A Type and Effect System for Deadlock Avoidance in Low-level Languages

Prodromos Gerakios [1] Nikolaos Papaspyrou [1] Konstantinos Sagonas [1,2]

[1] School of Electrical and Computer Engineering, National Technical University of Athens, Greece
[2] Department of Information Technology, Uppsala University, Sweden
{pgerakios,nickie,kostis}@softlab.ntua.gr

Abstract

The possibility to run into a deadlock is an annoying and commonly occurring hazard associated with the concurrent execution of programs. In this paper we present a polymorphic type and effect system that can be used to dynamically avoid deadlocks, guided by information about the order of lock and unlock operations which is computed statically. In contrast to most other type-based approaches to deadlock freedom, our system does not insist that programs adhere to a strict lock acquisition order or use locking primitives in a block-structured way. Lifting these restrictions is primarily motivated by our desire to target low-level languages, such as C with pthreads, but it also allows our system to be directly applicable in optimizing compilers for high-level languages, such as Java.

To show the effectiveness of our approach, we have also developed a tool that uses static analysis to instrument concurrent programs written in C/pthreads and then links these programs with a run-time system that avoids possible deadlocks. Although our tool is still in an early development stage, in the sense that currently its analysis only handles a limited class of programs, our benchmark results are very promising: they show that it is not only possible to avoid all deadlocks with a small run-time overhead, but also often achieve better throughput in highly concurrent programs by naturally reducing lock contention.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Concurrent programming structures; D.3.2 [*Programming Languages*]: Language Classifications—Concurrent, distributed and parallel languages; D.1.3 [*Software*]: Concurrent Programming—Parallel programming

General Terms Design, Languages, Performance, Theory

Keywords Deadlock avoidance, types and effects, C, pthreads

1. Introduction

In shared memory concurrent programming, deadlocks typically occur as a consequence of cyclic lock acquisition between threads. Two or more threads are deadlocked when each of them is waiting for a lock that has been acquired and is held by another thread. As deadlocks are a serious problem, several methods to achieve deadlock freedom have so far been proposed. In particular, type-based approaches aim for static deadlock freedom guarantees. Most of the proposed type systems in this category [6, 13, 18, 21] *prevent* deadlocks by imposing a strict (non-cyclic) lock acquisition order that must be respected throughout the entire program. However, insisting on a global lock ordering limits programming language expressiveness as many correct programs are rejected unnecessarily. Furthermore, the approach is intrinsically non-modular.

```
efs_lookup(struct inode *dir, struct dentry *dentry) {
  efs_ino_t inodenum;
  struct inode * inode = NULL;

  lock_kernel();
  inodenum = efs_find_entry(dir, dentry->d_name.name,
                                 dentry->d_name.len);
  if (inodenum) {
    if (!(inode = iget(dir->i_sb, inodenum))) {
      unlock_kernel();
      return ERR_PTR(-EACCES);
    }
  }
  unlock_kernel();

  d_add(dentry, inode);
  return NULL;
}
```

Listing 1. Code from Linux's EFS (`linux/fs/efs/namei.c`)

An alternative to deadlock prevention is to employ an approach that dynamically *avoids* deadlocks by utilizing information regarding future lock usage which is provided statically by program analysis. An interesting recent work in this direction is by Boudol [1] who presented a type and effect system for deadlock avoidance when locking is block-structured (e.g. as in Java's `synchronized` blocks). Unfortunately, in Boudol's system the fact that locking is block-structured is a crucial assumption that prohibits the use of his method in many situations. For example, there is a lot of important existing code where locking is used in an unstructured way; cf. the code in Listing 1, which is a typical example of systems code. Furthermore note that in low-level languages such as C, even if the programmer adheres to block-structured locking, this is nothing more than a convention: at the source level, any tool needs to deal with separate lock and unlock primitives. Finally, in almost all languages, the restriction that locking is block-structured is usually lifted at the low-level language of the compiler for optimization purposes. This is the type of languages that our work targets.

More specifically, in this paper we present a type-based method to dynamically avoid deadlocks guided by information about the order of lock and unlock operations which is computed statically via program analysis. The analysis is based on a type and effect system that is general enough to be applicable regardless of how locking is used. Our work is part of a long term effort to design and implement a language at the C-level of abstraction, which has explicit support for shared memory concurrency and provides static guarantees for various safety properties. Chief among these properties are memory safety, freedom from data races, and freedom from deadlocks. In this paper we focus exclusively on the last of them. While our work is primarily targeting low-level languages with unstructured locking, and is applied to multi-threaded C programs using the pthreads library, the main ideas in the type and effect system that we present are generic and language independent. To ease their exposition and simplify the presentation, the language we use in the main sections of this paper is a lambda calculus with recursion, conditionals, and of course primitives for creating, acquiring and releasing re-entrant locks. However, even in this simplified language, unstructured locking primitives and unrestricted lock aliasing introduce significant complexity to the type system compared with block-structured locking, where lock operations always match up with implicit unlock operations. Our type and effect system guarantees that locks are safely released and acquired in the presence of unrestricted lock aliasing.

It should be mentioned that this is not the first system for deadlock avoidance in the presence of unstructured locking that we have developed. In a recent workshop paper [9] we presented a rich type and effect system that, besides deadlock freedom, also guarantees race freedom and memory safety. Its effects contain elements that are pairs (n_1, n_2) associating memory cells with two capability counts: n_1 is a cell reference count, denoting whether the cell is live, while n_2 is the lock count, denoting how many times the cell has been locked (as locks are re-entrant). In addition, capabilities can be either unique or possibly aliased: the type system requires aliasing information so as to determine whether it is safe to pass lock capabilities to new threads. More importantly, it also requires that all functions are annotated with an explicit effect, which is used to type check their body. As a result, that type system is probably unsuitable for a language like C/pthreads; instead, it is relevant for a language like Cyclone [10] where it is commonplace for functions to have annotations. In contrast, the type and effect system we develop in this paper is much simpler. It focuses on deadlock avoidance only, captures the temporal order of lock and unlock operations, and imposes no restrictions with respect to aliasing. More importantly, its implementation is amenable to effect *inference*, and there is no requirement that functions are annotated with explicit effects. Instead, the type and effect system gathers effects and validates them at the beginning of the lexical scope of each lock. This simpler system is thus directly applicable to C/pthreads programs.

In short, the contributions of this paper are as follows:

- we present a polymorphic type and effect system that can be used to dynamically avoid deadlocks, guided by information about the order of lock and unlock operations which is computed statically, in a core language without references but with recursion, conditionals, and primitives for unstructured locking;
- we provide an operational semantics for deadlock avoidance in this language and state and provide proofs of the core soundness properties modeling and guaranteeing deadlock avoidance;
- we show the effectiveness of our approach by running existing C/pthreads programs in our prototype implementation and offer preliminary evidence that the approach is viable in practice.

