 3.6.3	\Rightarrow	$h'(\sigma, p) = v$	(9
(6)+(9)+(8)+(VAL)	\Rightarrow	$h', \sigma \vdash_{gb} e: l$	(
case (1) last was (VAR)			(5
(5)	\Rightarrow	e = l = p	(6
		$p \in \{\mathtt{x}, \mathtt{this}\}$	(7
$(6)+(7)+({ m VAR})$	\Rightarrow	$h', \sigma \vdash_{gb} e: l$	× .
case (1) last was (FIELD)			(5
(5)	\Rightarrow	e = p.f	(6
× /		l = p'.f	(7)
		$h, \sigma \vdash_{ab} p : p'$	(8
(4)+(7)	\Rightarrow	$\{p'\} \# F$	(9
(8) + (2) + (3) + (9) + IH	\Rightarrow	$h', \sigma \vdash_{ab} p : p'$	(10)
		$y_{0}r \cdot r$	(+ 0)

Lemma 3.6.7	Virgin guard	are preserved	over the	execution of	f other e	xpressions.
-------------	--------------	---------------	----------	--------------	-----------	-------------

Let		$h, \sigma \vdash_{ab} e : l$	(
200		Virgin(e)	(
		$_\vdash_, h \rightsquigarrow _, h'$	(
case (1) last was (UNIV)		(
(4)	\Rightarrow	$h, \sigma \vdash e : u$	(
		u eq any	(
		l = u	(
(5)+(3)+Lemma 3.5.4	\Rightarrow	$h', \sigma \vdash e: u$ _	(
(8)+(6)+(7)+(UNIV)	\Rightarrow	$h', \sigma \vdash_{gb} e: l$	
case (1) last was (VAR)			(
(4)	\Rightarrow	e = p	(
		l = p	
		$p \in \{ \mathtt{x}, \mathtt{this} \}$	
(5)+(6)+(7)+(VAR)	\Rightarrow	$h', \sigma \vdash_{gb} e: l$	
case (1) last was (FIELI))		
(4)	\Rightarrow	e = p.f	
		l = p'.f	
		$h, \sigma \vdash_{gb} p : p'$	
$(2)\!+\!(5)$	\Rightarrow	Virgin(p)	
$(7)+(8)+(3)+{ m IH}$	\Rightarrow	$h', \sigma \vdash_{gb} p : p'$	(
(5)+(6)+(9)+(FIELD)	\Rightarrow	$h', \sigma \vdash_{gb} e: l$	
case (1) last was (VAL)			(
(4)	\Rightarrow	e = v	
(5)	\Rightarrow	$\neg Virgin(e)$ which contradicts (2)	

Lemma 3.6.8 Types of virgin expressions are preserved over the execution of other expressions. ``

$$\left. \begin{array}{c} \mathbb{L}, h, \sigma \vdash e : F \\ - \vdash -, h \rightsquigarrow -, h' \\ Virgin(e) \end{array} \right\} \Longrightarrow \mathbb{L}, h', \sigma \vdash e : F$$

 Let

 $\begin{array}{c} \mathbb{L}, h, \sigma \vdash e : F \\ _\vdash _, h \rightsquigarrow _, h' \\ Virgin(e) \end{array}$ (1)(2)(3)

case (1) last was (VAR) (ADDR) (THIS) (NU	JLL) (N	IEW)	(4)
(4)	\Rightarrow	$e \in \{\texttt{x}, a, \texttt{this}, \texttt{null}, \texttt{new}\ t\}$	(5)
		$L = \emptyset$	(6)
(5)+(6)+(7)+(VAR)(ADDR)		$\Gamma = \psi$	(1)
(THIS)(NULL)(NEW)	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e : F$	
case (1) last was (CAST)			(4)
(4)	\Rightarrow	e = (t)e'	(5)
		$\mathbb{L}, h, \sigma \vdash e' : F$	(6)
(3)+(5)	\Rightarrow	Virgin(e')	(7)
$(6)\!+\!(2)\!+\!(7)\!+\!\mathrm{IH}$	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e' : F$	(8)
$(8)+(4)+({ m Cast})$	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e : F$	

case (1) last was (FIELD)			(4)
(4)	\Rightarrow	e = e'.f	(5)
		$\mathbb{L}, h, \sigma \vdash e' : F$	(6)
		$h, \sigma \vdash_{eh} e' : l$	(7)
		$l \in \mathbb{T}$	(8)
(5) + (2)	_	$U \subseteq \mathbb{Z}$	(0)
(3)+(3)	\rightarrow	v iigin(e)	(9)
$(0)+(2)+(9)+1\Pi$	\Rightarrow	$\mathbb{L}, n, \sigma \vdash e : F$	(10)
(7)+(2)+(3)+Lemma 3.6.7	\Rightarrow	$h', \sigma \vdash_{gb} e' : l$	(11)
$(5)+(10+(11)+(8)+({ m Field}))$	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e : F$	
case (1) last was (Assign) Similar to (Field)			(4)
case (1) last was (SUB)			(4)
(4)	\Rightarrow	$\mathbb{L}', h, \sigma \vdash e : F$	(5)
× /		$\mathbb{L}' \subset \mathbb{L}$	(6)
		$\overline{F' \subset F}$	(7)
		$\blacksquare \#F$	(1)
		$ \exists \# T \\ \forall m \in \mathbb{T} \cdot \mathbb{T} h \in \pi \vdash m \cdot $	(0)
(5) + (2) + (2) + III		$\forall p \in \mathbb{L} : \mathbb{L}, n, o \vdash p : _$	(9)
(0) + (2) + (3) + 111	\Rightarrow	$\mathbb{L}, n, \sigma \vdash e : F'$	(10)
(9)+(2)+Lemma A25	\Rightarrow	$\forall \in \mathbb{L} : \mathbb{L}, h' . \sigma \vdash p : _$	(11)
(10)+(6)+(7)+(8)+(11)+(SUB)	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e : F'$	
case (1) last was (SYNC) (4)	_	o — ama o'' o'''	(4)
(4)	\rightarrow	$e = \operatorname{Sync}_{e'} e e$	(0) (6)
		$n, \sigma \vdash_{gb} e : i$	(0)
		$h, \sigma \vdash_{gb} e'' : l$	(7)
		$\mathbb{L}, h, \sigma \vdash e' : F$	(8
		$\mathbb{L}, h, \sigma \vdash e'' : F$	(9)
		$\mathbb{L} \cup \{l\}, h, \sigma \vdash e''' : F$	(10)
(5)+(3)	\Rightarrow	Virgin(e')	(11)
	\Rightarrow	Virgin(e'')	(12)
	\Rightarrow	Virain(e''')	(13
(6)+(2)+(11)+Lemma 367	\Rightarrow	$h' \sigma \vdash \mu e' \cdot l$	(14
(0) + (2) + (12) + Lemma 3.6.7		$h', \sigma \vdash e'' : l$	(15
(1)+(2)+(12)+Lemma 5.0.7	\rightarrow	$\pi, 0 + gb e \cdot t$	(10)
$(8)+(2)+(11)+\underline{IH}$	\Rightarrow	$\mathbb{L}, n^{\circ}, \sigma \vdash e^{\circ} : F$	(10
$(9)+(2)+(12)+\underline{IH}$	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e'' : F$	(17
$(10)+(2)+(13)+\underline{\mathrm{IH}}$	\Rightarrow	$\mathbb{L} \cup \{l\}, h', \sigma \vdash e''' : F$	(18)
(5)+(14)+(15)+(16)+(17)+(18)+(Sync)	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e : F$	
case (1) last was (SPAWN)		,	(4
(4)	\Rightarrow	$e = ext{spawn} \ e'$	(5
		$L = \emptyset$	(6)
		$F = \emptyset$	(7)
		$\emptyset, h, \sigma \vdash e'$:	(8
(3)+(5)	\Rightarrow	Virgin(e')	(9
(8) + (2) + (9) + IH	\Rightarrow	$\emptyset, h', \sigma \vdash e'$:	(10)
(5) + (6) + (7) + (10) + (SPAWN)	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e: \overline{F}$	χ - ,
case (1) last was (CALL)			(4)
(4)	\Rightarrow	e = e'.m(e'')	(5)
		$\mathbb{L}, h, \sigma \vdash e' : F$	(6
		$\mathbb{L}, h, \sigma \vdash e'' : F$	(7
		$h \sigma \vdash e' \cdot u c$	(8)
		$\mathcal{T}\mathcal{H}(c,m) \vdash \subset F$	(0) (0)
		$\mathbb{L}_{JJ}(\mathbb{C},\mathbb{M})\downarrow_{2} \subseteq \mathbb{L}'$ $\mathbb{L}_{\mathcal{C}} \subset \mathbb{C} \mathbb{M}(\mathbb{C},\mathbb{M})\downarrow_{2}$	(9)
		$\mathbb{L} \in \mathcal{L}\mathcal{J}(\mathcal{C}, \mathcal{M}) \downarrow_{1}$	(10)
		$(h, \sigma, u, e, e') \gg \mathbb{L}' \subseteq \mathbb{L}$	(11)

$(4)\!+\!(3)$	\Rightarrow	Virgin(e')	(12)
		Virgin(e'')	(13)
$(12)+(6)+(2)+{ m IH}$	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e' : F$	(14)
$(13)\!+\!(7)\!+\!(2)\!+\!{ m IH}$	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e'' : F$	(15)
$(8) + (2) + { m Lemma} \ 3.5.4$	\Rightarrow	$h', \sigma \vdash e' : u \ c$	(16)
(11)+(12)+(13)+(2)+Lemma A18	\Rightarrow	$(h', \sigma, u, e, e') \bowtie \mathbb{L}' \subseteq \mathbb{L}$	(17)
(5)+(14)+(15)+(16)+(9)+(10)+(17)+(CALL)	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e : F$	

$\begin{array}{l} h(\sigma,p) = v \\ \sigma' \vdash e,h \rightsquigarrow _,h' \\ \emptyset,h,\sigma' \vdash e: _ \\ \mathbb{L},h,\sigma \vdash p: _ \\ \{h(a) \downarrow_1 h,\sigma \vdash_{gb} a: l,l \in \mathbb{L} \} \end{array}$	$\left. \begin{array}{l} \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \\$	
Let	$\begin{array}{l} h(\sigma,p) = v \\ \sigma' \vdash e, h \rightsquigarrow _, h' \\ \emptyset, h, \sigma' \vdash e : _ \\ \mathbb{L}, h, \sigma \vdash p : _ \\ \{h(a)\downarrow_1 h, \sigma \vdash_{gb} a : l, l \in \mathbb{L}\} \ \cap \ \{w Locked(e,w)\} = \emptyset \end{array}$	(1) (2) (3) (4) (5)
$\begin{array}{c} \text{case (4) last was (FIELD)} \\ (6) \qquad \qquad \Rightarrow \end{array}$	$p = p'.f$ $h, \sigma \vdash_{gb} p': l$ $l \in \mathbb{L}$	$ \begin{array}{c} \hline (6) \\ (7) \\ (8) \\ (9) \\ (10) \end{array} $
$(7)+(1)$ \Rightarrow	$ \begin{array}{ll} \mathbf{L}, h, \sigma \vdash p: \\ h(\sigma, p') = v' \ (11) \\ v = \begin{cases} \operatorname{null} & \text{if } v' = \operatorname{null} \\ (h(v')\downarrow_3(f) & \text{otherwise} \end{cases} $	(10)
$\begin{array}{ll} (11)+(2)+(3)+\\ (10)+(5)+\underline{\mathrm{IH}}\\ (11)+(8)+\mathrm{Lemma} \ 3.6.4 \end{array} \Rightarrow$	$ \begin{array}{l} h'(\sigma,p') = v' \\ h, \sigma \vdash_{gb} v' : l \end{array} $	(13) (14)
$\begin{array}{c} \hline \text{case v'=null} \\ (15)+(12) \Rightarrow v=\text{null} \\ (15)+(13) \Rightarrow h'(\sigma,p')= \\ (17)+(7) \Rightarrow h'(\sigma,p)= \\ (18)+(16) \Rightarrow h'(\sigma,p)= \end{array}$	null v	(15) (16) (17) (18)

$\begin{array}{lll} \text{case v'=a} & \\ (15)+(14) & \Rightarrow & h, \sigma \vdash_{gb} a:l \\ (16)+(9)+(5) & \Rightarrow & \neg Locked(e,h(a)\downarrow_1) \\ \text{Let} & \sigma' \vdash e, h \rightsquigarrow \beta, \end{array}$	$(14) \\ (16) \\ (17) \\ (18)$
$\begin{array}{ll} \mbox{case } \beta = \tau \\ (18) + (19) & \Rightarrow \forall a, f : a \in dom(h) \Rightarrow h'(a) \downarrow_3(f) = h(a) \downarrow_3(f) \\ (15) + (12) + (20) & \Rightarrow h'(v') \downarrow_3(f) = v \\ (7) + (21) + (13) + (15) & \Rightarrow h'(\sigma, p) = v \end{array}$	(19) (20) (21)
$\operatorname{case} \beta = a'$	(19)
$\begin{array}{rcl} (18)+(19)+(3)+\\ \text{Lemma } 3.6.14 & \Rightarrow & Locked(e,h(a')\downarrow_1)\\ (20+(17) & \Rightarrow & a \neq a'\\ (21) & \Rightarrow & h'(a)\downarrow_3(f) = h(a)\downarrow_3(f)\\ (22)+(15)+(12) & \Rightarrow & h'(a)\downarrow_3(f) = v\\ (7)+(13)+(15)+(23) & \Rightarrow & h'(\sigma,p) = v \end{array}$	(20) (21) (22) (23)
	•
	·
case (4) last was (ADDR)(6) $\Rightarrow p = a$ (7) $\Rightarrow h'(\sigma, p) = a$ (7)+(1) $\Rightarrow v = a$ (8)+(9) $\Rightarrow h'(\sigma, p) = v$	(6) (7) (8) (9)
case (4) last was (THIS)	(6) (7)
(6) $\Rightarrow p = \text{this}$ (7) $\Rightarrow h'(\sigma, p) = \sigma(\text{this})$	(7)
$\begin{array}{ll} (7)+(1) & \Rightarrow & v = \sigma(\texttt{this}) \\ (8)+(9) & \Rightarrow & h'(\sigma,p) = v \end{array}$	(9)
case (4) last was (VAR) Similar to (THIS)	(6)
case (4) last was (SUB)	(6)
$(6) \qquad \Rightarrow \mathbb{L}', h, \sigma \vdash p: _$	(7)
$\begin{array}{rcl} (8)+(5) & \Rightarrow & \{h(a)\downarrow_1 h, \sigma \vdash_{gb} a: l, l \in \mathbb{L}'\} \ \# \ \{w Locked(e, w)\} \\ (1)+(2)+(3)+(7)+(9)+\underline{\mathrm{IH}} & \Rightarrow & h'(\sigma, p) = v \end{array}$	(9)
case (4) last was (NULL) Similar to (ADDR)	(6)

Lemma 3.6.10 Guards are preserved over the execution of other expressions when locks do not collide.

$h, \sigma \vdash_{gb} e: l$	
$\sigma' \vdash e', h \rightsquigarrow _, h'$	
$\emptyset, h, \sigma' \vdash e':$ _	
$w = \{h(a)\downarrow_1 h, \sigma \vdash_{qb} a : l, l \in \mathbb{L}\}$	$\Rightarrow \Rightarrow n, \sigma \vdash_{gb} e: l$
$w \cap \{w Locked(e', w)\} = \emptyset$	
$l \in \in Path \Rightarrow \mathbb{L}, h, \sigma \vdash l: _$	
,	

Let

$$h, \sigma \vdash_{gb} e : l \tag{1}$$

$$\begin{array}{l}
\sigma' \vdash e', n \rightsquigarrow _, n \\
\emptyset, h, \sigma' \vdash e': _
\end{array}$$
(2)
(3)

$ \begin{array}{c} W \cap \{w Locked(e',w)\} = \emptyset \\ l \in \in Path \Rightarrow \mathbb{L}, h, \sigma \vdash l : _ \\ (f) \\ \hline case \ l = u \\ \hline case \ (l) \ last \ was \ (UNIV) \\ (8) + (9) \Rightarrow h, \sigma \vdash e : u \\ u \neq any \\ (11) \\ (10) + (2) + Lemma \ 3.5.4 \Rightarrow h', \sigma \vdash e : u \\ (12) + (11) + (UNIV) \Rightarrow h', \sigma \vdash_{gb} e : u \\ (13) + (8) \Rightarrow h', \sigma \vdash_{gb} e : u \\ \hline (13) + (8) \Rightarrow h', \sigma \vdash_{gb} e : l \\ \hline \hline case \ l = p \\ (8) + (6) \Rightarrow \mathbb{L}', h, \sigma \vdash p : _ \\ \hline case \ (l) \ last \ was \ (FIELD) \\ \hline (10) \qquad \Rightarrow p = p', f \\ (11) \\ e = p'', f \\ (12) \\ (13) + (2) + (3) + (4) \\ (13) + (2) + (3) + (4) \\ (13) + (2) + (5) + (11) + Lemma \ A.22 \\ \Rightarrow h, \sigma \vdash_{gb} p'' : p' \\ (15) \\ (15) + (11) + (12) + (FIELD) \\ \hline case \ (l) \ last \ was \ (VAL) \\ \hline \hline (10) \qquad \Rightarrow e = v \\ h(\sigma, \sigma \vdash_{gb} e : l \\ \hline \hline case \ (l) \ last \ was \ (VAL) \\ \hline (10) \qquad \Rightarrow e = v \\ h(\sigma, p) = v \\ (12) + (2) + (3) + (4) + (5) + (11) + Lemma \ 3.6.9 \Rightarrow h'(\sigma, p) = v \\ \hline (13) + (12) + (VAL) \\ \Rightarrow h'(\sigma \vdash_{gb} e : l \\ \hline \end{array} $	$W = \{h(a) \downarrow_1 h, \sigma \vdash_{gb} a : l, l \in \mathbb{L}\}$	}		(4)
$l \in e Path \Rightarrow \mathbb{L}, h, \sigma \vdash l: _ $ (f) case l = u (a) $(b) + (a) = b + (b) + (c) + (c)$	$W \cap \{w Locked(e',w)\} = \emptyset$			(5)
$\begin{array}{c} \csc l = u & (i) \\ \hline \csc (1) \text{ last was (UNIV)} & (9) \\ (8)+(9) & \Rightarrow h, \sigma \vdash e : u & (10) \\ & u \neq \operatorname{any} & (11) \\ (10)+(2)+\text{Lemma } 3.5.4 & \Rightarrow h', \sigma \vdash e : u & (12) \\ (12)+(11)+(UNIV) & \Rightarrow h', \sigma \vdash_{gb} e : u & (13) \\ (13)+(8) & \Rightarrow h', \sigma \vdash_{gb} e : l & & & & & \\ \hline & & & & & & & \\ \hline & & & &$	$l \in \in Path \Rightarrow \mathbb{L}, h, \sigma \vdash l: _$			(6)
$\begin{array}{c} \hline \mbox{case (1) last was (UNIV)} & (9) \\ (8)+(9) & \Rightarrow h, \sigma \vdash e: u & (10) \\ u \neq \mbox{any} & (11) \\ (10)+(2)+\text{Lemma 3.5.4} & \Rightarrow h', \sigma \vdash e: u & (12) \\ (12)+(11)+(UNIV) & \Rightarrow h', \sigma \vdash_{gb} e: u & (13) \\ (13)+(8) & \Rightarrow h', \sigma \vdash_{gb} e: l & & & & & \\ \hline \mbox{case } l = p & (6) \\ (8)+(6) & \Rightarrow \mbox{L}', h, \sigma \vdash p: _ & & & & & & \\ \hline \mbox{case } (1) \mbox{ last was (FIELD)} & & & & & & & \\ \hline \mbox{case } (1) \mbox{ last was (FIELD)} & & & & & & & \\ \hline \mbox{case } (1) \mbox{ last was (FIELD)} & & & & & & & \\ \hline \mbox{case } (1) \mbox{ last was (FIELD)} & & & & & & \\ \hline \mbox{case } (1) \mbox{ last was (FIELD)} & & & & & & \\ \hline \mbox{case } (1) \mbox{ last was (FIELD)} & & & & & \\ \hline \mbox{ (10)} & & & & & & & \\ \hline \mbox{ (11)} & & & & & & \\ \hline \mbox{ (12)} + (11)+\text{Lemma A22} & & & & & \\ \hline \mbox{ (13)} + (2)+(3)+(4) & & & & & \\ \hline \mbox{ (15)} + (11)+(12)+(FIELD) & & & & & & \\ \hline \mbox{ (16)} + (8) & & & & & & \\ \hline \mbox{ (16)} + (8) & & & & & \\ \hline \mbox{ (16)} & & & & & \\ \hline \mbox{ (10)} & & & & & & \\ \hline \mbox{ (10)} & & & & & & \\ \hline \mbox{ (10)} & & & & & & \\ \hline \mbox{ (10)} & & & & & & \\ \hline \mbox{ (10)} & & & & & & \\ \hline \mbox{ (10)} & & & & & & \\ \hline \mbox{ (11)} & & & & & & \\ \hline \mbox{ (12)} + (2)+(3)+(4)+(5)+(11)+\text{Lemma 3.6.9} & \Rightarrow h', \sigma \vdash_{gb} e: l & \\ \hline \mbox{ (13)} & & & & \\ \hline \mbox{ (14)} + (11)+(VAL) & & & & \Rightarrow h', \sigma \vdash_{gb} e: l & \\ \hline \mbox{ (13)} & & & \\ \hline \mbox{ (14)} & & & & \\ \hline \mbox{ (15)} & & & \\ \hline \mbox{ (16)} & & & \\ \hline \mbox{ (16)} & & & \\ \hline \mbox{ (17)} & & & \\ \hline \mbox{ (18)} & & & \\ \hline \mbox{ (10)} & & & \\ \hline \mbox{ (11)} & & & \\ \hline \mbox{ (12)} & & & \\ \hline \mbox{ (12)} & & \\ \hline \mbox{ (13)} & & \\ \hline \mbox{ (12)} & & \\ \hline \mbox{ (13)} & & \\ \hline \mbox{ (14)} & & \\ \hline \mbox{ (15)} & & \\ \hline \mbox{ (16)} & & \\ \hline $	case $l = u$			(8)
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	case (1) last was (UNIV)			(9)
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	$(8)+(9) \qquad \Rightarrow h, \sigma \vdash e: u$			(10)
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	$(0) + (0) \qquad \qquad$			(11)
$\begin{array}{cccc} (12) + (11) + (UNIV) & \Rightarrow & h', \sigma \vdash_{gb} e : u \\ (13) + (8) & \Rightarrow & h', \sigma \vdash_{gb} e : l \end{array} $ $\begin{array}{c} (13) \\ (13) \\ (13) + (8) & \Rightarrow & h', \sigma \vdash_{gb} e : l \end{array}$ $\begin{array}{c} (13) \\ (13) \\ (13) + (8) & \Rightarrow & h', \sigma \vdash_{gb} e : l \end{array}$ $\begin{array}{c} (13) \\ (13) \\ (13) + (8) & \Rightarrow & h', \sigma \vdash_{gb} e : l \end{array}$ $\begin{array}{c} (13) \\ (13) \\ (13) + (11) + (12) + (11) + (12) + (11) + (12) + ($	$(10)+(2)+$ Lemma 3.5.4 \Rightarrow $h'. \sigma \vdash e: u$			(12)
$\begin{array}{cccc} (13) + (8) & \Rightarrow & h', \sigma \vdash_{gb}^{s'} e: l \end{array} \\ \hline & & & & & & & \\ \hline case \ l = p & & & & & \\ (8) + (6) \ \Rightarrow & \mathbb{L}', h, \sigma \vdash p: _ & & & & \\ \hline case \ (1) \ last \ was \ (FIELD) & & & & \\ \hline (10) & & \Rightarrow & p = p'.f & & & \\ (11) & & & e = p''.f & & & \\ (12) & & & h, \sigma \vdash_{gb} p'': p' & & \\ (13) + (2) + (3) + (4) & & & \\ + (5) + (14) + \underline{H} & & \Rightarrow & h', \sigma \vdash_{gb} p'': p' & & \\ (15) + (11) + (12) + (FIELD) & & \Rightarrow & h', \sigma \vdash_{gb} e: p & & \\ (16) + (8) & & \Rightarrow & h', \sigma \vdash_{gb} e: l & & \\ \hline case \ (1) \ last \ was \ (VAL) & & & \\ \hline (10) & & \Rightarrow & e = v & & \\ (11) & & & h(\sigma, p) = v & & \\ (12) + (2) + (3) + (4) + (5) + (11) + Lemma \ 3.6.9 & \Rightarrow & h'(\sigma, p) = v & \\ (13) + (13) + (11) + (VAL) & & \Rightarrow & h', \sigma \vdash_{gb} e: l & \\ \hline \end{array}$	$(12)+(11)+(\text{UNIV}) \Rightarrow h', \sigma \vdash_{ab} e:$	u		(13)
$ \begin{array}{c c c c c c c c c c c c c c c c c c c $	$(13)+(8) \qquad \Rightarrow h', \sigma \vdash_{gb}^{go} e: b$	l		~ /
$\begin{array}{cccc} case \ l = p & (i) \\ (8) + (6) \Rightarrow \ \mathbb{L}', h, \sigma \vdash p : _ & (9) \\ \hline case \ (1) \ last \ was \ (FIELD) & (10) & (10) \\ (10) & \Rightarrow \ p = p'.f & (11) \\ e = p''.f & (12) \\ h, \sigma \vdash_{gb} p'' : p' & (13) \\ (13) + (2) + (3) + (4) & & \\ + (5) + (14) + \underline{IH} & \Rightarrow \ h', \sigma \vdash_{gb} p'' : p' & (15) \\ (15) + (11) + (12) + (FIELD) & \Rightarrow \ h', \sigma \vdash_{gb} e : p & (16) \\ (16) + (8) & \Rightarrow \ h', \sigma \vdash_{gb} e : l & \\ \hline case \ (1) \ last \ was \ (VAL) & (10) \\ (10) & \Rightarrow \ e = v & (11) \\ h(\sigma, p) = v & (12) \\ (12) + (2) + (3) + (4) + (5) + (11) + \text{Lemma } 3.6.9 & \Rightarrow \ h'(\sigma, p) = v & (13) \\ (8) + (13) + (11) + (VAL) & \Rightarrow \ h', \sigma \vdash_{gb} e : l & \\ \hline \end{array}$				·
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	case $l = p$			(8)
case (1) last was (FIELD) (10) (10) $\Rightarrow p = p'.f$ (11) $e = p''.f$ (12) $h, \sigma \vdash_{gb} p'': p'$ (13) (9)+(11)+Lemma A22 $\Rightarrow L', h, \sigma \vdash p'_{-}$ (14) (13)+(2)+(3)+(4) $\Rightarrow h', \sigma \vdash_{gb} p'': p'$ (15) (15)+(14)+ <u>IH</u> $\Rightarrow h', \sigma \vdash_{gb} e: p$ (16) (16)+(8) $\Rightarrow h', \sigma \vdash_{gb} e: p$ (16) (10) $\Rightarrow e = v$ (11) (10) $\Rightarrow e = v$ (11) (12)+(2)+(3)+(4)+(5)+(11)+Lemma 3.6.9 $\Rightarrow h'(\sigma, p) = v$ (12) (12)+(2)+(3)+(4)+(5)+(11)+Lemma 3.6.9 $\Rightarrow h'(\sigma, p) = v$ (13) (8)+(13)+(11)+(VAL) $\Rightarrow h', \sigma \vdash_{gb} e: l$ $\Rightarrow h', \sigma \vdash_{gb} e: l$	$(8)+(6) \Rightarrow \mathbb{L}', h, \sigma \vdash p: _$			(9)
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	case (1) last was (FIELD)			(10)
$(13)' \qquad \qquad$	(10)	\Rightarrow	n = n'. f	(11)
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	()	,	e = p'' f	(12)
$\begin{array}{cccc} (9)+(11)+\text{Lemma A22} & \Rightarrow & \mathbb{L}', h, \sigma \vdash gb \ p'' \vdash p' & (14) \\ (13)+(2)+(3)+(4) & \\ +(5)+(14)+\underline{\text{IH}} & \Rightarrow & h', \sigma \vdash_{gb} p'' \vdash p' & (15) \\ (15)+(11)+(12)+(\text{FIELD}) & \Rightarrow & h', \sigma \vdash_{gb} e \vdash p & (16) \\ (16)+(8) & \Rightarrow & h', \sigma \vdash_{gb} e \vdash l & \\ \hline \\ \hline \\ \hline \\ \hline \\ \hline \\ \hline \\ (10) & & \Rightarrow & e = v & (11) \\ h(\sigma, p) = v & (12) \\ (12)+(2)+(3)+(4)+(5)+(11)+\text{Lemma 3.6.9} & \Rightarrow & h'(\sigma, p) = v & (13) \\ \hline \\ $			$h, \sigma \vdash_{ab} p'' : p'$	(12)
$\begin{array}{cccc} (13)+(2)+(3)+(4) & \Rightarrow & h', \sigma \vdash_{gb} p'':p' & (15) \\ (13)+(14)+\underline{IH} & \Rightarrow & h', \sigma \vdash_{gb} p'':p' & (15) \\ (15)+(11)+(12)+(FIELD) & \Rightarrow & h', \sigma \vdash_{gb} e:p & (16) \\ (16)+(8) & \Rightarrow & h', \sigma \vdash_{gb} e:l & \\ \hline \hline \\ \hline$	(9)+(11)+Lemma A22	\Rightarrow	$\mathbb{L}', h, \sigma \vdash p'$	(14)
$\begin{array}{cccc} +(5)+(14)+\underline{\mathrm{IH}} & \Rightarrow & h', \sigma \vdash_{gb} p'':p' & (15) \\ (15)+(11)+(12)+(\mathrm{FIELD}) & \Rightarrow & h', \sigma \vdash_{gb} e:p & (16) \\ (16)+(8) & \Rightarrow & h', \sigma \vdash_{gb} e:l & \\ \hline \\$	(13)+(2)+(3)+(4)		, , , , , , , , , , , , , , , , , , ,	()
$(15)+(11)+(12)+(FIELD) \Rightarrow h', \sigma \vdash_{gb} e: p $ $(16)+(8) \Rightarrow h', \sigma \vdash_{gb} e: l $ (16) $(10) \qquad (10)$ $(10) \qquad (10)$ $(10) \qquad (10)$ $(10) \qquad (11)$ $h(\sigma, p) = v $ (11) $(12)+(2)+(3)+(4)+(5)+(11)+Lemma 3.6.9 \Rightarrow h'(\sigma, p) = v $ (13) $(8)+(13)+(11)+(VAL) \Rightarrow h', \sigma \vdash_{gb} e: l$	+(5)+(14)+IH	\Rightarrow	$h', \sigma \vdash_{ab} p'' : p'$	(15)
$\begin{array}{cccc} (16)+(8) & \Rightarrow & h', \sigma \vdash_{gb} e:l \\ \hline \\ \hline \\ \hline \\ case (1) last was (VAL) & (10) \\ (10) & \Rightarrow & e=v & (11) \\ & & h(\sigma,p)=v & (12) \\ (12)+(2)+(3)+(4)+(5)+(11)+Lemma 3.6.9 & \Rightarrow & h'(\sigma,p)=v & (13) \\ & & (8)+(13)+(11)+(VAL) & \Rightarrow & h', \sigma \vdash_{gb} e:l \end{array}$	(15)+(11)+(12)+(FIELD)	\Rightarrow	$h', \sigma \vdash_{ab} e : p$	(16)
case (1) last was (VAL) (10) (10) $\Rightarrow e = v$ (11) $h(\sigma, p) = v$ (12) (12)+(2)+(3)+(4)+(5)+(11)+Lemma 3.6.9 $\Rightarrow h'(\sigma, p) = v$ (13) (8)+(13)+(11)+(VAL) $\Rightarrow h', \sigma \vdash_{ab} e : l$ (13)	(16) + (8)	\Rightarrow	$h', \sigma \vdash_{gb}^{ge} e: l$	~ /
$\begin{array}{llllllllllllllllllllllllllllllllllll$	case (1) last was (VAL)			(10)
$\begin{array}{cccc} (10) & \rightarrow & c = v & (11) \\ & & & & h(\sigma, p) = v & (12) \\ (12) + (2) + (3) + (4) + (5) + (11) + \text{Lemma } 3.6.9 & \Rightarrow & h'(\sigma, p) = v & (13) \\ (8) + (13) + (11) + (\text{VAL}) & \Rightarrow & h', \sigma \vdash_{ab} e : l \end{array}$	(10)	\rightarrow	$\rho = \eta$	(11)
$\begin{array}{ccc} n(0,p) = v & (12) \\ (12)+(2)+(3)+(4)+(5)+(11)+\text{Lemma } 3.6.9 & \Rightarrow & h'(\sigma,p) = v \\ (8)+(13)+(11)+(\text{VAL}) & \Rightarrow & h', \sigma \vdash_{ab} e:l \end{array} $	(10)	\rightarrow	c = v $h(\sigma, n) = v$	(11) (12)
$(12) + (2) + (3) + (3) + (6) + (11) + \text{Echinia 5.6.5} \Rightarrow h'(0, p) = 0 $ (13) $(8) + (13) + (11) + (\text{VAL}) \qquad \Rightarrow h', \sigma \vdash_{ab} e : l$	(12)+(2)+(3)+(4)+(5)+(11)+Lemma 3.6.9	\rightarrow	h(0,p) = v $h'(\sigma, p) = v$	(12) (13)
	(12) + (2) + (3) + (1) + (3) + (11) + 10 (8)+(13)+(11)+(VAL)	\Rightarrow	$h'(\sigma,p) = c$ $h', \sigma \vdash_{gb} e : l$	(10)
case (1) last was (VAR) (10)	case (1) last was (VAR)		-	(10)
(10) $\Rightarrow e = l \in \{\mathbf{x} \text{ this}\} $ (11)	(10)	\Rightarrow	$e = l \in \{x \text{ this}\}$	(11)
$\begin{array}{ccc} (10) & \Rightarrow & c = i \in [\mathbf{X}, \text{ only}] \\ (11) + (\text{VAR}) & \Rightarrow & h', \sigma \vdash_{gb} e : l \end{array} $	(11) + (VAR)	\rightarrow	$h', \sigma \vdash_{qb} e: l$	(11)
				·

Lemma 3.6.11 Types are preserved over the execution of other expressions when locks do not collide.

 $\mathbb{L}, h, \sigma \vdash e : F$ $\left\} \Longrightarrow \mathbb{L}, h', \sigma \vdash e : F$ $\begin{array}{c} \sigma' \vdash e', h \rightsquigarrow _, h' \\ \emptyset, h, \sigma' \vdash e' : _ \\ Reachable(e) \end{array}$ $\begin{aligned} \forall w : \neg (Locked(e, w) \land Locked(e', w)) \\ \{h(a) \downarrow_1 | h, \sigma \vdash_{gb} a : l, l \in \mathbb{L} \} & \cap \{w | Locked(e', w)\} = \emptyset \end{aligned}$ $\forall w: \neg(Locked(e,w) \land Locked(e',w))$ $\mathbb{L}, h, \sigma \vdash e : F$ $\sigma' \vdash e', h \leadsto _, h'$ (2) $\emptyset, h, \sigma' \vdash e'$: (3)

$$\{h(a)\downarrow_1 | h, \sigma \vdash_{gb} a : l, l \in \mathbb{L} \} \cap \{w | Locked(e', w)\} = \emptyset$$

$$Reachable(e)$$

$$(4)$$

$$(5)$$

Let

(0)

(1)

(1)+Lemma A2	\Rightarrow	$\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$	(6
case (1) last was (FRAME)			(7)
(7)	\Rightarrow	$e = \texttt{frame} \sigma'' e''$	(8)
		$\forall w. \neg (Locked(e'', w) \land locked(e', w))$	(8b)
		$\sigma'' = (a, v)$	(9)
		$\mathbb{L} = (h, \sigma, u, a, v) \bowtie \mathbb{L}'$	(10)
		$\mathbb{L}', h, \sigma'' \vdash e'' : F$	(11)
		$h, \sigma \vdash a : u$	(12)
Let		$l' \in \mathbb{L}', a' \text{ such that } h, \sigma'' \vdash_{ab} a' : l' (13)$. ,
(13) + (10) + (12) + (9) +	Lemi	ma A24 \Rightarrow $h, \sigma \vdash_{ab} a' : l$	(14)
		$l \in \mathbb{L}$	(15)
(14) + (15) + (4)		$\Rightarrow \neg Locked(e', h(a')\downarrow_1)$	(16)
$(13) \rightarrow (16)$		$\{h(a')\downarrow_1 h, \sigma'' \vdash_{gb} a'' : l', l' \in \mathbb{L}'\} \cap \{w Locked(e', w\} =$	$\emptyset(17)$
(8)+(5)	\Rightarrow	Reachable(e'')	(18)
(8b)+(11)+(2)+(3)			
+(17)+(17)+IH	\Rightarrow	$\mathbb{L}', h', \sigma'' \vdash e'' : F$	(19)
(10+(2)+(3)+(4))			. ,
+(6) + Lemma A23	\Rightarrow	$(h', \sigma, u, a, v) \bowtie \mathbb{L}' = \mathbb{L}$	(20)
(8) + (5)	\Rightarrow	$h', \sigma \vdash a: u$	(21)
(8) + (9) + (20) + (19)		· _	、 /
+(21)+(FRAME)	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e : F$	
case (1) last was (SUD)			(7)
(7)	\rightarrow	\mathbb{I}' h $\sigma \vdash c \cdot F'$	(1)
	\rightarrow	$\mathbb{L}', n, 0 \in \mathbb{C} \cdot \mathbb{L}'$	(0) (0)
		$F' \subseteq F$	(ອ) (10)
		$L \simeq L$ I / # F'	(10)
		$\forall n \in \mathbb{I}' : \mathbb{I}'h \ \sigma \vdash n :$	(11)
(9) + (14)	<u> </u>	$\{h(a) \mid h \sigma \vdash a \circ l l \in \mathbb{I} \} \cap \{w \mid Locked(e' \mid w)\} = \emptyset$	(14) (13)
(0) + (8) + (2) + (3)	-7	$\left[\left[\left$	(10)
+(13)+(5)+IH	\Rightarrow	$\mathbb{L}', h', \sigma \vdash e : F'$	(14)
definition of \mathbb{I}'	\rightarrow	$\forall n \in \mathbb{I}' : Virain(n)$	(15)
(15)+(12)+(2)	\rightarrow	$v_P \subset \mathbb{I}_2$. $v_{orgono}(P)$	(10)
$\pm Lemma 368$	\Rightarrow	$\forall n \in \mathbb{I}' : \mathbb{I}, h' \sigma \vdash n$	(16)
(14)+(9)+(10)+(11)	\rightarrow	$p \subset \pi_{\sigma} \cdot \pi_{\sigma}, n, \sigma + p \cdot \underline{}$	(10)
+(16)+(SUB)	⇒	$\mathbb{L} h' \sigma \vdash e \cdot F$	
	7	,,,	(-)
case (1) last was (CALL) (\overline{a})			(7)
(I)	\Rightarrow	$e = e^{-t} \cdot m(e^{-t})$	(8)
		$\mathbb{L}, n, \sigma \vdash e^{\prime\prime} : F$	(9)
		$\mathbb{L}, n, \sigma \vdash e^{-:} : F$	(10)
		$n, \sigma \vdash e^{-} : u c$	(11)
		$\mathcal{L}_{\mathcal{H}}(c,m)\downarrow_{2}\subseteq F$	(12)
		$\mathbb{L} \in \mathcal{L}_{\mathcal{J}}((, c), m) \downarrow_{1}$	(13)
(0)		$(n, \sigma, u, e^{-}, e^{-}) \bowtie \square \subseteq \square$	(14)
(δ) (15) + (0)	⇒	$\forall w. Locked(e^{-}, w) \Leftrightarrow Locked(e, w)$	(10)
(10)+(0)	\Rightarrow	$\forall w. \neg Lockea(e^{r}, w) \land Lockea(e^{r}, w)$	(10)
(0) (17) + (0)	⇒	$\forall w. Lockea(e^{\dots}, w) \Leftrightarrow Lockea(e, w)$	(17)
(1) + (0)	⇒	$\forall w. \neg Lockea(e^{(e)}, w) \land Lockea(e^{(e)}, w)$	(18)
(0)	\Rightarrow	reachable(e'')	(19)
(16) + (0) + (0) + (0)		<i>Reachable</i> (e)	(20)
(10)+(9)+(2)+(3)		\mathbb{I} $h' = 1 e'' \cdot E$	(01)
$+(4)+(19)+\underline{111}$ (18) + (10) + (2) + (2)	\Rightarrow	$\mathbb{L}, n, o \vdash e : r$	(21)
(10)+(10)+(2)+(3) + $(4)+(20)+111$		$\mathbb{I} b' \in c''' : F$	(99)
$+(4)+(20)+\underline{10}$ (11) + (2) + Lomma 2.5.4	⇒ _	$\mathbf{L}, \mathbf{u}, \mathbf{o} \models \mathbf{e}^{U} : \mathbf{u} = \mathbf{e}^{U}$	(22)
(11)+(2)+Lemma $3.3.4$	\Rightarrow		(∠ə)

$(14)+(2)+(3)+(4) \ +(6)+{ m Lemma} \ { m A23} \ (8)+(21)+(22)$	\Rightarrow	$(h',\sigma,u,e'',e''') \Vdash \mathbb{L}' \subseteq \mathbb{L}$	(24)
+(23)+(12)+(13) +(24)+(m CALL)	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e : F$	
case (1) last was (SYNCH	ED)		(7)
(7)	\rightarrow	$e = \mathtt{synced}_{e_0} w e''$	(8)
		$\mathbb{L}, h, \sigma \vdash e_0 : F$	(9)
		$h(a)\downarrow_1 = w$	(10)
		$h, \sigma \vdash_{gb} a: l$	(11)
		$h, \sigma \vdash_{gb} e_0 : l$	(12)
(-)		$\mathbb{L} \cup \{l\}, h, \sigma \vdash e'' : F$	(13)
(8)	\Rightarrow	Locked(e, w)	(14)
(14)+(0)	\Rightarrow	$\neg Locked(e', w)$	(15)
(5)	\Rightarrow	$V irgin(e_0)$	(16)
(8) + (0) + (15)	,	Reachable(e'')	(17)
(8)+(0)+(15) (10)+(11)+1 omma A15	\Rightarrow	$\forall w : \neg (Lockea(e, n) \land Lockea(e, w))$ $\forall a': h, \sigma \vdash a': l$	(10)
(10)+(11)+Lemma A13 (10)+(4)+(15)	\rightarrow	$ \{ u : h, 0 + gb \ u : l \\ \{ b(a') \mid b \ \sigma \vdash x \ a' \cdot l' \ l' \in \mathbb{I} \cup \{ l \} \} $	(19)
(13)+(4)+(15)	\rightarrow	$ \bigcap \{w Locked(e' w)\} = \emptyset $	(20)
(18)+(13)+(2)+(3)		$\{w \mid Lockcu(c, w)\} = \emptyset$	(20)
+(20)+(17)+IH	\Rightarrow	$\mathbb{L} \cup \{l\}, h', \sigma \vdash e'' : F$	(21)
(13)+Lemma A2	\Rightarrow	$\forall p \in \mathbb{L} \cup \{l\} : \mathbb{L} \cup \{l\}, h, \sigma \vdash p :$	(22)
(22)	\Rightarrow	$l \in Path \Rightarrow \mathbb{L} \cup \{l\}, h, \sigma \vdash l:$	(23)
(11) + (23) + (2) + (3)			· · · · ·
+(20) +Lemma 3.6.10	\Rightarrow	$h', \sigma \vdash_{qb} a: l$	(24)
(12) + (16)			
+(2)+Lemma 3.6.7	\Rightarrow	$h', \sigma \vdash_{gb} e_0: l$	(25)
$(9)\!+\!(16)$			
$+(2)+ { m Lemma}~3.6.8$	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e_0 : F$	(26)
(10)+(2)+Lemma 3.5.3	\Rightarrow	$h'(a)\downarrow_1 = w$	(27)
(8)+(26)+(27)+(24)			
+(23)+(21)+(5YNCED)	\Rightarrow	$\mathbb{L}, n', \sigma \vdash e : F$	
case (1) last was $(SYNC)$			(7)
(7)	\Rightarrow	$e = \mathtt{sync}_{e_0} \; e'' \; e_1$	(8)
		$\mathbb{L}, h, \sigma \vdash e_0 : F$	(9)
		$\mathbb{L}, h, \sigma \vdash e'' : F$	(10)
		$h, \sigma \vdash_{gb} e_0 : l$	(11)
		$h, \sigma \vdash_{gb} e'' : l$	(12)
(\mathbf{r}) (\mathbf{o})	,	$\mathbb{L} \cup \{l\}, h, \sigma \vdash e_1 : F$	(13)
(3)+(8)	\Rightarrow	<i>Reachable</i> (e)	(14) (15)
		$Virgin(e_0)$ $Virgin(e_1)$	(15) (16)
(9)+(2)+(15)		V $irgin(c_1)$	(10)
$\pm Lemma 368$	\Rightarrow	\mathbb{I} , $h' \sigma \vdash e_0 : F$	(17)
(13)+(2)+(16)	,		(1.)
$+ Lemma \ 3.6.8$	\Rightarrow	$\mathbb{L} \cup \{l\}, h', \sigma \vdash e_1 : F$	(18)
(11) + (2) + (15)			· · · ·
+Lemma 3.6.7	\Rightarrow	$h, \sigma \vdash_{ab} e_0 : l$	(19)
Let		$l = p$ for some $p \in Path$	(20)
$(20)\!+\!(11)\!+\!(15)\!+\!\mathrm{Le}$	emma A	$26 \Rightarrow e_0 = p$	(21)
(21)+(9)		\Rightarrow L, $h, \sigma \vdash p : _$	(22)
$(20) \rightarrow (22)$	\Rightarrow	$l \in Path \Rightarrow \mathbb{L}, h, \sigma \vdash l: _$	(23)
(12) + (23) + (2) + (3)			

\Rightarrow	$\forall w \cdot \neg (Locked(e'' \mid w) \land Locked(e' \mid w))$	(96)
	$v\omega$ · $(\text{Deneu(c}, \omega) \land \text{Deneu(c}, \omega))$	(20)
\Rightarrow	$\mathbb{L}, h', \sigma \vdash e'' : F$	(27)
\Rightarrow	$\mathbb{L}, h', \sigma \vdash e : F$	
)		(7)
\Rightarrow	e = e''.f := e'''	(8)
	$\mathbb{L}, h, \sigma \vdash e'' : F$	(9
	$\mathbb{L}, h, \sigma \vdash e''' : F$	(10
	$f \in F$	(11
	$h, \sigma \vdash e'' : \mathbb{L}$	(12)
	$l \in \mathbb{L}$	(13
\Rightarrow	$\forall w : Locked(e'', w) \Leftrightarrow Locked(e', w)$	(14
,	$\forall w : Locked(e'' w) \Leftrightarrow Locked(e' w)$	(15
\rightarrow	$\forall w : \neg (Locked(e''' w) \land Locked(e' w))$	(16
	$\forall w : \neg (Locked(e'', w)) \land Locked(e', w))$	(17
_	Reachable(e'')	(18
\rightarrow	Reachable(e''')	(10
	neuchaore(e)	(13
\rightarrow	$\mathbb{I} h' \sigma \vdash c'' \cdot F$	(20
\rightarrow	$\mathbb{L}, n, o \in \mathcal{E}$. If	(20
,	\mathbb{I} $h' = 1 \cdot t'' \cdot \overline{F}$	(91)
\Rightarrow	$\mathbb{L}, n, o \vdash e$: F	(21
		(00)
\Rightarrow	$n^{*}, \sigma \vdash_{gb} e^{\cdot i} : i$	(22)
\Rightarrow	$\mathbb{L}, h', \sigma dash e : F'$	
		(7)
		(7)
\Rightarrow	$e = {\tt spawn} \; e''$	(8
	$\emptyset, h, \sigma \vdash e'':$ _	(9
	$\mathbb{L} = \emptyset \wedge F = \emptyset$	(9b
\Rightarrow	Virgin(e'')	(10
\Rightarrow	$\emptyset, h', \sigma \vdash e'':$ _	(11
\Rightarrow	$\mathbb{L}, h', \sigma \vdash e : F$	
		(7
\Rightarrow	e = (t)e''	(8
	$\mathbb{L}, h, \sigma \vdash e'' : F$	(9
\Rightarrow	Reachable(e'')	(10
\Rightarrow	$\neg (Locked(e'', w) \land Locked(e', w))$	(11
,	((11
\Rightarrow	$\mathbb{L}, h', \sigma \vdash e'' : F$	(12
7	$\square, h' \sigma \vdash e \cdot F$	(12
\rightarrow		
\rightarrow		
		$ \Rightarrow \mathbb{L}, h', \sigma \vdash e'' : F $ $ \Rightarrow \mathbb{L}, h', \sigma \vdash e : F $ $ \Rightarrow \mathbb{L}, h, \sigma \vdash e'' : F $ $ \mathbb{L}, h, \sigma \vdash e'' : F $ $ h, \sigma \vdash e'' : L $ $ l \in \mathbb{L} $ $ \Rightarrow \forall w : Locked(e'', w) \Leftrightarrow Locked(e', w) $ $ \forall w : -(Locked(e''', w) \land Locked(e', w)) $ $ \forall w : \neg (Locked(e'', w) \land Locked(e', w)) $ $ \Rightarrow \forall w : \neg (Locked(e'', w) \land Locked(e', w)) $ $ \Rightarrow \forall w : \neg (Locked(e''') \Rightarrow Locked(e'', w)) $ $ \Rightarrow \mathbb{L}, h', \sigma \vdash e'' : F $ $ \Rightarrow \mathbb{L}, h', \sigma \vdash e'' : F $ $ \Rightarrow \mathbb{L}, h', \sigma \vdash e'' : F $ $ \Rightarrow \mathbb{L}, h', \sigma \vdash e' : F $ $ \Rightarrow \mathbb{L}, h', \sigma \vdash e'' : F $ $ \Rightarrow \mathbb{L}, h', \sigma \vdash e'' : F $ $ \Rightarrow \mathbb{L}, h', \sigma \vdash e'' : F $ $ \Rightarrow \mathbb{L}, h', \sigma \vdash e'' : F $ $ \Rightarrow \mathbb{L}, h', \sigma \vdash e'' : F $ $ \Rightarrow \mathbb{L}, h', \sigma \vdash e'' : F $ $ \Rightarrow \mathbb{L}, h', \sigma \vdash e'' : F $ $ \Rightarrow \mathbb{L}, h', \sigma \vdash e'' : F $ $ \Rightarrow \mathbb{L}, h', \sigma \vdash e'' : F $ $ \Rightarrow \mathbb{R}eachable(e'') $ $ \Rightarrow \neg (Locked(e'', w) \land Locked(e', w)) $ $ \Rightarrow \mathbb{L}, h', \sigma \vdash e'' : F $

$ \left. \begin{array}{c} \mathbb{L}, h, \sigma \vdash e : F \\ \sigma \vdash e, h \rightsquigarrow e', h' \\ h, \sigma \vdash e : t \\ \vdash h \\ Reachable(e) \end{array} \right\} \Longrightarrow \begin{array}{c} \mathbb{L} \\ \mathbb{R} \\ R \end{array} $	$h', \sigma \vdash e' : F$ eachable (e')	
f(euchable(e))		
Let	$\mathbb{L}, h, \sigma \vdash e : F$	(1)
	$\sigma dash e, h \leadsto e', h'$	(2)
	$h,\sigma \vdash e:t$	(3)
	$\vdash h$	(4)
	Reachable(e)	(5)
case (1) last was (VAR) (NUL)	L) (THIS) (NEW)	(6)
(6) + (2)	$\Rightarrow e' = v$	(7)
(7)+(NULL)+(ADDR)	$\Rightarrow \emptyset, h', \sigma \vdash e' : \emptyset$	(8)
(1) + Lemma A2	$\Rightarrow \mathbb{L}\#F$	(9)
	$\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$	(10)
(10)+(2)+Lemma A25	$\Rightarrow \forall p \in \mathbb{L} : \mathbb{L}, h', \sigma \vdash p : _$	(11)
(8)+(9)+(11)+(SUB)	$\Rightarrow \mathbb{L}, h', \sigma \vdash e' : F'$	
(7)	\Rightarrow Reachable(e')	
case (1) last was $(ADDR)$		(6)
(6)	$\Rightarrow e = a$	(7)
contradicts (2), $addresses$	cannot reduce	
case (1) last was (CAST)		(6)
case (2) last was (CTX)		(7)
(6)+(7)	$\Rightarrow e = (t')e''$	(8)
	$\sigma \vdash e'', h \rightsquigarrow e''', h'$	(9)
	e' = (t')e'''	(10)
$(1)\!+\!(6)\!+\!(8)$	$\Rightarrow \mathbb{L}, h, \sigma \vdash e'' : F$	(11)
$(3)+(8)+{ m Lemma}~{ m A1}$	$\Rightarrow h, \sigma \vdash e'' : t''$	(12)
(8)+(5)	\Rightarrow Reachable(e'')	(13)
(11) + (9) + (12) + (4) + (13) + (1	$+\underline{\mathrm{IH}} \Rightarrow \mathbb{L}, h', \sigma \vdash e''' : F$	(14)
	\Rightarrow Reachable(e''')	(15)
(14)+(10)+(CAST)	$\Rightarrow \mathbb{L}, h', \sigma \vdash e' : F$	
(15)+(10)	\Rightarrow Reachable(e')	
case (2) last was $(CAST)$		(7)

 $\Rightarrow e' = a$

 $\emptyset, h', \sigma \vdash e' : \emptyset$

 $\Rightarrow \quad \mathbb{L}, h', \sigma \vdash e': F$

Reachable(e')

 $\Rightarrow \quad \mathbb{L}\#F, \forall p \in \mathbb{L}: \mathbb{L}, h, \sigma \vdash p: _$

 $\Rightarrow \quad \forall p \in \mathbb{L} : \mathbb{L}, h', \sigma \vdash p : _$

 \Rightarrow

 \Rightarrow

Theorem 3.6.12 The type of a thread is preserved over the execution of that thread.

case (1) last was (FIELD)

(1)+Lemma A2

(10)+(2)+Lemma A25

(9)+(10)+(11)+(SUB)

(6)+(7)

(8)

(8)

(6)

·

(8)

(9)

(10)

(11)

case (2) last was (CTX)			(7)
(6) + (7)	\Rightarrow	$e = e^{\prime\prime}.f$	(8)
		$\sigma \vdash e^{\prime\prime}, h \rightsquigarrow e^{\prime\prime\prime}, h^\prime$	(9)
		e' = e'''.f	(10)
$(3)+(8)+{ m Lemma}~{ m A1}$	\Rightarrow	$h, \sigma \vdash e'' : t'$	(11)
(6) + (8) + (FIELD)	\Rightarrow	$\mathbb{L}, h, \sigma \vdash e'' : F$	(12)
		$h, \sigma \vdash_{gb} e'' : l$	(13)
		$l \in \mathbb{L}^{-1}$	(14)
$(8)\!+\!(15)$	\Rightarrow	Reachable(e'')	(15)
$(12)\!+\!(9)\!+\!(11)\!+\!(4)\!+\!(15)\!+\!\mathrm{\underline{IH}}$	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e''' : F$	(16)
		Reachable(e''')	(17)
$(13)+(9)+(4)+{ m Lemma}3.6.4$		$h', \sigma \vdash_{gb} e'' : l$	(18)
$(16)+(18)+(14)+(10)+({ m Field})$	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e' : F$	
(17)+(10)	\Rightarrow	Reachable(e')	
case (2) last was (FIELD)			(7)
(6)+(7)	\Rightarrow	e = a.f	(8)
		$e' = h(a)\downarrow_3(f) = v$	(9)
(9)+(ADDR) + (NULL)	\Rightarrow	$\emptyset, h', \sigma \vdash e': \emptyset$	(10)
(1) + Lemma A2	\Rightarrow	$\emptyset, h', \sigma \vdash e': \emptyset$	(11)
	\Rightarrow	$\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$	(12)
(12)+(2)+Lemma A25	\Rightarrow	$\forall p \in \mathbb{L} : \mathbb{L}, h', \sigma \vdash p : _$	(13)
$(10)\!+\!(11)\!+\!(13)\!+\!(\mathrm{Sub})$	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e' : F$	
(9)	\Rightarrow	Reachable(e')	
			·
case (1) last was (SUB)			(6)
(6) \Rightarrow	\mathbb{L}'	$\subseteq \mathbb{L}$	(7)
	F'	$\subseteq F$	(8)
	$\mathbb{L}_{\#}$	$\neq F$	(9)
	$\mathbb{L}',$	$h, \sigma \vdash e : F'$	(10)
	$\forall p$	$\in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p :$	(11)
$(10)+(2)+(3)+(4)+(5)+\underline{IH} \Rightarrow$	$\mathbb{L}',$	$h', \sigma \vdash e' : F'$	(12)
	Re	eachable(e')	
$(11)+(2)+$ Lemma A25 \Rightarrow	$\forall p$	$\in \mathbb{L}: \mathbb{L}, h', \sigma \vdash p: _$	(13)
$(12)+(7)+(8)+(9)+(13)+(S_{\rm UB}) \Rightarrow$	$\mathbb{L},$	$h', \sigma \vdash e': F$. ,
case (1) last was (CALL)			(6)

$\begin{array}{cccccccccccccccccccccccccccccccccccc$
$ \begin{array}{c ccccccccccccccccccccccccccccccccccc$
$ \begin{array}{cccccccccccccccccccccccccccccccccccc$
$\begin{array}{cccccccccccccccccccccccccccccccccccc$
$\begin{array}{cccccccccccccccccccccccccccccccccccc$
$ \begin{array}{cccccccccccccccccccccccccccccccccccc$
$\begin{array}{cccccccccccccccccccccccccccccccccccc$
$ \begin{array}{cccccccccccccccccccccccccccccccccccc$
$\begin{array}{cccccccccccccccccccccccccccccccccccc$
$\begin{array}{ccccc} (10)+(13)+(18)+\text{Lemma } 3.6.8 &\Rightarrow & \mathbb{L}, h', \sigma \vdash e_{0} : F & (22) \\ (11)+(13)+(4)+\text{Lemma } 3.5.6 &\Rightarrow h', \sigma \vdash e_{2} : u \ c' & (23) \\ (4)+(16)+(13)+(18)+\text{Lemma } A3 &\Rightarrow (h', \sigma, u, e_{2}, e_{0}) \Vdash \mathbb{L}' \subseteq \mathbb{L} & (24) \\ (20)+(22)+(23)+(14)+(15) &\Rightarrow & \mathbb{L}, h', \sigma \vdash e' : F \\ (21)+(18)+(12) &\Rightarrow & Reachable(e') \\ \hline \hline case (2) \text{ last was (CTX) with } E[\bullet] = a.m(\bullet) & (7) \\ (6)+(7) &\Rightarrow & e = a.m(e_{1}) & (8) \\ \mathbb{L}, h, \sigma \vdash a : F & (9) \\ \mathbb{L}, h, \sigma \vdash e_{1} : F & (10) \\ h, \sigma \vdash a : u \ c' & (11) \\ e' = a.m(e_{2}) & (12) \\ \sigma \vdash e_{1}, h \rightsquigarrow e_{2}, h' & (13) \\ \mathbb{L}ff(c', m)\downarrow_{2} \subseteq F & (14) \\ \mathbb{L}' \in \pi ff(c'\ m)\downarrow_{1} \subseteq F & (15) \end{array}$
$\begin{array}{cccccccccccccccccccccccccccccccccccc$
$\begin{array}{cccc} (11)^{+}(10)^{+}(11)^{+}\text{Lemma A3} &\Rightarrow & (h', \sigma, u, e_2, e_0) \Vdash \mathbb{L}' \subseteq \mathbb{L} \\ (20)^{+}(22)^{+}(23)^{+}(14)^{+}(15) \\ &+(24)^{+}(12)^{+}(\text{CALL}) &\Rightarrow & \mathbb{L}, h', \sigma \vdash e' : F \\ (21)^{+}(18)^{+}(12) &\Rightarrow & Reachable(e') \end{array} $ $\begin{array}{cccc} (2) \text{ last was (CTX) with } E[\bullet] = a.m(\bullet) & (7) \\ \hline (6)^{+}(7) &\Rightarrow & e = a.m(e_1) & (8) \\ \mathbb{L}, h, \sigma \vdash a : F & (9) \\ \mathbb{L}, h, \sigma \vdash e_1 : F & (10) \\ h, \sigma \vdash a : u c' & (11) \\ e' = a.m(e_2) & (12) \\ \sigma \vdash e_1, h \rightsquigarrow e_2, h' & (13) \\ \mathbb{L}ff(c', m)_{\downarrow_2} \subseteq F & (14) \\ \mathbb{L}' \in \pi ff(c' m)_{\downarrow_2} \subseteq F & (14) \end{array}$
$(1) + (2) + (2) + (10) + 12 \text{ minimum} + 10^{-1} + (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0$
$\begin{array}{ccc} (25) + (22) + (25) + (11) + (15) \\ + (24) + (12) + (CALL) & \Rightarrow & \mathbb{L}, h', \sigma \vdash e' : F \\ (21) + (18) + (12) & \Rightarrow & Reachable(e') \end{array}$ $\hline case (2) last was (CTX) with E[\bullet] = a.m(\bullet) & (7) \\ (6) + (7) & \Rightarrow & e = a.m(e_1) & (8) \\ \mathbb{L}, h, \sigma \vdash a : F & (9) \\ \mathbb{L}, h, \sigma \vdash e_1 : F & (10) \\ h, \sigma \vdash a : u c' & (11) \\ e' = a.m(e_2) & (12) \\ \sigma \vdash e_1, h \rightsquigarrow e_2, h' & (13) \\ \mathbb{E}ff(c', m) \downarrow_2 \subseteq F & (14) \\ \mathbb{L}' \in \mathcal{F}ff(c', m) \downarrow_2 \subseteq F & (14) \end{array}$
(21)+(12)+(0112) + (0112) +
$(11)^{+}(10)^{+}(12)^{-} \qquad (12)^{-} \qquad (12)$
case (2) last was (CTX) with $E[\bullet] = a.m(\bullet)$ (7) (6)+(7) $\Rightarrow e = a.m(e_1)$ (8) $\mathbb{L}, h, \sigma \vdash a : F$ (9) $\mathbb{L}, h, \sigma \vdash e_1 : F$ (10) $h, \sigma \vdash a : u c'$ (11) $e' = a.m(e_2)$ (12) $\sigma \vdash e_1, h \rightsquigarrow e_2, h'$ (13) $\mathfrak{Eff}(c', m)\downarrow_2 \subseteq F$ (14) $\mathbb{L}' \in \mathfrak{Fff}(c', m)\downarrow_2$ (15)
$(6)+(7) \qquad \Rightarrow e = a.m(e_1) \qquad (8) \\ \mathbb{L}, h, \sigma \vdash a : F \qquad (9) \\ \mathbb{L}, h, \sigma \vdash e_1 : F \qquad (10) \\ h, \sigma \vdash a : u \ c' \qquad (11) \\ e' = a.m(e_2) \qquad (12) \\ \sigma \vdash e_1, h \rightsquigarrow e_2, h' \qquad (13) \\ \mathfrak{Eff}(c', m) \downarrow_2 \subseteq F \qquad (14) \\ \mathbb{L}' \in \mathfrak{Tff}(c' \ m) \downarrow_2 \qquad (15) \end{cases}$
$h, \sigma \vdash a : u c' $ $e' = a.m(e_2) $ $\sigma \vdash e_1, h \rightsquigarrow e_2, h' $ $\mathfrak{Lff}(c', m) \downarrow_2 \subseteq F $ $\mathbb{I}(12)$ (13) $\mathfrak{Lff}(c', m) \downarrow_2 \subseteq F $ (14) $\mathbb{I}(-c) \mathfrak{Lff}(c', m) \downarrow_2 \qquad (15)$
$e' = a.m(e_2) $ $\sigma \vdash e_1, h \rightsquigarrow e_2, h' $ $\mathfrak{Eff}(c', m) \downarrow_2 \subseteq F $ $\mathbb{I}(12)$ $\mathbb{I}' \in \mathfrak{Eff}(c', m) \downarrow_2 \qquad (15)$
$\sigma \vdash e_1, h \rightsquigarrow e_2, h' \tag{13}$ $\mathcal{E}ff(c', m) \downarrow_2 \subseteq F \tag{14}$ $\mathbb{I}' \in \mathcal{F}ff(c', m) \downarrow_2 \tag{15}$
$\mathcal{E}ff(c',m)\downarrow_2 \subseteq F \tag{14}$ $\mathbb{I}' \in \mathcal{F}ff(c',m)\downarrow_2 \tag{15}$
$\mathbb{I}' \in \mathcal{F}f(c' m) $ (15)
$\square \in \mathcal{D}_{\mathcal{J}}(\mathcal{C},\mathcal{M}) \downarrow_{1} $
$(h, \sigma, u, a, e_1) \bowtie \mathbb{L}' \subseteq \mathbb{L} $ (16)
$(8)+(3) + \text{Lemma A1} \qquad \Rightarrow h, \sigma \vdash e_1 : _ \tag{17}$
$(5)+(8) \qquad \Rightarrow Virgin(e_1) \lor Reachable(e_1) \tag{18}$
$(17) + \text{Lemma A7} \qquad \Rightarrow Reachable(e_1) \tag{19}$
$(10)+(13)+(17)+(4)+(19)+\underline{\mathrm{IH}} \qquad \Rightarrow \mathbb{L}, h', \sigma \vdash e_2 : F \tag{20}$
$\Rightarrow Reachable(e_2) \tag{21}$
(1) \perp Lemma A2 $\rightarrow \mathbb{I} \# F$ (22)
$(1) + \text{Lemma 112} \qquad \qquad \rightarrow \square \# 1 \qquad \qquad (22)$
$ \exists \# T \qquad (22) \\ \Rightarrow \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ $ (23)
$(1) + \text{Lemma A22} \qquad \qquad \Rightarrow \exists \# T \qquad (22) \\ \Rightarrow \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ \qquad (23) \\ \Rightarrow \forall p \in \mathbb{L} : \mathbb{L}, h', \sigma \vdash p : _ \qquad (24)$
(1) + Lemma A2 \rightarrow $\mathbb{L}_{\#}^{+1}$ (22) \Rightarrow $\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p :(23)(23) + (2) + Lemma A25\Rightarrow \forall p \in \mathbb{L} : \mathbb{L}, h', \sigma \vdash p :(24)(ADDR)\Rightarrow \emptyset, h', \sigma \vdash a : \emptyset(25)$
(1) + Lemma A2 \Rightarrow $\mathbb{L}_{\#}^{+1}$ (22) \Rightarrow $\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p :(23)(23) + (2) + Lemma A25\Rightarrow \forall p \in \mathbb{L} : \mathbb{L}, h', \sigma \vdash p :(23)(ADDR)\Rightarrow \emptyset, h', \sigma \vdash a : \emptyset(25)(25) + (SUB) + (22) + (24)\Rightarrow \mathbb{L}, h', \sigma \vdash a : F(26)$
(1) + Lemma A2 \Rightarrow $\mathbb{L}_{\#}^{+1}$ (22) \Rightarrow $\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (23)(23)+(2)+Lemma A25 \Rightarrow $\forall p \in \mathbb{L} : \mathbb{L}, h', \sigma \vdash p : _$ (23)(ADDR) \Rightarrow $\emptyset, h', \sigma \vdash a : \emptyset$ (25)(25)+(SUB) +(22)+(24) \Rightarrow $\mathbb{L}, h', \sigma \vdash a : F$ (26)(11)+(13)+Lemma 3.5.4 \Rightarrow $h', \vdash a : u c'$ (27)
$\begin{array}{cccccccccccccccccccccccccccccccccccc$
$\begin{array}{cccccccccccccccccccccccccccccccccccc$
$(1) + \text{Lemma } 12 \qquad \qquad \Rightarrow \mathbb{L}_{\#} 1 \qquad (22) \\ \Rightarrow \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ \qquad (23) \\ (23) + (2) + \text{Lemma } A25 \qquad \qquad \Rightarrow \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ \qquad (24) \\ (ADDR) \qquad \qquad \Rightarrow \emptyset, h', \sigma \vdash a : \emptyset \qquad (25) \\ (25) + (SUB) + (22) + (24) \qquad \qquad \Rightarrow \mathbb{L}, h', \sigma \vdash a : F \qquad (26) \\ (11) + (13) + \text{Lemma } 3.5.4 \qquad \qquad \Rightarrow h', \vdash a : u \ c' \qquad (27) \\ (4) + (16) + (13) + (10) + (22) + \text{Lemma } A4 \qquad \Rightarrow (h', \sigma, u, a, e_2) \Vdash \mathbb{L}' \subseteq \mathbb{L} \qquad (28) \\ (20) + (26) + (27) + (14) \\ + (15) + (28) + (12) + (CALL) \qquad \qquad \Rightarrow \mathbb{L}, h', \sigma \vdash e' : F$