To make the paper self-contained, we review existing type-based approaches to deadlock freedom (Sect. 2), including the recent work of Boudol, and explain why his approach cannot guarantee deadlock freedom in the presence of unstructured locking (first half of Sect. 3). Most of this material is taken more or less verbatim from our previous workshop paper [9]. In the main body of the current paper, we first describe informally how our approach manages to avoid deadlocks when unstructured locking is used (second half of Sect. 3), and then present the syntax of our language, its operational semantics, and a type and effect system for this language (Sect. 4) which we prove type safe (Sect. 5). To show the effectiveness of our approach, we briefly describe our current implementation (Sect. 6) and its performance (Sect. 7). The paper ends with a comparison of our approach with other techniques for providing deadlock freedom (Sect. 8), and with some concluding remarks.

2. Deadlock Freedom and Related Work

According to Coffman *et al.* [3], a set of threads reaches a *deadlocked state* when the following conditions hold:

- *Mutual exclusion*: Threads claim exclusive control of the locks that they acquire.
- *Hold and wait*: Threads already holding locks may request (and wait for) new locks.
- *No preemption*: Locks cannot be forcibly removed from threads; they must be released explicitly by the thread that acquired them.
- *Circular wait*: Two or more threads form a circular chain, where each thread waits for a lock held by the next thread in the chain.

Therefore, deadlock freedom can be guaranteed by denying at least one of these conditions *before* or *during* program execution. Thus, the following three strategies guarantee deadlock freedom:

- *Deadlock prevention*: At each point of execution, *ensure* that at least one of the above conditions is not satisfied. Thus, programs that fall into this category are correct by design.
- *Deadlock detection and recovery*: A dedicated observer thread *determines* whether the above conditions are satisfied and preempts some of the deadlocked threads, releasing (some of) their locks, so that the remaining threads can make progress.
- *Deadlock avoidance*: Using static information regarding thread resource allocation, *determine* at run time whether granting a lock will bring the program to an *unsafe* state, i.e., a state which can result in deadlock, and only grant locks that lead to safe states.

The majority of literature for language-based approaches to deadlock freedom falls under the first two strategies. In the deadlock prevention category, one finds type and effect systems [2, 6, 13, 18, 21] that guarantee deadlock freedom by statically enforcing a global lock-acquisition ordering, which must be respected by all threads. In this setting, lock handles are associated with type-level lock names via the use of singleton types. Thus, handle $\mathtt{lk}_\iota$ is of type $\mathtt{Lk}(\iota)$. The same applies to lock handle variables. The effect system tracks the order of lock operations on handles or variables and determines whether all threads acquire locks in the same order.

Using a strict lock acquisition order is a constraint we want to avoid. It is not hard to come up with an example that shows that imposing a partial order on locks is too restrictive. The simplest of such examples can be reduced to program fragments of the form:

$$(\mathtt{lock}\ x\ \mathtt{in}\ \ldots\ \mathtt{lock}\ y\ \mathtt{in}\ \ldots) \parallel$$
$$(\mathtt{lock}\ y\ \mathtt{in}\ \ldots\ \mathtt{lock}\ x\ \mathtt{in}\ \ldots)$$

In a few words, there are two parallel threads which acquire two distinct locks, x and y, in reverse order. When trying to find a partial order $\leq$ on locks for this program, the type system or static analysis tool will fail: it will deduce that $x \leq y$ must be true, because of the first thread, and that $y \leq x$ must be true, because of the second. In short, there is no partial order that satisfies these constraints. Thus, programs containing such patterns will be rejected, both in the system of Flanagan and Abadi which requires annotations [6] and in the system of Kobayashi which employs inference [13] as there is no single lock order for *both* threads. Similar considerations apply to the more recent works of Suenaga [18] and Vasconcelos *et al.* [21] dealing with unstructured locking primitives. Finally, such programs cannot be handled even by the type and effect system of Boyapati *et al.* [2], which allows for some controlled changes to the partial order of locks at runtime by permitting conservative updates on directed acyclic lock graphs, because there is no acyclic data structure that captures the cyclic dependencies between locks x and y of this program fragment.

Recently, Boudol developed a type and effect system for deadlock freedom [1], which is based on *deadlock avoidance*. The effect system calculates for each expression the set of acquired locks and annotates lock operations with the "future" lockset. The run-time system utilizes the inserted annotations so that each lock operation can only proceed when its "future" lockset is *available* to the requesting thread. The main advantage of Boudol's type system is that it allows a larger class of programs to type check and thus increases the programming language expressiveness as well as concurrency by allowing arbitrary locking schemes.

The previous example can be rewritten in Boudol's language as follows, assuming that the only lock operations in the two threads are those visible:

$$(\texttt{lock}_{\{y\}}\ x\ \texttt{in}\ \ldots\ \texttt{lock}_{\emptyset}\ y\ \texttt{in}\ \ldots) \parallel$$
$$(\texttt{lock}_{\{x\}}\ y\ \texttt{in}\ \ldots\ \texttt{lock}_{\emptyset}\ x\ \texttt{in}\ \ldots)$$

This program is accepted by Boudol's type system which, in general, allows locks to be acquired in *any* order. At run-time, the first lock operation of the first thread must ensure that y has not been acquired by the second (or any other) thread, before granting x. The second lock operation need not ensure anything, as its future lockset is empty. (The handling is symmetric for the second thread.)

3. Type System Overview

Boudol's work heavily relies on the assumption that locking is block-structured. In fact, the soundness of his system in the presence of lock aliasing is guaranteed by assuming that locks are re-entrant and are released in the reverse order in which they were acquired. In this section, we discuss the main ideas of a novel type system for a simple language with unstructured locking primitives, recursion, and conditionals, which guarantees deadlock freedom and safe use of operations that acquire and release locks in the presence of aliasing. We first show that a naïve extension of Boudol's system is insufficient to guarantee deadlock freedom when locking is unstructured. The example program in Fig. 1(a) illustrates this point: It uses three shared variables, x, y and z, ensuring at each step that no unnecessary locks are held. It is assumed here that the long computations do not acquire or release any locks.[1]

In our naïvely extended (and broken, as we will see) version of Boudol's type and effect system, the program in Fig. 1(a) will type check. The future lockset annotations of the three locking operations in the body of f are $\{y\}$, $\{z\}$ and $\emptyset$, respectively. (This can be easily verified by observing the lock operations between a specific lock and unlock pair.) Now, function f is used by instantiating both x and y with the same variable a, and instantiating z with a distinct variable b. The result of this substitution is shown in Fig. 1(b). The first thing to notice is that, if we want this program to work, locks have to be *re-entrant*. This roughly means that if a thread holds some lock, it can try to acquire the same lock again; this will immediately succeed, but then the thread will have to release the lock *twice*, before it is actually released.