(2) last was (CALL)			
(2) fast was (CALL) (7) $\rightarrow a = a m(v)$			
$e^{-(1)} \Rightarrow e^{-a.m(v)}$			
$e = \text{frame } \sigma S(e)$	Ъ)		
$\sigma^{r} = (a, v)$			
$e_b = \mathcal{M}\mathcal{B}ody(c,m)$			
$c = n(a)\downarrow_2$			
$n = n^{-1}$			
$\mathbb{L}, h, \sigma \vdash a : F$			
$\mathbb{L}, h, \sigma \vdash v : F$			
$h, \sigma \vdash a : u c'$			
$F'' = \mathcal{E}ff(c',m)\downarrow_2$			
$F'' \subseteq F'$			
$\mathbb{L}' \in \mathcal{E}\!f\!f(c',m)\!\downarrow_1$	_		
$(h, \sigma, u, a, v) \mathrel{\mathbb{B}} \mathbb{L}'$	$\subseteq \mathbb{L}$		
case (3) last was (SUB)			(
1(21)	\Rightarrow	$h, \sigma dash e: t'$	(2
		$t' \leq t$	(2
$(1)+(2)+(22)+(4)+(5)+\underline{IH}$	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e' : F$	
		Reachable(e')	
case (3) last was (CALL)			(
$1 \ (8) + (21)$	\Rightarrow	$h, \sigma \vdash a: u'' \ c''$	(2
		$\mathcal{M}((,c)'',m) = m(u'' \bowtie t''_x)$	(2
		$h, \sigma \vdash v : t''$	(2
(16) + (12) + Lemma A5	\Rightarrow	$c' \ge c$	(2
(22) + (12) + Lemma A5	\Rightarrow	c'' > c	(2
$(\vdash c) + (17) + (19) + (25)$	\Rightarrow	$\mathbb{L}'' \in \mathfrak{E}f\!\!f(c,m) \downarrow_1$	(2
		$\mathbb{L}'' \subset \mathbb{L}'$	(2
		$\mathcal{E}ff(c,m) _{c_2} \subset F'$	(2)
		$\mathcal{M}(c,m) = m(t_r)$	(3
		$\Gamma = (\text{self } c, t_x)$	(3
		$\mathbb{L}'', \Gamma \vdash e_h : \mathcal{F}ff(c, m) _{\mathfrak{S}}$	(3
(1)+Lemma A2	\Rightarrow	\mathbb{L}_{+}^{+}	(3
	/	$\forall n \in \mathbb{I} : \mathbb{I}, \ h \ \sigma \vdash n$	(3
(34)+(2)+Lemma A25	\rightarrow	$\forall p \in \mathbb{L}' : \mathbb{L}', h, \sigma \vdash p : _$	(3
$(\vdash c') + (19) + (17)$	\rightarrow	MBody(c' m)	(3
$(36) \perp Lemma 3.6.1$	$\overline{\rightarrow}$	$\mathbb{I}' \# F'$	(3
(SUB) + (37) + (32)	\rightarrow	$\mathbb{L}' \ \Gamma \vdash e_1 \cdot F'$	(3
(12) + (10) + (A ddr)	\rightarrow	$h \sigma' \Gamma a$ self c	(3
(12) + (10) + (100)		$h, \sigma' \vdash v : u'' \mapsto t''$	(4
(22) + (24) + 100000000000000000000000000000000000	\rightarrow	$u'' \gg t'' < t$	(4
(30) + (1 c) + (20) + (25) (40) + (41) + (Sup)		$\begin{array}{ccc} u & \triangleright & \iota_x \geq \iota_x \\ h & \sigma' \vdash v : t \end{array}$	(4
(40) + (41) + (50B) (42) + (21) + (20) + (28)	\rightarrow	$n, o + v \cdot \iota_x$	(4
(42)+(31)+(39)+(38)	,	$\mathbb{I}' = -I + C(c) + E'$	(1
+(10)+Lemma 3.0.2 (16) + (10) + (42) + (EDAME)	\Rightarrow	$\mathbb{L}, h, 0 \vdash S(e_b) : F$ $\mathbb{L}''' = \int f(e_b) \cdot F'$	(4
(10)+(10)+(43)+(FRAME)	\Rightarrow	$\mathbb{L}^{n}, n, \sigma \vdash \text{irame } \sigma \ S(e_b) : F$	(4
(0) + (44) + (45) + (18)		$\mathbb{L} = (n, \sigma, u, a, v) \Vdash \mathbb{L}$	(4
(3) + (43) + (43) + (10) + $(20) + (23) + (25) + (6mp)$		\mathbb{I} $b' = c' \cdot F$	
+(20)+(30)+(30)+(30B)	\Rightarrow	$\underline{\omega}, n, 0 + e \cdot F$ $Vincin(S(a))$	(4
(11) $(46) \pm \mathbf{I}$ amma $\mathbf{A}7$	⇒	$V \text{ irgin}(\mathcal{S}(e_b))$	(4
1401+Lemma A(\Rightarrow	$neuchaole(S(e_b))$	(4
(47) + (0)		$D_{-} = -1 = 1 + 1 = (-1)$	

·

e(1) last was (Assign)		(6
case (2) last was (CTX) with $E[\bullet] =$	$f \bullet . f := \dots$	(7)
(6) + (7)	$\Rightarrow e = e_1 \cdot f := e_0$	(8)
	$\sigma \vdash e_1, h \rightsquigarrow e_2, h'$	(9)
	$e' = e_2.f := e_0$	(10)
	$\mathbb{L}, h, \sigma \vdash e_1 : \overset{\circ}{F}$	(11)
	$\mathbb{L}, h, \sigma \vdash e_0 : F$	(12)
	$h, \sigma \vdash_{ab} e_1 : l$	(13)
	$l \in \mathbb{L}$	(14)
	$f \in F$	(15)
(9)	$\Rightarrow e_1 \neq a$	(16)
(5)+(8)+(16)	$\Rightarrow Virgin(e_0)$	(17)
(12)+(17)+(9)+Lemma 3.6.8	$\Rightarrow \mathbb{L}, h', \sigma \vdash e_0 : F$	(18)
(3) + (8) + Lemma A1	$\Rightarrow h, \sigma \vdash e_1 : t'$	(19)
(5)+(8)	$\Rightarrow Reachable(e_1)$	(20)
(11)+(9)+(19)+(4)+(20)+IH	$\Rightarrow \mathbb{I}_{4}, h', \sigma \vdash e_{2} : F$	(21)
	$Reachable(e_2)$	(22)
(13)+(4)+(9)+Lemma 3.6.4	$\Rightarrow h' \cdot \sigma \vdash_{ab} e_2 : l$	(23)
(21)+(18)+(23)+(14)+(15)+(A SSIG)	$ (N) \Rightarrow [L, h', \sigma \vdash e_2, f := e_0 : F] $	(24)
(24) + (10) + (20) + (21) + (20) + (100) + (20) +	$\Rightarrow \mathbb{I}_{e} h' \sigma \vdash e' : F$	(= -)
(22) + (17)	$\Rightarrow Beachable(e_2, f := e_0)$	(25)
(25)+(10)	$\Rightarrow Reachable(e')$	()
(2) last map (Cmr) with $E[z]$		(7)
case (2) last was (CTX) with $E[\bullet] =$	$a.f := \bullet$	(7)
(0)+(7)	$\Rightarrow e = a.f := e_1$	(8)
	$\sigma \vdash e_1, n \rightsquigarrow e_2, n$	(9)
	$e^{r} = a.f := e_2$	(10)
	$\mathbb{L}, n, \sigma \vdash a : F$	(11)
	$\mathbb{L}, n, \sigma \vdash e_1 : F$	(12)
	$n, \sigma \vdash_{gb} a: l$	(13)
	$l \in \mathbb{L}$	(14)
(z) + (o)	$J \in \Gamma$	(13) (16)
(3)+(8)	\Rightarrow Reachable(e ₁)	(10)
(3)+(8)+Lemma AI (12)+(0)+(17)+(4)+(16)+111	$\Rightarrow h, \sigma \vdash e_1 : t$	(17)
$(12)+(9)+(17)+(4)+(10)+\underline{IH}$	$\Rightarrow \mathbb{L}, h^{r}, \sigma \vdash e_{2} : F$	(18)
(1) I	$\pi eachable(e_2)$	(19)
(1) +Lemma A2 (A_{DDD})	$\Rightarrow \mathbb{L}\#F$	(20)
(ADDR)	$\Rightarrow \emptyset, n \sigma \vdash a : \emptyset$	(21)
(21)+(20)+(50B) (12) + (0) + (12) + (14) + (20) + L array =	$\Rightarrow \mathbb{L}, n, \sigma \vdash a : F$	(22)
(13)+(9)+(12)+(14)+(20)+Lemma (18)+(22)+(14)+(15)+(10)+($3.0.0 \Rightarrow h, \sigma \vdash_{gb} d: l$	(23)
(18)+(22)+(23)+(14)+(15)+(10)+(19)+(10)+(10)+(10)+(10)+(10)+(10)+(10)+(10	$\begin{array}{llllllllllllllllllllllllllllllllllll$	
case (2) last was $(ASSIGN)$		(7)
$(6)+(7) \qquad \Rightarrow e = c$	u.f := v	(8)
e' =	v	(9)
(1)+Lemma A2 $\Rightarrow \mathbb{L}\#F$	·	(10)
$\forall p \in$	$\mathbb{L}:\mathbb{L},h,\sigma\vdash p:_$	(11)
$(11)+(2)$ +Lemma A25 $\Rightarrow \forall p \in$	$\mathbb{L}:\mathbb{L},h',\sigma\vdash p:_$	(12)
$(9) + (ADDR) \qquad \Rightarrow \qquad \emptyset, h',$	$\sigma \vdash e' : \emptyset$	(13)
$(11) + (10) + (12) + (SUB) \Rightarrow \mathbb{L}, h'$	$,\sigmadasheve e':F$	
$(9) \qquad \Rightarrow Read$	ehable(e')	
		·

case (2) last was (CTX)			(()
6)+(7)	\Rightarrow	e = frame(a, v) e''	(8)
		$\mathbb{L} = (h, \sigma, u, a, v) \bowtie \mathbb{L}'$	(9)
		$h, \sigma \vdash a : u$	(10)
		$\mathbb{L}', h, (a, v) \vdash e'' : F$	(11)
		$(a,v) \vdash e'', h \rightsquigarrow e''', h'$	(12)
		frame $(av) e'''$	(13)
(8)	\Rightarrow	Reachable(e'')	(14)
$(3)+(8)+{ m Lemma}~{ m A11}$	\Rightarrow	$h, (a, v) \vdash e'' : t'$	(15)
$(11) + (12) + (15) + (4) + (14) + \underline{\mathrm{IH}}$	\Rightarrow	$\mathbb{L}, h', (a, v) \vdash e''' : F$	(16)
		Reachable(e''')	(17)
(1) + Lemma A2	\Rightarrow	$\mathbb{L}\#F$	(18)
(9)+(12)+(11)+(18)+Lemma A10	\Rightarrow	$h', \sigma, u, a, v) \mathrel{\blacktriangleright} \mathbb{L}' = \mathbb{L}$	(19)
$(10) + (12) + Lemma \ 3.5.4$	\Rightarrow	$h', \sigma \vdash a: u$ _	(20)
$(16)\!+\!(19)\!+\!(20)\!+\!(13)\!+\!({ m Frame})$	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e' : F$	
(17) + (13)	\Rightarrow	Reachable(e')	
case (2) last was (FRAME2)			(()
6) + (7)	\Rightarrow	e = v	(8)
(1) + Lemma A2	\Rightarrow	$\mathbb{L}\#F$	(9)
		$\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p :$	(10)
(10)+(2)+Lemma A25	\Rightarrow	$\forall p \in \mathbb{L} : \mathbb{L}, h', \sigma \vdash p :$	(11)
(8)+(ADDR) +(NULL)	\Rightarrow	$\emptyset, h', \sigma \vdash e' : \emptyset$	(12)
(12)+(9)+(11)+(SUB)	\Rightarrow	$\mathbb{L}, h, \sigma \vdash e' : F$. /
(8)	\Rightarrow	Reachable(e')	
		· · /	

·

case (1) last was $(SYNC)$			(6)
case (2) last was (CTX)			(()
6)+(7)	\Rightarrow	$e = \operatorname{sync}_{e_0} e_1 e''$	(8)
		$h, \sigma \vdash_{gb} e_0 : l$	(9)
		$h, \sigma \vdash_{gb} e_1 : l$	(10)
		$\mathbb{L}, h, \sigma \vdash e_0 : F$	(11)
		$\mathbb{L}, h, \sigma \vdash e_1 : F$	(12)
		$\mathbb{L} \cup \{l\}, h, \sigma \vdash e'' : F$	(13)
		$\sigma \vdash e_1, h \rightsquigarrow e_2, h'$	(14)
		$e^\prime = \mathtt{sync}_{e_0} \; e_2 \; e^{\prime\prime}$	(15)
(8) + (5)	\Rightarrow	$Virgin(e_0) \land Virgin(e'')$	(16)
		$Reachable(e_1)$	(17)
$(9)+(16)+(14)+{ m Lemma}~3.6.7$	\Rightarrow	$h', \sigma \vdash_{gb} e_0: l$	(18)
$(10)+(4)+(14)+{ m Lemma}~3.6.4$	\Rightarrow	$h', \sigma \vdash_{gb} e_2: l$	(19)
$(11) + (16) + (14) + Lemma \ 3.6.8$	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e_0 : F$	(20)
$(3) + { m Lemma~A1}$	\Rightarrow	$h, \sigma \vdash e: t'$	(21)
$(12)\!+\!(14)\!+\!(21)\!+\!(4)\!+\!(17)\!+\!{ m IH}$	\Rightarrow	$\mathbb{L}, h, \sigma \vdash e_2 : F$	(22)
		$Reachable(e_2)$	(23)
$(13)\!+\!(16)\!+\!(14)\!+\!{ m Lemma3.6.8}$	\Rightarrow	$\mathbb{L} \cup \{l\}, h', \sigma \vdash e'' : F$	(24)
(18) + (19) + (20) + (22) + (24) + (15) + (Sync)	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e' : F$	
$(23)\!+\!(16)\!+\!(15)$	\Rightarrow	Reachable(e')	

			(()
case (2) last was (CTX)			(()
6) + (7)	\Rightarrow	$e = \mathtt{synced}_{e_0} \ w \ e_1$	(8)
		$\sigma \vdash e_1, h \rightsquigarrow e_2, h'$	(9)
		$h, \sigma \vdash_{gb} e_0 : l$	(10)
		$h, \sigma \vdash_{gb} a: l$	(11)
		$h(a)\downarrow_1 = w$	(12)
		$\mathbb{L}, h, \sigma \vdash e_0 : F$	(13)
		$\mathbb{L} \cup \{l\}, h, \sigma \vdash e_1 : F$	(14)
		$e' = \texttt{synced}_{e_0} \ w \ e_2$	(15)
$(3)+(8)+{ m Lemma}~{ m A1}$	\Rightarrow	$h, \sigma \vdash e_1: t'$	(16)
(5)+(8)	\Rightarrow	$Virgin(e_0)$	(17)
		$Reachable(e_1)$	(18)
$(14)\!+\!(9)\!+\!(16)\!+\!(4)\!+\!(18)\!+\!{ m IH}$	\Rightarrow	$\mathbb{L} \cup \{l'\}, h, \sigma \vdash e_2 : F$	(19)
		$Reachable(e_2)$	(20)
$(10) + (13) + (9) + Lemma \ 3.6.7$	\Rightarrow	$h', \sigma \vdash_{gb} e_1 : l$	(21)
(14) + Lemma A2	\Rightarrow	$\mathbb{L} \cup \{l\} \# F$	(22)
(22)	\Rightarrow	$\{l\}\#F$	(23)
$(13)+(17)+(9)+(23)+{ m Lemma}~3.6.6$	\Rightarrow	$h', \sigma \vdash_{gb} a: l$	(24)
$(13)+(17)+(9)+{ m Lemma}~3.6.8$	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e_0 : F$	(25)
(21)+(24)+(25)+(12)+(19)+(15)+(Synced)	\Rightarrow	$\mathbb{L}, h', \sigma \vdash e' : F$	
(20)+(17)+(15)	\Rightarrow	Reachable(e')	

Theorem 3.6.13 Well-typedness of the system is preserved over execution.

 $\vdash h$ $\begin{array}{l} h, \sigma \vdash e_{1...n} \\ \sigma \vdash e_{1...n}, h \rightsquigarrow e'_{1...n}, h' \\ h, \sigma \vdash e_{1...n} : t_{1...n} \end{array}$ $\left\{ \implies \begin{array}{l} h', \sigma \vdash e'_{1\dots m} \\ \forall i \in \{1 \dots m\} : Reachable(e'_i) \end{array} \right.$ $\forall i \in \{1 \dots n\} : Reachable(e_i)$ Let $\vdash h$ (1) $h, \sigma \vdash e_{1...n}$ (2) $\sigma \vdash e_{1...n}, h \rightsquigarrow e'_{1...n}, h'$ (3) $h, \sigma \vdash e_{1...n} : t_{1...n}$ (4) $\forall i \in \{1 \dots n\} : Reachable(e_i)$ (5) $\begin{array}{l} \Rightarrow \quad \forall i \in \{1 \dots n\} : \emptyset, h, \sigma \vdash e_i : _ \\ \quad \forall i, j \in \{1 \dots n\}, w : Locked(e_i, w) \land Locked(e_j, w) \Rightarrow i = j \end{array}$ (2)+(THREADS)(6)(7)(4)+(THREADS) $\Rightarrow \forall i \in \{1 \dots n\} : h, \sigma \vdash e_i : t_i$ (8)case (1) last was (INTERLEAVE) (9)(9) $\Rightarrow i \in \{1 \dots n\}$ (10) $\begin{array}{l} e_i \vdash h, e_i' \rightsquigarrow h', \\ \forall j \neq i : e_j' = e_j \end{array}$ (11)(12)m = n(13)(1)+(10)+(6)+(8)+(5)+(11) $+Lemma \ 3.6.12$ $\emptyset, h, \sigma \vdash e'_i$: (14) \Rightarrow $Reachable(e'_i)$ (14b)Let $j \neq i$ (15) $\Rightarrow \emptyset, h, \sigma \vdash e_j : _$ (16)(15)+(6)(10)+(6) $\Rightarrow \emptyset, h, \sigma \vdash e_i : _$ (17)(16)+(17)+(11)+(5)+(7)+Lemma 3.6.11 $\Rightarrow \emptyset, h', \sigma \vdash e_j : _$ (18) $(15) \rightarrow (18)$ $\Rightarrow \quad \forall j \neq i : \emptyset, h', \sigma \vdash e_j : _$ (19)

(12) + (19)	\Rightarrow	$\forall j \neq i : \emptyset, h', \sigma \vdash e'_i :$	(20)
(13)+(14)+(20)	\Rightarrow	$\forall i \in \{1 \dots m\} : \emptyset, h', \sigma \vdash e_i :$	(21)
(12) + (5)	\Rightarrow	$\forall j \neq i : Reachable(e'_i)$	(22)
(22) + (14b)	\Rightarrow	$\forall i \in \{1 \dots n\} : Reachable(e'_i)$	(23)
(12)	\Rightarrow	$\forall i \neq i : \forall w : Locked(e', w) \Leftrightarrow Locked(e_i, w)$	(24)
(11)+Lemma A12	\Rightarrow	$\forall w : Locked(e; w) \Leftrightarrow Locked(e', w)$	(25)
(24)+(25)+(13)	\rightarrow	$\forall i \in \{1, w\} : Locked(e', w) \Leftrightarrow Locked(e, w)$	(26)
(21) + (20) + (10) Let		$i i w$ such that $Locked(e', w) \land Locked(e', w)$	(20) (27)
$(27) \pm (26) \rightarrow$	Locke	$d(e, w) \wedge Locked(e, w)$	(21)
$(28) + (7) \rightarrow (28) + (7)$	i - i	$u(c_i, w) \land Lochcu(c_j, w)$	(20) (29)
$(20) + (1) \rightarrow (27) \rightarrow (29)$	$\iota = J$	$\forall i \in \{1 \ m\} \cdot (Locked(e', w) \land Locked(e', w)) \Rightarrow i = i$	(23) (30)
(21) + (23) (30) + (21)		$v_i \in \{1, \dots, m\}$ (Locheu $(e_i, w) \land Locheu(e_j, w)\} \Rightarrow i = j$	(00)
(50) + (21)	~	$n, 0 + c_{1} \dots n_{f}$	
(23)+(13)	\Rightarrow	$\forall i \in \{1 \dots m\}$: Reachable(e_i)	
case (1) last was (SP)	AWN)		(9)
(9)	\Rightarrow	$e_i = C[\texttt{spawn} \ e']$	(10)
		$\sigma' = Active(\sigma, C[\bullet])$	(11)
		m = n + 1	(12)
		$e_m' = \texttt{frame} \sigma' e'$	(13)
		$e'_i = C[\texttt{null}]$	(14)
		$\forall j \notin \{i, m\} : e'_i = e_i$	(15)
		h' = h	(16)
(NULL)	\Rightarrow	$\forall \sigma', t': h, \sigma' \vdash \mathtt{null}: t'$	(17)
Lemma A20	\Rightarrow	$\forall \mathbb{L}', \sigma', F' : \mathbb{L}', h, \sigma' \vdash \text{spawn } e' : F' \Rightarrow \mathbb{L}', h, \sigma' \vdash v : F'$	(18)
clearly		spawn $e' \notin Path$	(19)
(5)	\Rightarrow	$Reachable(e_i)$	(20)
(20) + (10)			
+Lemma A28	\Rightarrow	spawn $e' \Rightarrow Virgin(e')$	(21)
(21)	\Rightarrow	$\frac{1}{4} \text{ frame } \in e'$	(22)
(22)	\Rightarrow	$\forall \sigma' : Active(\sigma', e') = \sigma'$	(23)
(6)+(10)	\Rightarrow	$\emptyset, h, \sigma \vdash C[\text{spawn } e']:$	(24)
(24)+(17)+(18)	,	,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,	()
+(19)+(23)			
+Lemma A13+(11)	\Rightarrow	$\mathbb{L}'', h, \sigma' \vdash $ spawn $e' : F''$	(25)
	,	$\mathbb{L}, h, \sigma \vdash C[null]: F$	(26)
(25)+Lemma A31	\Rightarrow	$\emptyset \ h \ \sigma' \vdash e' :$	(23)
WLOG let	,	$(a, v) = \sigma'$ and $b, \sigma \vdash a : v \in \sigma'$	(28)
clearly		$(a, b) = b$ and $(a, b) + b$. $a \in b$	(20)
(27)+(28)+(29)	\rightarrow	$\emptyset = (n, 0, u, u, 0) \lor \psi$ $\emptyset = h \sigma \vdash \text{frame } \sigma' e':$	(20)
(21) + (23) + (23) (30) + (13)	\rightarrow	$\emptyset, h, \sigma \vdash e'$	(31)
(36) + (13) $(26) \pm (14) \pm (31)$	\rightarrow	$\psi, n, 0 + c_m \cdot \underline{}$	(01)
$(20)^+(14)^+(01)^+$ $\pm(12)^\pm(15)^\pm(6)^-$	\rightarrow	$\forall i \in \{1, \dots, m\} : \emptyset \ h \ \sigma \vdash e'$	(32)
(12) + (10) + (0) (32) + (16)		$\forall i \in \{1, \dots, n\} : \emptyset, n, 0 + c_i : _$ $\forall i \in \{1, \dots, m\} : \emptyset, h', \sigma \vdash c' :$	(32)
(15) (15)		$\forall i \in \{1, \dots, n\} : \forall i, n, o \in c_i : _$ $\forall i \notin \{i, m\} w : Locked(e', w) \Leftrightarrow Locked(e, w)$	(30)
(10) (13) + (91)	\rightarrow	$\forall j \not\subseteq \{i, m\}, \& : Deckeu(e_j, \&) \Leftrightarrow Deckeu(e_j, \&) \\ \forall w = I \ ockeu(e_j, \&) \\ \forall w = I \ ockeu$	(34)
(13)+(21) (21)+(16)+(14)		$\forall w. \forall Locked(C[\bullet], w) \leftrightarrow Locked(o, w) \leftrightarrow Locked(o', w)$	(30)
(21)+(10)+(14) (24)+(26)	\rightarrow	$\forall w. Lockeu(C[\bullet], w) \Leftrightarrow Lockeu(e_i, w) \Leftrightarrow Lockeu(e_i, w)$ $\forall i \in \{1, \dots, n\} \text{ an } : Lockeu(e', w) \Leftrightarrow Lockeu(e_i, w)$	(30)
$(34) \pm (30)$	\Rightarrow	$\forall i \in \{1, \dots, n\}, w \in \text{Lockea}(e_i, w) \Leftrightarrow \text{Lockea}(e_i, w)$ $\forall i \ i \in \{1, \dots, n\}, w \in \mathbb{N}$	()()
(01)	\Rightarrow	$v_i, j \in \{1 \dots n\}, w :$ $(I colord(c', w) \land I colord(c', w))$	
		$(Loched(e_j, w) \land Loched(e_j, w))$ $\rightarrow (Loched(e_j, w) \land Loched(e_j, w))$	(90)
(29) + (27)		$\rightarrow (Lockeu(e_i, w) \land Lockeu(e_j, w))$ $\forall i \in [1, \dots, m] \text{ and } i$	(38)
(16)+(0()	\Rightarrow	$v, j \in \{1 \dots n_j, w\}$	(20)
(19) + (20) + (25)		$(Lockeu(e_i, w) \land Lockeu(e_j, w)) \Rightarrow i = j$	(99)
(12)+(39)+(30)	\Rightarrow	$\forall, j \in \{1 \dots m\}, w:$ $(I colord(cl, w)) \land I colord(cl, w)) \land i = i$	(40)
(17) + (7)		$(Lockea(e_i, w) \land Lockea(e_j, w)) \Rightarrow i = j$	(40)
(10)+(0)	\Rightarrow	$\forall j \notin \{i, m\}$: $Keachable(e'_j)$	(41)
(13)+(21)			
--	---------------	--	-------
+ Lemma A7	\Rightarrow	$Reachable(e'_m)$	(42)
clearly		Reachable(v)	(43)
(10)+(43)+(14)			
+(20)+Lemma A28	\Rightarrow	$Reachable(e'_i)$	(44)
(41)+(42)+(44)	\Rightarrow	$\forall i \in \{1 \dots m\} : Reachable(e'_i)$	
(40)+(33)	\Rightarrow	$h'\sigma \vdash e_{1m}$	
case (1) last was (Loc	к)		(9
(9)	\Rightarrow	$i \in \{1 \dots n\}$	(10
· ·		m = n, h = h'	(11
		$e_i = C[\operatorname{sync}_{e''} a \ e''']$	(12
		$e'_i = C[\texttt{synced}_{e''} \ w \ e''']$	(13)
		$\forall j \neq i : e'_j = e_j$	(14)
		$h(a)\downarrow_1 = w$	(15)
		$\forall j \in \{1 \dots n\} : Locked(e_j, w) \Rightarrow i = j$	(16
(10)+(12)+(6)	\Rightarrow	$\emptyset, h, \sigma \vdash C[$ sync $e'' a e''']$:	(17
(15)+Lemma A16	\Rightarrow	$\forall \sigma', t': h, \sigma' \vdash \text{sync } e'' a e''' : t' \Rightarrow$	
		$h, \sigma' \vdash \mathtt{synced}_{e''} \ w \ e''' : t'$	(18
(15)+Lemma A17	\Rightarrow	$\forall \sigma', \mathbb{L}', F' : \mathbb{L}', h, \sigma' \vdash \texttt{sync} \ e'' \ ae''' : F' \Rightarrow$	
		$\mathbb{L}', h, \sigma' \vdash \mathtt{synced}_{e''} \ w \ e''' : F'$	(19
clearly		$sync_{e''} a e''' \notin Path$	(20
(5)+(10)	\Rightarrow	$Reachable(e_i)$	(21
(21)+(12)			
+Lemma A28	\Rightarrow	$Reachable(sync_{e''} a e''')$	(22)
(22)	\Rightarrow	$Virgin(e'') \land Virgin(e''')$	(22b)
(22b)	\Rightarrow	$\forall \sigma' : Active(\sigma', \text{sync } e'' \ ae''')$	(23
(17)+(18)+(19)+(20)			
+(23)+Lemma A13	\Rightarrow	$\emptyset, h, \sigma \vdash C[\texttt{synced}_{e''} \ w \ e''']:$	(24)
(24) + (13)	\Rightarrow	$\emptyset, h, \sigma \vdash e'_i$:	(25)
(11)+(25)+(14)+(6)	\Rightarrow	$\forall i \in \{1 \dots m\} : \emptyset, h', \sigma \vdash e'_i : _$	(26)
Let		$j,k \in \{1 \dots m\}, w' \text{ such that}$	
		$Locked(e'_j,w') \wedge Locked(e'_k,w')$	(27)
case $i \neq i, k \neq i$			(28)
(11)+(28)+(27)+(27)+(27)+(27)+(27)+(27)+(27)+(27	(14)	\Rightarrow Locked $(e_i, w') \land Locked(e_k, w')$	(29)
(29) + (7)	· /	$\Rightarrow j = k$	· · ·
		-	(00)
case $j = i, k \neq i$	(1.4)	T = 1 = 1(-1)	(28)
(11)+(20)+(29)+(29)+(29)+(29)+(29)+(29)+(29)+(29	(14)	$\Rightarrow Lockea(e_k, w)$	(29)
case $w \neq w'$			(30)
$(12)\!+\!(13)$		$\Rightarrow \forall w'' \neq w : Locked(e'_i, w'') \Leftrightarrow Locked(e_i, w'')$	(31)
(31) + (30) + (2)	(28)+($27) \Rightarrow Locked(e_j, w')$	(32)
$(29)\!+\!(32)\!+\!(5)$	7)	$\Rightarrow j = k$ which contradicts (28)	
case $w = w'$			(30)
(29) + (16) + (3)	30)	$\Rightarrow k = i \text{ which contradicts (28)}$	< - /
	·		
case $j \neq i, k = i$ as above			(28)
			(00)
case $i = j = \kappa$ (28)		$\rightarrow i-k$	(28)
(40) Thus		$\neg j = h$ i = k in all cases	(28)
1 1145		$J = \kappa$ III all cases	(20)
			·

$(27) \rightarrow (28)$	\Rightarrow	$\forall j, k \in \{1 \dots m\}, w' :$ $(Locked(e', w') \land Locked(e', w')) \Rightarrow i = k$	(29)
(11)+(14)+(5)	\Rightarrow	$\forall i \neq i : Reachable(e'_{k}) $	(20)
(11) + (14) + (0) (12) + (13) + (21)	\rightarrow	$V_j \neq i$. Reachable(e_j) Reachable(e')	(30)
(12) + (10) + (21) (30+(31)+(10)	\rightarrow	$\forall i \in \{1, \dots, m\}: Reachable(e'_i)$	(01)
(30+(31)+(10)) (29)+(26)	\rightarrow	$h' \sigma \vdash e'$	
(20) + (20)	_	<i>n</i> , <i>o</i> + <i>o</i> _{1<i>m</i>}	
case (1) last was (UNL)	лоск)		(9)
(9)	\Rightarrow	$i \in \{1 \dots n\}$	(10)
		m = n	(11)
		h = h'	(12)
		$e_i = C[\text{synced}_{e''} \ w \ v]$	(13)
		$e'_i = C[v]$	(14)
		$\forall j \neq i : e'_j = e_j$	(15)
(5)+(10)+(13)			<i>,</i> ,
+Lemma A28	\Rightarrow	$Reachable(synced_{e''} w v)$	(16)
(16)	\Rightarrow	Virgin(e'')	(17)
(17)	\Rightarrow	$\forall \sigma' : Active(\sigma', \texttt{synced}_{e''} \ w \ v) = \sigma'$	(18)
(13) + (10) + (6)	\Rightarrow	$\emptyset, h, \sigma \vdash C[\texttt{synced}_{e''} \ w \ v] : _$	(19)
Lemma A20	\Rightarrow	$orall L', \sigma', F':$	<i>,</i> ,
		$\mathbb{L}', h, \sigma' \vdash \texttt{synced}_{e''} \ w \ v: F' \Rightarrow \mathbb{L}', h, \sigma' \vdash v: F'$	(20)
Lemma A19	\Rightarrow	$\forall \sigma', t': h, \sigma' \vdash \texttt{synced}_{e''} \ w \ v: t' \Rightarrow h, \sigma' \vdash v: t'$	(21)
clearly	\Rightarrow	$\mathtt{synced}_{e^{\prime\prime}} w v \notin Path$	(22)
(19)+(21)+(20)+(22)			()
+(18)+Lemma A13	\Rightarrow	$\emptyset, h, \sigma \vdash C[v]:$	(23)
(23)+(14)+(15)			()
+(6)+(11)+(12)	\Rightarrow	$\forall i \in \{1 \dots m\} : \emptyset, h', \sigma \vdash e'_i : _$	(24)
Let		$j,k \in \{1 \dots n\}, w'$	(25)
		such that $Locked(e'_j, w') \wedge Locked(e'_k, w')$	(25)
case $j = k = i$			(26)
(26)	\Rightarrow	j = k	
		•	(0 c)
case $j = i \land \kappa \neq i$			(20)
$(26)\!+\!(25)\!+\!(15)$	\Rightarrow	$Locked(e_k, w')$	(27)
case $w \neq w'$			(28)
(13) + (14)		$\Rightarrow \forall w'' \neq w : Locked(e'_i, w'') \Leftrightarrow Locked(e_i, w'')$	(29)
(29) + (28) +	25)+(26) \Rightarrow Locked (e_i, w')	(30)
(25) + (27) +	30) + (7) $\Rightarrow j = k$ which contradicts (26)	· /
/			(00)
case $w = w$		T = 1 = 1 (T)	(28)
(28)+(13)		$\Rightarrow Locked(e_i, w')$	(29)
(29) + (20)	<u>م</u> ر ا	$\Rightarrow Lockea(e_j, w')$	(30)
(25)+(27)+(3)	o∪)+($j = \kappa \text{ which contradicts (26)}$	
			·
case $j \neq i \land k = i$			(26)
1			
$as \ above$			
case $j \neq i \land k \neq i$			(26)

(25) + (26) + (15)	\Rightarrow	$Locked(e_j, w') \land Locked(e_k, w')$	(27)
$(25)\!+\!(27)\!+\!(7)$	\Rightarrow	j = k	

Conclude that		$\forall j,k \in \{1 \dots n\}, w':$	
		$(Locked(e'_{i}, w') \land Locked(e'_{k}, w')) \Rightarrow j = k$	(26)
(26)+(11)	\Rightarrow	$\forall j,k \in \{1\dots m\}, w':$	
		$(Locked(e'_{i}, w') \land Locked(e'_{k}, w')) \Rightarrow j = k$	(27)
(10)+(5)	\Rightarrow	$Reachable(C[synced_{e''} w v])$	(28)
Clearly		Reachable(v)	(29)
(28) + (29)			
+Lemma A28	\Rightarrow	Reachable(C[v])	(30)
(30)	\Rightarrow	$Reachable(e'_i)$	(31)
(31)+(5)+(15)			
+(10)+(11)	\Rightarrow	$\forall i \in \{1 \dots m\} : Reachable(e'_i)$	
(27)+(24)			
+(THREADS)	\Rightarrow	$h', \sigma dash e'_{1m}$	

Lemma 3.6.14	Objects are	only accessed	while their	owners a	are locked.
--------------	-------------	---------------	-------------	----------	-------------

$ \begin{array}{c} \mathbb{L}, h, \sigma \vdash e : _ \\ \sigma \vdash e, h \stackrel{a}{\leadsto} _, _ \end{array} \right\} \Longrightarrow$	$ (\exists l \in \mathbb{L} : h, \sigma \vdash_{gb} a : l) \lor Locked(e, h(a)\downarrow_1) $	
Let	$\mathbb{L}, h, \sigma \vdash e:$ _	(1)
	$\sigma \vdash e, h \stackrel{a}{\leadsto} _, _$	(2)
case (1) last was (FRAME))	(3)
$(0) \qquad \qquad \Rightarrow \qquad $	$e = \texttt{frame} \sigma' e'$	(4)
	$\sigma' = (a, v)$	(5)
	$\mathbb{L}', h, \sigma \vdash e':$	(6)
	$\mathbb{L} = (h, \sigma, u, a', v) \bowtie \mathbb{L}'$	(7)
	$h, \sigma \vdash a': u$	(8)
$(4)+(2)+(FRAME1) \Rightarrow$	$\sigma' \vdash e', h \stackrel{a}{\rightsquigarrow} _, _$	(9)
$(6)+(9)+\underline{\mathrm{IH}} \qquad \Rightarrow$	$(\exists l' \in \mathbb{L}': h, \sigma' \vdash_{gb} a: l') \ \lor \ Locked(e', h(a){\downarrow_1})$	(10)
case (10) RHS		(11)
(11)+(4)	$\Rightarrow Locked(e, h(a) \downarrow_1)$	
case (10) LHS		(11)
(11) + (7) + (8) + (5) + L	emma A24 $\Rightarrow \exists l \in \mathbb{L} : h, \sigma \vdash_{gb} a : l$	× /
		·

case (1) last was (SY)	NCED		(3)
(3)	\Rightarrow	$e = \mathtt{synced}_{e_0} \ w \ e'$	(4)
		$\sigma' = h, \sigma \vdash_{gb} a' : l$	(5)
		$h(a')\downarrow_1 = w$	(6)
		$\mathbb{L} \cup \{l\}, h, \sigma \vdash e' : _$	(7)
(3)+(2)+(FRAME)	\Rightarrow	$\sigma \vdash e', h \stackrel{a}{\leadsto} ,$	(8)
$(7)+(8)+\underline{\mathrm{IH}}$	\Rightarrow	$\exists l' \in \mathbb{L} \cup \{l\}: h, \sigma \vdash_{gb} a: l'$	(10)

case (10) RHS (11)+(4) \Rightarrow Locked(e, h(a) \downarrow_1)	(11)
case (10) LHS	(11)
$\begin{array}{ll} \operatorname{case} l \neq l' \\ (11) + (10) + (12) \end{array} \Rightarrow \exists l' \in \mathbb{L} : h, \sigma \vdash_{gb} a : l' \end{array}$	(12)
$\begin{array}{llllllllllllllllllllllllllllllllllll$	$(12) \\ (13) \\ (14)$
	· · · · · · · · · · · · · · · · · · ·
$\begin{array}{c} \text{case (1) last was (CAST)} \\ (3) \qquad \Rightarrow e = (t)e' \\ \qquad $	$ \begin{array}{c} (3) \\ (4) \\ (5) \\ (6) \\ (7) \end{array} $
(8) $\exists l \in \mathbb{L} : h, \sigma \vdash_{gb} a : l) \lor Locked(e, h(a) \downarrow_1)$	(8)
case (7) RHS (8)+(4) \Rightarrow Locked(e, h(a) \downarrow_1)	(8)
	·
$\begin{array}{c} \text{case (1) last was (CALL)} \\ (3) \qquad \Rightarrow e = e'.m(e'') \\ \qquad \qquad \mathbb{L}, h, \sigma \vdash e': \\ \qquad \qquad \qquad \mathbb{L}, h, \sigma \vdash e'': \\ \end{array}$	(3) (4) (5) (6)
case (2) last was (CTX) as above	(7)
case (2) last was (CALL) contradicts (2)	(7)
	·
case (1) last was (Assign)	(3)
case (2) last was (CTX) as above	(4)
$egin{array}{llllllllllllllllllllllllllllllllllll$	(4)
	·
case (1) last was (FIELD) as (ASSIGN)	(3)
case (1) last was (NEW) (NULL) (THIS) (VAR) (ADDR) contradicts (2)	(3)
case (1) last was (SYNC) as (CAST)	(3)

case (1) last was ((SUB)	(3)
(3)	$\Rightarrow \mathbb{L}', h, \sigma \vdash e: _$	(4)
	$\mathbb{L}' \subseteq \mathbb{L}$	(5)
$(4)+(2)+\underline{\mathrm{IH}}$	$\Rightarrow (\exists l' \in \mathbb{L}' : h, \sigma \vdash_{gb} a : l') \ \lor \ Locked(e, h, (a) \downarrow_1)$	(6)
case (6) LHS		(7)
$(5)+(7) \Rightarrow$	$l'\in\mathbb{L}$	(8)
$(7)+(8) \Rightarrow$	$\exists l' \in \mathbb{L} : h, \sigma \vdash_{gb} a : l'$	
case (6) RHS		(7)
$(7) \qquad \Rightarrow \qquad \qquad$	$(\exists l \in \mathbb{L} : h, \sigma \vdash_{gb} a : l) \ \lor \ Locked(e, h(a) \downarrow_1)$	
		·

Theorem 3.6.15 Objects are only accessed while their owners are locked by the corresponding thread.

$$\begin{array}{c} h, \sigma \vdash e_{1..n} \\ \sigma \vdash e_{1..n}, h \stackrel{(i,a)}{\leadsto}_{-}, _ \end{array} \end{array} \} \Longrightarrow Locked(e_i, h(a)\downarrow_1) \\ Let \qquad h, \sigma \vdash e_{1..n}, h \stackrel{(i,a)}{\leadsto}_{-}, _ \qquad (1) \\ \hline \sigma \vdash e_{1..n}, h \stackrel{(i,a)}{\leadsto}_{-}, _ \qquad (2) \end{array}$$

(3)	\Rightarrow	$\sigma \vdash e_i, h \stackrel{a}{\rightsquigarrow}$,	
(1)	\Rightarrow	$\forall i \in 1n : \emptyset, \overline{h}, \overline{\sigma} \vdash e_i : _$	
(4)+(5)+Lemma 3.6.14	\Rightarrow	$(\exists l \in \emptyset : h, \sigma \vdash_{gb} a : l) \lor Locked(e_i, h(a) \downarrow_1)$	
clearly		$\nexists l \in \emptyset$	
(6)+(7)	\Rightarrow	Locked(e, h(a)).	

$h, \sigma \vdash e : t$ $e \in \{e' \ f \ ($	(t')e' e' f := e	" e"	f := e' spawn e'	
$e \in \{e, j, k\}$ sync _e e'.m(e	$e''(e')e'', e'''', sync_{e'}e'''), e''.m(e') $, e'''	$e', \operatorname{synced}_{e''} w e', \end{cases} \Longrightarrow h, \sigma \vdash e' : _$	_
	Let		$h, \sigma \vdash e : t e \in \{e'.f, (t')e', \dots\}$	(1) (2)
-	case (1) last w $(2)+(3)$	$\operatorname{vas}\left(\overline{\mathrm{F}} ight)$	TIELD) e = e'.f $h, \sigma \vdash e' : _$	$(3) \\ (4)$
-	case (1) last w (3)	as (A)		$ \begin{array}{c} (3) \\ (4) \\ (5) \\ (6) \end{array} $
	${(2)+(4)}\ {(7)+(5)+(6)}$	$\Rightarrow \Rightarrow$	e' = e'' or $e' = e'''h, \sigma \vdash e':$	(7)
_	$\begin{array}{c} \text{case (1) last w} \\ \text{(3)} \end{array}$	as(C)	$ \begin{array}{l} \text{CALL})\\ e = e''.m(e''')\\ h, \sigma \vdash e'': u \ c \end{array} $	(3) (4) (5)
	(2) + (4)	\Rightarrow	$h, \sigma \vdash e''' : t''$ $e' = e'' \text{ or } e' = e'''$	$(6) \\ (7)$

$(7)+(5)+(6) \Rightarrow $	$h, \sigma \vdash e' : _$	
case (1) last was (CAST)	(3)
$(2)+(3) \Rightarrow$	e = (t')e'	(4)
	$h, \sigma \vdash e':$ _	
case (1) last was (SPAWN)	(3)
$(2)+(3) \qquad \Rightarrow \qquad$	$e = {\tt spawn} \; e'$	(4)
	$h, \sigma \vdash e':$ _	
case (1) last was (Synced)	(3)
$(2)+(3) \qquad \Rightarrow \qquad$	$e = \texttt{synced}_{e^{\prime\prime}} w e^\prime$	(4)
	$h, \sigma \vdash e':$ _	
case (1) last was (Sync)	(3)
$(3) \qquad \qquad \Rightarrow \qquad $	$e = \texttt{sync}_{e'''} \ e'' \ e''''$	(4)
	$h, \sigma \vdash e'' : u \ c$	(5)
	$h, \sigma \vdash e^{\prime\prime\prime}: t^{\prime\prime}$	(6)
$(2)+(4) \qquad \Rightarrow \qquad$	e' = e'' or $e' = e'''$	(7)
$(7)+(5)+(6) \Rightarrow$	$h, \sigma \vdash e':_$	

 $\mathbb{L}, h, \sigma \vdash e : F \Longrightarrow \mathbb{L} \# F, \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p :$ $\mathbb{L}, h, \sigma \vdash e : F$ Let (1)case (1) last was (VAR) (THIS) (NULL) (ADDR) (NEW) (SPAWN) (2)(3)(2) \Rightarrow $\mathbb{L} = \emptyset$ $F = \emptyset$ (4) $\mathbb{L}\#F, \forall p \in \mathbb{L}: \mathbb{L}, h, \sigma \vdash p: _$ (3)+(4) \Rightarrow case (1) last was (CAST) (2) \Rightarrow $\mathbb{L}, h, \sigma \vdash e' : F$ (3)(2)e = (t)e'(4) $\Rightarrow \quad \mathbb{L}\#F, \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ $(3) + \underline{IH}$ case (1) last was (FIELD) (2)(3) $\Rightarrow e = e'.f$ (2) $\mathbb{L}, h, \sigma \vdash e' : F$ (4) $(4) + \underline{IH}$ $\mathbb{L}\#F, \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ \Rightarrow (2)case (1) last was (Assign) $\Rightarrow \quad e = e'.f := e''$ (3)(2) $\mathbb{L}, h, \sigma \vdash e': F$ (4) $(4) + \underline{IH}$ $\Rightarrow \quad \mathbb{L}\#F, \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p :$ case (1) last was (SUB) (2) $\Rightarrow \quad \mathbb{L} \# F, \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (3)(2)(2)case (1) last was (FRAME) $e = \texttt{frame} \ \sigma' \ e'$ (2) \Rightarrow (3) $\mathbb{L}, h, \sigma \vdash e' : F$ (4) $\Rightarrow \quad \mathbb{L}\#F, \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ $(4) + \underline{IH}$ case (1) last was (SYNC) (2)(3) $\Rightarrow e = \operatorname{sync}_{e'''} e' e''$ (2) $\mathbb{L}, h, \sigma \vdash e' : F$ (4) $(4) + \underline{\mathrm{IH}} \quad \Rightarrow \quad \mathbb{L} \# F, \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$

case (1)	last v	vas (Synced)	(2)
(2)	\Rightarrow	$e = \texttt{synced}_{e'} \ w \ e''$	(3)
		$\mathbb{L}, h, \sigma \vdash e': F$	(4)
$(4) + \underline{\mathrm{IH}}$	\Rightarrow	$\mathbb{L}\#F, \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$	
case (1)	last v	vas (CALL)	(2)
(2)	\Rightarrow	e = e' m(e'')	(2) (3)
(2)		$\mathbb{L}, h, \sigma \vdash e' : F$	(4)
$(4) + \underline{IH}$	\Rightarrow	$\mathbb{L} \# F, \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$	· · · · · · · · · · · · · · · · · · ·

$\left. \begin{array}{c} \vdash h \\ (h, \sigma, u, e_1, e) \triangleright l = l' \\ \sigma \vdash e_1, h \rightsquigarrow e_2, h' \\ Virgin(e) \end{array} \right\} \Longrightarrow (h)$	$l', \sigma, u, e_2, e) \bowtie l = l'$
---	---

 $\vdash h$ (0)Let $(h,\sigma,u,e_1,e) \Vdash l = l'$ (1) $\sigma \vdash e_1, h \rightsquigarrow e_2, h'$ (2)Virgin(e)(3)case l = u'(4)l' = u''(4)(5) \Rightarrow $u^{\prime\prime}=u \mathrel{\textcircled{}} u^\prime$ (6) $u^{\prime\prime} \neq \texttt{any}$ (7)(5)+(6)+(7) $\forall h', \sigma, e_2, e: (h', \sigma, u, e_2, e) \mathrel{\blacktriangleright} u = l'$ (8) \Rightarrow $(h', \sigma, u, e_2, e) \bowtie l = l'$ (8)+(4) \Rightarrow case $l = \mathbf{x}.f_{1...n}$ (4) $l' = p.f_{1...n}$ (4) \Rightarrow (5) $h, \sigma \vdash_{gb} e : p$ (6) $\begin{array}{l} h', \sigma \vdash_{gb} e : p \\ (h', \sigma, _, _, e) \bowtie l = l' \end{array}$ (6)+(3)+(2)+Lemma 3.6.7(7) \Rightarrow (4) + (5) + (7) \Rightarrow case $l = \texttt{this}.f_{1...n}$ (4)(4) $l' = p.f_{1...n}$ (5) \Rightarrow $h, \sigma \vdash_{gb} e_1 : p$ (6)(6)+(3)+(2)+Lemma 3.6.4 $h', \sigma \vdash_{gb} e_2 : p$ (7) \Rightarrow (4) + (5) + (7) \Rightarrow $(h',\sigma,_,e_2,_) \Vdash l = l'$

Lemma A4

$\vdash h$	
$(h,\sigma,u,v,e_1) \Vdash l = l'$	
$\sigma \vdash e_1, h \rightsquigarrow e_2, h'$	$\Longrightarrow (h', \sigma, u, v, e_2) \bowtie l = l'$
$_, h, \sigma \vdash e_1 : F$	
$\{l'\}\#F$	

 Let

$\vdash h$	(1)
$(h,\sigma,u,v,e_1) \Vdash l = l'$	(2)
$\sigma \vdash e_1, h \rightsquigarrow e_2, h'$	(3)
$h \sigma \vdash a \cdot F$	(4)

case $l = \texttt{this}.f_{1n}$			(4)
(6)	\Rightarrow	$l' = p.f_{1n}$	(7)
		$h, \sigma \vdash_{ab} v : p$	(8)
(5)+(7)	\Rightarrow	$\{p\}\#\check{F}$	(9)
(8)+(3)+(4)+(9)+Lemma 3.6.6	\Rightarrow	$h', \sigma \vdash_{ab} v : p$	(10)
(6)+(7)+(10)	\Rightarrow	$(h'\sigma,_,v,_) \mathrel{\blacktriangleright} l = l'$	
case otherwise			(4)
Similar to Lemma A3 cases i	l = u	and $l = this.f_{1n}$	

$\begin{array}{l} h, \sigma \vdash a : _ c \\ h(a) \downarrow_2 = c \end{array}$	$\left. \begin{array}{c} \\ \end{array} \right\} \Longrightarrow c' \ge c$			
	Let		$\begin{array}{l} h, \sigma \vdash a: _ c' \\ h(a) \downarrow_2 = c \end{array}$	(1) (2)
	case (1) last	was (Sub)	(3)
	(3)	\Rightarrow	$\begin{array}{l}h, \sigma \vdash a : _ c''\\c'' \le c'\end{array}$	(4) (5)
	$egin{array}{l} (4)+(2)+{{ m IH}}\ (6)+(5) \end{array}$	$\begin{array}{c} \Rightarrow \\ \Rightarrow \end{array}$	$c'' \ge c$ $c' \ge c'' \ge c$	(6)
	case (1) last	was (Addr)	(3)
	${(3)} \ {(4)+(2)}$	$\Rightarrow \Rightarrow$	$ \begin{array}{l} h(a){\downarrow_2} = c'\\ c' = c \end{array} $	(4)

Lemma A6

 $\left. \begin{array}{c} h,\sigma\vdash a:u \\ \sigma'(\texttt{this}) = a \\ h,\sigma\vdash v:t \end{array} \right\} \Longrightarrow h,\sigma'\vdash v:u \vDash t$

Not proved here, this is a general property of universe types.

Lemma A7

 $Virgin(e) \Longrightarrow Reachable(e)$

Let		Virgin(e)	(1)
case $e \in \{\mathbf{x}, \mathbf{x}\}$	this,r	ull, new t	(2)
(2)	\Rightarrow	Reachable(e)	
case $e = (t)e$	/		(2)
(1) + (2)	\Rightarrow	Virgin(e')	(3)
$(3)+\underline{\mathrm{IH}}$	\Rightarrow	Reachable(e')	(4)
(4) + (2)	\Rightarrow	Reachable(e)	
case $e = e'.f$			(2)
(1) + (2)	\Rightarrow	Virgin(e')	(3)
$(3) + \underline{\mathrm{IH}}$	\Rightarrow	Reachable(e')	(4)

(4) + (2)	\Rightarrow	Reachable(e)	
case $e = e'.f$:= e''		(2)
(1)+(2)	\Rightarrow	Virgin(e')	(3)
		Virgin(e'')	(4)
$(3) + \underline{\mathrm{IH}}$	\Rightarrow	Reachable(e')	(5)
(5)+(4)+(2)	\Rightarrow	Reachable(e)	
case $e = e'.m$	(e'')		(2)
(1) + (2)	\Rightarrow	Virgin(e')	(3)
		Virgin(e'')	(4)
$(3) + \underline{\mathrm{IH}}$	\Rightarrow	Reachable(e')	(5)
(5)+(4)+(2)	\Rightarrow	Reachable(e)	
case $e = span$	n e'		(2)
(1) + (2)	\Rightarrow	Virgin(e')	(3)
(3)+(2)	\Rightarrow	Reachable(e)	
case $e = sync$	e' e'	/	(2)
(1)+(2)	\Rightarrow	Virgin(e')	(3)
		Virgin(e'')	(4)
$(3) + \underline{\mathrm{IH}}$	\Rightarrow	Reachable(e')	(5)
(5)+(4)+(2)	\Rightarrow	Reachable(e)	

 $\left. \begin{array}{l} h, \sigma \vdash \sigma'(\texttt{this}) : u \\ h, \sigma' \vdash v : t \end{array} \right\} \Longrightarrow h, \sigma \vdash v : u \bowtie t$

Not proved here, this is a general property of universe types.

Lemma A9

$\begin{array}{l} h, \sigma \vdash a : u \\ h, \sigma \vdash a' : u' \\ u \neq \texttt{any} \\ u' \neq \texttt{any} \\ u \leq u' \end{array}$	$\left\} \Longrightarrow h(a) \downarrow_1 = h(a') \downarrow_1$	
Let	$egin{array}{lll} h,\sigmadash a:u\h,\sigmadash a':u'\u & u'\u & any\u' eq any\u & u' \end{array}$	(1)(2)(3)(4)(5)
case (1) last was $(A$ (6)	$\begin{array}{llllllllllllllllllllllllllllllllllll$	(6) (7) (8) (9)

case $u = u' = rep$		(10)
$(10) \Rightarrow h(a){\downarrow_1} =$	$h(a'){\downarrow_1} = b$	
case $u, u' \in \{\texttt{peer}, \texttt{se}\}$	elf}	(10)
$(10) \Rightarrow h(a){\downarrow_1} =$	$h(a'){\downarrow_1}=h(b){\downarrow_1}$	
case otherwise		(10)
Conflict with (3)	, (4), (5)	
		·
case (1) last was (SUB)		(6
(6)	$\Rightarrow h, \sigma \vdash a : u''$	(7
	$u''_{\cdots} \leq u$	(8
(8) + (3)	\Rightarrow $u'' \neq ext{any}$	(6
(8)+(5)	$\Rightarrow u'' \leq u'$	(10
$(7)+(2)+(9)+(4)+(10)+\underline{1}$	$\underline{\mathbf{H}} \Rightarrow h(a) \downarrow_1 = h(a') \downarrow_1$	
case (2) last was (SUB)		(6
(6)	$\Rightarrow h, \sigma \vdash a' : u''$	(7
	$u'' \leq u$	(8
(8)+(3)	$\Rightarrow u'' \neq any$	() ()
(9)+(3)+(8)+(5)	$\Rightarrow u'' \leq u \text{ or } u \leq u''$	(10
(1)+(7)+(3)+(9)+(10)+	$\underline{\mathbf{H}} \Rightarrow h(a)\downarrow_1 = h(a')\downarrow_1$	
		[

 $\begin{array}{l} (h,\sigma,u,a,v) \Vdash l' = l \\ \sigma' \vdash e,h \rightsquigarrow _,h' \\ _,h,\sigma' \vdash e:F \\ \{l\} \# F \end{array} \end{array} \right\} \Longrightarrow (h',\sigma,u,a,v) \Vdash l' = l$

case $l' = u$	(1)
Similar to Lemma A3 case $l = u'$	
case $l' = \texttt{this}.f_{1n}$	(1)
Similar to Lemma A4 case $l = this.f_{1n}$	
case $l' = \mathbf{x}.f_{1n}$	(1)
Similar to Lemma A4 case $l = this.f_{1n}$	

Lemma A11

 $h, \sigma \vdash \texttt{frame} \ \sigma' \ e' : t \Longrightarrow h, \sigma' \vdash e' : _$

Let	$h,\sigma\vdash\texttt{frame }\sigma'\ e':t$	(1)
case (1) last	was (SUB)	(2)
$(2) \qquad \Rightarrow \qquad \qquad$	$h, \sigma \vdash \texttt{frame} \ \sigma' \ e' : t'$	(3)
	$t' \leq t$	(4)
$(3)+\underline{\mathrm{IH}} \Rightarrow$	$h,\sigma'\vdash e':_$	
case (1) last	was (FRAME)	(2)
$(2) \qquad \Rightarrow \qquad \qquad$	$h, \sigma' \vdash e' : _$	

$\sigma \vdash e, h \rightsquigarrow e', h' \Longrightarrow \forall w : Locked(e, w) \Leftrightarrow Locked(e', w)$	
Let $\sigma \vdash e, h \rightsquigarrow e', h'$	(1)
case (1) last was (THIS) (VAR) (NEW) (FIELD) (ASSIGN) (CAST) (FRA	ME2) (2)
(2) $\Rightarrow \forall w : \neg Locked(e, w)$, , , , ,
$\forall w : \neg Locked(e', w)$	
case (1) last was (FRAME1)	(2)
(2) $\Rightarrow e = \text{frame } \sigma' e''$	(3)
$e'={\tt frame}\sigma'e'''$	(4)
$\sigma' dash e'', h \leadsto e''', h'$	(5)
(4) \Rightarrow Locked(e, w) = Locked(e'', w)	(6)
(3) \Rightarrow Locked(e', w) = Locked(e''', w)	(7)
(5) +IH $\Rightarrow \forall w : Locked(e'', w) \Leftrightarrow Locked(e''', w)$	(8)
$(8) + (\overline{6}) + (7) \Rightarrow \forall w : Locked(e, w) \Leftrightarrow Locked(e', w)$	
case (1) last was (CTX)	(2)
case $E[\bullet] \neq synced \dots$	(3)
Similar to (FRAME1)	()
case $E[\bullet] = \texttt{synced} \dots$	(3)
$(3) \Rightarrow e = \texttt{synced}_{e''} \ w' \ e'''$	(4)
$e' = \texttt{synced}_{e''} w' e''''$	(5)
$\sigma \vdash e^{\prime\prime\prime}, h \rightsquigarrow e^{\prime\prime\prime\prime}, h^\prime$	(6)
Let w	
case $w \neq w'$	(7)
Similar to (FRAME1)	
case $w = w'$	(7)
(4) \Rightarrow Locked(e, w)	(8)
Locked(e', w)	(9)
$(8)+(9) \Rightarrow Locked(e,w) \Leftrightarrow Locked(e',w)$	(-)
	•
	•
case (1) last was (CALL)	(2)
Let w	(9)

Let <i>w</i>	`	,	· · · · · · · · · · · · · · · · · · ·
(2)	\Rightarrow	$\neg Locked(e, w)$	(3)
		$e' = \texttt{frame} \ \sigma' \ e''$	(4)
		$e'' = \mathcal{M}(c,m)$	(5)
(5)	\Rightarrow	$\neg Locked(e'', w)$	(6)
(4) + (6)	\Rightarrow	$\neg Locked(e', w)$	(7)
(3) + (7)	\Rightarrow	$Locked(e,w) \Leftrightarrow Locked(e',w)$	

Lemma A13

$$\begin{bmatrix} \mathbb{L}, h, \sigma \vdash C[e] : F \\ \forall t', \sigma' : h, \sigma' \vdash e : t' \Rightarrow h, \sigma' \vdash e' : t' \\ \forall \mathbb{L}', F', \sigma' : \mathbb{L}', h, \sigma' \vdash e : F' \Rightarrow \mathbb{L}', h, \sigma' \vdash e' : F' \\ e \notin Path \\ \forall \sigma' : Active(\sigma', e) = \sigma' \end{bmatrix} \Rightarrow \begin{bmatrix} \mathbb{L}'', F'' : \\ \mathbb{L}', h, Active(\sigma, C[e]) \vdash e : F'' \\ \mathbb{L}, h, \sigma \vdash C[e'] : F \end{bmatrix}$$
Let
$$\mathbb{L}, h, \sigma \vdash C[e] : F \qquad (1$$

(1)

$\forall t', \sigma': h, \sigma' \vdash e: t' \Rightarrow t$	h,σ'	$\vdash e':t'$	(2)
$\forall \mathbb{L}', F', \sigma' : \mathbb{L}', h, \sigma' \vdash e$	e:F'	$\Rightarrow \mathbb{L}', h, \sigma' \vdash e' : F'$	(3)
$e \notin Path$			(4)
$\forall \sigma': Active(\sigma', e) = \sigma'$			(5)
case $C[e] = e$			(6)
$(6)+(1) \Rightarrow \mathbb{L}, h, \sigma \vdash e: F$			(7)
$(6)+(5) \Rightarrow Active(\sigma, C[e]) = \sigma$			(8)
$(7)+(8) \Rightarrow \mathbb{L}, h, Active(\sigma, C[e]) \vdash e$	e:F		
$(7)+(3) \Rightarrow \mathbb{L}, h, \sigma \vdash e' : F$			(9)
$(9)+(6) \Rightarrow \mathbb{L}, h, \sigma \vdash C[e']: F$			
$\frac{1}{2} \cos C[a] \neq a$			(6)
			(0)
case (1) last was (FRAME)			(7)
(7)	\Rightarrow	$C[e] = $ frame $\sigma''' e'''$	(8)
		$\mathbb{L}''', h, \sigma''' \vdash e''' : F$	(9)
		$\sigma^{\prime\prime\prime} = (a, v)$	(10)
		$\mathbb{L} = (h, \sigma, u, a, v) \mathbb{P} \mathbb{L}'''$	(11)
(6) + (8)	\Rightarrow	$e^{\prime\prime\prime} = C^{\prime}[e]$	(12)
$(12)\!+\!(9)\!+\!(2)\!+\!(3)\!+\!(4)\!+\!(5)\!+\!{ m IH}$	\Rightarrow	$\exists \mathbb{L}'', F'' : \mathbb{L}'', h, Active(\sigma''', C'[e]) \vdash e : F'$	'(13)
		$\mathbb{L}''', h, \sigma''' \vdash C'[e'] : F$	(14)
(14) + (10) + (11) + (Frame)	\Rightarrow	$\mathbb{L}, h, \sigma \vdash \texttt{frame} \ \sigma''' \ C'[e'] : F$	(15)
$(15)\!+\!(8)\!+\!(12)$	\Rightarrow	$\mathbb{L}, h, \sigma \vdash C[e'] : F$	
$(8)\!+\!(12)\!+\!(13)$	\Rightarrow	$\exists \mathbb{L}'', F'' : \mathbb{L}'', h, Active(_, C[e]) \vdash e : F''$	
case (1) last was (CALL)			(7)
(7) \Rightarrow $C[e] = e'''.m(e'''')$			(8)
$\mathbb{L}, h, \sigma \vdash e''' : F$			(9)
$\mathbb{L}, h, \sigma \vdash e'''' : F$			(10)
$h, \sigma \vdash e''' : u \ c$			(11)
$\mathcal{E}ff(c,m) _{\mathfrak{d}} \subseteq F$			(12)
$\mathbb{L}' \in \mathcal{F}ff(c, m) _{1}$			(13)
$(h, \sigma, u, e''', e'''') \gg \mathbb{L}' \subseteq$	T.		(14)
			(17)
case $e^{-1} = C^{*}[e]$ (15) + (0) + (2) + (2)			(15)
(15)+(5)+(2)+(5) + $(4)+(5)+111$	<u> </u>	$\exists \mathbb{I}'' F'' \cdot \mathbb{I}'' h Action(\sigma C[c]) \vdash c \cdot F''$	(16)
$+(4)+(0)+\underline{111}$	\Rightarrow	$\square , I' . \square , I', Active(0, \cup [e]) \vdash e : F \\ \square h \sigma \vdash C'[e'] : F$	(10)
(9) + (15)	ς.	$\mathbf{L}, \mathbf{n}, 0 \vdash \mathbf{O} [\mathbf{e}] : \mathbf{F}$ $Active(\mathbf{\sigma} C'[\mathbf{e}]) = Active(\mathbf{\sigma} C[\mathbf{e}])$	(11)
(0)+(10) (15) + (11) + (9) + Lemme - A14	\Rightarrow	$Active(0, \cup [e]) = Active(0, \cup [e])$	(10)
(13)+(11)+(2)+Lemma A14	\Rightarrow	$u, o \vdash \cup [e] : u c$	(19)
(4)+(10)+(14) (17)+(10)+(10)+(12)+(12)	\Rightarrow	$\forall e_1, e_2 : (n, \sigma, u, e_1, e_2) \bowtie \mathbb{L} \subseteq \mathbb{L}$	(20)
(17)+(10)+(19)+(12)+(13)			(01)
+(20)+(CALL)	\Rightarrow	$\mathbb{L}, n, \sigma \vdash C'[e'].m(e''''): F'$	(21)
(21)+(15)+(8)	\Rightarrow	$\mathbb{L}, h, \sigma \vdash C[e'] : F'$	
(16) + (18)	\Rightarrow	$\exists \mathbb{L}'', F''' : \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e : F''$	
case $e^{\prime\prime\prime\prime} = C^{\prime}[e]$			(15)