(a) before substitution	(b) after substitution
$\texttt{let}\ f = \lambda x.\,\lambda y.\,\lambda z.$	
$\quad\texttt{lock}_{\{y\}}\ x;$	$\texttt{lock}_{\{a\}}\ a;$
$\quad\textit{some\_long\_computation}\ x;$	$\textit{some\_long\_computation}\ a;$
$\quad\texttt{lock}_{\{z\}}\ y;$	$\texttt{lock}_{\{b\}}\ a;$
$\quad\textit{another\_long\_computation}\ x\ y;$	$\textit{another\_long\_computation}\ a\ a;$
$\quad\texttt{unlock}\ x;$	$\texttt{unlock}\ a;$
$\quad\texttt{lock}_{\emptyset}\ z;$	$\texttt{lock}_{\emptyset}\ b;$
$\quad\textit{another\_long\_computation}\ y\ z;$	$\textit{another\_long\_computation}\ a\ b;$
$\quad\texttt{unlock}\ z;$	$\texttt{unlock}\ b;$
$\quad\texttt{unlock}\ y$	$\texttt{unlock}\ a$
$\texttt{in}\ f\ a\ a\ b$	

Figure 1. A program which is typable by a naïve extension of Boudol's system before substitution (a) but not after (b).

(a) before substitution	(b) after substitution
$\texttt{let}\ f = \lambda x.\,\lambda y.\,\lambda z.$	
$\quad\texttt{lock}_{[y+,\,x-,\,z+,\,z-,\,y-]}\ x;$	$\texttt{lock}_{[a+,\,a-,\,b+,\,b-,\,a-]}\ a;$
$\quad\textit{some\_long\_computation}\ x;$	$\textit{some\_long\_computation}\ a;$
$\quad\texttt{lock}_{[x-,\,z+,\,z-,\,y-]}\ y;$	$\texttt{lock}_{[a-,\,b+,\,b-,\,a-]}\ a;$
$\quad\textit{another\_long\_computation}\ x\ y;$	$\textit{another\_long\_computation}\ a\ a;$
$\quad\texttt{unlock}\ x;$	$\texttt{unlock}\ a;$
$\quad\texttt{lock}_{[z-,\,y-]}\ z;$	$\texttt{lock}_{[b-,\,a-]}\ b;$
$\quad\textit{another\_long\_computation}\ y\ z;$	$\textit{another\_long\_computation}\ a\ b;$
$\quad\texttt{unlock}\ z;$	$\texttt{unlock}\ b;$
$\quad\texttt{unlock}\ y$	$\texttt{unlock}\ a$
$\texttt{in}\ f\ a\ a\ b$	

Figure 2. The program of Fig. 1 with continuation effect annotations; now the program is typable in both cases.

Even with re-entrant locks, however, it is easy to see that the program in Fig. 1(b) does not type check with the present annotations. The first lock operation for a now matches with the *last* (and not the first) unlock operation; this means that a will remain locked during the whole execution of the program. In the meantime b is locked, so the future lockset annotation of the first lock operation should contain b, but it does not. (The annotation of the second lock operation contains b, but blocking there if lock b is not available does not prevent a possible deadlock; lock a has already been acquired.) So, the technical failure of our naïvely extended language is that the preservation lemma breaks. From a more pragmatic point of view, if a thread running in parallel with the thread in Fig. 1(b) already holds b and, before releasing it, is about to acquire a, a deadlock can occur. The naïve extension also fails for another reason: Boudol's system is based on the assumption that calling a function cannot affect the set of locks that are held. This is obviously not true, if non lexically-scoped locking operations are to be supported.

To avoid such problems, our type system precisely tracks effects as a sequence of lock and unlock events. A *continuation effect* of an expression represents the effect of the function code following that expression (i.e., our continuation effects are intra-procedural, in contrast to the work of Hicks *et al.* [11] where the continuation effects are inter-procedural). As shown in the example of Fig. 1, the future lockset for unstructured locking operations cannot be computed statically as a result of lock aliasing. Therefore, the computation of future locksets given the continuation effects is deferred until run-time (i.e., after substitution has taken place), in contrast to Boudol's system.

The program in Fig. 2 is similar to the program in Fig. 1, except that lock operations are now annotated with continuation effects.

[1] For simplicity, in the examples of this section, we assume that there is one (implicit) lock for every shared program variable, which is used to avoid data races when this shared variable is accessed. Therefore, by x we denote both the shared variable x and its implicit lock. As we will see, in Sect. 4 we will simplify presentation even further by completely omitting shared variables and mutable state in general from the language.

$\texttt{let}\ f = \lambda\_.\ \ g()_{[z+]};$
$\qquad\qquad\ \ \texttt{lock}_{\emptyset}\ z\,;$

$\texttt{let}\ g = \lambda\_.\ \ \boxed{\texttt{lock}_{[y+,\,y-]}\ x;}$
$\qquad\qquad\ \ \texttt{lock}_{[y-]}\ y;$
$\qquad\qquad\ \ \texttt{unlock}\ y\,;$

$f()_{[z-,\,x-]};\ \texttt{unlock}\ z;\ \texttt{unlock}\ x$

(a) lock/unlock in different scope

Stack

$z-,\ x-$

$z+$

Lock/Continuation

$x+ \quad y+, y-$

lockset $= \{\,y, z\,\}$

(b) run-time state

Figure 3. An example program where lock and unlock operations are not in the same scope (a) and the run-time state of this program when the boxed term is executed (b).

For example, the annotation $[y+,\ x-,\ z+,\ z-,\ y-]$ at the first lock operation means that in the future (i.e., after this lock operation) y will be acquired, then x will be released, and so on. If x and y are instantiated with distinct values, the run-time system will compute the future lockset $\{y\}$ from the continuation effect. In terms of the continuation effect, $y+$ precedes $x-$ (i.e., the matching unlock operation).

On the other hand, if x and y are instantiated with the same lock handle a and z with b, the continuation effect of the first lock operation becomes $[a+,\ a-,\ b+,\ b-,\ a-]$ and the future lockset is now correctly calculated as $\{a, b\}$: $a+$ and $b+$ precede the matching unlock operation, which is the last $a-$. More generally, the future lockset computation algorithm takes as input a lock x, a continuation effect γ, assumes an empty future lockset and adds all $y+$ events of γ to the future lockset until the matching unlock operation for x is found.

Our continuation effects are intra-procedural, as mentioned earlier. Therefore, the matching unlock operation for y may not be located in γ. We resolve this issue by annotating application terms with their continuation effect. At run-time, when a function application redex is evaluated, its continuation effect is pushed on a stack of continuation effects for the duration of the function evaluation.[2] When the matching unlock operation is not located in a continuation effect, the algorithm proceeds with the remaining continuation effects on the run-time stack. The type system ensures that for each lock operation there exists a matching unlock operation. Therefore, the lockset computation algorithm is guaranteed to terminate. A lock operation succeeds only when both the lock and its future lockset are available. However, the locks in the future lockset are not prematurely acquired, as this would damage the program's degree of parallelism.

Fig. 3(a) illustrates a program where lock and unlock operations reside in different scopes. For instance, x is locked in function g, but it is unlocked in the outermost scope. Application terms are annotated with their continuation effects. For instance, the application of f is annotated with the continuation effect $[z-,\ x-]$ as it is succeeded by two unlock operations on z and x respectively. Fig. 3(b) shows the run-time state of the program when control reaches the lock operation on x: the run-time stack (which grows downwards in the figure) contains the continuation effects of f and g and the lockset computation algorithm starts off with the continuation effect of the lock operation. The algorithm adds $y+$ to the future lockset of x and then considers the continuation effects on the stack, from top to bottom. Thus, $z+$ is added to the future lockset and the matching unlock operation is found on the next element of the stack. The resulting lockset is $\{y, z\}$.

Our language provides support for conditional expressions and recursion. A shortcoming of representing effects as ordered events is that, when typing conditional expressions, it is too restrictive to require that both branches have the same effect. Consider the following example:

$$\begin{array}{l}\texttt{if}\ e\ \texttt{then}\ (\texttt{lock}_{[y]}\ x;\ \ldots\ \texttt{lock}_{\emptyset}\ y;\ \ldots\ \texttt{unlock}\ y)\\ \quad\ \texttt{else}\ (\texttt{lock}_{[x]}\ z;\ \ldots\ \texttt{lock}_{\emptyset}\ x;\ \ldots\ \texttt{unlock}\ z)\end{array}$$

The lock operations of the two branches differ: the effect of the first branch is $[x+,\ y+,\ y-]$ and that of the second is $[z+,\ x+,\ z-]$. Although the overall effect of the two branches (as most programmers understand it) is the same, a simple type and effect system would have to reject this program.

Our system is able to overcome this issue by keeping track of the effects in both branches. For the example shown above, we make the effect of the conditional expression $[x+,\ y+,\ y-]\ ?\ [z+,\ x+,\ z-]$. Given this effect the lockset calculation algorithm computes the lockset of the two branches separately. The resulting lockset is formulated by joining the two locksets. However, for each lock, we impose the restriction that the number of unmatched lock/unlock operations must be equal in both branches.

Additional problems need to be addressed when dealing with recursive function definitions. Consider the following example:

$$\begin{array}{l}\texttt{letrec}\ f = \lambda x.\,\lambda y.\,\lambda z.\\ \quad \texttt{if}\ z > 0\ \texttt{then}\ (\texttt{lock}_{\gamma}\ x;\ \ f\ x\ y\ (z-1);\ \texttt{unlock}\ x)\\ \qquad\qquad\ \texttt{else}\ (\texttt{lock}_{\emptyset}\ y;\ldots\texttt{unlock}\ y;)\end{array}$$

In this case, if we employ the usual typing for `letrec`, the effect of f must equal the effect of its body. However, this is impossible, as the two effects cannot be structurally equivalent: in fact, the effect of f is *contained* in the effect of its body, due to the recursive call. To overcome this issue, our system assigns f a *summary* of the effect of its body. A detailed discussion of effect summaries is deferred until Sect. 4.3 but let us briefly see how our system can infer the effect of function f in the example above. Suppose that γ_f is the (unknown) effect of f. Then, the effect of *the body* of f as a function of γ_f is expressed as

$$\gamma_b(\gamma_f) = ([x+] :: \gamma_f :: [x-])\ ?\ [y+,\ y-]$$

where $\gamma_1 :: \gamma_2$ denotes the appending of two effects γ_1 and γ_2. We are looking for a solution to the equation

$$\gamma_f = \mathsf{summary}(\gamma_b(\gamma_f))$$

At this point, we can start with $\gamma_0 = \emptyset$ (noticing that function f has no unmatched lock or unlock operations) and look for the limit of the sequence $\gamma_{n+1} = \mathsf{summary}(\gamma_b(\gamma_n))$ in other words, for a fixed point of the summarized function's body. We have

$$\gamma_1 = \mathsf{summary}(\gamma_b(\gamma_0)) = \mathsf{summary}([x+,\ x-]\ ?\ [y+,\ y-])$$

Although we have not formally defined what function $\mathsf{summary}$ does, a possible (but conservative) choice here would be to "merge" the effects of the two branches in the summary. (In Sect. 4.3 we discuss how exactly this "merging" takes place and also discuss less conservative alternatives.) Therefore,

$$\gamma_1 = [x+,\ x-,\ y+,\ y-]$$

We can then proceed in the same way and take

$$\gamma_2 = \mathsf{summary}([x+,\ x+,\ x-,\ y+,\ y-,\ x-]\ ?\ [y+,\ y-])$$

If we are outside function f, we don't care if inside f lock x is taken more than once. Nor do we care if x is held or not, at the moment when y is taken. We are just happy to know that x and y are taken and released. Therefore, by merging again:

$$\gamma_2 = [x+,\ x-,\ y+,\ y-] = \gamma_1$$

We reached a fixed point and we can take γ_1 as the summarized effect of function f. Therefore, the effect γ in the annotation of the first lock operation in the example is equal to $[x+,\ x-,\ y+,\ y-,\ x-]$.

[2] As we will see in Sect. 6, this is a constant time operation.

Expression	e	$::=$	$x \mid v \mid (e\ e)^{\xi} \mid (e)\,[r] \mid \texttt{pop}_{\gamma}\ e$
		$\mid$	$\texttt{newlock}\ \rho, x\ \texttt{in}\ e \mid \texttt{lock}_{\gamma}\ e \mid \texttt{unlock}\ e$
		$\mid$	$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e$
Value	v	$::=$	$() \mid \texttt{true} \mid \texttt{false} \mid f \mid \texttt{lk}_{\iota}$
Function	f	$::=$	$\lambda x.\, e \mid \Lambda\rho.\, f \mid \texttt{fix}\ x.\, f$
Type	τ	$::=$	$\langle\rangle \mid \texttt{Bool} \mid \texttt{Lk}(r) \mid \tau \xrightarrow{\gamma} \tau \mid \forall\rho.\, \tau$
Lock	r	$::=$	$\rho \mid \iota$
Calling mode	ξ	$::=$	$\mathsf{seq}(\gamma) \mid \mathsf{par}$
Operation	κ	$::=$	$+ \mid -$
Effect	γ	$::=$	$\emptyset \mid r^{\kappa}, \gamma \mid \gamma\,?\,\gamma, \gamma$

Figure 4. Language and type syntax.