$\begin{array}{c} \begin{array}{c} \begin{array}{c} \begin{array}{c} \label{eq:constraints} \mathbb{L},h,\sigma\vdash e^{im'}:F\\ \mathbb{L},h,\sigma\vdash e^{im'}:F\\ \mathbb{L},h,\sigma\vdash e^{im'}:F\\ \mathbb{L},h,\sigma\vdash e^{im'}:F\\ \mathbb{L},h,\sigma\vdash e^{im'}:F\\ \mathbb{L},h,\sigma\vdash e^{im'}:E^{im'}:\mathbb{L}'',h,Active(\sigma,C'[e])\vdash e:F''\\ (1)\\ \mathbb{L}\in\mathbb{L}\\ f\in F \end{array} \end{array} $	case (1) last was (ASSIGN) (7) \Rightarrow $C[e] = e''' \cdot f := e$,,,,		
$ \begin{split} \mathbb{L}, h, \sigma \vdash e''' : F & (f = h + h + h + h + h + h + h + h + h + h$	$\mathbb{L}, h, \sigma \vdash e''' : F$			
$\begin{array}{c} h \vdash_{gb} \sigma : e'''l \\ l \in \mathbb{L} \\ f \in F \end{array} (1)$ $\begin{array}{c} f \in F \end{array} (2)$ $\begin{array}{c} (1) \\ (1) + (9) + (2) + (3) \\ + (4) + (5) + \underline{H} \\ + (1) + \underline{L} \\ (4) + (2) + (14) \\ + (11) + \underline{L} \\ (1) + (10) + (12) \\ + (13) + (ASIGN) \end{array} \Rightarrow \mathbb{L}, h, \sigma \vdash_{gb} C'[e] : l \qquad (1')$ $\begin{array}{c} (1) \\ (16) + (17) + (10) + (12) \\ + (13) + (ASIGN) \end{array} \Rightarrow \mathbb{L}, h, \sigma \vdash_{C}[e'] : F \qquad (18) \\ (18) + (8) + (14) \\ \Rightarrow \\ Active(\sigma, C'[e]) = Active(\sigma, C[e]) \\ (15) + (19) \\ \Rightarrow \\ \exists \mathbb{L}'', F'' : \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e : F'' \\ \hline \\ \hline \\ \hline \\ case e^{I'''} = C'[e] \\ Very similar. \qquad (12) \\ \hline \\ case (1) last was (FIELD) \\ Similar to (ASSIGN) \\ \hline \\ case (1) last was (SYNCED) \\ Similar to (ASSIGN) \\ \hline \\ case (1) last was (CAST) \\ Similar to (ASSIGN) \\ \hline \\ case (1) last was (SUB) \\ (7) \\ \Rightarrow \\ \begin{array}{c} \mathbb{L}'', h, \sigma \vdash e : F''' \\ \mathbb{L}''' \subseteq \mathbb{L} \\ F''' \subseteq F \\ \mathbb{L} \\ \#F \\ (1) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ \\ (8) + (2) + (3) + (4) + (5) + \mathbf{IH} \\ \Rightarrow \\ \end{array} \Rightarrow \\ \begin{array}{c} \mathbb{L}'', h, \sigma \vdash C[e'] : F'' \\ \mathbb{L}''', h, \sigma \vdash C[e'] : F'' \\ \hline \\ \hline \\ (13) + (9) + (10) + (11) + (12) \\ \Rightarrow \\ \mathbb{L}, h, \sigma \vdash C[e'] : F'' \\ \hline \\ case (1) last was (VAR) (THIS) (ADDR) (NULL) (NEW) (SPAWN) \\ \hline \\ contradicts (6) \\ \end{array}$	$\mathbb{L}, h, \sigma \vdash e'''' : F$			(
$\begin{array}{c} l \in \mathbb{L} \\ f \in F \\ \hline \\ \hline \\ case e''' = C'[e] \\ (14) + (9) + (2) + (3) \\ + (4) + (5) + III \\ + (4) + (5) + III \\ \hline \\ (4) + (2) + (14) \\ + (11) + Lemma A21 \\ + (13) + (Assign) \\ + (13) + (Assign) \\ + (13) + (Assign) \\ \hline \\ \hline \\ (16) + (17) + (10) + (12) \\ + (13) + (Assign) \\ \hline \\ \hline \\ (16) + (17) + (10) + (12) \\ \hline \\ + (13) + (Assign) \\ \hline \\ $	$h \vdash_{gb} \sigma : e'''l$			(
$\begin{array}{c} f \in F \\ \hline case e''' = C'[e] \\ (14) + (9) + (2) + (3) \\ + (4) + (5) + \underline{H} \\ & \Rightarrow \\ \exists L'', F'' : L'', h, Active(\sigma, C'[e]) \vdash e : F'' \\ & \downarrow, h, \sigma \vdash C'[e] : F \\ (14) + (2) + (14) \\ + (11) + Lemma A21 \\ + (13) + (Assign) \\ & \Rightarrow \\ \downarrow, h, \sigma \vdash C'[e] : F \\ (18) + (8) + (14) \\ & \Rightarrow \\ Active(\sigma, C'[e]) = Active(\sigma, C[e]) \\ & (15) + (19) \\ \hline case e'''' = C'[e] \\ & Very similar. \\ \hline case (1) last was (FIELD) \\ Similar to (ASSIGN) \\ \hline case (1) last was (SYNCED) \\ Similar to (ASSIGN) \\ \hline case (1) last was (SYNCED) \\ Similar to (ASSIGN) \\ \hline case (1) last was (SYNCED) \\ Similar to (ASSIGN) \\ \hline case (1) last was (SUB) \\ (7) \\ & \Rightarrow \\ L''', h, \sigma \vdash e : F''' \\ & L''' \subseteq L \\ F''' \subseteq F \\ & L\#F \\ & (8) + (2) + (3) + (4) + (5) + I\underline{H} \\ \Rightarrow \\ \exists L'', F'' : L'', h, Active(\sigma, C[e]) \vdash e : F'' \\ \hline case (1) last was (VAR) (THIS) (ADDR) (NULL) (NEW) (SPAWN) \\ Contradicts (6) \\ \hline \end{array}$	$l\in\mathbb{L}$			(
$\begin{array}{c} \hline \operatorname{case} e''' = C'[e] & (1 \\ (14) + (9) + (2) + (3) \\ + (4) + (5) + \underline{\mathrm{H}} & \Rightarrow & \exists \mathbb{L}'', F'' : \mathbb{L}'', h, Active(\sigma, C'[e]) \vdash e : F'' & (1 \\ & \mathbb{L}, h, \sigma \vdash C'[e] : F & (1 \\ & (4) + (2) + (14) \\ + (11) + \operatorname{Lemma} A21 & \Rightarrow h, \sigma \vdash_{gb} C'[e] : l & (1' \\ (16) + (17) + (10) + (12) & \\ & (13) + (Assign) & \Rightarrow & \mathbb{L}, h, \sigma \vdash C'[e], f := e''' : F & (14) \\ & (13) + (Assign) & \Rightarrow & \mathbb{L}, h, \sigma \vdash C[e'] : F & (14) \\ & (15) + (14) & \Rightarrow & Active(\sigma, C'[e]) = Active(\sigma, C[e]) & (15) \\ & (15) + (19) & \Rightarrow & \exists \mathbb{L}'', F'' : \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e : F'' & \\ \hline & \\ & \\$	$f \in F$			(
$\begin{array}{cccc} (14)+(9)+(2)+(3) \\ +(4)+(5)+\underline{\mathbf{H}} & \Rightarrow & \exists \mathbb{L}'', F'': \mathbb{L}'', h, Active(\sigma, C'[e]) \vdash e:F'' & (14) \\ & \mathbb{L}, h, \sigma \vdash C'[e]:F & (14) \\ +(11)+\text{Lemma A21} & \Rightarrow & h, \sigma \vdash_{gb} C'[e]:l & (11) \\ (16)+(17)+(10)+(12) \\ +(13)+(AssiGN) & \Rightarrow & \mathbb{L}, h, \sigma \vdash C'[e];f \coloneqq e''':F & (18)+(14) & \Rightarrow & Active(\sigma, C'[e]) = Active(\sigma, C[e]) & (15)+(19) & \Rightarrow & \exists \mathbb{L}'', F'': \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e:F'' \\ \hline \hline (15)+(19) & \Rightarrow & \exists \mathbb{L}'', F'': \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e:F'' & \\ \hline \hline case e''' = C'[e] & (14) \\ \hline very similar. & (14) \\ \hline case (1) last was (FIELD) & \\ \hline Similar to (AssiGN) & \\ \hline case (1) last was (SYNCE) & \\ \hline Similar to (AssiGN) & \\ \hline case (1) last was (SYNCED) & \\ \hline Similar to (AssiGN) & \\ \hline case (1) last was (CAST) & \\ \hline Similar to (AssiGN) & \\ \hline case (1) last was (SUB) & \\ \hline (7) & \Rightarrow & \mathbb{L}'', h, \sigma \vdash e:F''' & \\ \hline \mathbb{L}'' \subseteq \mathbb{L} & \\ F''' \subseteq F & (14) \\ \hline & & \forall p \in \mathbb{L}: \mathbb{L}, h, \sigma \vdash p: _ \\ \hline (8)+(2)+(3)+(4)+(5)+\underline{\mathbf{H}} & \Rightarrow & \exists \mathbb{L}'', F'': \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e:F'' & \\ \hline (13)+(9)+(10)+(11)+(12) & \Rightarrow & \mathbb{L}, h, \sigma \vdash C[e']:F'' & \\ \hline case (1) last was (VAR) (THIS) (ADDR) (NULL) (NEW) (SPAWN) \\ \hline contradicts (6) & \\ \hline \end{array}$	case $e^{\prime\prime\prime} = C^{\prime}[e]$			(14
$\begin{array}{cccc} +(4)+(5)+\underline{\mathrm{II}} & \Rightarrow & \exists \mathbb{L}^{n}, F^{n}:\mathbb{L}^{n}, h, \operatorname{Active}(\sigma, C^{n}[e])\vdash e:F^{n} & (14)\\ & & \mathbb{L}, h, \sigma \vdash C^{n}[e]:F & (14)\\ & & (4)+(2)+(14) & \\ +(11)+\operatorname{Lemma} A21 & \Rightarrow & h, \sigma \vdash_{gb} C^{n}[e]:l & (11)\\ & & (16)+(17)+(10)+(12) & \\ +(13)+(ASIGN) & \Rightarrow & \mathbb{L}, h, \sigma \vdash C^{n}[e]:F & (18)\\ & & (18)+(8)+(14) & \Rightarrow & \operatorname{Active}(\sigma, C^{n}[e]) & (19)\\ & & (15)+(19) & \Rightarrow & \exists \mathbb{L}^{n}, F^{n}:\mathbb{L}^{n}, h, \operatorname{Active}(\sigma, C^{n}[e]) & (19)\\ & & (15)+(19) & \Rightarrow & \exists \mathbb{L}^{n}, F^{n}:\mathbb{L}^{n}, h, \operatorname{Active}(\sigma, C^{n}[e]) & e:F^{n} & (18)\\ & & & (16)+(17)+(19) & \Rightarrow & \exists \mathbb{L}^{n}, F^{n}:\mathbb{L}^{n}, h, \operatorname{Active}(\sigma, C^{n}[e]) & (19)+(10)+(11)+(12) & \Rightarrow & \mathbb{L}^{n}, \sigma \vdash e:F^{nn} & (18)\\ & & & & & & & & & & & & & & & & & & &$	(14)+(9)+(2)+(3)			(1 =
$(4)+(2)+(14) + \text{Lemma A21} \Rightarrow h, \sigma \vdash_{gb} C'[e] : l \qquad (1'$ $(1)+\text{Lemma A21} \Rightarrow h, \sigma \vdash_{gb} C'[e] : l \qquad (1'$ $(1)+(1)+(10)+(12) + (13)+(Assign) \Rightarrow \mathbb{L}, h, \sigma \vdash C'[e] : F \qquad (18)$ $(18)+(8)+(14) \Rightarrow \mathbb{L}, h, \sigma \vdash C[e'] : F \qquad (18)$ $(18)+(14) \Rightarrow Active(\sigma, C'[e]) = Active(\sigma, C[e]) \qquad (19)$ $(15)+(19) \Rightarrow \exists \mathbb{L}'', F'' : \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e : F'' \qquad (18)$ $(15)+(19) \Rightarrow \exists \mathbb{L}'', F'' : \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e : F'' \qquad (18)$ $(13)+(19)+(10)+(11)+(12) \Rightarrow \mathbb{L}, h, \sigma \vdash e : F''' \qquad (18)+(12)+(12)+(12)+(12)+(12)+(12)+(12)+(12$	$+(4)+(5)+\underline{IH}$	\Rightarrow	$\exists \mathbb{L}^n, F^n : \mathbb{L}^n, h, Active(\sigma, C'[e]) \vdash e : F^n \\ \mathbb{L}, h, \sigma \vdash C'[e] : F$	(15) (16)
$(1)^{+1}(1)^$	(4)+(2)+(14)	`	$h = \int C'[c] \cdot l$	(15
$\begin{array}{rcl} (\mathrm{dS}) & (\mathrm{dS}) & (\mathrm{dS}) \\ + (\mathrm{dS}) + (\mathrm{dS}) & (\mathrm{dS}) \\ + (\mathrm{dS}) + (\mathrm{dS}) & (\mathrm{dS}) \\ \hline \\ $	+(11)+Lemma A21 (16)+ (17) + (10) + (12)	\Rightarrow	$n, o \vdash_{gb} \mathbb{C}[e] \cdot i$	(17
$\begin{array}{cccc} (18)+(8)+(14) &\Rightarrow & \mathbb{L}, h, \sigma \vdash C[e'] : F \\ (8)+(14) &\Rightarrow & Active(\sigma, C'[e]) = Active(\sigma, C[e]) &(19) \\ \hline (15)+(19) &\Rightarrow & \exists \mathbb{L}'', F'' : \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e : F'' \\ \hline \hline case e'''' = C'[e] &(14) \\ \hline case e'''' = C'[e] &(14) \\ \hline very similar. &(14) \\ \hline case (1) last was (FIELD) \\ \hline Similar to (ASSIGN) \\ \hline case (1) last was (SYNCED) \\ \hline Similar to (ASSIGN) \\ \hline case (1) last was (CAST) \\ \hline Similar to (ASSIGN) \\ \hline case (1) last was (CAST) \\ \hline Similar to (ASSIGN) \\ \hline case (1) last was (CAST) \\ \hline similar to (ASSIGN) \\ \hline case (1) last was (SUB) \\ \hline (7) &\Rightarrow & \mathbb{L}'', h, \sigma \vdash e : F''' \\ \hline \mathbb{L}''' \subseteq \mathbb{L} \\ F''' \subseteq F \\ \hline (8)+(2)+(3)+(4)+(5)+\underline{H} &\Rightarrow & \exists \mathbb{L}'', F'' : \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e : F'' \\ \hline \mathbb{L}''', h, \sigma \vdash C[e'] : F'' \\ \hline case (1) last was (VAR) (THIS) (ADDR) (NULL) (NEW) (SPAWN) \\ \hline contradicts (6) \\ \hline \end{array}$	+(13)+(A SSIGN)	\Rightarrow	$\mathbb{L}, h, \sigma \vdash C'[e], f := e'''' : F$	(18
$(8)+(14) \implies Active(\sigma, C'[e]) = Active(\sigma, C[e]) \qquad (19)$ $(15)+(19) \implies \exists \mathbb{L}'', F'': \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e: F''$ (14) $(15)+(19) \implies \exists \mathbb{L}'', F'': \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e: F''$ (14) $(15)+(19) \implies \exists \mathbb{L}'', F'': \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e: F''$ (14) (14) $(15)+(10) + (11) + (12) \implies dx$ $(11) \implies Active(\sigma, C[e]) \vdash e: F''$ $(13)+(9)+(10) + (11) + (12) \implies dx$ $(11) (Addressed $	(18) + (8) + (14)	\Rightarrow	$\mathbb{L}, h, \sigma \vdash C[e'] : F$	X
$\begin{array}{c} (15)+(19) \qquad \Rightarrow \exists \mathbb{L}'', F'': \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e: F'' \\ \hline case e'''' = C'[e] & (1e^{-i}) \\ \hline Very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ \hline very \ similar. \end{array} $ $\begin{array}{c} (1e^{-i}) \\ very \ simi$	(8) + (14)	\Rightarrow	$Active(\sigma, C'[e]) = Active(\sigma, C[e])$	(19)
$case e^{H''} = C'[e]$ Very similar.(14 $case (1)$ last was (FIELD) Similar to (ASSIGN)(15 $case (1)$ last was (SYNC) Similar to (ASSIGN)(11) $case (1)$ last was (SYNCED) Similar to (ASSIGN)(11) $case (1)$ last was (CAST) Similar to (ASSIGN)(11) $case (1)$ last was (CAST) Similar to (ASSIGN)(11) $case (1)$ last was (SUB) 	$(15)\!+\!(19)$	\Rightarrow	$\exists \mathbb{L}'', F'' : \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e : F''$	
Very similar. case (1) last was (FIELD) Similar to (Assign) case (1) last was (SYNCED) Similar to (Assign) case (1) last was (SYNCED) Similar to (Assign) case (1) last was (CAST) Similar to (Assign) case (1) last was (CAST) Similar to (Assign) (7) L''', $h, \sigma \vdash e : F'''$ L''', $h, \sigma \vdash e : F'''$ L''', $b, \sigma \vdash e : F'''$ L#F (6) U#F (8)+(2)+(3)+(4)+(5)+IH > IL''', $F'' : L', h, \sigma \vdash p : (8)+(2)+(3)+(4)+(5)+IH > IL''', F'' : L'', h, Active(\sigma, C[e]) \vdash e : F'' L''', h, \sigma \vdash C[e'] : F''' (1) (1) (2) (2) (2) (2) (2) (3) (2) (2$	case $e^{\prime\prime\prime\prime} = C^{\prime}[e]$			(14
$\begin{array}{c} \text{case (1) last was (FIELD)} \\ Similar to (ASSIGN) \\ \hline \text{case (1) last was (SYNC)} \\ Similar to (ASSIGN) \\ \hline \text{case (1) last was (SYNCED)} \\ Similar to (ASSIGN) \\ \hline \text{case (1) last was (CAST)} \\ Similar to (ASSIGN) \\ \hline \text{case (1) last was (SUB)} \\ \hline (7) \qquad \Rightarrow \mathbb{L}'', h, \sigma \vdash e : F''' \\ \mathbb{L}''' \subseteq \mathbb{L} \\ F''' \subseteq F \\ \mathbb{L}\#F \\ (6) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ \\ (8) + (2) + (3) + (4) + (5) + \underline{\text{H}} \\ \Rightarrow \exists \mathbb{L}'', F'' : \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e : F'' \\ \mathbb{L}''', h, \sigma \vdash C[e'] : F'' \\ \hline (13) + (9) + (10) + (11) + (12) \\ \Rightarrow \mathbb{L}, h, \sigma \vdash C[e'] : F \\ \hline \text{case (1) last was (VAR) (THIS) (ADDR) (NULL) (NEW) (SPAWN) \\ Contradicts (6) \end{array}$	Very similar.			
case (1) last was (FIELD) Similar to (ASSIGN) case (1) last was (SYNC) Similar to (ASSIGN) case (1) last was (SYNCED) Similar to (ASSIGN) case (1) last was (CAST) Similar to (ASSIGN) case (1) last was (SUB) (7) $\Rightarrow \mathbb{L}'', h, \sigma \vdash e : F'''$ $\mathbb{L}''' \subseteq \mathbb{L}$ $F''' \subseteq \mathbb{F}$ (($\mathbb{L}\#F$ (($\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F$ (($\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F$ (($\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F$ (($\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F$ (($\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F$ (($\mathbb{L}\#F$ (($\mathbb{L}\#F$ (($\mathbb{L}\#F) \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L}\#F \in \mathbb{L} : $				[
$\begin{array}{c} Similar \ to \ (A \ SSIGN) \\ \hline case \ (1) \ last \ was \ (SYNCED) \\ Similar \ to \ (A \ SSIGN) \\ \hline case \ (1) \ last \ was \ (SYNCED) \\ Similar \ to \ (A \ SSIGN) \\ \hline case \ (1) \ last \ was \ (C \ A \ SSIGN) \\ \hline case \ (1) \ last \ was \ (C \ A \ SSIGN) \\ \hline case \ (1) \ last \ was \ (SUB) \\ \hline (7) \qquad \Rightarrow \begin{array}{c} \mathbb{L}'', \ h, \ \sigma \vdash e : F''' \\ \mathbb{L}''' \subseteq \mathbb{L} \\ F''' \subseteq F \\ \mathbb{L} \# F \\ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, \ h, \ \sigma \vdash p : _ \\ (1) \ (1) + (1) + (1) + (1) \\ \Rightarrow \mathbb{L}'', \ F'' : \mathbb{L}'', \ h, \ Active(\sigma, C[e]) \vdash e : F'' \\ \mathbb{L}''', \ h, \ \sigma \vdash C[e'] : F'' \\ \hline (13) + (9) + (10) + (11) + (12) \\ \Rightarrow \mathbb{L}, \ h, \ \sigma \vdash C[e'] : F \\ \hline \hline case \ (1) \ last \ was \ (VAR) \ (THIS) \ (ADDR) \ (NULL) \ (NEW) \ (SPAWN) \\ \hline Contradicts \ (6) \end{array}$	case (1) last was (FIELD)			
$\begin{array}{l} \begin{array}{l} \begin{array}{l} \begin{array}{l} \operatorname{case} \left(1\right) \operatorname{last} \operatorname{was} \left(\operatorname{SYNC}\right) \\ Similar \ to \ \left(\operatorname{ASSIGN}\right) \end{array} \end{array} \\ \end{array} \\ \begin{array}{l} \begin{array}{l} \operatorname{case} \left(1\right) \operatorname{last} \operatorname{was} \left(\operatorname{SYNCED}\right) \\ Similar \ to \ \left(\operatorname{ASSIGN}\right) \end{array} \end{array} \\ \end{array} \\ \begin{array}{l} \begin{array}{l} \operatorname{case} \left(1\right) \operatorname{last} \operatorname{was} \left(\operatorname{CAST}\right) \\ \end{array} \\ \end{array} \\ \begin{array}{l} \begin{array}{l} \operatorname{case} \left(1\right) \operatorname{last} \operatorname{was} \left(\operatorname{SUB}\right) \end{array} \end{array} \end{array} \\ \end{array} \\ \begin{array}{l} \begin{array}{l} \begin{array}{l} \end{array} \\ \end{array} \\ \end{array} \\ \end{array} \\ \begin{array}{l} \end{array} \\ \end{array} \\ \end{array} \\ \end{array} \\ \begin{array}{l} \end{array} \\ \end{array} \\ \end{array} \\ \begin{array}{l} \end{array} \\ \end{array} \\ \end{array} \\ \begin{array}{l} \end{array} \\ \end{array} \\ \end{array} \\ \end{array} \\ \begin{array}{l} \end{array} \\ \end{array} \\ \end{array} \\ \begin{array}{l} \end{array} \\ \end{array} \\ \end{array} \\ \end{array} \\ \begin{array}{l} \end{array} \\ \end{array} \\ \end{array} \\ \end{array} \\ \begin{array}{l} \end{array} \\ \end{array} \\ \end{array} \\ \end{array} \\ \begin{array}{l} \end{array} \\ \end{array} \\ \end{array} \\ \begin{array}{l} \end{array} \\ \end{array} \\ \end{array} \\ \end{array} \\ \begin{array}{l} \end{array} \\ \end{array} \\ \end{array} \\ \end{array} \\ \begin{array}{l} \end{array} \\ \end{array} \\ \end{array} \\ \begin{array}{l} \end{array} \\ \end{array} \\ \end{array} \\ \end{array} \\ \end{array} \\ \end{array} \\ \begin{array}{l} \end{array} \\ \begin{array}{l} \end{array} \\ \begin{array}{l} \end{array} \\ \end{array} $	Similar to (Assign)			
$\begin{array}{l} \begin{array}{c} \text{Similar to (ASSIGN)} \\ \hline \text{case (1) last was (SYNCED)} \\ \hline \text{Similar to (ASSIGN)} \\ \hline \text{case (1) last was (CAST)} \\ \hline \text{Similar to (ASSIGN)} \\ \hline \text{case (1) last was (SUB)} \\ \hline (7) \qquad \Rightarrow \mathbb{L}^{\prime\prime\prime}, h, \sigma \vdash e : F^{\prime\prime\prime} \\ \mathbb{L}^{\prime\prime\prime} \subseteq \mathbb{L} \\ F^{\prime\prime\prime} \subseteq F \\ \hline (\mu \# F \\ (\phi p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (\phi (p \in \mathbb{L}))] = f = F^{\prime\prime} \\ \hline (13) + (9) + (10) + (11) + (12) \\ \hline \text{case (1) last was (VAR) (THIS) (ADDR) (NULL) (NEW) (SPAWN)} \\ \hline \text{Contradicts (6)} \end{array}$	case (1) last was (SYNC) Cimilar to (A SECON)			
case (1) last was (SYNCED) Similar to (ASSIGN) case (1) last was (CAST) Similar to (ASSIGN) (7) $\Rightarrow \mathbb{L}'', h, \sigma \vdash e : F'''$ $\mathbb{L}''' \subseteq \mathbb{L}$ $F''' \subseteq F$ (($\mathbb{L} \# F$ (($\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L} \# F$ (($\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{L} \# F$ (($\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{H} \# F$ (($\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{H} \# F$ (($\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{H} \# F$ (($\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (($\mathbb{H} \# F$ (($\mathbb{H} \# F$ (($\mathbb{H} \# F$ (($\mathbb{H} \# F$ (($\mathbb{H} \oplus \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (\mathbb{L} \oplus \mathbb{L} \oplus $	Simular to (ASSIGN)			
$Similar to (ASSIGN)$ $case (1) last was (CAST)$ $Similar to (ASSIGN)$ $case (1) last was (SUB)$ $(7) \qquad \Rightarrow \qquad \mathbb{L}'', h, \sigma \vdash e : F'''$ $\mathbb{L}''' \subseteq \mathbb{L}$ $F''' \subseteq F$ (4) $\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ $(8) + (2) + (3) + (4) + (5) + \underline{IH} \Rightarrow \exists \mathbb{L}'', F'' : \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e : F''$ $\mathbb{L}''', h, \sigma \vdash C[e'] : F''$ $(13) + (9) + (10) + (11) + (12) \Rightarrow \qquad \mathbb{L}, h, \sigma \vdash C[e'] : F$ $(13) + (9) + (10) + (11) + (12) \Rightarrow \qquad \mathbb{L}, h, \sigma \vdash C[e'] : F$ $(13) + (9) + (10) + (11) + (12) \Rightarrow \qquad \mathbb{L}, h, \sigma \vdash C[e'] : F$	case (1) last was (SYNCED)			
$\begin{array}{l} \text{case (1) last was (CAST)} \\ Similar to (ASSIGN) \\ \hline \text{case (1) last was (SUB)} \\ (7) \qquad \Rightarrow \mathbb{L}''', h, \sigma \vdash e : F''' \\ \mathbb{L}''' \subseteq \mathbb{L} \\ F''' \subseteq F \\ \mathbb{L} \# F \\ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ \\ (13) + (4) + (5) + \underline{\text{IH}} \\ \Rightarrow \exists \mathbb{L}'', F'' : \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e : F'' \\ \mathbb{L}''', h, \sigma \vdash C[e'] : F'' \\ \hline \text{case (1) last was (VAR) (THIS) (ADDR) (NULL) (NEW) (SPAWN)} \\ Contradicts (6) \end{array}$	Similar to (Assign)			
$Similar to (A SSIGN)$ $case (1) last was (SUB)$ $(7) \qquad \Rightarrow \qquad \mathbb{L}''', h, \sigma \vdash e : F''' \\ \qquad \mathbb{L}''' \subseteq \mathbb{L} \\ F''' \subseteq F \\ \qquad \mathbb{L} \#F \qquad (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L}, h, \sigma \vdash p : _ (0) \\ \forall p \in \mathbb{L}, \mu, \mu \in \mathbb{L}, \mu, \mu, \mu \in \mathbb{L}, \mu, \mu, \mu \in \mathbb{L}, \mu, \mu,$	case (1) last was (CAST)			
$\begin{array}{l} \text{case (1) last was (SUB)} \\ (7) \qquad \Rightarrow \qquad \mathbb{L}''', h, \sigma \vdash e : F''' \\ \qquad \mathbb{L}''' \subseteq \mathbb{L} \\ F''' \subseteq F \\ (1) \\ (8) + (2) + (3) + (4) + (5) + \underline{\text{IH}} \\ (8) + (2) + (3) + (4) + (5) + \underline{\text{IH}} \\ (8) + (2) + (3) + (4) + (5) + \underline{\text{IH}} \\ (8) + (2) + (3) + (4) + (5) + \underline{\text{IH}} \\ (8) + (2) + (3) + (4) + (5) + \underline{\text{IH}} \\ (8) + (2) + (3) + (4) + (5) + \underline{\text{IH}} \\ (8) + (2) + (3) + (4) + (5) + \underline{\text{IH}} \\ (8) + (2) + (3) + (4) + (5) + \underline{\text{IH}} \\ (8) + (2) + (3) + (4) + (5) + \underline{\text{IH}} \\ (8) + (2) + (3) + (4) + (5) + \underline{\text{IH}} \\ (8) + (2) + (3) + (4) + (5) + \underline{\text{IH}} \\ (8) + (2) + (3) + (4) + (5) + \underline{\text{IH}} \\ (8) + (2) + (3) + (4) + (5) + \underline{\text{IH}} \\ (8) + (2) + (3) + (4) + (5) + \underline{\text{IH}} \\ (8) + (2) + (3) + (3) + (3) + (3) + (2) + (3) + ($	Similar to (Assign)			
(7) $\Rightarrow \mathbb{L}''', h, \sigma \vdash e : F'''$ $\mathbb{L}''' \subseteq \mathbb{L}$ $F''' \subseteq F$ $\mathbb{L} \# F$ (6) $\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (7) $\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ (8)+(2)+(3)+(4)+(5)+ <u>IH</u>) $\Rightarrow \exists \mathbb{L}'', F'' : \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e : F''$ $\mathbb{L}''', h, \sigma \vdash C[e'] : F'''$ (13)+(9)+(10)+(11)+(12)) $\Rightarrow \mathbb{L}, h, \sigma \vdash C[e'] : F$ (13)+(9)+(10)+(11)+(12)) $\Rightarrow \mathbb{L}, h, \sigma \vdash C[e'] : F$ (13)+(9)+(10)+(11)+(12)) (ADDR) (NULL) (NEW) (SPAWN) <i>Contradicts (6)</i>	case (1) last was (SUB)			
$ \begin{array}{c} \mathbb{L}^{\prime\prime\prime} \subseteq \mathbb{L} \\ F^{\prime\prime\prime} \subseteq F \\ (1) \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ \\ (1) \\ (3) + (2) + (3) + (4) + (5) + \underline{\mathrm{IH}} \\ (3) + (2) + (3) + (1) + (1) + (12) \\ (3) \\ (3) + (2) + (10) + (11) + (12) \\ (3) \\$	(7)	\Rightarrow	$\mathbb{L}^{\prime\prime\prime}, h, \sigma \vdash e: F^{\prime\prime\prime}$	
$F''' \subseteq F$ $\mathbb{L}\#F$ (0) $\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$ $(13) + (9) + (10) + (11) + (12) \Rightarrow \mathbb{L}, h, \sigma \vdash C[e'] : F''$ $(13) + (9) + (10) + (11) + (12) \Rightarrow \mathbb{L}, h, \sigma \vdash C[e'] : F$ $(13) + (9) + (10) + (11) + (12) \Rightarrow \mathbb{L}, h, \sigma \vdash C[e'] : F$ $(13) + (9) + (10) + (11) + (12) \Rightarrow \mathbb{L}, h, \sigma \vdash C[e'] : F$			$\mathbb{L}''' \subseteq \mathbb{L}$	
$ \begin{array}{c} \mathbb{L}\#F \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ \\ (6) + (2) + (3) + (4) + (5) + \underline{\text{IH}} \\ \Rightarrow \\ \mathbb{L}'', F'' : \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e : F'' \\ \mathbb{L}''', h, \sigma \vdash C[e'] : F''' \\ (13) + (9) + (10) + (11) + (12) \\ \Rightarrow \\ \mathbb{L}, h, \sigma \vdash C[e'] : F \end{array} $ $ \begin{array}{c} (6) \\ (6) \\ (7) \\ $			$F''' \subseteq F$	(
$ \begin{array}{c} \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ \\ (8) + (2) + (3) + (4) + (5) + \underline{\text{IH}} & \Rightarrow & \exists \mathbb{L}'', F'' : \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e : F'' \\ & & \mathbb{L}''', h, \sigma \vdash C[e'] : F''' \\ (13) + (9) + (10) + (11) + (12) & \Rightarrow & \mathbb{L}, h, \sigma \vdash C[e'] : F \\ \hline \text{case (1) last was (VAR) (THIS) (ADDR) (NULL) (NEW) (SPAWN)} \\ \hline \text{Contradicts (6)} \end{array} $			$\mathbb{L}\#F$	(
$ \begin{array}{rcl} (8)+(2)+(3)+(4)+(5)+\underline{IH} & \Rightarrow & \exists \mathbb{L}'', F'':\mathbb{L}'', h, Active(\sigma, C[e]) \vdash e:F'' \\ & & \mathbb{L}''', h, \sigma \vdash C[e']:F''' \\ (13)+(9)+(10)+(11)+(12) & \Rightarrow & \mathbb{L}, h, \sigma \vdash C[e']:F \\ \hline \\ $			$\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _$	(
	$(8)+(2)+(3)+(4)+(5)+\underline{IH}$	\Rightarrow	$\exists \mathbb{L}'', F'' : \mathbb{L}'', h, Active(\sigma, C[e]) \vdash e : F''$,
$\begin{array}{ll} (13)+(9)+(10)+(11)+(12) & \Rightarrow & \mathbb{L}, h, \sigma \vdash C[e']: F\\ \hline case \ (1) \ last \ was \ (VAR) \ (THIS) \ (ADDR) \ (NULL) \ (NEW) \ (SPAWN)\\ \hline Contradicts \ (6) \end{array}$	(10) + (0) + (10) + (11) + (12)		$\mathbb{L}^{\prime\prime\prime}, h, \sigma \vdash C[e^{\prime}] : F^{\prime\prime\prime}$	(
case (1) last was (VAR) (THIS) (ADDR) (NULL) (NEW) (SPAWN) <i>Contradicts (6)</i>	(13)+(9)+(10)+(11)+(12)	\Rightarrow	$\mathbb{L}, h, \sigma \vdash C[e'] : F'$	
Contraarcts (b)	case (1) last was (VAR) (The Constant of Constant o	HIS)	(Addr) (Null) (New) (Spawn)	
	Contraaicts (b)			

$$\begin{array}{l} \forall t', \sigma': h, \sigma' \vdash e: t' \Rightarrow h, \sigma' \vdash e': t' \\ h, \sigma \vdash C[e]: t \end{array} \right\} \Longrightarrow h, \sigma \vdash C[e']: t$$

$$\begin{array}{l} \text{Let} \qquad \forall t', \sigma': h, \sigma' \vdash e: t' \Rightarrow h, \sigma' \vdash e': t' \\ h, \sigma \vdash C[e]: t \end{array}$$