4. Formal Semantics and Metatheory

The syntax of our language is illustrated in Fig. 4, where x and ρ range over term and lock variables, respectively, and ι ranges over lock constants. In this paper, to make the presentation as simple as possible, we do not include any mutable shared state in our language. In other words, we study locks in isolation: locks do not serve any other purpose than thread synchronization (mutual exclusion). Without shared mutable references, locks may seem a bit pointless. However, our primary goal is to develop a simple and understandable type and effect system that guarantees deadlock avoidance. Including shared memory and achieving other interesting properties, such as memory safety and data race freedom, are goals which are more or less orthogonal to deadlock freedom. For one way on how to achieve them, we refer the reader to our previous work [8] and to the workshop paper [9] mentioned in the introduction.

The language core comprises of constants (`true`, `false` and () — the "unit" value), functions (f), and function application. Functions can be monomorphic ($\lambda x.\, e$), lock polymorphic ($\Lambda\rho.\, f$), and recursive ($\texttt{fix}\ x.\, f$). The application of lock polymorphic functions is explicit ($e[r]$, where r is a metavariable ranging over lock constants and variables). The application of monomorphic functions is annotated with a *calling mode* (ξ), which is $\mathsf{seq}(\gamma)$ for normal sequential application and par for parallel application.[3] The semantics of parallel application is that, once the application term is evaluated to a redex, it is moved to a new thread of execution and the spawning thread can proceed with the remaining computation in parallel with the new thread. Conditional expressions ($\texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2$) are standard.

The construct $\texttt{newlock}\ \rho, x\ \texttt{in}\ e$ allocates a fresh lock, which is initially unlocked, and associates it with variables ρ and x within expression e. The type variable ρ is bound to the type-level representation of the fresh lock and allows the type system to statically track uses of it, whereas the term variable x is bound to the fresh lock's handle. Handles can be used as arguments in operations $\texttt{lock}_{\gamma}\ e$ and $\texttt{unlock}\ e$, which have been explained in Sect. 3. It is worth noting that run-time locks are re-entrant, so each lock is associated with a count which is modified after each successful lock/unlock operation. As mentioned, the run-time system inspects the lock annotation γ to determine whether it is safe to lock e.

The term $\texttt{pop}_{\gamma}\ e$ encloses a function body e and cannot exist at the source-level; it only appears during evaluation. The same applies to constant lock handles $\texttt{lk}_{\iota}$.

The syntax of types is more or less standard; a function's type is annotated with the function's effect. Effects (γ) are sequences of events, in the way that was explained in Sect. 3. An atomic event

[3] Notice that sequential application terms are annotated with γ, the *continuation effect*, as mentioned earlier in Sect. 3.

Lock Store	S	$::=$	$\emptyset \mid S, \iota \mapsto n; n; \epsilon; \epsilon$
Threads	T	$::=$	$\emptyset \mid T, n : e$
Configuration	C	$::=$	$S\,; T$
Lockset	ϵ	$::=$	$\emptyset \mid \epsilon, \iota$
Context	E	$::=$	$\Box \mid E[F]$
Frame	F	$::=$	$(\Box\ e)^{\xi} \mid (v\ \Box)^{\xi} \mid (\Box)\,[r] \mid \texttt{pop}_{\gamma}\ \Box$
		$\mid$	$\texttt{lock}_{\gamma_1}\ \Box \mid \texttt{unlock}\ \Box \mid \texttt{if}\ \Box\ \texttt{then}\ e\ \texttt{else}\ e$

Figure 5. Operational semantics syntax and evaluation context.

can either be r^+ or r^-, representing acquire and release operations on a lock handle of type $\texttt{Lk}(r)$. Events also include $\gamma_1\,?\,\gamma_2$, where γ_1 and γ_2 are the continuation effects corresponding to the two branches of a conditional expression.

4.1 Operational Semantics

We define a *small-step* operational semantics for our language in Fig. 5 and 6.[4] The evaluation relation transforms *configurations*. A configuration C consists of an abstract *lock store* S and a thread map T.[5] A store S maps constant locks (ι) to tuples of the form $(n_1; n_2; \epsilon_1; \epsilon_2)$. The first two elements of the tuple are natural numbers representing the thread identifier that owns ι and the count of ι, respectively. The remaining two elements are locksets; they bear no operational significance but are necessary for the type safety proof. The first lockset (ϵ_1) represents the set of all locks in S when ι was last locked (when its n_2 went from zero to one). The second lockset (ϵ_2) represents the future lockset of ι when it was last locked.

A thread map T associates thread identifiers to expressions (i.e., threads). A *frame* F is an expression with a *hole*, represented as $\Box$. The hole indicates the position where the next reduction step can take place. A *thread evaluation context* E is defined as a stack of nested frames. Our notion of evaluation context imposes a call-by-value evaluation strategy to our language. Subexpressions are evaluated in a left-to-right order. We assume that concurrent reduction events can be totally ordered [14]. At each step, a *random* thread (n) is chosen from the thread list for evaluation. Therefore, the evaluation rules are *non-deterministic*.

When a parallel function application redex is detected within the evaluation context of a thread, a new thread is created (rule *E-SN*). The redex is replaced with the unit value in the currently executed thread and a new thread is added to the thread list, with a *fresh* thread identifier. The calling mode of the application term is changed from parallel to sequential, with an empty continuation effect. When evaluation of a thread reduces to a unit value, the thread is removed from the thread list (rule *E-T*). The sequential function application rule (*E-A*) reduces an application redex to a `pop` expression, which contains the body of the function and is annotated with the same effect as the application term. `Pop` expressions are used to form the run-time stack of continuation effects, explained in the example of Fig. 3(b). When the expression contained within a `pop` has been reduced to a value, then enclosing `pop` is removed and the value is returned to the context (rule *E-PP*). The rules for evaluating the application of polymorphic functions (*E-RP*) and recursive functions (*E-FX*) are standard, as well as the rules for evaluating conditionals (*E-IT* and *E-IF*). Rule *E-NG* appends to S a fresh lock ι, which is initially unlocked.

The most interesting rule is *E-LK0*, which dynamically computes the future lockset (ϵ) of lock ι. To achieve this, function stack

[4] A full formalization is given in the Appendix.