$$\begin{array}{l} (1) \\ (2) \end{array}$$

$ \begin{array}{c} (2) \Rightarrow h, \sigma \vdash e:t \\ (1) \Rightarrow h, \sigma \vdash e:t \\ (2) \Rightarrow h, \sigma \vdash C[e']:t \\ \hline \\ \hline \\ \hline \\ case (2) last was (FRAME) \\ \hline \\ (4) \Rightarrow C[e] = frame \sigma'' e'' \\ (5) \\ h, \sigma'' \vdash \sigma'' th'' \\ \hline \\ \\ \hline \\ \\ case (2) last was (FRAME) \\ \hline \\ (4) \Rightarrow C[e] = frame \sigma'' e'' \\ \hline \\ \\ \\ \hline \\ \hline \\ \\ \hline \\ \\ \hline \\ \\ \hline \\ \hline \\ \hline \\ \\ \hline \\ \hline \\ \hline \\ \hline \\ \\ \hline \hline \\ \hline \\ \hline \\ \hline \\ \hline \hline \\ \hline \hline \\ \hline \\ \hline \\ \hline \\ \hline \hline \\ \hline \\ \hline \\ \hline \\ \hline \\ \hline \hline \\ \hline \\ \hline \\$	se $C[e] = e$	(3
$ \begin{array}{cccccc} (1) & \Rightarrow & h, \sigma \vdash C[e']:t \\ (5) & \Rightarrow & h, \sigma \vdash C[e']:t \\ (5) & \Rightarrow & h, \sigma \vdash C[e']:t \\ \hline case (2) last was (FRAME) & \Rightarrow & C[e] = frame \sigma'' e'' & (5) \\ & h, \sigma'' \vdash \sigma''(this):u & (4) \\ (4) & \Rightarrow & C[e] = frame \sigma'' e'' & (5) \\ & h, \sigma'' \vdash \sigma''(this):u & (7) \\ & t = u \triangleright t'' & (6) \\ & h, \sigma'' \vdash \sigma''(this):u & (7) \\ & t = u \triangleright t'' & (8) \\ \hline (5) + (3) & \Rightarrow & \exists c'' : e'' = C''[e] & (9) \\ (1) + (6) + (9) + III & \Rightarrow & h, \sigma \vdash C''[e']:t' & (10) \\ (10) + (7) + (8) + (FRAME) & \Rightarrow & h, \sigma \vdash frame \sigma'' C''[e']:t & (11) \\ (11) + (9) + (5) & \Rightarrow & h, \sigma \vdash C[e']:t \\ \hline case (2) last was (CALL) & \Rightarrow & h, \sigma \vdash C[e']:t \\ \hline case (2) last was (CALL) & \Rightarrow & h, \sigma \vdash C''[e']:u c & (6) \\ & \mathcal{M}(c,m) = t m(u \triangleright t_a) & (7) \\ & h, \sigma \vdash e'': t_a & (8) \\ \hline \hline case e'' = C'[e] & (9) \\ \hline (6) + (9) + (1) + III & \Rightarrow & h, \sigma \vdash C'[e']:u c & (10) \\ (10) + (7) + (8) + (CALL) & \Rightarrow & h, \sigma \vdash C'[e']:u c & (10) \\ \hline (10) + (7) + (8) + (CALL) & \Rightarrow & h, \sigma \vdash C'[e']:u c & (10) \\ \hline (10) + (7) + (8) + (CALL) & \Rightarrow & h, \sigma \vdash C'[e']:u c & (10) \\ \hline case e'' = C'[e] & (9) \\ \hline very similar & \Box \\ \hline case (2) last was (FIELD) (SYNC) (ASSIGN) (SYNCED) (CAST) & (4) \\ Similar to (CALL) & \Rightarrow & h, \sigma \vdash C[e]:t' & (5) \\ \hline t' < t & (6) \\ \hline (5) + (1) + III \Rightarrow & h, \sigma \vdash C[e]:t' & (7) \\ \hline case (2) last was (SUB) & (4) \\ \hline (4) & \Rightarrow & h, \sigma \vdash C[e]:t' & (5) \\ \hline t' < t & (6) \\ \hline (5) + (1) + III \Rightarrow & h, \sigma \vdash C[e']:t' & (7) \\ \hline case (2) last was (NULL) (VAR) (THIS) (ADDR) (NEW) (SPAWN) & (4) \\ \hline contradict (3) & \Box \\ \hline \end{array}$	(+) $(+)$	(4
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	$ (+(1) \implies h, \sigma \vdash e' : t $	(5
$\begin{array}{c} cC[e] \neq e \\ \hline case (2) \text{ last was (FRAME)} & (4) \\ (4) & \Rightarrow & C[e] = \texttt{frame } \sigma'' e'' & (5) \\ & h, \sigma'' \vdash e'' : t'' & (6) \\ & h, \sigma'' \vdash e'' : t'' & (6) \\ & h, \sigma'' \vdash e'' : t'' & (6) \\ & h, \sigma' \vdash \sigma'' (\texttt{this}) : u _ & (7) \\ & t = u \geqslant t'' & (8) \\ \hline (5) + (3) & \Rightarrow & \exists c'' : e'' = C''[e] & (9) \\ (1) + (6) + (9) + \coprod & \Rightarrow & h, \sigma \vdash C'[e'] : t' & (10) \\ (10) + (7) + (8) + (FRAME) \Rightarrow & h, \sigma \vdash frame \sigma'' C''[e'] : t & (11) \\ (11) + (9) + (5) & \Rightarrow & h, \sigma \vdash C[e'] : t & (11) \\ \hline (11) + (9) + (5) & \Rightarrow & h, \sigma \vdash C[e'] : t & (11) \\ \hline (11) + (9) + (5) & \Rightarrow & h, \sigma \vdash C[e'] : u c & (6) \\ & \mathcal{M}(c,m) = t \ m(u \geqslant t_a) & (7) \\ & h, \sigma \vdash e'' : u c & (6) \\ & \mathcal{M}(c,m) = t \ m(u \geqslant t_a) & (7) \\ & h, \sigma \vdash e''' : t_a & (8) \\ \hline \hline case \ e'' = C'[e] & (9) \\ \hline (10) + (7) + (8) + (CALL) \Rightarrow & h, \sigma \vdash C'[e'] : u c & (10) \\ \hline (10) + (7) + (8) + (CALL) \Rightarrow & h, \sigma \vdash C'[e'] : u c & (11) \\ \hline (11) + (5) + (9) \Rightarrow & h, \sigma \vdash C[e'] : t & (11) \\ \hline case \ e'' = C'[e] & (9) \\ \hline very \ similar & \hline \\ \hline case \ (2) \ last \ was \ (FIELD) \ (SYNC) \ (ASSIGN) \ (SYNCED) \ (CAST) & (4) \\ \hline similar \ to \ (CALL) & \hline \\ case \ (2) \ last \ was \ (SUB) & (4) \\ \hline (4) \ \Rightarrow \ h, \sigma \vdash C[e] : t' & (5) \\ & t' < t & (6) \\ \hline (5) + (1) + \coprod \ \Rightarrow \ h, \sigma \vdash C[e] : t' & (5) \\ & t' < t & (6) \\ \hline (5) + (1) + \coprod \ \Rightarrow \ h, \sigma \vdash C[e] : t' & (7) \\ \hline (7) + (6) + (SUB) \Rightarrow \ h, \sigma \vdash C[e] : t' & (7) \\ \hline (7) + (6) + (SUB) \Rightarrow \ h, \sigma \vdash C[e] : t' & (7) \\ \hline case \ (2) \ last \ was \ (NULL) \ (VAR) \ (THIS) \ (ADDR) \ (NEW) \ (SPAWN) & (4) \\ \hline contradict \ (3) \\ \hline \hline \end{array}$	$+(5) \Rightarrow n, o \vdash C[e]: i$	
$\begin{array}{c} \mbox{case (2) last was (FRAME)} & (4) \\ (4) & \Rightarrow & C[e] = frame \sigma'' e'' & (5) \\ & h, \sigma'' \vdash e'' : t'' & (6) \\ & h, \sigma'' \vdash e'' : t'' & (6) \\ & h, \sigma'' \vdash e'' : t'' & (7) \\ & t = u \geqslant t'' & (8) \\ (5) + (3) & \Rightarrow & \exists c'' : e'' = C''[e] & (9) \\ (1) + (6) + (9) + IH & \Rightarrow & h, \sigma \vdash C''[e'] : t' & (10) \\ (10) + (7) + (8) + (FRAME) & \Rightarrow & h, \sigma \vdash C''[e'] : t' & (11) \\ (10) + (7) + (8) + (FRAME) & \Rightarrow & h, \sigma \vdash C''[e'] : t' & (11) \\ (11) + (9) + (5) & \Rightarrow & h, \sigma \vdash C'[e'] : t & (11) \\ (4) & \Rightarrow & C[e] = e''.m(e''') & (5) \\ & h, \sigma \vdash e'' : u c & (6) \\ & \mathcal{M}(c,m) = t m(u \geqslant t_a) & (7) \\ & h, \sigma \vdash e'' : u c & (6) \\ & \mathcal{M}(c,m) = t m(u \geqslant t_a) & (7) \\ & h, \sigma \vdash e'' : t_a & (8) \\ \hline & \hline$	se $C[e] \neq e$	(3)
$ \begin{array}{cccccccccccccccccccccccccccccccccccc$	case (2) last was (FRAME)	(4)
$\begin{array}{c} h, \sigma'' \vdash e'': t'' & (6) \\ h, \sigma'' \vdash \sigma'' (this): u & (7) \\ t = u \geqslant t'' & (8) \\ (5)+(3) & \Rightarrow \exists c'': e'' = C''[e] & (9) \\ (1)+(6)+(9)+\amalg & \Rightarrow & h, \sigma \vdash C''[e']: t'' & (10) \\ (10)+(7)+(8)+(FRAME) & \Rightarrow & h, \sigma \vdash frame \sigma'' C''[e']: t & (11) \\ (11)+(9)+(5) & \Rightarrow & h, \sigma \vdash C[e']: t & (11) \\ (11)+(9)+(5) & \Rightarrow & h, \sigma \vdash C[e']: t & (11) \\ \hline \\ case (2) last was (CALL) & (4) \\ (4) & \Rightarrow & C[e] = e''.m(e''') & (5) \\ h, \sigma \vdash e'': u c & (6) \\ M(c, m) = t m(u \geqslant t_a) & (7) \\ h, \sigma \vdash e'': t_a & (8) \\ \hline \\ \hline \\ case e'' = C'[e] & (9) \\ (6)+(9)+(1)+\amalg & \Rightarrow & h, \sigma \vdash C'[e']: u c & (10) \\ (10)+(7)+(8)+(CALL) & \Rightarrow & h, \sigma \vdash C'[e']: u c & (10) \\ (10)+(7)+(8)+(CALL) & \Rightarrow & h, \sigma \vdash C'[e']: t & (11) \\ \hline \\ case e''' = C'[e] & (9) \\ very similar & \hline \\ \hline \\ case (2) last was (FIELD) (SYNC) (ASSIGN) (SYNCED) (CAST) & (4) \\ Similar to (CALL) & \hline \\ case (2) last was (SUB) & (4) \\ (4) & \Rightarrow & h, \sigma \vdash C[e]: t' & (5) \\ t' < t & (6) \\ (5)+(1)+\amalg & \Rightarrow & h, \sigma \vdash C[e]: t' & (5) \\ t' < t & (6) \\ (5)+(1)+\amalg & \Rightarrow & h, \sigma \vdash C[e']: t & (7) \\ \hline \\ case (2) last was (NULL) (VAR) (THIS) (ADDR) (NEW) (SPAWN) & (4) \\ Contradict (3) & \hline \\ \hline \end{array}$	(4) $\Rightarrow C[e] = \texttt{frame } \sigma'' e''$	(5)
$\begin{array}{c} h, \sigma'' \vdash \sigma''(\operatorname{this}) : u _ (7) \\ t = u \triangleright t'' & (8) \\ (5) + (3) \Rightarrow \exists c'' : e'' = C''[e] & (9) \\ (1) + (6) + (9) + \amalg \Rightarrow h, \sigma \vdash C''[e'] : t'' & (10) \\ (10) + (7) + (8) + (\operatorname{FRAME}) \Rightarrow h, \sigma \vdash \operatorname{frame} \sigma'' C''[e'] : t & (11) \\ (11) + (9) + (5) \Rightarrow h, \sigma \vdash C[e'] : t & (11) \\ (11) + (9) + (5) \Rightarrow h, \sigma \vdash C[e'] : t & (11) \\ (11) + (9) + (5) \Rightarrow h, \sigma \vdash C[e'] : t & (11) \\ (11) + (9) + (5) \Rightarrow h, \sigma \vdash C'[e] = e''.m(e''') & (5) \\ h, \sigma \vdash e''' : u c & (6) \\ \mathcal{M}(c, m) = t m(u \triangleright t_a) & (7) \\ h, \sigma \vdash e''' : t_a & (8) \\ \hline case \ e'' = C'[e] & (9) \\ (10) + (7) + (8) + (CALL) \Rightarrow h, \sigma \vdash C'[e'] : u c & (10) \\ (10) + (7) + (8) + (CALL) \Rightarrow h, \sigma \vdash C'[e'] .m(e''') : t & (11) \\ (11) + (5) + (9) \Rightarrow h, \sigma \vdash C'[e'] .m(e''') : t & (11) \\ \hline case \ e''' = C'[e] & (9) \\ very \ similar \\ \hline \hline case \ (2) \ last \ was \ (FIELD) \ (SYNC) \ (ASSIGN) \ (SYNCED) \ (CAST) & (4) \\ Similar \ to \ (CALL) & (4) \\ \hline d(4) \Rightarrow h, \sigma \vdash C[e] : t' & (5) \\ t' < t & (6) \\ (5) + (1) + \amalg \Rightarrow h, \sigma \vdash C[e] : t' & (5) \\ t' < t & (6) \\ \hline (5) + (1) + \amalg \Rightarrow h, \sigma \vdash C[e'] : t' & (7) \\ \hline case \ (2) \ last \ was \ (NULL) \ (VAR) \ (THIS) \ (ADDR) \ (NEW) \ (SPAWN) & (4) \\ \hline contradict \ (3) \\ \hline \end{array}$	$h,\sigma'' \vdash e'':t''$	(6)
$\begin{array}{c} t = u \triangleright t'' \qquad (8) \\ (5)+(3) \qquad \Rightarrow \exists c'':e'' = C''[e] \qquad (9) \\ (1)+(6)+(9)+\amalg \qquad \Rightarrow h, \sigma \vdash C'[e']:t'' \qquad (10) \\ (10)+(7)+(8)+(FRAME) \Rightarrow h, \sigma \vdash frame \sigma'' C''[e']:t \qquad (11) \\ (11)+(9)+(5) \qquad \Rightarrow h, \sigma \vdash frame \sigma'' C''[e']:t \qquad (11) \\ (11)+(9)+(5) \qquad \Rightarrow h, \sigma \vdash C[e']:t \qquad (4) \\ (4) \qquad \Rightarrow C[e] = e''.m(e''') \qquad (5) \\ h, \sigma \vdash e'':u c \qquad (6) \\ \mathcal{M}(c,m) = t m(u \triangleright t_a) \qquad (7) \\ h, \sigma \vdash e''':t_a \qquad (8) \\ \hline \hline case e'' = C'[e] \qquad (9) \\ (6)+(9)+(1)+\amalg \Rightarrow h, \sigma \vdash C'[e']:u c \qquad (10) \\ (10)+(7)+(8)+(CALL) \Rightarrow h, \sigma \vdash C'[e'].m(e'''):t \qquad (11) \\ (11)+(5)+(9) \qquad \Rightarrow h, \sigma \vdash C'[e']:t \qquad (11) \\ \hline case e''' = C'[e] \qquad (9) \\ very similar \qquad \Box \\ \hline \hline case (2) last was (FIELD) (SYNC) (ASSIGN) (SYNCED) (CAST) \qquad (4) \\ Similar to (CALL) \qquad \Box \\ case (2) last was (FIELD) (SYNC) (ASSIGN) (SYNCED) (CAST) \qquad (4) \\ Similar to (CALL) \qquad \Box \\ case (2) last was (SUB) \qquad (4) \\ (4) \qquad \Rightarrow h, \sigma \vdash C[e]:t' \qquad (5) \\ t' < t \qquad (6) \\ (5)+(1)+\amalg \Rightarrow h, \sigma \vdash C[e]:t' \qquad (5) \\ t' < t \qquad (6) \\ (5)+(1)+\amalg \Rightarrow h, \sigma \vdash C[e']:t \qquad (7) \\ (7)+(6)+(SUB) \Rightarrow h, \sigma \vdash C[e']:t \qquad (7) \\ case (2) last was (NULL) (VAR) (THIS) (ADDR) (NEW) (SPAWN) \qquad (4) \\ Contradict (3) \\ \hline \end{array}$	$h,\sigma''dash\sigma''(this):u$ _	(7)
$ \begin{array}{cccccccccccccccccccccccccccccccccccc$	$t = u \triangleright t''$	(8)
$\begin{array}{cccc} (1)+(6)+(9)+\underline{H} & \Rightarrow & h, \sigma \vdash C''[c']:t' & (10) \\ (10)+(7)+(8)+(FRAME) & \Rightarrow & h, \sigma \vdash frame \sigma'' C''[e']:t & (11) \\ (11)+(9)+(5) & \Rightarrow & h, \sigma \vdash C[e']:t & (11) \\ (11)+(9)+(5) & \Rightarrow & h, \sigma \vdash C[e']:t & (11) \\ (4) & \Rightarrow & C[e] = e''.m(e''') & (5) \\ & h, \sigma \vdash e'':u & c & (6) \\ & \mathcal{M}(c,m) = t & m(u \vDash t_a) & (7) \\ & h, \sigma \vdash e''':t_a & (8) \\ \hline & \hline$	$(5)+(3) \qquad \Rightarrow \exists c'': e'' = C''[e]$	(9)
$\begin{array}{cccccccccccccccccccccccccccccccccccc$	$(1)+(6)+(9)+\underline{\mathrm{IH}} \qquad \Rightarrow h,\sigma \vdash C''[e']:t''$	(10)
$\begin{array}{cccc} (11)+(9)+(3) & \Rightarrow & h, \sigma \vdash C[e']:t \\ \hline \\ \hline \\ case (2) last was (CALL) & (4) \\ (4) & \Rightarrow & C[e] = e''.m(e''') & (5) \\ & h, \sigma \vdash e'':u c & (6) \\ & \mathcal{M}(c,m) = t m(u \vDash t_a) & (7) \\ & h, \sigma \vdash e''':t_a & (8) \\ \hline \\ \hline \\ \hline \\ case e'' = C'[e] & (9) \\ (6)+(9)+(1)+IH & \Rightarrow & h, \sigma \vdash C'[e']:u c & (10) \\ (10)+(7)+(8)+(CALL) & \Rightarrow & h, \sigma \vdash C'[e'].m(e'''):t & (11) \\ (11)+(5)+(9) & \Rightarrow & h, \sigma \vdash C'[e']:t & (11) \\ \hline \\ \hline \\ case e''' = C'[e] & (9) \\ \hline \\ very similar & (9) \\ \hline \\ case (2) last was (FIELD) (SYNC) (ASSIGN) (SYNCED) (CAST) & (4) \\ Similar to (CALL) & \hline \\ \hline \\ case (2) last was (SUB) & (4) \\ (4) & \Rightarrow & h, \sigma \vdash C[e]:t' & (5) \\ & t' < t & (6) \\ (5)+(1)+IH & \Rightarrow & h, \sigma \vdash C[e']:t' & (7) \\ (7)+(6)+(SUB) & \Rightarrow & h, \sigma \vdash C[e']:t & (7) \\ \hline \\ case (2) last was (NULL) (VAR) (THIS) (ADDR) (NEW) (SPAWN) & (4) \\ \hline \\ \hline \\ \hline \\ \hline \end{array}$	$(10)+(7)+(8)+(FRAME) \implies h, \sigma \vdash \text{frame } \sigma'' C'''[e']:t$	(11)
$\begin{array}{c} \operatorname{case}\left(2\right) \operatorname{last} \operatorname{was}\left(\operatorname{CALL}\right) & (4) \\ (4) & \Rightarrow & C[e] = e''.m(e''') & (5) \\ & h, \sigma \vdash e'': u \ c & (6) \\ & \mathcal{M}(c,m) = t \ m(u \vDash t_a) & (7) \\ & h, \sigma \vdash e''': t_a & (8) \end{array}$ $\begin{array}{c} \overline{\operatorname{case}\ e'' = C'[e]} & (9) \\ (6) + (9) + (1) + \underline{\mathrm{H}} & \Rightarrow \ h, \sigma \vdash C'[e']: u \ c & (10) \\ (10) + (7) + (8) + (\operatorname{CALL}) & \Rightarrow \ h, \sigma \vdash C'[e'] : m(e'''): t & (11) \\ (11) + (5) + (9) & \Rightarrow \ h, \sigma \vdash C'[e'] : t & (11) \\ \hline (11) + (5) + (9) & \Rightarrow \ h, \sigma \vdash C[e']: t & (2) \\ \hline \operatorname{case}\ e''' = C'[e] & (9) \\ very\ similar & (2) \operatorname{last}\ was\ (\mathrm{FIELD})\ (\mathrm{SYNC})\ (\mathrm{ASSIGN})\ (\mathrm{SYNCED})\ (\mathrm{CAST}) & (4) \\ Similar\ to\ (\mathrm{CALL}) & & t' < t & (6) \\ (5) + (1) + \underline{\mathrm{H}} & \Rightarrow \ h, \sigma \vdash C[e]: t' & (5) \\ & t' < t & (6) \\ (5) + (1) + \underline{\mathrm{H}} & \Rightarrow \ h, \sigma \vdash C[e']: t' & (7) \\ (7) + (6) + (\mathrm{SUB}) & \Rightarrow \ h, \sigma \vdash C[e']: t' & (7) \\ (7) + (6) + (\mathrm{SUB}) & \Rightarrow \ h, \sigma \vdash C[e']: t & (7) \\ \operatorname{case}\ (2) \operatorname{last}\ was\ (\mathrm{NULL})\ (\mathrm{VAR})\ (\mathrm{THIS})\ (\mathrm{ADDR})\ (\mathrm{NEW})\ (\mathrm{SPAWN}) & (4) \\ \hline \end{array}$	$(11)+(9)+(5) \qquad \Rightarrow h, \sigma \vdash C[e^{\epsilon}]:t$	
$(4) \qquad \Rightarrow C[e] = e''.m(e''') \qquad (5) \\ h, \sigma \vdash e'': u c \qquad (6) \\ \mathcal{M}(c,m) = t m(u \bowtie t_a) \qquad (7) \\ h, \sigma \vdash e''': t_a \qquad (8) \\ \hline case e'' = C'[e] \qquad (9) \\ (6) + (9) + (1) + \underline{H} \Rightarrow h, \sigma \vdash C'[e']: u c \qquad (10) \\ (10) + (7) + (8) + (CALL) \Rightarrow h, \sigma \vdash C'[e']: m(e'''): t \qquad (11) \\ (11) + (5) + (9) \Rightarrow h, \sigma \vdash C'[e']: m(e'''): t \qquad (11) \\ (11) + (5) + (9) \Rightarrow h, \sigma \vdash C[e']: t \qquad (9) \\ \hline case e''' = C'[e] \qquad (9) \\ very similar \qquad \qquad$	case (2) last was (CALL)	(4)
$\begin{array}{c} h, \sigma \vdash e'': u \ c & (6) \\ \mathcal{M}(c,m) = t \ m(u \vDash t_a) & (7) \\ h, \sigma \vdash e''': t_a & (8) \end{array}$ $\hline \begin{array}{c} case \ e'' = C'[e] & (9) \\ (6) + (9) + (1) + \underline{H} & \Rightarrow \ h, \sigma \vdash C'[e']: u \ c & (10) \\ (10) + (7) + (8) + (CALL) & \Rightarrow \ h, \sigma \vdash C'[e']: m(e'''): t & (11) \\ (11) + (5) + (9) & \Rightarrow \ h, \sigma \vdash C[e']: t & (11) \end{array}$ $\hline \begin{array}{c} case \ e''' = C'[e] & (9) \\ very \ similar & \end{array}$ $\hline \begin{array}{c} \hline case \ e''' = C'[e] & (9) \\ very \ similar & very \ similar & very \ similar & (2ALL) & (2ALL) & (2ASIGN) \ (SYNCED) \ (CAST) & (4) \\ Similar \ to \ (CALL) & (2ALL) & (2ASIGN) \ (SYNCED) \ (CAST) & (4) \\ \hline \begin{array}{c} case \ (2) \ last \ was \ (SUB) & (10) \\ (4) & \Rightarrow \ h, \sigma \vdash C[e]: t' & (5) \\ t' < t & (6) \\ (5) + (1) + \underline{H} & \Rightarrow \ h, \sigma \vdash C[e']: t' & (5) \\ (7) + (6) + (SUB) & \Rightarrow \ h, \sigma \vdash C[e']: t & (7) \\ \hline \end{array}$	(4) $\Rightarrow C[e] = e''.m(e''')$	(5)
$ \begin{aligned} & \mathcal{M}(c,m) = t \ m(u \bowtie t_a) & (7) \\ & h, \sigma \vdash e''' : t_a & (8) \end{aligned} \\ \hline \hline & case \ e'' = C'[e] & (9) \\ & (6) + (9) + (1) + IH \Rightarrow h, \sigma \vdash C'[e'] : u \ c & (10) \\ & (10) + (7) + (8) + (CALL) \Rightarrow h, \sigma \vdash C'[e'] : m(e''') : t & (11) \\ & (11) + (5) + (9) \Rightarrow h, \sigma \vdash C[e'] : t & (11) \\ & (11) + (5) + (9) \Rightarrow h, \sigma \vdash C[e'] : t & (9) \\ \hline & case \ e''' = C'[e] & (9) \\ & very \ similar & (9) \\ \hline & case \ e''' = C'[e] & (9) \\ & very \ similar & (9) \\ \hline & case \ (2) \ last \ was \ (FIELD) \ (SYNC) \ (ASSIGN) \ (SYNCED) \ (CAST) & (4) \\ & Similar \ to \ (CALL) & (4) \\ & (4) & \Rightarrow \ h, \sigma \vdash C[e] : t' & (5) \\ & t' < t & (6) \\ & (5) + (1) + IH \ \Rightarrow \ h, \sigma \vdash C[e'] : t' & (6) \\ & (5) + (1) + IH \ \Rightarrow \ h, \sigma \vdash C[e'] : t' & (7) \\ & (7) + (6) + (SUB) \ \Rightarrow \ h, \sigma \vdash C[e'] : t & (7) \\ \hline & case \ (2) \ last \ was \ (NULL) \ (VAR) \ (THIS) \ (ADDR) \ (NEW) \ (SPAWN) & (4) \\ & Contradict \ (3) \\ \hline \hline & \Box & \Box & \Box & \Box & \Box \\ \hline & \Box \\ \hline & \Box &$	$h,\sigma dash e'': u \; c$	(6)
$h, \sigma \vdash e''' : t_a $ (8) $case e'' = C'[e] \qquad (9) (6) + (9) + (1) + IH \Rightarrow h, \sigma \vdash C'[e'] : u c \qquad (10) (10) + (7) + (8) + (CALL) \Rightarrow h, \sigma \vdash C'[e'] : m(e''') : t \qquad (11) (11) + (5) + (9) \Rightarrow h, \sigma \vdash C[e'] : t \qquad (11) (11) + (5) + (9) \Rightarrow h, \sigma \vdash C[e'] : t \qquad (9) case e''' = C'[e] \qquad (9) very similar \qquad (9) case (2) last was (FIELD) (SYNC) (ASSIGN) (SYNCED) (CAST) \qquad (4) Similar to (CALL) case (2) last was (SUB) \qquad (4) (4) \Rightarrow h, \sigma \vdash C[e] : t' \qquad (5) t' < t \qquad (6) (5) + (1) + IH \Rightarrow h, \sigma \vdash C[e'] : t' \qquad (6) (5) + (1) + IH \Rightarrow h, \sigma \vdash C[e'] : t' \qquad (7) (7) + (6) + (SUB) \Rightarrow h, \sigma \vdash C[e'] : t \qquad (7) case (2) last was (NULL) (VAR) (THIS) (ADDR) (NEW) (SPAWN) \qquad (4) Contradict (3) $	$\mathcal{M}(c,m) = t \ m(u \bowtie t_a)$	(7)
$\begin{array}{c} \hline \operatorname{case} e'' = C'[e] & (9) \\ (6) + (9) + (1) + \underline{\mathrm{H}} & \Rightarrow h, \sigma \vdash C'[e'] : u c & (10) \\ (10) + (7) + (8) + (\operatorname{CALL}) & \Rightarrow h, \sigma \vdash C'[e'] . m(e''') : t & (11) \\ (11) + (5) + (9) & \Rightarrow h, \sigma \vdash C[e'] : t & (9) \\ \hline \\ \hline \\ \hline \\ \operatorname{case} e''' = C'[e] & (9) \\ very similar & & & \\ \hline \\ \operatorname{case} (2) \text{ last was (FIELD) (SYNC) (ASSIGN) (SYNCED) (CAST)} & (4) \\ \\ Similar to (CALL) & & & \\ \operatorname{case} (2) \text{ last was (SUB)} & (4) \\ (4) & \Rightarrow h, \sigma \vdash C[e] : t' & (5) \\ & t' < t & (6) \\ (5) + (1) + \underline{\mathrm{H}} & \Rightarrow h, \sigma \vdash C[e'] : t' & (7) \\ (7) + (6) + (\mathrm{SUB}) & \Rightarrow h, \sigma \vdash C[e'] : t & & \\ \hline \\ \operatorname{case} (2) \text{ last was (NULL) (VAR) (THIS) (ADDR) (NEW) (SPAWN)} & (4) \\ \hline \\ \hline \\ \hline \\ \hline \end{array}$	$h,\sigma dash e''':t_a$	(8)
$(6)+(9)+(1)+\underline{\mathbf{H}} \implies h, \sigma \vdash C'[e'] : u c $ (10) $(10)+(7)+(8)+(CALL) \implies h, \sigma \vdash C'[e'].m(e''') : t $ (11) $(11)+(5)+(9) \implies h, \sigma \vdash C[e'] : t $ (9) $very similar $ (9) $very similar $ (10) (11)	case $e'' = C'[e]$	(9)
$(10)+(7)+(8)+(CALL) \Rightarrow h, \sigma \vdash C'[e'].m(e'''):t $ (11) $(11)+(5)+(9) \Rightarrow h, \sigma \vdash C[e']:t $ (9) $very similar $ (9) $very similar $ (11) (11) $(11)+(5)+(9) \Rightarrow h, \sigma \vdash C[e']:t $ (9) (11) (11) (12) $(12$	$(6)+(9)+(1)+\underline{\mathrm{IH}} \qquad \Rightarrow h,\sigma \vdash C'[e']: u \ c$	(10)
$(11)+(5)+(9) \Rightarrow h, \sigma \vdash C[e']:t$ (9) $very similar$ (9) $very similar$ $(11)+(5)+(9) \Rightarrow h, \sigma \vdash C[e']:t$ $(11)+(1)+(1) \Rightarrow h, \sigma \vdash C[e]:t'$ $(11)+(1) \Rightarrow h, \sigma \vdash C[e]:t'$ $(11)+(1) \Rightarrow h, \sigma \vdash C[e']:t'$ $(11)+(1) \Rightarrow h, \sigma \vdash C[e'$	$(10) + (7) + (8) + (CALL) \Rightarrow h, \sigma \vdash C'[e'].m(e'''):t$	(11)
case $e''' = C'[e]$ very similar(9) \vdots case (2) last was (FIELD) (SYNC) (ASSIGN) (SYNCED) (CAST)(4)Similar to (CALL)(4)case (2) last was (SUB) (4) (4)(4) \Rightarrow $h, \sigma \vdash C[e] : t'$ $t' < t$ (5) $t' < t$ (5)+(1)+III $(7)+(6)+(SUB)$ \Rightarrow $h, \sigma \vdash C[e'] : t'$ $(7)+(6)+(SUB)$ (7)case (2) last was (NULL) (VAR) (THIS) (ADDR) (NEW) (SPAWN)(4) $Contradict (3)$	$(11)+(5)+(9) \qquad \Rightarrow h, \sigma \vdash C[e']:t$	
$\frac{very \ similar}{\Box}$ $\frac{very \ similar}{\Box}$ $case (2) \ last \ was \ (FIELD) \ (SYNC) \ (A \ SSIGN) \ (SYNCED) \ (CAST) \qquad (4)$ $Similar \ to \ (CALL)$ (4) (4) (4) (4) (4) (4) (4) (4) (4) (5) $t' < t \qquad (5)$ $(5)+(1)+IH \Rightarrow h, \sigma \vdash C[e] : t' \qquad (5)$ (7) $(7)+(6)+(SUB) \Rightarrow h, \sigma \vdash C[e'] : t' \qquad (7)$ $(7)+(6)+(SUB) \Rightarrow h, \sigma \vdash C[e'] : t$ $(ase \ (2) \ last \ was \ (NULL) \ (VAR) \ (THIS) \ (ADDR) \ (NEW) \ (SPAWN) \qquad (4)$ $Contradict \ (3)$	case $e^{\prime\prime\prime} = C^{\prime}[e]$	(9)
Case (2) last was (FIELD) (SYNC) (ASSIGN) (SYNCED) (CAST)(4)Similar to (CALL)(4)(4) \Rightarrow $h, \sigma \vdash C[e] : t'$ (5) $t' < t$ (5)+(1)+IH \Rightarrow $h, \sigma \vdash C[e'] : t'$ (7)+(6)+(SUB) \Rightarrow $h, \sigma \vdash C[e'] : t$ (2) last was (NULL) (VAR) (THIS) (ADDR) (NEW) (SPAWN)(4)(5)(7)(7)(7)(7)(1)(2)(3)	very similar	
$\begin{array}{c} \text{case (2) last was (FIELD) (SYNC) (ASSIGN) (SYNCED) (CAST)} & (4) \\ Similar to (CALL) & (4) \\ \text{(a)} & \Rightarrow h, \sigma \vdash C[e] : t' & (5) \\ & t' < t & (6) \\ (5) + (1) + \underline{\text{IH}} & \Rightarrow h, \sigma \vdash C[e'] : t' & (7) \\ (7) + (6) + (\text{SUB}) & \Rightarrow h, \sigma \vdash C[e'] : t & (7) \\ \text{case (2) last was (NULL) (VAR) (THIS) (ADDR) (NEW) (SPAWN)} & (4) \\ Contradict (3) & $		·
$\begin{array}{c} Similar \ to \ (CALL) \end{array} \tag{4}$ $\begin{array}{c} (4) \\ (4) \\ (4) \\ (4) \\ (4) \\ (5) \\ (5) \\ (7) \\ (6) \\ (5) \\ (7) \\ (6) \\ (5) \\ (7) \\ (6) \\ (SUB) \\ (7) \\ (6) \\ (SUB) \\ (7) \\ (6) \\ (Call \\ (SUB) \\ (7) \\ (6) \\ (Call \\ (SUB) \\ (7) \\ (7) \\ (6) \\ (SUB) \\ (7) \\ (7) \\ (6) \\ (SUB) \\ (7) \\ (7) \\ (6) \\ (SUB) \\ (7) \\ (7) \\ (7) \\ (6) \\ (SUB) \\ (7) \\ (7) \\ (7) \\ (6) \\ (SUB) \\ (7) \\ (7) \\ (7) \\ (7) \\ (6) \\ (SUB) \\ (7) \\ (7) \\ (7) \\ (7) \\ (6) \\ (SUB) \\ (7) \\ (7) \\ (7) \\ (7) \\ (7) \\ (7) \\ (6) \\ (SUB) \\ (7) \\ ($	case (2) last was (FIELD) (SYNC) (ASSIGN) (SYNCED) (CAST)	(4)
$\begin{array}{ccc} case \ (2) \ last \ was \ (SUB) & (4) \\ (4) & \Rightarrow & h, \sigma \vdash C[e] : t' \\ & t' < t \\ (5) \\ (5) + (1) + \underline{IH} & \Rightarrow & h, \sigma \vdash C[e'] : t' \\ (7) + (6) + (SUB) & \Rightarrow & h, \sigma \vdash C[e'] : t \\ case \ (2) \ last \ was \ (NULL) \ (VAR) \ (THIS) \ (ADDR) \ (NEW) \ (SPAWN) \\ Contradict \ (3) \end{array} $ (4)	Similar to (CALL)	
$ \begin{array}{cccc} (4) & \Rightarrow & h, \sigma \vdash C[e] : t' & (5) \\ & t' < t & (6) \\ (5) + (1) + \underline{\mathrm{I\!H}} & \Rightarrow & h, \sigma \vdash C[e'] : t' & (7) \\ (7) + (6) + (\mathrm{SUB}) & \Rightarrow & h, \sigma \vdash C[e'] : t & (7) \\ \hline & & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & & \\ & & & & & \\ & & & & \\ & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & & \\ & & & & \\ & & & & & \\ & & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & & & \\ & & &$	case (2) last was (SUB)	(4)
$t' < t $ (6) $(5)+(1)+\underline{IH} \Rightarrow h, \sigma \vdash C[e']:t'$ $(7)+(6)+(SUB) \Rightarrow h, \sigma \vdash C[e']:t$ (7) $(7)+(6)+(SUB) \Rightarrow h, \sigma \vdash C[e']:t$ (7) $(7$	(4) $\Rightarrow h, \sigma \vdash C[e] : t'$	(5)
$(5)+(1)+\underline{\mathrm{IH}} \Rightarrow h, \sigma \vdash C[e']:t' $ $(7)+(6)+(\mathrm{SUB}) \Rightarrow h, \sigma \vdash C[e']:t$ (7) $(7$	t' < t	(6)
$(7)+(6)+(SUB) \implies h, \sigma \vdash C[e']:t$ case (2) last was (NULL) (VAR) (THIS) (ADDR) (NEW) (SPAWN) (4) Contradict (3)	$(5)+(1)+\underline{\mathrm{IH}} \qquad \Rightarrow h,\sigma \vdash C[e']:t'$	(7)
case (2) last was (NULL) (VAR) (THIS) (ADDR) (NEW) (SPAWN) (4) Contradict (3)	$(7)+(6)+(SUB) \Rightarrow h, \sigma \vdash C[e']: t$	
Contradict (3)	case (2) last was (NULL) (VAR) (THIS) (ADDR) (NEW) (SPAWN)	(4)
	Contradict (3)	()
		·
		Г

$\left. \begin{array}{l} h, \sigma \vdash_{gb} a: l \\ h(a) \downarrow_1 = w \end{array} \right\} \Longrightarrow \forall a': h, a' \in \mathbb{R}$	$\sigma \vdash_{gb} a' : l \Rightarrow h(a') \downarrow_1 = w$	
Let	$\begin{array}{l} h, \sigma \vdash_{gb} a : l \\ h(a) \downarrow_1 = w \\ h, \sigma \vdash_{gb} a' : l \end{array}$	(1) (2) (3)
case (1) last was (1)	Univ)	(4)
(4)	$\Rightarrow \iota = u \\ h, \sigma \vdash a : u _$	(5) (6)

		u eq any	(7)
(7)+(5)+(4)+(3)	\Rightarrow	$h, \sigma \vdash a' : u$	(8)
(6)+(7)+(8)+Lemma A26	\Rightarrow	$h(a){\downarrow_1} = h(a'){\downarrow_1}$	(9)
$(2)\!+\!(9)$	\Rightarrow	$h(a'){\downarrow_1} = w$	(10)
case (1) last was (VAL)			(4)
(4)	\Rightarrow	l = p	(5)
		$h(\sigma, p) = a$	(6)
(5)+(4)+(3)	\Rightarrow	$h(\sigma, p) = u'$	(7)
(6)+(7)	\Rightarrow	a = a'	(8)
(-) (-)	`	h(a') = au	(n)

$h, \sigma \vdash \operatorname{sym} h(a) \downarrow_1 = 0$	$\left. \begin{array}{c} \underset{e}{\operatorname{c}}_{e} \ a \ e' : t \\ w \end{array} \right\} \Longrightarrow l$	$h, \sigma \vdash$	$\texttt{synced}_e \ w \ e': t$	
	Let		$h, \sigma \vdash \texttt{sync}_e \ a \ e' : t$	(1)
			$h(a){\downarrow_1} = w$	(2)
	case (1) last was	(SUB	:)	(3)
	(3)	\Rightarrow	$h, \sigma \vdash \texttt{sync}_e \ a \ e' : t'$	(4)
			$t' \leq t$	(5)
	$(4)+(2)+{ m IH}$	\Rightarrow	$h, \sigma \vdash \texttt{synced}_e \ w \ e' : t'$	(6)
	(6)+(5)+(Sub)	\Rightarrow	$h,\sigma\vdash \texttt{synced}_e \ w \ e':t$	
	case (1) last was	(SYN	IC)	(3)
	(3)	\Rightarrow	$h, \sigma \vdash e': t$	(4)
	(4)+(Synced)	\Rightarrow	$h, \sigma \vdash \texttt{synced}_e \ a \ e' : t$	

Lemma A17

 $\begin{array}{c} \mathbb{L}, h, \sigma \vdash \texttt{sync}_e \ a \ e' : F \\ w = h(a) \downarrow_1 \end{array} \right\} \Longrightarrow \mathbb{L}, h, \sigma \vdash \texttt{synced}_e \ w \ e' : F \end{array}$

Let		$\mathbb{L}, h, \sigma \vdash \mathtt{sync}_e \ a \ e' : F$	(1)
		$w = h(a)\downarrow_1$	(2)
case (1) last was (SUB)			(3)
(3)	\Rightarrow	$\mathbb{L}', h, \sigma \vdash \mathtt{sync}_e \ a \ e' : F'$	(4)
		$\mathbb{L}' \subseteq \mathbb{L}$	(5)
		$F' \subseteq F$	(6)
		$\mathbb{L} \# F, \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p :$	(7)
$(4)+(2)+\overline{\mathrm{IH}}$	\Rightarrow	$\mathbb{L}', h, \sigma \vdash \mathtt{synced}_e \ w \ e' : F'$	(8)
(8)+(5)+(6)+(7)+(SUB)	\Rightarrow	$\mathbb{L}, h, \sigma \vdash \texttt{synced}_e^- w \ e' : F$	
case (1) last was (SYNC)			(3)
(3)	\Rightarrow	$h, \sigma \vdash_{ab} e: l$	(4)
		$h, \sigma \vdash_{ab} a : l$	(5)
		$\mathbb{L} \cup \{l\}, h, \sigma \vdash e' : F$	(6)
		$\mathbb{L}, h, \sigma \vdash e : F$	(7)
(2)+(7)+(5)+(4)+(6)+(Synced)	\Rightarrow	$\mathbb{L}, h, \sigma \vdash \texttt{synced}_e \ w \ e' : F$	

Lemma A18				
(h, σ, u, e, q) $_\vdash_, h \sim$ Virgin(e) Virgin(e')	$ \begin{array}{c} e') \bowtie l = l' \\ {\rightarrow} _, h' \\ {\rightarrow} \end{array} \right\} \Longrightarrow (h, \sigma, u, $	e, e')	ightarrow l=l'	
Le	t		$(h, \sigma, u, e, e') \bowtie l = l'$	(1)
			$_ \vdash _, n \rightsquigarrow _, n$ Vingin(a)	(2)
			V ingin(e) Vingin(a')	(3)
			v ii gin(e)	(4)
cas	se $l = u'$			(5)
(5))	\Rightarrow	$l' = u \triangleright u'$	(6)
			l' = tany	(7)
(5))+(6)+(7)	\Rightarrow	$(h',\sigma,u,e,e') \Vdash l = l'$	
cas	se $l = \mathbf{x} \cdot f_{1 \dots n}$			(5)
(5))	\Rightarrow	$l' = l[p/\mathbf{x}]$	(6)
× /			$h, \sigma \vdash_{ab} e' : p$	(7)
(7)	+(4)+(2)+Lemma 3.6.7	\Rightarrow	$h', \sigma \vdash_{qb} e' : p$	(8)
(5))+(8)+(6)	\Rightarrow	$(h,\sigma,u,e,e') \Vdash l = l'$	
cas	se $l = \texttt{this}. f_{1 \dots n}$			(5)
(5))	\Rightarrow	l' = l[p/this]	(6)
			$h, \sigma \vdash_{ab} e : p$	(7)
(7)	+(3)+(2)+Lemma 3.6.7	\Rightarrow	$h', \sigma \vdash_{gb} e : p$	(8)
(5))+(8)+(6)	\Rightarrow	$(h,\sigma,u,e,e') \mathrel{\blacktriangleright} l = l'$	

 $h, \sigma \vdash \texttt{synced}_e \ w \ v: t \Longrightarrow h, \sigma \vdash v: t$

Let	$h, \sigma \vdash \texttt{synced}_e \ w \ v: t$	(1)
case (1) last was (Su	3)	(()
$2) (2) \qquad \Rightarrow \qquad $	$h, \sigma \vdash \texttt{synced}_e w v: t'$	(3)
	$t' \leq t$	(4)
$(3)+\underline{\mathrm{IH}}$ \Rightarrow	$h, \sigma \vdash v: t'$	(5)
$(5)+(4)+(SUB) \Rightarrow$	$h,\sigma\vdash v:t$	
case (1) last was (SY	NCED)	(()
$2) (2) \qquad \Rightarrow \qquad \qquad$	$h, \sigma \vdash v: t$	

Lemma A20

 $\mathbb{L}, h, \sigma \vdash e : F \Longrightarrow \forall v : \mathbb{L}, h, \sigma \vdash v : F$

Let
$$\mathbb{L}, h, \sigma \vdash e : F$$
 (1)
(1)+Lemma A2 $\rightarrow \mathbb{L} \# F$ (2)

$$\begin{array}{cccc}
(1) + \operatorname{Lemma} A2 & \Rightarrow & \mathbb{L}\#F' & (2) \\ & & \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ & (3) \\ (\operatorname{NULL}) + (\operatorname{ADDR}) & \Rightarrow & \forall v : \emptyset, h, \sigma \vdash v : \emptyset & (4) \\ (4) + (2) + (3) + (\operatorname{SUB}) & \Rightarrow & \forall v : \mathbb{L}, h, \sigma \vdash v : F \end{array}$$

$$e \in Path \\ \forall \sigma', t' : h, \sigma' \vdash e' : t' \\ h, \sigma \vdash_{gb} C[e] : l \\ \end{pmatrix} \Longrightarrow h, \sigma \vdash_{gb} C[e'] : l \\ (1) \\ \forall \sigma', t' : h, \sigma' \vdash e' : t' \\ (2) \\ h, \sigma \vdash_{gb} C[e] : l \\ (3) \\ \hline case (3) last was (PATH) \\ Contradicts (1) \\ \hline case (3) last was (UNIV) \\ (4) \\ (4) \\ (4) \\ (4) \\ (4) \\ (4) \\ (4) \\ (4) \\ (4) \\ (4) \\ (5) \\ u \neq any \\ (6) \\ l = u \\ (7) \\ (2) + (5) + Lemma A14 \\ \Rightarrow h, \sigma \vdash C[e'] : u c \\ (8) \\ (8) + (6) + (7) + (UNIV) \\ \Rightarrow h, \sigma \vdash C[e'] : l \\ \hline$$

$\mathbf{Lemma} \ \mathbf{A22}$

 $\mathbb{L}, h, \sigma \vdash p.f: F \Longrightarrow \mathbb{L}, h, \sigma \vdash p: F$

Let		$\mathbb{L}, h, \sigma \vdash p.f : F$	(1)
case (1) last was (SUB)			(2)
(2)	\Rightarrow	$\mathbb{L}', h, \sigma \vdash p.f : F'$	(3)
		$\mathbb{L}' \subseteq \mathbb{L}$	(4)
		$F' \subseteq F$	(5)
		$\mathbb{L}\#F$	(6)
$(3)+\overline{\mathrm{IH}}$	\Rightarrow	$\mathbb{L}', h, \sigma \vdash p : F'$	(7)
(7)+(4)+(5)+(6)+(SUB)	\Rightarrow	$\mathbb{L}, h, \sigma \vdash p : F$	(8)
case (1) last was (FIELD)			(2)
(2)	\Rightarrow	$\mathbb{L}, h, \sigma \vdash p : F$	()

Lemma A23

 $\begin{array}{l} (h, \sigma, u, e', e'') \bowtie l' = l \\ \sigma' \vdash e, h \rightsquigarrow _, h' \\ \emptyset, h, \sigma' \vdash e : _ \\ \{h(a)\downarrow_1 | h, \sigma \vdash a : l, l \in \mathbb{L}\} \cap \{w | Locked(e, w)\} = \emptyset \\ l \in \mathbb{L} \\ \forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ \end{array} \right\} \Longrightarrow (h', \sigma, u, e', e'') \bowtie l' = l$

 Let

$$(h, \sigma, u, e', e'') \bowtie l' = l \tag{1}$$
$$\sigma' \vdash e, h \rightsquigarrow _, h' \tag{2}$$
$$\emptyset, h, \sigma' \vdash e : _ \tag{3}$$
$$\{h(a)\downarrow_1 | h, \sigma \vdash a : l, l \in \mathbb{L}\} \cap \{w | Locked(e, w)\} = \emptyset \tag{4}$$
$$l \in \mathbb{L} \tag{5}$$
$$\forall p \in \mathbb{L} : \mathbb{L}, h, \sigma \vdash p : _ \tag{6}$$

case l' = p'

1 .

	case $p' =$	$x.f_{1n}$			(8)
	(1)+(7)+	(8)	\Rightarrow	$l = p'[p''/\mathbf{x}]$	(9)
				$h, \sigma \vdash_{qb} e'' : p''$	(10)
	(7)+(8)+	-(9)	\Rightarrow	$l = p''.f_{1n}$	(11)
	(9)+(5)+	-(6)	\Rightarrow	$\mathbb{L}, h, \sigma \vdash l$:	(12)
	(11) + (12))+Lemma A22	\Rightarrow	$\mathbb{L}, h, \sigma \vdash p''$:	(13)
	(10)+(2)	+(3)+(4)+(13)+Lemma 3.6.10	\Rightarrow	$h', \sigma \vdash_{qb} e'' : \overline{p''}$	(14)
	(8)+(14)	+(9)+(7)	\Rightarrow	$(h',\sigma,u,e',e'') \Vdash l' = l$	
	case $p' = Very$	this. f_{1n} similar			(8)
					·
cas	l' = u'				(7)
(7)	$+(1) \Rightarrow$	$l = u \triangleright u'$			(8)
		l eq any			(9)
(8)	$+(9) \Rightarrow$	$(h',\sigma,u,e',e'') \Vdash l' = l$			、 <i>,</i>

Lemma A24

$\left. \begin{array}{l} h, \sigma' \vdash_{gb} a' : l' \\ l = (h, \sigma, u, a, v) \bowtie l' \\ h, \sigma \vdash a : u \\ \sigma' = (a, v) \end{array} \right\} \Longrightarrow$	$h, \sigma \vdash_{gb} a' : l$	
Let	$h, \sigma' \vdash_{ab} a' : l'$	(1)
	$l = (h, \sigma, u, a, v) \bowtie l'$	(2)
	$h, \sigma \vdash a: u$ _	(3)
	$\sigma' = (a, v)$	(4)
case (1) last was (UNIV)		(5)
(5)	$\Rightarrow l' = u'$	(6)
	$h, \sigma' \vdash a' : u'$	(7)
(\mathbf{C}) (\mathbf{Q})	$u \neq any$	(8)
(0)+(2)	$\Rightarrow l = u^{\prime}$	(9)
	$u = u \triangleright u$	(10) (11)
(3)+(4)+(7)+(10)+Lemma A8	$a \neq any$ $\Rightarrow h \sigma \vdash a' : u''$	(11) (12)
(12)+(9)+(11)+(UNIV)	$ \Rightarrow h, \sigma \vdash_{gb} a' : l $	(12)
case (1) last was (VAL)		(5)
(5)	$\Rightarrow l' = p'$	(6)
	$h(\sigma',p')=a'$	(7)
case $p' = \mathbf{x}.f_{1n}$		(8)
$(6)+(8)+(2) \qquad \Rightarrow l=p$	$p'[p/\mathtt{x}]$	(9)
h, σ	$\vdash_{gb} v: p$	(10)
$(10)+(VAL) \Rightarrow h(\sigma,$	p) = v	(11)
$(4) \qquad \Rightarrow h(\sigma')$	$(\mathbf{x}) = v$	(12)
$(11)+(12)+(7) \implies h(\sigma,$	l) = a'	(13)
$(13)+(\text{VAL}) \qquad \Rightarrow h,\sigma$	$\vdash_{gb} a':l$	
case $p' = \texttt{this}.f_{1n}$		(8)
Very Similar		
		•

Lemma A25

$\forall p \in \mathbb{L} : \\ _ \vdash _, h$	$ \begin{bmatrix} \mathbb{L}, h, \sigma \vdash p : - \\ n \rightsquigarrow _, h' \end{bmatrix} = $	$\Rightarrow \forall p \in \mathbb{L} : \mathbb{L}, h', \sigma \vdash p : _$	
Let	$\forall p \in \mathbb{L}: \mathbb{L}, h, \sigma$	$\vdash p:$ _	(1)
	$_\vdash_,h\leadsto_,$	h'	(2)
Let	$p \in \mathbb{L}$		(3)
(3)		$\Rightarrow Virgin(p)$	(4)
(1)+(3))	\Rightarrow L, $h, \sigma \vdash p$:	(5)
(4)+(5)	+(2)+Lemma 3.6.3	$8 \Rightarrow \mathbb{L}, h', \sigma \vdash p:$	(6)
$(3) \rightarrow (6) =$	$\Rightarrow \forall p \in \mathbb{L} : \mathbb{L}, h', c$	$\sigma \vdash p:$ _	

$\left. \begin{array}{c} h, \sigma \vdash a': u' \\ h, \sigma \vdash a': u' \\ u, u' \neq \text{any} \\ u = \texttt{rep} \Leftrightarrow u' = \texttt{rep} \end{array} \right\} \Longrightarrow h(a)\downarrow_1 = h(a')\downarrow_1$	
Let $h, \sigma \vdash a : u$ $h, \sigma \vdash a' : u'$ $u, u' \neq any$ $u = rep \Leftrightarrow u' = rep$	(1) (2) (3) (4)
$\begin{array}{l} \text{case (1) last was (SUB)} \\ (5) \Rightarrow h, \sigma \vdash a : u'' _ \\ u'' \leq u \end{array}$	(5) (6) (7)
$\begin{array}{lll} \mbox{case } u = \mbox{rep} \\ (8)+(4) & \Rightarrow & u' = \mbox{rep} \\ (8)+(7) & \Rightarrow & u'' = \mbox{rep} \\ (9)+(10) & \Rightarrow & u'' = \mbox{rep} \Leftrightarrow u' = \mbox{rep} \\ (9)+(10) & \Rightarrow & u, u' \neq \mbox{any} \\ (6)+(2)+(12)+(11)+\underline{\rm IH} & \Rightarrow & h(a){\downarrow}_1 = h(a'){\downarrow}_1 \end{array}$	(8)(9)(10)(11)(12)
$\begin{array}{ccc} case \ u \neq \texttt{rep} \\ (3)+(8)+(7) & \Rightarrow & u'' \neq \texttt{rep} \\ & & u'' \neq \texttt{any} \\ (8)+(4) & \Rightarrow & u' \neq \texttt{rep} \\ (9)+(11) & \Rightarrow & u'' = \texttt{rep} \Leftrightarrow u' = \texttt{rep} \\ (6)+(2)+(10)+(3)+(12)+\underline{IH} & \Rightarrow & h(a)\downarrow_1 = h(a')\downarrow_1 \end{array}$	(8)(9)(10)(11)(12)
case (2) last was (SUB) As above	. (5)
case (1) and (2) last were (ADDR) (5) $\Rightarrow a'', v \vdash a, h(a) \downarrow_1 : u$ $a'', v \vdash a', h(a') \downarrow_1 : u'$ $a'' = \sigma(\texttt{this})$ $v = h(a'') \downarrow_1$	(5) (6) (7) (8) (9)

$$\begin{array}{l} a'',v\vdash a',h(a')\downarrow_1:a'\\ a''=\sigma(\texttt{this})\\ v=h(a'')\downarrow_1 \end{array}$$

case $u = rep$			(10)
(10)+(4)	\Rightarrow	u' = rep	(11)
(6)+(7)+(11)	\Rightarrow	$h(a){\downarrow_1} = h(a'){\downarrow_1} = a''$	
case $u \neq rep$			(10)
(10)+(4)	\Rightarrow	$u' eq { t rep}$	(11)
$(10)\!+\!(11)\!+\!(3)$	\Rightarrow	$u, u' \in \{\texttt{peer}, \texttt{self}\}$	(12)
(6)+(7)+(12)	\Rightarrow	$h(a)\downarrow_1 = h(a')\downarrow_1 = v$	
			·

$Virgin(e)$ $h, \sigma \vdash_{gb} e$	$\left. \begin{array}{c} p \\ \vdots p \end{array} \right\} \Longrightarrow p = e$		
	Let	$\begin{array}{l} Virgin(e) \\ h, \sigma \vdash_{gb} e : p \end{array}$	(1) (2)
	case (2) last was	(FIELD)	(3)
	(3) =	e = p'.f	(4)
		p = p''.f	(5)
		$h, \sigma \vdash_{gb} p' : p''$	(6)
	$(4)+(1) \Rightarrow$	\rightarrow Virgin(p')	(7)
	$(7)+(6)+\underline{\mathrm{IH}}$	> p' = p''	(8)
	$(8) + (4) + (5) \Rightarrow $	$\Rightarrow p = e$	
	case (2) last was	(VAR)	(3)
	(3) =	$\Rightarrow e = p$	
	case (2) last was	(VAL)	(3)
	(3) =	$\Rightarrow e = a$	(4)
	(4) =	$\rightarrow \neg Virgin(e)$ (contradiction)	

Lemma A28

	$\begin{cases} e \neq v \\ Reachable(C[e]) \\ Reachable(e') \end{cases} \implies \begin{cases} Reachable(C[e']) \\ Reachable(e) \end{cases}$	
	Let $e \neq v$	(1)
	Reachable(C[e])	(2)
	Reachable(e')	(3)
_	case $C[e] = (t)C'[e]$	(4)
	$(4)+(2) \qquad \Rightarrow Reachable(C'[e])$	(5)
	$(1)+(5)+(3)+\underline{IH} \Rightarrow Reachable(C'[e'])$	(6)
	Reachable(e)	
	$(6)+(4) \qquad \Rightarrow Reachable(C[e'])$	
_	$\begin{array}{l} \text{case } C[e] \in \{C'[e].f, v.f := C'[e], v.m(C'[e]), \texttt{sync}_{e_0} \ C'[e] \ e_1 \} \\ Similar \ to \ above \end{array}$	(4)
_	case $C[e] \in \{ \text{synced}_{e_0} \ w \ C'[e], \text{frame } \sigma \ C'[e] \}$ Similar to above	(4)
	case $C[e] = C'[e] \cdot f := e''$	(4)
	$(4)+(2) \qquad \Rightarrow (Reachable(C'[e]) \land Virgin(e'')) \lor (C'[e] = v \land Reachable(e''))$	(5)

case LHS of (5)			(6)
(6)	\Rightarrow	Reachable(C'[e])	(7)
		Virgin(e'')	(8)
(1)+(7)+(3)+IH	\Rightarrow	Reachable(C'[e'])	(9)
		Beachable(e)	(0)
(9)+(4)+(8)	\rightarrow	Reachable(C[e'])	
$(\mathbf{J}) + (\mathbf{T}) + (\mathbf{O})$	-	neaenable(e [e])	
case RHS of (5)			(6)
(6)	\Rightarrow	C'[e] = v	(7)
		Reachable(e'')	(8)
(7)	\Rightarrow	e = v	(9)
(9)		contradicts (1)	(-)
(0)			
case $C[e] = C'[e]$.	$m(\overline{e''})$		(4)
Similar to abo	ve		

Let] ; ;	$\begin{array}{l} \Gamma \vdash_{gb} e: l \\ h, \sigma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \\ h, \sigma \vdash \mathtt{this} : \Gamma(\mathtt{this}) \end{array}$	(1) (2) (3)
case (1) last was (UN)	(V)		(4)
(4)	\Rightarrow]	$\Gamma \vdash e: u$	(5)
	1	u eq any	(6)
	1	u = l	(7)
(5)+(2)+(3)+Lemma	$3.5.5 \Rightarrow l$	$h, \sigma \vdash S(e) : u$	(8)
(8)+(6)+(7)+(UNIV)	\Rightarrow 1	$h, \sigma \vdash_{gb} S(e) : l$	
case (1) last was (PA	гн)		(4)
(4)	\Rightarrow e	e = l = p	(5)
case $p = \mathbf{x}$			(6)
(5)+(6)	$\Rightarrow e = l$	= x	(7)
(7) + (VAR)	$\Rightarrow h, \sigma \vdash$	${gb} e: l$	(8)
(7)	\Rightarrow $S(e)$	= e	(9)
(8) + (9)	$\Rightarrow h, \sigma \vdash$	${gb} S(e): l$	
case $p = \texttt{this}$			(6)
Similar to p	= x		
case $p = p', f$			(6)
(PATH)	$\Rightarrow \Gamma \vdash_{ab}$	p':p'	(7)
$(7)+(2)+(3)+\mathrm{IH}$	$\Rightarrow h, \sigma \vdash$	-ab S(p'): p'	(8)
(6)+(8)+(FIELD)	$\Rightarrow h, \sigma \vdash$	$-g_b S(p') \cdot f : p$	(9)
(6)	$\Rightarrow S(p)$	= S(p').f	(10)
(9) + (10)	$\Rightarrow h, \sigma \vdash$	${gb} S(p) : p$	
			•

nma A30			
$\left. \begin{array}{l} l = (u, e, e') \bowtie l' \\ h, \sigma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \\ h, \sigma \vdash \mathtt{this} : \Gamma(\mathtt{this}) \end{array} \right\} =$	$\Rightarrow (h,$	$\sigma, u, S(e), S(e')) \bowtie l' = l$	
Let		$l = (u, e, e') \bowtie l'$	(1)
		$h, \sigma \vdash \mathbf{x} : \Gamma(\mathbf{x})$	(2)
		$h, \sigma \vdash \texttt{this}: \Gamma(\texttt{this})$	(3)
case $l' = u'$			(4)
(1) + (4)	\Rightarrow	l=u ightarrow u'	(5)
$(4)\!+\!(5)$	\Rightarrow	$\forall h, \sigma, S(e), S(e') : (h, \sigma, u, S(e), S(e')) \bowtie l' = l$	
case $l' = \mathbf{x} \cdot f_{1n}$			(4
(1)+(4)	\Rightarrow	e' = p'	(5)
		$l = l'[p'/\mathbf{x}]$	(6)
(Path)	\Rightarrow	$\Gamma \vdash_{gb} p' : p'$	(7)
(7)+(2)+(3)+Lemma A29	\Rightarrow	$h, \sigma \vdash_{gb} S(p') : p'$	(8)
(5)	\Rightarrow	S(e') = S(p')	(9)
(8) + (9)	\Rightarrow	$h, \sigma \vdash_{gb} S(e') : p'$	(10)
(6) + (4) + (10)	\Rightarrow	$\forall u, S(e) : (h, \sigma, a, S(e), S(e')) \bowtie l' = l$	
case $l' = \texttt{this}.f_{1n}$			(4)
Similar to $\mathbf{x}.f_{1n}$			

 $\mathbb{L}, h, \sigma; \texttt{spawn} \ e: F \Longrightarrow \emptyset, h, \sigma \vdash e: _$

Let	$\mathbb{L}, h, \sigma;$ spawn $e: F$	(1)
$\begin{array}{c} \hline \text{case (1) l} \\ (2) \\ (3) + \underline{\text{IH}} \end{array}$	ast was (SUB) $\Rightarrow \mathbb{L}', h, \sigma \vdash \text{spawn } e : F'$ $\Rightarrow \emptyset, h, \sigma \vdash e : _$	$(2) \\ (3)$
$\begin{array}{c} \text{case (1) l} \\ \text{(2)} \end{array}$	ast was (SPAWN) $\Rightarrow \emptyset, h, \sigma \vdash e: _$	(2)

Lemma A32

 Let

$$\left. \begin{array}{c} ,h,\sigma \vdash e:F\\ \sigma \vdash e,h \rightsquigarrow e',h'\\ h(a)\downarrow_3(f) \neq h'(a)\downarrow_3(f) \end{array} \right\} \Longrightarrow f \in F$$

$,h,\sigmadash e:F$	(1)
$\overline{\sigma} dash e, h \rightsquigarrow e', h'$	(2)
$h(a){\downarrow}_3(f)\neq h'(a){\downarrow}_3(f)$	(3)

case (1) la $h = h'$	st was (THIS) (VAR) (NULL) , thus contradicts (3)	(4)
case (1) la Contre	st was (ADDR) (SPAWN) adicts (2)	(4)
case (1) la	st was (NEW)	(4)
(4)	$\Rightarrow a' \notin dom(h)$	(5)
	$\forall f, a'' \neq a' : h(a'') {\downarrow}_3(f) = h'(a'') {\downarrow}_3(f)$	(6)

	$\Rightarrow a' \neq a$	
(7)+(6)	$\Rightarrow h(a)\downarrow_3(f) = h'(a)\downarrow_3(f)$	
(8)+(3)	\Rightarrow contradiction	
case (1) last was	(SUB)	
(4)	$\Rightarrow \underline{,} h, \sigma \vdash e : F'$	
(5) + (2) + (3) + IH	$F \subseteq F$ $\rightarrow f \subset F'$	
(7)+(6)	$\begin{array}{ccc} \rightarrow & f \in I \\ \Rightarrow & f \in F \end{array}$	
case (1) last was	(Sync)	
(4)	$\Rightarrow e = \operatorname{sync}_{e'} e'' e'''$	
	$_,h,\sigma \vdash e'':F$	
(5)+(3)+(CTX)	$\Rightarrow \sigma \vdash e'', h \rightsquigarrow e'''', h'$	
$(6)+(7)+(3)+\underline{IH}$	$\Rightarrow f \in F'$	
case (1) last was	(FIELD)	
In one case,	h = h', in the other case proceed as in (Sync)	
case (1) last was	(Call) (Cast) (Synced) (Frame)	
case (1) last was	(Call) (Cast) (Synced) (Frame)	
case (1) last was As (FIELD)	(Call) (Cast) (Synced) (Frame)	
case (1) last was $As (FIELD)$ case (1) last was	(CALL) (CAST) (SYNCED) (FRAME) (Assign)	
case (1) last was As (FIELD) case (1) last was (4)	(CALL) (CAST) (SYNCED) (FRAME) (Assign) $\Rightarrow e = e''.f' := e'''$	
case (1) last was $As (FIELD)$ case (1) last was (4)	(CALL) (CAST) (SYNCED) (FRAME) (Assign) $\Rightarrow e = e''.f' := e'''$ $f' \in F$	
case (1) last was $As \text{ (FIELD)}$ case (1) last was (4) $\hline \text{case } e'' = a'$	(CALL) (CAST) (SYNCED) (FRAME) (ASSIGN) $\Rightarrow e = e''.f' := e'''$ $f' \in F$	
case (1) last was $As \text{ (FIELD)}$ case (1) last was (4) $\hline case e'' = a'$ (7)	(CALL) (CAST) (SYNCED) (FRAME) (ASSIGN) $\Rightarrow e = e'' \cdot f' := e'''$ $f' \in F$ $\Rightarrow e' = e''' = v$	(
case (1) last was $As \text{ (FIELD)}$ case (1) last was (4) $\hline case \ e'' = a'$ (7)	$(CALL) (CAST) (SYNCED) (FRAME)$ $(ASSIGN)$ $\Rightarrow e = e'' \cdot f' := e'''$ $f' \in F$ $\Rightarrow e' = e''' = v$ $h' = h[a' \downarrow_3(f') \mapsto v]$	(
case (1) last was $As (FIELD)$ case (1) last was (4) $\hline case e'' = a'$ (7) $= (9)+(3) = (2)$	$(CALL) (CAST) (SYNCED) (FRAME)$ $(ASSIGN)$ $\Rightarrow e = e''.f' := e'''$ $f' \in F$ $\Rightarrow e' = e''' = v$ $h' = h[a' \downarrow_3(f') \mapsto v]$ $\Rightarrow a = a' \land f = f'$	
case (1) last was As (FIELD) case (1) last was (4) $\hline case e'' = a'$ (7) = (9)+(3) = (10)+(6)	$(CALL) (CAST) (SYNCED) (FRAME)$ $(ASSIGN)$ $\Rightarrow e = e'' f' := e'''$ $f' \in F$ $\Rightarrow e' = e''' = v$ $h' = h[a'\downarrow_3(f') \mapsto v]$ $\Rightarrow a = a' \land f = f'$ $\Rightarrow f \in F$	(1
case (1) last was As (FIELD) case (1) last was (4) $\hline case e'' = a'$ (7) = (9)+(3) = (10)+(6) = $\hline case e = E[e]$	$(CALL) (CAST) (SYNCED) (FRAME)$ $(ASSIGN)$ $\Rightarrow e = e'' \cdot f' := e'''$ $f' \in F$ $\Rightarrow e' = e''' = v$ $h' = h[a' \downarrow_3(f') \mapsto v]$ $\Rightarrow a = a' \land f = f'$ $\Rightarrow f \in F$	(()(())((())(())(())((())(())((())((())((())((())((())((())((((
case (1) last was $As (FIELD)$ case (1) last was (4) (4) (4) (7) $(9)+(3)$ $(10)+(6)$ $(ase \ e = E[e]$ $Case \ an$	$(CALL) (CAST) (SYNCED) (FRAME)$ $(ASSIGN)$ $\Rightarrow e = e'' \cdot f' := e'''$ $f' \in F$ $\Rightarrow e' = e''' = v$ $h' = h[a'\downarrow_3(f') \mapsto v]$ $\Rightarrow a = a' \land f = f'$ $\Rightarrow f \in F$ $\bullet]$ $allysis of E, proceed in each case via IH$	(((((((((((((((((((((((((((((((((((((((
case (1) last was $As (FIELD)$ case (1) last was (4) $\hline case e'' = a' \\ (7) = (10) + (3) = (10) + (6) = (10) + (10) + (6) = (10) + (10) + (6) = (10) + (10) $	$(CALL) (CAST) (SYNCED) (FRAME)$ $(ASSIGN)$ $\Rightarrow e = e'' \cdot f' := e'''$ $f' \in F$ $\Rightarrow e' = e''' = v$ $h' = h[a' \downarrow_3(f') \mapsto v]$ $\Rightarrow a = a' \land f = f'$ $\Rightarrow f \in F$ $\bullet]$ $alysis of E, proceed in each case via IH$) ((() ()

Appendix B

Correctness of Path Graph Analysis

Here we give the Isabelle/HOL ProofGeneral script that defines the semantics of a model CFG language, our analysis on the CFG, and a correctness theorem. We then give the correctness proof. The syntax is as close as we could manage to the earlier presentation (§4.4). For example, we had to use two dots in field loads / stores, e.g. \mathbf{x} ..f because the single dot is used by Isabelle/HOL as part of its quantification syntax.

```
theory PG imports Main Nat List
begin
types node = nat
datatype field = ff | fg | fh | Field nat
datatype var = vx | vy | vz | Var nat
                            ("[_:=_,_]")
datatype st = Copy var var node
        FLoad var var field node
                            ("[_=_.._,_]")
        | FStore var field var node ("[_.._=_,_]")
        New var node
                            ("[_=new,_]")
                            ("[?,_,_]")
        | Branch node node
types prog = "node ~=> st"
types addr = nat
datatype val = Null
        | Addr addr ("$_")
types heap = "addr ~=> field \<Rightarrow> val"
types stack = "var ~=> val"
types action = "addr option"
```

```
consts filter_addr :: "addr \<Rightarrow> (action list)
                            \<Rightarrow> (action list)"
defs filter_addr[simp]: "filter_addr a A ==
                             (map (\<lambda>x. if x=Some a then None else x) A)"
consts accesses :: "(prog * heap * stack * node * (action list))
                    \<Rightarrow> bool"
recdef accesses "measure(\<lambda>(P,h,s,n,A). length A)"
   "accesses (P, h, s, n, []) = True"
   "accesses (P, h, s, n, (act#A)) = (case P n of
   Some st \<Rightarrow> (case st of
        [x:=y, n'] \<Rightarrow> (case s y of
            Some v \<Rightarrow> (act=None) & accesses (P,h,(s(x:=Some v)),n',A)
          | None \<Rightarrow> False
        )
      | [x=y .. f, n'] \<Rightarrow> (case s y of
            Some u \<Rightarrow> (case u of
                $a \<Rightarrow> (case h a of
                    Some fs \<Rightarrow> ((act=Some a)
                        & accesses (P,h,(s(x:=Some (fs f))),n',A))
                  | None \<Rightarrow> False
                )
              | Null \<Rightarrow> False
            )
          | None \<Rightarrow> False
        )
      | [x .. f=y, n'] \<Rightarrow> (case s y of
            Some y' \<Rightarrow> (case s x of
                Some x' \<Rightarrow> (case x' of
                    $a \<Rightarrow> (case h a of
                        Some fs \<Rightarrow>
                             ( (act=Some a) & accesses (P,
                                                         (h(a:=Some(fs(f:=y')))),
                                                         s,n',A) )
                      | None \<Rightarrow> False
                    )
                  | Null \<Rightarrow> False
                )
              | None \<Rightarrow> False
            )
          | None \<Rightarrow> False
        )
      [x=new,n'] \<Rightarrow> (
            (act = None) &
            (\<exists>a.
                ((h a) = None) \&
                (\<exists>A'. accesses (P,
                                          (h(a:=Some (\<lambda>f. Null))),
                                          (s(x:=Some $a)), n', A')
                               & A = filter_addr a A')
            )
        )
      | [?,n',n''] \<Rightarrow> (
            (act = None) &
```

```
(accesses (P, h, s, n', A) | accesses (P, h, s, n'', A))
        )
    )
  | None \<Rightarrow> False
)"
(hints recdef_cong: rev_conj_cong)
(* sanity *)
consts MyProg :: prog
defs MyProg_def[simp]: "MyProg == [1\<mapsto>[vx=new,2],
                                    2 \geq [vx:=vy,3],
                                    3\<mapsto>[vz .. ff=vz,4]]"
lemma "(b~=Some a & c~=Some a) \<longrightarrow> (filter_addr a [b,c] = [b,c])"
by simp
lemma "(b~=Some a) \<longrightarrow> (filter_addr a [b,Some a] = [b,None])"
by (simp)
lemma "accesses (MyProg,
                 [a\<mapsto>(\<lambda>f. Null)],
                 [vz\<mapsto>$a],
                 З,
                 [Some a])"
by (simp)
lemma always_another_address1[simp]: "(EX aa::addr . a ~= aa)"
apply(rule_tac x = "Suc(a)" in exI)
apply(simp)
done
lemma always_another_address2[simp]: "(EX aa::addr . aa ~= a)"
apply(rule_tac x = "Suc(a)" in exI)
by (simp)
lemma "accesses (MyProg,
                 [a\<mapsto>(\<lambda>f. Null)],
                 [vz\<mapsto>$a,vy\<mapsto>$b],
                 1,
                 [None,None,Some a])"
apply(auto)
apply(rule_tac ?x="Suc(a)" in exI)
apply(auto)
apply(rule_tac ?x="[None, Some a]" in exI)
apply(simp)
done
```

```
datatype edge = VarEdge var node
                                           (infix "\<rightarrow>" 100)
              | FieldEdge node field node ("_\<rightarrow>_\<rightarrow>_" 100)
types pgraph = "edge set"
types analysis = "node \<Rightarrow> pgraph"
consts access:: "node \<Rightarrow> st \<Rightarrow> pgraph"
defs access_def[simp]: "access n st == case st of
        [x:=y,n'] \<Rightarrow> {}
      | [x=y .. f,n'] \<Rightarrow> {y\<rightarrow>n}
      | [x .. f=y,n'] \<Rightarrow> {x\<rightarrow>n}
      | [x=new,n'] \<Rightarrow> {}
      | [?,n',n''] \<Rightarrow> {}"
consts translate:: "node \<Rightarrow> st \<Rightarrow> analysis
                    \<Rightarrow> pgraph"
defs translate_def[simp]: "translate n st X == case st of
        [x:=y,next] \<Rightarrow> ((X next)
                                   - {x\<rightarrow>n' |
                                      n'. x\<rightarrow>n'\<in>(X next)})
                                  \<union> {y\<rightarrow>n' |
                                            n'. x\<rightarrow>n'\<in>(X next)}
      | [x=y .. f,next] \<Rightarrow> ((X next)
                                       - {x\<rightarrow>n' |
                                          n'. x\<rightarrow>n'\<in>(X next)})
                                  \<union> {n\<rightarrow>f\<rightarrow>n' |
                                            n'. x\<rightarrow>n'\<in>(X next)}
      | [x .. f=y,next] \<Rightarrow> ((X next) - {e. \<exists>n' n''.
                                          (e=(n'\<rightarrow>f\<rightarrow>n''))
                                        & x\<rightarrow>n'\<in>(X next)
                                        & \<not>(\<exists>z.(z~=x) &
                                                 z\<rightarrow>n'\<in>(X next))
                                        & \<not>(\<exists>n'', f.
                                             (n'''\<rightarrow>f\<rightarrow>n')
                                                  <in>(X next))})
                                      \<union>
                                      {y\<rightarrow>n' |
                                       n'. (< xists > n''.
                                            (n''\<rightarrow>f\<rightarrow>n')
                                                <in>(X next))}
      [x=new,next] \<Rightarrow>
                                     (X next)
                                      - {x\<rightarrow>n' |
                                         n'. x\<rightarrow>n'\<in>(X next)}
      | [?,tnext,fnext] \<Rightarrow> (X tnext) \<union> (X fnext)
...
consts wfpgraph:: "prog \<Rightarrow> analysis \<Rightarrow> node
                   \<Rightarrow> bool"
defs wfpgraph_def[simp]: "wfpgraph P X n == case P n of
```

```
Some st \<Rightarrow> (X n) = (access n st) \<union> (translate n st X)
      | None \<Rightarrow> True"
consts wfanalysis:: "prog \<Rightarrow> analysis \<Rightarrow> bool"
    (infix "\<turnstile>" 95)
defs wfanalysis_def: "P\<turnstile>X == \<forall>n. wfpgraph P X n"
(*
(* sanity *)
consts anal0:: analysis
defs anal0[simp]: "anal0 == \<lambda>n.{}"
lemma "wfanalysis MyProg (anal0(1:={vz\<rightarrow>3},
                               2:={vz\<rightarrow>3},
                               3:={vz\<rightarrow>3}))"
by (simp add: wfanalysis_def)
lemma "wfanalysis [1\<mapsto>[vy=new,2],
                  2 \geq [vx:=vy,3],
                  3\<mapsto>[vx .. f=vz,4]]
                  (anal0(2:={vy\<rightarrow>3}, 3:={vx\<rightarrow>3}))"
by (simp add: wfanalysis_def)
lemma "wfanalysis [1\<mapsto>[vx=new,2],
                  2 \geq [vx:=vy,3],
                  3\<mapsto>[vx .. f=vz,4]]
                  (anal0(1:={vy\<rightarrow>3},
                        2:={vy\<rightarrow>3},
                        3:={vx\<rightarrow>3}))"
by (simp add: wfanalysis_def)
*)
types pgass = "node \<Rightarrow> (addr set)"
consts pgassle :: "pgass \<Rightarrow> pgass \<Rightarrow> bool"
    (infixl "\<sqsubseteq>" 65 )
defs pgassle_def: "\<phi> \<sqsubseteq> \<phi>'
                  == \<forall>n. (\<phi> n) \<subseteq> (\<phi>' n)"
consts pgassinter :: "pgass \<Rightarrow> pgass \<Rightarrow> pgass"
    (infixl "<sqinter>" 65 )
defs pgassinter_def: "\<phi> \<sqinter> \<phi>'
                     == \langle ambda \rangle n. (\langle phi \rangle n) \langle inter \rangle (\langle phi \rangle' n)''
consts pgassInter :: "pgass set \<Rightarrow> pgass" ("\<Sqinter> _" 65 )
defs pgassInter_def: "\<Sqinter> \<phi>s
```

```
== \<lambda>n. \<Inter> { (\<phi> n) |
                                               <phi>. <phi><in><phi>s }"
constdefs \<phi>0:: "pgass"
               lemma \<phi>0_le[simp]: "\<phi>0 \<sqsubseteq> \<phi>"
apply(simp add: pgassle_def \<phi>0_def)
done
lemma \<phi>0_union1[simp]: "\<phi> \<sqinter> \<phi>' \<sqsubseteq> \<phi>"
apply(simp add: pgassinter_def subset_def pgassle_def)
done
lemma \<phi>0_union2[simp]: "\<phi> \<sqinter> \<phi>' \<sqsubseteq> \<phi>'"
apply(simp add: pgassinter_def subset_def pgassle_def)
done
lemma \<phi>0_union_id[simp]: "\<phi> \<sqinter> \<phi> = \<phi>"
apply(simp add: pgassinter_def subset_def)
done
lemma pgass_ref[simp]: "\<phi> \<sqsubseteq> \<phi>"
apply(simp add: pgassle_def)
done
lemma pgass_id[simp]: "\<phi> \<sqsubseteq> \<phi>0
                      \<Longrightarrow> \<phi>=\<phi>0"
apply(simp add: \<phi>0_def pgassle_def)
apply(rule ext)
apply(auto)
done
consts pgassaddr :: "pgass \<Rightarrow> (addr set)"
defs pgassaddr_def: "pgassaddr \<phi> == {a. \<exists>n. a\<in>(\<phi> n)}"
consts wfpgass :: "heap \<Rightarrow> stack \<Rightarrow> pgraph
                       \<Rightarrow> pgass \<Rightarrow> bool"
    ("(_,_\<turnstile>_:_)")
defs wfpgass_def: "wfpgass h s G \<phi> ==
    (\<forall>x. \<forall>n.
        (x\<rightarrow>n)\<in>G\ <longrightarrow>
            (\<exists>x'.
                  ((s x) = x') \&
                  (\<forall>a.(x'=Some $a) \<longrightarrow> a\<in>(\<phi> n))))
  & (\<forall>f. \<forall>n. \<forall>n'.
        (n\<rightarrow>f\<rightarrow>n')\<in>G \<longrightarrow>
            {a. < xists > a'. a' < in > (< phi > n)
                          & (\<exists>fs. ((h a')=Some fs) & ((fs f)=$a))}
            \<subseteq> (\<phi> n'))"
```