[5] The order of elements in comma-separated lists, e.g., in a store S or in a list of threads T, is unimportant; we consider all list permutations as equivalent. However, in sequences (e.g., effects), order is important.

$$\frac{\text{fresh } n'}{S;T,n:E[(v'\ v)^{\mathsf{par}}] \rightsquigarrow S;T,n:E[()],n':\Box[(v'\ v)^{\mathsf{seq}(\emptyset)}]}\ \textit{(E-SN)} \qquad \frac{}{S;T,n:\Box[()] \rightsquigarrow S;T}\ \textit{(E-T)}$$

$$\frac{}{S;T,n:E[((\lambda x.\,e_1)\ v)^{\mathsf{seq}(\gamma)}] \rightsquigarrow S;T,n:E[\mathtt{pop}_\gamma\ e_1[v/x]]}\ \textit{(E-A)} \qquad \frac{}{S;T,n:E[\mathtt{pop}_\gamma\ v] \rightsquigarrow S;T,n:E[v]}\ \textit{(E-PP)}$$

$$\frac{}{S;T,n:E[(\Lambda\rho.\,f)[\iota]] \rightsquigarrow S;T,n:E[f[\iota/\rho]]}\ \textit{(E-RP)} \qquad \frac{v' = \mathtt{fix}\ x.\,f}{S;T,n:E[(v'\ v)^{\mathsf{seq}(\gamma)}] \rightsquigarrow S;T,n:E[(f[v'/x]\ v)^{\mathsf{seq}(\gamma)}]}\ \textit{(E-FX)}$$

$$\frac{}{S;T,n:E[\mathtt{if\ true\ then}\ e_1\ \mathtt{else}\ e_2] \rightsquigarrow S;T,n:E[e_1]}\ \textit{(E-IT)} \qquad \frac{}{S;T,n:E[\mathtt{if\ false\ then}\ e_1\ \mathtt{else}\ e_2] \rightsquigarrow S;T,n:E[e_2]}\ \textit{(E-IF)}$$

$$\frac{\text{fresh } \iota \qquad S' = S, \iota \mapsto n;0;\emptyset;\emptyset}{S;T,n:E[\mathtt{newlock}\,\rho,x\ \mathtt{in}\ e_1] \rightsquigarrow S';T,n:E[e_1[\iota/\rho][\mathtt{lk}_\iota/x]]}\ \textit{(E-NG)}$$

$$\frac{\begin{array}{c} S(\iota) = n_1;0;\epsilon_1;\epsilon_2 \qquad S' = S[\iota \mapsto n;1;\mathsf{dom}(S);\epsilon] \\ \epsilon = \mathsf{run}(\mathsf{stack}(E[\mathtt{pop}_{\gamma_1}\ \Box]),\iota,1) \qquad \epsilon \cup \{\iota\} \subseteq \mathsf{available}(S,n) \end{array}}{S;T,n:E[\mathtt{lock}_{\gamma_1}\ \mathtt{lk}_\iota] \rightsquigarrow S';T,n:E[()]}\ \textit{(E-LK0)}$$

$$\frac{S(\iota) = n;n_2;\epsilon_1;\epsilon_2 \qquad n_2 > 0 \qquad S' = S[\iota \mapsto n;n_2+1;\epsilon_1;\epsilon_2]}{S;T,n:E[\mathtt{lock}_{\gamma_1}\ \mathtt{lk}_\iota] \rightsquigarrow S';T,n:E[()]}\ \textit{(E-LK1)} \qquad \frac{S(\iota) = n;n_2;\epsilon_1;\epsilon_2 \qquad n_2 > 0 \qquad S' = S[\iota \mapsto n;n_2-1;\epsilon_1;\epsilon_2]}{S;T,n:E[\mathtt{unlock}\,\mathtt{lk}_\iota] \rightsquigarrow S';T,n:E[()]}\ \textit{(E-UL)}$$

Figure 6. Operational semantics.

assembles the overall (stacked) continuation effect by concatenating the continuation effect annotations of pop expressions that are found in the stack of the evaluation context. The lockset computation is modeled by function $\mathsf{run}(\gamma,\iota,k)$, which accepts the stacked effect γ, the lock ι whose lockset is to be computed and the number k of unmatched unlock events (ι^-) in the stack. It returns a subset of the lock events (r^+) located in the stack, such that each element of the subset is locked *before* the last unmatched unlock operation of ι. Function run is defined only when all unlock events for ι are found in the stacked effect. The future lockset of ι (ϵ) is equal to $\mathsf{run}(\gamma,\iota,1)$. Rule *E-LK0* also requires that both ι and its future lockset are available — $\epsilon \cup \{\iota\} \subseteq \mathsf{available}(S,n)$. Function available takes as input a lock store S and a thread identifier n and returns a set of locks, such that each element of the set can be acquired by thread n (i.e., locks whose thread identifier either equals n or their count is zero). If the availability premise holds, the lock count of ι is set to one and the thread identifier is set to n. In addition, both ϵ_1 and ϵ_2 (the last two elements of $S(\iota)$) are replaced with $\mathsf{dom}(S)$ (all locks allocated in the program) and ϵ, respectively.

The rules for acquiring or releasing a held lock (*E-LK1* or *E-UL*) require that the count of that lock is positive and that it is owned by the thread that is performing the unlock/lock operation. Otherwise, the semantics will get stuck. We will soon present a type system for this language and also the type safety formulation that guarantees that well-typed programs cannot reach a stuck state.

Note that, although rule *E-LK0* ensures that all locks in the future lockset ϵ are available before proceeding, our semantics only acquires the requested lock ι and not any of the locks in ϵ. As a possible optimization, an implementation could choose to acquire additionally some subset $\epsilon' \subseteq \epsilon$. These locks are all available at this point and an implementation might not want to recheck for their availability and more importantly risk having to wait for them at the time they are needed, in case some other thread has got hold of them until then. Pre-acquisition of locks, however, may reduce parallelism and an implementation should use it only when an analysis shows that the locks will definitely be needed, and not "too late" in the future. (Additional information could statically be placed in the effects to guide such an implementation.) The type safety of our system, stated in Sect. 5, can be proved even if the semantics pre-acquires a subset of the future lockset in this rule.

4.2 Static Semantics

The syntax of types and effects is given in Fig. 4 (on page 5). Basic types consist of the boolean and the unit type, denoted by $\langle\rangle$; lock handle types $\mathtt{Lk}(r)$ are singleton types parameterized by a type-level lock name r; and monomorphic function types carry the function's effect. Effects (γ) are used to statically track lock ownership information; they are ordered *sequences* of events, which can be either r^κ or $\gamma_1\,?\,\gamma_2$.

The typing relation is denoted by $M;\Delta;\Gamma \vdash e : \tau\,\&\,(\gamma;\gamma')$. It takes an expression e, the typing context $M;\Delta;\Gamma$, and an input effect γ, and produces the type τ assigned to expression e as well as an output effect γ'. Here, M is a set of lock constants, Δ is a set of lock variables, and Γ is a mapping of term variables to types.