consts minpgass :: "heap \<Rightarrow> stack \<Rightarrow> pgraph

```
\<Rightarrow> pgass"
    ("(\<Phi>___)")
defs minpgass_def: "\<Phi> h s G == \<Sqinter> { \<phi>. wfpgass h s G \<phi> }"
(* sanity *)
lemma "wfpgass [100\<mapsto>(\<lambda>f. Null)(f:=$100)]
               [vx\<mapsto>$100]
               {vx\<rightarrow>1,1\<rightarrow>f\<rightarrow>1}
               ((\<lambda>n.{})(1:={100}))"
apply(simp add:wfpgass_def)
done
lemma "wfpgass [100\<mapsto>((\<lambda>f. Null)(ff:=$101))]
               empty
               {1\<rightarrow>ff\<rightarrow>2}
               ((\<lambda>n.{})(1:={100},2:={101}))"
apply(simp add:wfpgass_def)
done
consts actionlistaddr:: "(action list) \<Rightarrow> (addr set)"
defs actionlistaddr_def: "actionlistaddr A == {a. (Some a) mem A}"
(* sanity *)
lemma "actionlistaddr [Some a, None, Some b, Some b] = {a,b}"
apply(simp add:actionlistaddr_def)
apply(auto)
done
lemma noaction [simp]: "actionlistaddr (None#A) = actionlistaddr A"
apply(simp add: actionlistaddr_def)
done
lemma someaction [simp]: "actionlistaddr ((Some a)#A)
                          = {a} \<union> actionlistaddr A"
apply(simp add: actionlistaddr_def)
apply(auto)
done
lemma wfpgass_preserved_over_subset: "\<lbrakk> G'\<subseteq>G ;
                                                wfpgass h s G \<phi> \<rbrakk>
                                      \<Longrightarrow> wfpgass h s G' \<phi>"
apply(simp add: wfpgass_def)
apply(blast)
```

done

```
(* wellformed assignments can be joined to form another assignment
   ultimately this can help us find a minimal assignment *)
lemma wfpgass_inter: "\<lbrakk> \<forall> \<phi>\<in>\<phi>s.
                                    wfpgass h s G \<phi> \<rbrakk>
                      \<Longrightarrow> wfpgass h s G (\<Sqinter> \<phi>s)"
apply(simp add: pgassInter_def subset_def wfpgass_def)
apply(safe)
apply(blast)
apply(blast)
done
(* clearly the smallest valid assignment is valid *)
lemma minpgass_sufficient: "wfpgass h s G (minpgass h s G)"
apply(simp add: minpgass_def)
apply(insert wfpgass_inter)
apply(blast)
done
lemma wfpgass_always_le: "wfpgass h s G \<phi>
                          \<Longrightarrow> (\<exists>\<phi>'.
                                                 minpgass h s G = \<phi>' \<and>
                                                 <phi>'<sqsubseteq><<phi>)"
apply(simp add: minpgass_def pgassInter_def pgassle_def)
apply(blast)
done
lemma wfpgass_shrink: "wfpgass h s G \<phi>
                       \<Longrightarrow> (minpgass h s G) \<sqsubseteq> \<phi>"
apply(drule wfpgass_always_le)
apply(blast)
done
lemma source_aux: "\<lbrakk> \<forall>x. x \<rightarrow> n \<in> G
                                          \<longrightarrow> s x \<noteq> Some $a;
                   \<forall>n' f. (n'<rightarrow>f<rightarrow>n) <in> G
                                   \<longrightarrow>
                                   (\<forall>a' fs. h a' = Some fs
                                                    \<longrightarrow>
                                                    fs f \<noteq> $a) \<rbrakk>
                   \<Longrightarrow> \<exists>\<phi>. a\<notin>\<phi> n \<and>
                                                       wfpgass h s G \<phi>"
apply (rule_tac ?x="(\<lambda>n . UNIV)(n:=UNIV-{a})" in exI)
apply (simp add: wfpgass_def)
apply(auto)
done
lemma source: "a\<in>(minpgass h s G) n
               \<Longrightarrow> (\<exists>x. x\<rightarrow>n\<in>G \<and>
                                              (s x = Some $a))
                                 \langle or \rangle
```

```
(\<exists>n' f a' fs.
                                       (n'\<rightarrow>f\<rightarrow>n)\<in>G
                                       \langle and \rangle (h a'=Some fs) \langle and \rangle (fs f = $a))"
apply (subgoal_tac "\<forall>h s G n a. a\<in>(minpgass h s G) n
                                          \<longrightarrow>
                                          (\geq xists > x.
                                              x\<rightarrow>n\<in>G \<and>
                                               (s x = Some $a))
                                          \langle or \rangle
                                          (\<exists>n' f a' fs.
                                               (n'\<rightarrow>f\<rightarrow>n)
                                                    \langle in \rangle G \langle and \rangle
                                               (h a'=Some fs) \<and>
                                              (fs f = $a))")
apply(blast)
apply(thin_tac "a \<in> minpgass h s G n")
apply(simp add: minpgass_def pgassInter_def)
apply(rule classical)
apply(simp)
apply(clarify)
apply(simp)
apply(drule_tac ?h="h" and ?s="s" and ?G="G" in source_aux)
apply(auto)
done
lemma thinactionlistaddr: " actionlistaddr L - {a} \leq S
                            \<Longrightarrow>
                            actionlistaddr (
                                 map ((\<lambda>x. x)(Some a := None)) L)
                            \ \ S''
apply(subgoal_tac "actionlistaddr (map ((\<lambda>x. x)(Some a := None)) L)
                    = actionlistaddr L - {a}")
apply(simp)
apply(simp add: actionlistaddr_def)
apply(thin_tac "{a. Some a mem L} - {a} \<subseteq> S")
apply(induct_tac L)
apply(simp)
apply(simp,blast)
done
theorem soundness: "(accesses (P,h,s,n,A)) \<and> (wfanalysis P X)
                     \<longrightarrow> (actionlistaddr A
                                         \<subseteq>
                                         pgassaddr (minpgass h s (X n)))"
apply(induct_tac P h s n A rule: accesses.induct)
apply(simp add: actionlistaddr_def)
apply(rule impI|rule allI|erule conjE|rule conjI)+
apply(case_tac "P n")
apply(simp)
apply(rename_tac st)
apply(case_tac st)
```

```
(*** x=y ***)
apply(rename_tac x y n')
apply(simp)
apply(fold fun_upd_def)
apply(case_tac "s y")
apply(simp)
apply(rename_tac v)
apply(simp)
apply(erule conjE)
apply(simp add: wfanalysis_def)
apply(drule_tac ?x="n" in spec)
apply(simp)
apply(simp only:minpgass_def pgassInter_def)
apply(erule subset_trans)
apply(simp only:subset_def Ball_def pgassaddr_def wfpgass_def Inter_def)
apply(clarify)
apply(rename_tac a n'')
apply(rule_tac ?x="n'' in exI)
apply(simp)
apply(clarify)
apply(rename_tac z \<phi>)
apply(drule_tac ?x="\<phi> n''" in spec)
apply(erule impE)
apply(rule_tac ?x="\<phi>" in exI)
apply(simp add: wfpgass_def)
apply(rule impI|rule allI|rule conjI|erule conjE)+
apply(rename_tac z n'' a)
apply(case_tac "y=x")
apply(simp)
apply(case_tac "z=x")
apply(simp)
apply(drule_tac ?x="z" in spec)
apply(drule_tac ?x="n''' in spec)
apply(simp)
apply(drule_tac ?x="z" in spec)
apply(drule_tac ?x="n'' in spec)
apply(simp)
apply(assumption)
(*** case x=y.f ***)
apply(simp)
apply(fold fun_upd_def)
apply(rename_tac x y f n')
apply(case_tac "s y")
apply(simp)
apply(simp)
apply(rename_tac yv)
apply(case_tac yv)
apply(simp)
```

```
apply(rename_tac targ)
apply(simp)
apply(case_tac "h targ")
apply(simp)
apply(rename_tac fs)
apply(simp)
apply(erule conjE)
apply(subgoal_tac "actionlistaddr A \<subseteq> pgassaddr (minpgass h s (X n))")
apply(rule conjI)
apply(thin_tac "accesses (P, h, s(x \<mapsto> fs f), n', A)")
apply(thin_tac "actionlistaddr A <<subseteq> pgassaddr <<Phi> h s (X n)")
apply(thin_tac "act = Some targ")
apply(thin_tac "actionlistaddr A
                \<subseteq> pgassaddr (\<Phi> h s(x \<mapsto> fs f) (X n'))")
apply(thin_tac "yv = $targ")
apply(thin_tac "st = [x=y.. f,n']")
apply(thin_tac "h targ = Some fs")
apply(simp add: minpgass_def pgassInter_def wfpgass_def
                wfanalysis_def pgassaddr_def)
apply(drule_tac ?x="n" in spec)
apply(rule_tac ?x="n" in exI)
apply(simp)
apply(rule allI)
apply(rename_tac addrs)
apply(rule impI)
apply(erule exE)
apply(erule conjE)+
apply(drule_tac ?x="y" in spec)
apply(drule_tac ?x="n" in spec)
apply(erule conjE)+
apply(drule mp)
apply(simp)
apply(simp)
apply(simp)
apply(erule subset_trans)
apply(thin_tac "st = [x=y.. f,n']")
apply(thin_tac "yv = $targ")
apply(thin_tac "act = Some targ")
apply(thin_tac "accesses (P, h, s(x \<mapsto> fs f), n', A)")
apply(simp add: minpgass_def pgassInter_def subset_def pgassaddr_def)
apply(rule allI)
apply(rename_tac a)
apply(rule impI)
apply(erule exE)
apply(rename_tac n'')
apply(rule_tac ?x="n'' in exI)
apply(rule allI)
apply(rename_tac addrs)
apply(rule impI)
apply(drule_tac ?x="addrs" in spec)
apply(erule exE)
apply(rename_tac \<phi>)
apply(drule mp)
apply(rule_tac ?x="\<phi>" in exI)
apply(erule conjE)
```

```
apply(erule conjI)
apply(simp add: wfanalysis_def wfpgass_def)
apply(drule_tac ?x="n" in spec,simp)
apply(clarify)
apply(rename_tac z)
apply(rule conjI)
apply(rule impI)
apply(rule allI)
apply(rename_tac n'')
apply(clarify)
apply(drule_tac ?x="f" in spec)
apply(drule_tac ?x="n" in spec)
apply(drule_tac ?x="n'' in spec)
apply(erule conjE)
apply(drule mp)
back
apply(simp)
apply(simp add: subset_def)
apply(drule_tac ?x="a" in spec)
apply(drule mp)
apply(rule_tac ?x="targ" in exI)
apply(rule conjI)
apply(blast)
apply(blast)
apply(assumption)
apply(blast)
apply(assumption)
(*** case x.f=y ***)
apply(rename_tac x f y n')
apply(simp)
apply(fold fun_upd_def)
apply(thin_tac "st = [x..f=y,n']")
apply(case_tac "s y")
apply(simp)
apply(rename_tac yv)
apply(case_tac "s x")
apply(simp)
apply(rename_tac xv)
apply(simp)
apply(case_tac xv)
apply(simp)
apply(rename_tac a)
apply(case_tac "h a")
apply(simp)
apply(rename_tac fs)
apply(simp)
apply(thin_tac "xv = $a")
apply(erule conjE)
apply(subgoal_tac "actionlistaddr A
                   \<subseteq> pgassaddr (minpgass h s (X n)) ")
```

```
apply(simp)
apply(simp add: pgassaddr_def minpgass_def pgassInter_def)
apply(rule_tac ?x="n" in exI)
apply(rule allI)
apply(rename_tac addrs)
apply(rule impI)
apply(erule exE)
apply(thin_tac "actionlistaddr A
                \<subseteq> {aa.
                              \<exists>n.
                                  \langle forall \rangle x.
                                      (<exists><phi>. x = <phi> n <and>
                                                         wfpgass (h(a \<mapsto>
                                                                    fs(f := yy)))
                                                                 s (X n') \langle phi \rangle
                                      \<longrightarrow> aa \<in> x}")
apply(thin_tac "s y = Some yv")
apply(thin_tac "h a = Some fs")
apply(thin_tac "act = Some a")
apply(thin_tac "accesses (P,(h(a \<mapsto> fs(f := yv))),s,n',A)")
apply(thin_tac "actionlistaddr A \<subseteq>
                {a. \<exists>na. \<forall>x.
                       (\<exists>\<phi>. x = \<phi> na \<and>
                                         wfpgass h s (X n) \<phi>)
                      \<longrightarrow> a \<in> x}")
apply(erule conjE)
apply(simp add: wfpgass_def wfanalysis_def)
apply(drule_tac ?x="n" in spec)
apply(simp)
apply(blast)
apply(thin_tac "act = Some a")
apply(thin_tac "accesses(P,(h(a \<mapsto> fs(f := yv))),s,n',A)")
apply(erule subset_trans)
apply(simp add:subset_def)
apply(clarify)
apply(rename_tac a')
apply(simp add: pgassaddr_def)
apply(clarify)
apply(rename_tac n'')
apply(rule_tac ?x="n'' in exI)
apply(simp add: minpgass_def pgassInter_def)
apply(clarify)
apply(rename_tac addrs \<phi>)
apply(simp)
apply(drule_tac ?x="(minpgass h s (X n)) n''" in spec)
apply(drule mp)
apply(rule_tac ?x="minpgass h s (X n)" in exI)
apply(simp)
prefer 2
apply(drule wfpgass_shrink)
apply(unfold pgassle_def)
apply(blast)
apply(thin_tac "h,s\<turnstile>X n:\<phi>")
apply(subgoal_tac "h,s\<turnstile>X n:(minpgass h s (X n))")
prefer 2
```
```
apply(rule minpgass_sufficient)
apply(simp only: wfanalysis_def wfpgraph_def)
apply(drule_tac ?x="n" in spec)
apply(simp|clarify)+
apply(thin_tac "X n = insert (x \<rightarrow> n)
         (X n' -
          {e. \<exists>n'a.
                  (<exists>n''. e = (n'a<<rightarrow>f<<rightarrow>n'')) <and>
                    x <rightarrow> n'a <in> X n' <and>
                    (<forall>z. z = x <or>
                                 z \<rightarrow> n'a \<notin> X n') \<and>
                    (\<forall>n'', f.
                        (n'''\<rightarrow>f\<rightarrow>n'a) \<notin> X n')
          } \<union> {y \<rightarrow> n'a | n'a.
                          \<exists>n''.
                              (n''\<rightarrow>f\<rightarrow>n'a) \<in> X n'})")
apply(thin_tac "P n = Some [x..f=y,n']")
apply(simp (no_asm_simp) only: wfpgass_def)
apply(clarify|simp)+
apply(intro conjI impI allI)
apply(simp only: wfpgass_def)
apply(clarify|simp)+
apply(rename_tac n1 n2 a2 a1)
apply(case_tac "a1 = a")
apply(simp)
apply(clarify)
apply(simp (no_asm) add: minpgass_def pgassInter_def)
apply(simp (no_asm) only: wfpgass_def)
apply(clarify)
apply(simp (no_asm_simp))
apply(drule_tac ?x="y" in spec)
apply(drule_tac ?x="n2" in spec)
apply(drule mp)
apply(blast)
apply(blast)
apply(simp|clarify)+
apply(rename_tac "fs1")
apply(frule source)
apply(simp|clarify)+
apply(erule disjE)
apply(simp|clarify)+
apply(rename_tac z)
apply(erule disjE)
apply(simp|clarify)+
apply(erule disjE)
apply(simp only: wfpgass_def)
apply(simp|clarify)+
apply(drule_tac ?x="f" in spec)
apply(drule_tac ?x="n1" in spec)
apply(drule_tac ?x="n2" in spec)
apply(simp|clarify)+
apply(drule mp)
apply(subgoal_tac "z\<noteq>x")
```

```
apply(blast)
apply(simp|clarify)+
apply(drule_tac ?x="z" in spec)
apply(drule_tac ?x="n1" in spec)
apply(simp|clarify)+
apply(blast)
apply(simp|clarify)+
apply(rename_tac n3)
apply(simp only: wfpgass_def)
apply(simp|clarify)+
apply(drule_tac ?x="f" in spec)
apply(drule_tac ?x="n1" in spec)
apply(drule_tac ?x="n2" in spec)
apply(simp|clarify)+
apply(drule mp)
apply(blast)
apply(blast)
apply(simp|clarify)+
apply(rename_tac n3 f3 a9 fs9)
apply(simp only: wfpgass_def)
apply(erule conjE)
apply(drule_tac ?x="f" in spec)
apply(drule_tac ?x="n1" in spec)
apply(drule_tac ?x="n2" in spec)
apply(drule mp)
back
apply(blast)
apply(simp only: subset_def)
apply(simp only: Ball_def)
apply(drule_tac ?x="a2" in spec)
apply(blast)
apply(rename_tac f0 n1 n2)
apply(clarify|simp)+
apply(rename_tac a2 a1)
apply(subgoal_tac "a1 \<in> minpgass h s
               (insert (x \<rightarrow> n)
                 (X n' -
                  {e. \<exists>n'a.
                      (\<exists>n''. e = (n'a\<rightarrow>f\<rightarrow>n''))
                                      \langle and \rangle
                                      x \<rightarrow> n'a \<in> X n'
                                      \langle and \rangle
                                      (\ z = x \ or)
                                                   z \<rightarrow> n'a
                                                        \<notin> X n')
                                      \langle and \rangle
                                      (\<forall>n''' f.
                                          (n'''\<rightarrow>f\<rightarrow>n'a)
                                              \<notin> X n')
                  } \<union> { y \<rightarrow> n'a |n'a.
                      \<exists>n''. (n''\<rightarrow>f\<rightarrow>n'a)
                                          \langle in \rangle X n'
                  }))
```

```
n1 \leqand>
        (\leq xists \leq s. (h a1 = Some fs) \leq and (fs f0 = a2))")
apply(simp|clarify)+
apply(rename_tac fs1)
apply(simp only: wfpgass_def)
apply(simp|clarify)+
apply(drule_tac ?x="f0" in spec)
apply(drule_tac ?x="n1" in spec)
apply(drule_tac ?x="n2" in spec)
apply(blast)
apply(blast)
(*** new ***)
apply(rename_tac x n')
apply(simp)
apply(fold fun_upd_def)
apply(erule conjE)
apply(erule exE)
apply(erule conjE)
apply(erule exE)
apply(rename_tac "A'")
apply(drule_tac ?x="a" in spec)
apply(drule_tac ?x="A'" in spec)
apply(simp)
apply(thin_tac "st = [x=new,n']")
apply(thin_tac "act = None")
apply(thin_tac "h a = None")
apply(erule conjE)
apply(thin_tac "A = map ((\<lambda>x. x)(Some a := None)) A'")
apply(thin_tac "accesses (P, h(a \<mapsto> \<lambda>f. Null),
                          s(x \<mapsto> $a), n', A')")
apply(simp add: wfanalysis_def)
apply(drule_tac ?x="n" in spec)
apply(simp)
apply(thin_tac "P n = Some [x=new,n']")
apply(thin_tac "X n = X n' - {x \<rightarrow> n'a |
                                  n'a. x \<rightarrow> n'a \<in> X n'}")
apply(simp)
apply(rule thinactionlistaddr)
apply(simp add: minpgass_def actionlistaddr_def pgassInter_def pgassaddr_def)
apply(simp add: subset_def Ball_def)
apply(intro allI impI)
apply(rename_tac a')
apply(erule conjE)
apply(drule_tac ?x="a'" in spec)
apply(simp)
apply(thin_tac "Some a' mem A'")
apply(erule exE)
apply(rename_tac "n")
apply(rule_tac ?x="n" in exI)
```

```
apply(rule allI)
apply(rename_tac "addrs")
apply(rule impI)
apply(drule_tac ?x="addrs \<union> {a}" in spec)
apply(drule mp)
apply(erule exE)
apply(erule conjE)
apply(thin_tac "a' \<noteq> a")
apply(simp)
apply(rule_tac ?x="\<lambda>n.{a} \<union> \<phi> n" in exI)
apply(simp)
apply(clarify)
apply(simp)
apply(unfold wfpgass_def)
apply(rule conjI)
apply(clarify)
apply(simp)
apply(blast)
apply(simp)
apply(blast)
apply(simp)
(*** if ***)
apply(rename_tac n' n'')
apply(simp)
apply(erule conjE)
apply(erule disjE)
apply(thin_tac "accesses (P, h, s, n'', A) \<longrightarrow>
                actionlistaddr A \<subseteq> pgassaddr \<Phi> h s (X n'')")
apply(simp)
apply(simp add: wfanalysis_def pgassaddr_def)
apply(drule_tac ?x="n" in spec)
apply(simp)
apply(subgoal_tac "{a. \<exists>n. a \<in> (\<Phi> h s (X n')) n}
                    \<subseteq> {a. \<exists>n. a \<in>
                                     (\<Phi> h s ((X n') \<union> (X n''))) n}")
apply(blast)
apply(thin_tac "accesses (P, h, s, n', A)")
apply(thin_tac "act = None")
apply(thin_tac "X n = X n', \<union> X n',")
apply(thin_tac "P n = Some [?,n',n'']")
apply(thin_tac "actionlistaddr A \<subseteq>
                    {a. \<exists>n. a \<in> (\<Phi> h s (X n')) n}")
apply(thin_tac "st = [?,n',n'']")
apply(simp add: minpgass_def subset_def pgassInter_def)
apply(clarify)
apply(rule_tac ?x="n" in exI)
apply(clarify)
apply(drule_tac ?x="\<phi> n" in spec)
apply(drule mp)
apply(simp add: subset_def Ball_def)
apply(rule_tac ?x="\<phi>" in exI)
```

```
apply(insert wfpgass_preserved_over_subset)
apply(blast)
apply(blast)
apply(thin_tac "accesses (P, h, s, n', A) <longrightarrow>
                                   actionlistaddr A <<subseteq> pgassaddr <<Phi> h s (X n')")
apply(simp)
apply(simp add: wfanalysis_def pgassaddr_def)
apply(drule_tac ?x="n" in spec)
apply(simp)
apply(subgoal_tac "{a. \<exists>n. a \<in>(\<Phi> h s (X n'')) n}
                                          \ \a. a. \a. a. \a. \a. \a. \a. \a. a. \a. \a. \a. a. \a. a. \a. a. \a. \a. a. \a. a
                                                                               (\<Phi> h s ((X n') \<union> (X n''))) n}")
apply(blast)
apply(thin_tac "accesses (P, h, s, n'', A)")
apply(thin_tac "act = None")
apply(thin_tac "X n = X n' \<union> X n''")
apply(thin_tac "P n = Some [?,n',n'']")
apply(thin_tac "actionlistaddr A
                                          apply(thin_tac "st = [?,n',n'']")
apply(simp add: minpgass_def subset_def pgassInter_def)
apply(clarify)
apply(rule_tac ?x="n" in exI)
apply(clarify)
apply(drule_tac ?x="\<phi> n" in spec)
apply(drule mp)
apply(simp add: subset_def Ball_def)
apply(rule_tac ?x="\<phi>" in exI)
apply(insert wfpgass_preserved_over_subset)
apply(blast)
apply(blast)
```

done

end

References

- 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] Martin Abadi and Luca Cardelli. A Theory of Objects. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.
- [3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers: principles, techniques, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
- [4] Al Viro. Fix dnotify/close race (CVE-2008-1375), May 2008. http://www.kernel. org/pub/linux/kernel/v2.6/ChangeLog-2.6.24.6.
- [5] 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, volume 1785/2000, pages 38-42, London, UK, 2000. Springer-Verlag.
- [6] Various Authors. Boost. Free peer-reviewed portable C++ source libraries http: //www.boost.org.
- [7] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio J. Serrano. Thin locks: featherweight synchronization for java. SIGPLAN Not., 39(4):583-595, 2004.
- [8] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect of Java without data races. In Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications, pages 382–400, Minneapolis, Minnesota, October 2000.
- [9] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. Commun. ACM, 13(7):422-426, 1970.
- [10] Shekhar Borkar. Design challenges of technology scaling. IEEE Micro, 19(4):23-29, 1999.
- [11] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In Proceedings of the 16th Annual ACM Symposium on Object-Oriented Programming Systems, Languages, and Applications, Tampa Bay, Florida, October 2001.
- [12] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 211–230, New York, NY, USA, 2002. ACM Press.

- [13] 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), Tucson, Arizona, June 2008.
- [14] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98), volume 33 of ACM SIGPLAN Notices, pages 48-64, New York, October 1998. ACM Press.
- [15] Dave Cunningham. Lock inference proofs, 2008. available from http://www.doc. ic.ac.uk/~dc04/FTfJP2008.thy.
- [16] Dave Cunningham, Werner Dietl, Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander Summers. Universe Types for Topology and Encapsulation. Formal Methods for Components, 5382:72–112, 2008.
- [17] Dave Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Universe Types for Race Safety. In VAMP 07, pages 20–51, September 2007.
- [18] Dave Cunningham, Sophia Drossopoulou, and Susan Eisenbach. Lock Inference Proven Correct. In FTfJP, July 2008.
- [19] 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.
- [20] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, pages 336–346, New York, NY, USA, 2006. ACM.
- [21] Daniel Jalkut. Apple mac OS X foundation 'NSURLConnection' cache management race condition security vulnerability, March 2008. http://www.securityfocus.com/ bid/28359/discuss.
- [22] C. J. Date. An Introduction to Database Systems. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1991.
- [23] developerWorks. Java theory and practice: Going atomic, November 2004. http: //www-128.ibm.com/developerworks/java/library/j-jtp11234/.
- [24] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. Journal of Object Technology (JOT), 4(8):5-32, October 2005.
- [25] Werner Dietl, Sophia Drossopoulou, and Peter Müller. Generic Universe Types. In Berlin, volume 4609 of Lecture Notes in Computer Science, pages 28–53, August 2007.
- [26] Java API documentation. java.util.concurrent.locks.Lock.
- [27] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Gordon Woodhull. Graphviz - open source graph drawing tools. In *Graph Drawing*, pages 483–484, 2001.

- [28] 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.
- [29] 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.
- [30] Cormac Flanagan and Martin Abadi. Object types against races. In International Conference on Concurrency Theory, pages 288–303, 1999.
- [31] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. ACM SIGPLAN Notices, 35(5):219–232, 2000.
- [32] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 256-267, New York, NY, USA, 2004. ACM.
- [33] Cormac Flanagan and Stephen N. Freund. Automatic synchronization correction. In Synchronization and Concurrency in Object-Oriented Languages (SCOOL), 2005.
- [34] Cormac Flanagan, Stephen N. Freund, and Marina 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.
- [35] Cormac Flanagan and Shaz 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.
- [36] Cormac Flanagan and Shaz 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.
- [37] Roberto Giacobazzi, editor. Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings, volume 3148 of Lecture Notes in Computer Science. Springer, 2004.
- [38] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification Second Edition. Addison-Wesley, Boston, Mass., 2000.
- [39] J. N. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. *Readings in database systems (2nd ed.)*, pages 181–208, 1994.
- [40] Khilan Gudka. Implementing atomic sections using lock inference (distinguished project). http://www.doc.ic.ac.uk/~khilan/, June 2007.
- [41] Richard L. Halpert, Christopher J. F. Pickett, and Clark Verbrugge. Componentbased lock allocation. In PACT'07: Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques, September 2007. To appear.

- [42] Tim Harris and Keir Fraser. Language support for lightweight transactions. In OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Objectoriented programing, systems, languages, and applications, pages 388–402, New York, NY, USA, 2003. ACM Press.
- [43] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 48–60, New York, NY, USA, 2005. ACM Press.
- [44] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. SIGPLAN Not., 41(6):14-25, 2006.
- [45] Maurice Herlihy. Wait-free synchronization. ACM Trans. Program. Lang. Syst., 13(1):124-149, 1991.
- [46] Maurice Herlihy. DSTM 2.1, 2008. available from http://www.cs.brown.edu/~mph/.
- [47] Maurice Herlihy, J. Eliot, and B. Moss. Transactional memory: architectural support for lock-free data structures. In in Proceedings of the 20th Annual International Symposium on Computer Architecture, pages 289–300, 1993.
- [48] Hewlett Packard. ftpd: SITE CHMOD / race condition vulnerability, April 1995. http://www.securityfocus.com/advisories/1532.
- [49] 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.
- [50] 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.
- [51] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99), volume 34, pages 132-146, N. Y., 1999.
- [52] International Organization for Standardization. Information technology-Portable Operating System Interface (POSIX). IEEE, 1990. International standard ISO/IEC 9945. IEEE Std 1003.1-1990 (revision of IEEE Std 1003.1-1988). Part 1. System application program interface (API) [C language].
- [53] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods, pages 137-147, Washington, DC, USA, 2005. IEEE Computer Society.
- [54] Josh Bressers. (CVE-2007-5794) CVE-2007-5794 nss_ldap randomly replying with wrong user's data, November 2007. https://bugzilla.redhat.com/show_bug.cgi? id=367461.

- [55] Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. ACM Trans. Program. Lang. Syst., 30(1):1, 2007.
- [56] Eric Koskinen and Maurice Herlihy. Dreadlocks: efficient deadlock detection. In SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures, pages 297–303, New York, NY, USA, 2008. ACM.
- [57] Doug Lea. Concurrent Programming in Java. Second Edition: Design Principles and Patterns. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
- [58] G. Leavens, E. Poll, C. Clifton, Y. Cheon, and C. Ruby. JML reference manual, 2002.
- [59] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. Commun. ACM, 18(12):717-721, 1975.
- [60] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In Proceedings of an ACM conference on Language design for reliable software, pages 128–137, 1977.
- [61] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, Calif., pages 47–57. ACM, January 1988.
- [62] Mark Dowd. Portable openSSH GSSAPI remote code execution vulnerability, September 2006. http://sunsolve.sun.com/search/document.do?assetkey= 1-66-264429-1.
- [63] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. SIGPLAN Not., 41(1):346–358, 2006.
- [64] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures, pages 73–82, New York, NY, USA, 2002. ACM.
- [65] Sun Microsystems. Java virtual machine documentation. http://java.sun.com/ j2se/1.4.2/docs/guide/vm/index.html.
- [66] 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.
- [67] P. Müller. Modular Specification and Verification of Object-Oriented programs, volume 2262. Springer-Verlag, 2002.
- [68] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. SIGPLAN Not., 42(1):327–338, 2007.
- [69] Armand Navabi, Xiangyu Zhang, and Suresh Jagannathan. Quasi-static scheduling for safe futures. In PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pages 23-32, New York, NY, USA, 2008. ACM.

- [70] Bradford Nichols, Dick Buttlar, and Jacqueline Proulx Farrell. Pthreads programming. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.
- [71] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. Principles of Program Analysis. Springer Verlag, second printing edition, 2005.
- [72] James Noble, John Potter, David Holmes, and Jan Vitek. Flexible alias protection. In Proceedings of ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, volume 1445 of Lecture Notes in Computer Science. Springer, 1998.
- [73] America Online. AOLserver, a highly scalable, multi-threaded application server. http://aolserver.com/.
- [74] John K. Ousterhout. Why threads are A bad idea (for most purposes). Presentation given at the 1996 Usenix Annual Technical Conference http://home.pacbell.net/ ouster/threads.pdf.
- [75] David A. Patterson and John L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.
- [76] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S Muchnick and Neil D Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, 1981.
- [77] Nir Shavit and Dan Touitou. Software transactional memory. In Symposium on Principles of Distributed Computing, pages 204–213, 1995.
- [78] Matthew Smith and Sophia Drossopoulou. Cheaper Reasoning with Ownership Types. In ECOOP International Workshop on Aliasing, Confinement and Ownership (IWACO 2003), 2003.
- [79] Sun Microsystems. Race condition security vulnerability in solaris auditing related to extended file attributes may allow local unprivileged users to panic the system, July 2009. http://sunsolve.sun.com/search/document.do?assetkey= 1-66-264429-1.
- [80] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. In Dr. Dobb's Journal, March 2005. http://www.gotw.ca/publications/ concurrency-ddj.htm.
- [81] Herb Sutter. Lock-free code: A false sense of security. In Dr. Dobb's Journal, August 2008. http://www.ddj.com/cpp/209903274.
- [82] Tim Sweeney. The next mainstream programming language: a game developer's perspective. SIGPLAN Not., 41(1):269-269, 2006.
- [83] John D. Valois. Lock-free linked lists using compare-and-swap. In In Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, pages 214–222, 1995.
- [84] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. SIGPLAN Not., 41(1):334–345, 2006.

REFERENCES

[85] Cheng Wang, Victor Ying, and Youfeng Wu. Supporting legacy binary code in a software transaction compiler with dynamic binary translation and optimization. In *CC '08: Proc. International Conference on Compiler Construction*, volume 4959, pages 291–306, March 2008.