As lock operations and application terms are annotated with their continuation effect, it is natural that effects *flow backwards* through the type system: the input effect to an expression e represents the events that follow in the future of e, that is, after e is evaluated. On the other hand, the output effect represents the combined sequence of events caused by e and its future. In fact, the typing relation does not modify the input effect but rather appends to it: the input effect is always a *suffix* of the output effect, in chronological order. (This is ensured by the typing relation and the typing context well-formedness.) The typing rules are given in Fig. 7.[6] The typing rules *T-U*, *T-T*, *T-F*, *T-V*, *T-L*, *T-RF*, *T-RP* and *T-FN* are standard. Notice, that in the case of rule *T-FN*, the input effect of the function's body e_1 is empty. The typing rule for sequential function application (*T-SA*) appends the input effect γ to the function's effect γ_a and propagates the new effect to expression e_2, which in turn propagates its output effect to e_1. The output effect of the sequential function application is the output effect of expression e_1. The annotation of the application must match with the input effect γ. Rule *T-PP* acts as a bridge between the body of a function that is being executed and its calling environment, by appending the continuation effect to the effect of the function's body. The rule for parallel application (*T-PA*) is similar to the sequential application rule, except that the function's effect (γ_a) is not combined with the input effect (as the function will be evaluated in a new thread) and the function's return type must be unit. In addition, all locks in the function's effect must be released before and after the function's execution — $\forall r.\, r;0 \vdash_{ok} \gamma_a$. The relation $r;n \vdash_{ok} \gamma$ checks

[6] A complete formalization appears in the Appendix.

$$\frac{M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma}{M;\Delta;\Gamma \vdash () : \langle\rangle \,\&\, (\gamma;\gamma)}\ \textit{(T-U)} \qquad \frac{x : \tau \in \Gamma \quad M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma}{M;\Delta;\Gamma \vdash x : \tau \,\&\, (\gamma;\gamma)}\ \textit{(T-V)}$$

$$\frac{M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma \quad \tau \equiv \tau_1 \xrightarrow{\gamma_b} \tau_2 \quad M;\Delta \vdash \tau \quad M;\Delta;\Gamma, x : \tau_1 \vdash e_1 : \tau_2 \,\&\, (\emptyset;\gamma_b)}{M;\Delta;\Gamma \vdash \lambda x.\, e_1 : \tau \,\&\, (\gamma;\gamma)}\ \textit{(T-FN)}$$

$$\frac{M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma}{M;\Delta;\Gamma \vdash \mathtt{true} : \mathtt{Bool} \,\&\, (\gamma;\gamma)}\ \textit{(T-T)} \qquad \frac{M;\Delta,\rho;\Gamma \vdash f : \tau \,\&\, (\gamma;\gamma)}{M;\Delta;\Gamma \vdash \Lambda\rho.\, f : \forall\rho.\,\tau \,\&\, (\gamma;\gamma)}\ \textit{(T-RF)}$$

$$\frac{r \in M \cup \Delta \quad M;\Delta;\Gamma \vdash e_1 : \forall\rho.\,\tau \,\&\, (\gamma;\gamma')}{M;\Delta;\Gamma \vdash (e_1)\,[r] : \tau[r/\rho] \,\&\, (\gamma;\gamma')}\ \textit{(T-RP)}$$

$$\frac{M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma}{M;\Delta;\Gamma \vdash \mathtt{false} : \mathtt{Bool} \,\&\, (\gamma;\gamma)}\ \textit{(T-F)} \qquad \frac{M;\Delta \vdash \gamma \quad M;\Delta;\Gamma \vdash e : \tau \,\&\, (\emptyset;\gamma')}{M;\Delta;\Gamma \vdash \mathtt{pop}_\gamma\ e : \tau \,\&\, (\gamma;\gamma' :: \gamma)}\ \textit{(T-PP)}$$

$$\frac{M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma \quad \iota \in M}{M;\Delta;\Gamma \vdash \mathtt{lk}_\iota : \mathtt{Lk}(\iota) \,\&\, (\gamma;\gamma)}\ \textit{(T-L)}$$

$$\frac{M;\Delta;\Gamma \vdash e : \mathtt{Lk}(r) \,\&\, (r^+,\gamma;\gamma')}{M;\Delta;\Gamma \vdash \mathtt{lock}_\gamma\, e : \langle\rangle \,\&\, (\gamma;\gamma')}\ \textit{(T-LK)} \qquad \frac{M;\Delta;\Gamma \vdash e : \mathtt{Lk}(r) \,\&\, (r^-,\gamma;\gamma')}{M;\Delta;\Gamma \vdash \mathtt{unlock}\, e : \langle\rangle \,\&\, (\gamma;\gamma')}\ \textit{(T-UL)}$$

$$\frac{M;\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_a} \tau_2 \,\&\, (\gamma_1;\gamma') \quad M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_a :: \gamma;\gamma_1)}{M;\Delta;\Gamma \vdash (e_1\ e_2)^{\mathsf{seq}(\gamma)} : \tau_2 \,\&\, (\gamma;\gamma')}\ \textit{(T-SA)}$$

$$\frac{\forall r \in \mathsf{dom}(\gamma_a).\ r;0 \vdash_{ok} \gamma_a \quad M;\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_a} \langle\rangle \,\&\, (\gamma_1;\gamma') \quad M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma;\gamma_1)}{M;\Delta;\Gamma \vdash (e_1\ e_2)^{\mathsf{par}} : \langle\rangle \,\&\, (\gamma;\gamma')}\ \textit{(T-PA)}$$

$$\frac{M;\Delta \vdash \tau \quad \rho \notin \mathsf{dom}(\gamma) \quad \rho;0 \vdash_{ok} \gamma' \quad M;\Delta,\rho;\Gamma, x : \mathtt{Lk}(\rho) \vdash e_1 : \tau \,\&\, (\gamma;\gamma')}{M;\Delta;\Gamma \vdash \mathtt{newlock}\ \rho, x\ \mathtt{in}\ e_1 : \tau \,\&\, (\gamma;\gamma' \setminus \rho)}\ \textit{(T-NG)}$$

$$\frac{M;\Delta;\Gamma, x : \tau_a \vdash f : \tau_b \,\&\, (\gamma;\gamma) \quad \tau_a \equiv \tau_1 \xrightarrow{\gamma_a} \tau_2 \quad \tau_b \equiv \tau_1 \xrightarrow{\gamma_b} \tau_2 \quad \gamma_a = \mathsf{summary}(\gamma_b)}{M;\Delta;\Gamma \vdash \mathtt{fix}\ x.\, f : \tau_a \,\&\, (\gamma;\gamma)}\ \textit{(T-FX)}$$

$$\frac{M;\Delta;\Gamma \vdash e_1 : \mathtt{Bool} \,\&\, (\gamma_1 \mathbin{?} \gamma_2, \gamma;\gamma') \quad M;\Delta;\Gamma \vdash e_2 : \tau \,\&\, (\gamma;\gamma_1 :: \gamma) \quad M;\Delta;\Gamma \vdash e_3 : \tau \,\&\, (\gamma;\gamma_2 :: \gamma)}{M;\Delta;\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau \,\&\, (\gamma;\gamma')}\ \textit{(T-IF)}$$

Figure 7. Typing rules.

that there exist exactly n unmatched unlock events in γ for lock r (it is used at the same time to make sure that r is never released more times than it has been acquired).

The rule for typing recursive functions (*T-FX*) is the standard one, if we ignore the effects γ_a and γ_b on the function types. As mentioned in Sect. 3, it may be impossible to assign the recursive function variable x the same effect as the function body f (i.e., γ_b). The intuition here is that x must be assigned an effect γ_a that *summarizes* γ_b, and this effect can be computed as a least fixed point with the procedure that was sketched in Sect. 3. We postpone the discussion on summaries for a little longer, until Sect. 4.3.

The rule for creating new locks (*T-NG*) passes the input effect γ to e_1, the body of `let`, assigns the lock handle variable x the singleton type $\mathtt{Lk}(\rho)$ and adds ρ to the lock variable context for the scope of e_1. The output effect of the lock creation construct is equal to the output effect of e_1 minus any events of the form ρ^κ. The rule also requires that ρ is unlocked before and after the execution of e_1 — $\rho;0 \vdash_{ok} \gamma'$. Rule *T-LK* prepends r^+ to the input effect and propagates the resulting effect to e_1. Notice, that the input effect must match the lock annotation (the continuation effect of the lock operation must be valid). The typing rule *T-UL* prepends r^- to the input effect and propagates the resulting effect to e_1.

The typing rule for conditional expressions (*T-IF*) propagates the input effect of the conditional expression to its branches e_2 and e_3 respectively. We know that γ is a common suffix of the output effect of e_2 and e_3. Let us assume that γ_1 and γ_2 are the prefixes of the two branches respectively. Thus, the input effect of the guard expression e_1 is $\gamma_1 \mathbin{?} \gamma_2, \gamma$, which tells us that the type system records the effects of both branches but it does not unify them.

The typing rules *T-SA*, *T-LK* and *T-PP* ensure that the effect annotations in sequential applications, `lock` and `pop` expressions are equal to the expression's input effect. This means that, even in this language (and much more so in a language like C), programmers are not really expected to explicitly annotate such expressions: it is easy for the type and effect system to infer the annotations.

4.3 Summarizing Recursive Functions

We have already discussed why it is necessary to summarize the effects of recursive functions. However, the function $\mathsf{summary}$ can be correctly defined in different ways. In principle, any possible definition will do, as long as it satisfies Lemmata 1 and 2.

Lemma 1 (Consistency of Summary). *Let σ be a substitution of lock variables with lock constants and γ_s be a continuation effect. If $\gamma_a = \mathsf{summary}(\gamma_b)$ then for all ι and n we have*

$$\mathsf{run}(\sigma(\gamma_b :: \gamma_s), \iota, n) \subseteq \mathsf{run}(\sigma(\gamma_a :: \gamma_s), \iota, n)$$

Before we proceed to Lemma 2, we provide an informal definition for function $\mathsf{startup}$. This function takes an effect γ and finds all unmatched lock and unlock operations in γ. It produces an effect γ' which has all the unmatched lock operations, followed by all the unmatched unlock operations. E.g.

$$\mathsf{startup}([x+,\ x-] \mathbin{?} [y+,\ y-]) = \emptyset$$
$$\mathsf{startup}([z-,\ y+] \mathbin{?} [x+,\ y+,\ x-,\ z-]) = [y+,\ z-]$$

We can also define the notion of compositionality for functions on effects. Informally, a function $F(\gamma)$ is *compositional* if γ can only be used as a sub-effect in the result (i.e. in the way that our type and effect system uses effects).

Lemma 2 (Fixed Point of Summary). *Let $F(\gamma)$ be a compositional function, $\gamma_0 = \mathsf{startup}(F(\emptyset))$, and $\gamma_{n+1} = \mathsf{summary}(F(\gamma_n))$. Then there exists a k such that for all $n > k$ we have $\gamma_k = \gamma_n$.*

If a $\mathsf{summary}$ function satisfies Lemma 2, then the procedure described in Sect. 3 can be used to compute the fixed point of all recursive functions. This fixed point can be used by the type system to determine type τ_a in rule *T-FX*. Furthermore, if a $\mathsf{summary}$ function satisfies Lemma 1, then it is safe for the run-time system to use the summarized effect in the place of the real effect of a function's body. In all cases, the future lockset that will be computed based on the summary will be a superset of the future lockset that would be computed based on the body's real effect.

In our implementation, we use a conservative function $\mathsf{summary}$ that can be shown to satisfy Lemmata 1 and 2. For any effect γ, we define $\mathsf{summary}(\gamma)$ as follows. We take $\mathsf{startup}(\gamma)$ and split it in two components: γ_+, which contains the unmatched lock operations, and γ_-, which contains the unmatched unlock operations. We reorder the events in γ_+ and γ_- using any total order relation on lock variables ρ. (This *normalization* is required for ensuring that a fixed point exists — Lemma 2.) We then build a third component: γ_0, which contains one pair of $[x+,\ x-]$ for each lock x that is acquired at any time in γ, excluding the ones that are in γ_+. Again, we normalize γ_0 by reordering the events that it contains. Finally, we take $\mathsf{summary}(\gamma) = \gamma_+ :: \gamma_0 :: \gamma_-$.

As an example, consider the conditional statement of Sect. 3, copied here without the annotations to lock operations:

```
if e then (lock x; ... lock y; ... unlock y)
     else (lock z; ... lock x; ... unlock z)
```

The corresponding effect is:

$$\gamma = [x+,\ y+,\ y-]\ ?\ [z+,\ x+,\ z-]$$

There is one unmatched lock operation (for x, which occurs in both branches of the conditional), therefore $\mathsf{startup}(\gamma) = [x+]$. We take $\gamma_+ = [x+]$ and $\gamma_- = \emptyset$. Then, we build $\gamma_0 = [y+,\ y-,\ z+,\ z-]$, by taking one matching pair for each of the lock operations that occur in γ and are not contained in γ_+ (these are $y+$ and $z+$, and we order lock variables lexicographically). Thus:

$$\mathsf{summary}(\gamma) = [x+,\ y+,\ y-,\ z+,\ z-]$$

More accurate summary functions can also be constructed, not merging branching effects and respecting the nested structure of lock/unlock operations. However, we are not convinced of their practical importance and, in particular, whether the future locksets $\mathsf{run}(\sigma(\gamma_a :: \gamma_s), \iota, n)$ that they produce are indeed more accurate.

As a last note here, summarization is not only necessary for dealing with recursive functions. It is useful for reducing the size of the effects of non-recursive functions, to improve the performance of the run-time system.

5. Type Safety and Deadlock Freedom

In this section we present the fundamental theorems that prove type safety for our language, together with very brief proof sketches.[7] Type safety, which in this system implies deadlock freedom, is based on proving the *preservation*, *deadlock freedom* and *progress* lemmata. Informally, a program written in our language is safe when each thread of execution can perform an evaluation step or is waiting for a lock (*blocked*). In addition, there must not exist threads that have reached a deadlocked state.

As discussed in Sect. 4.1, a thread may become stuck when it performs an ill-typed operation, or when it attempts to compute the future lockset of a malformed stack, or when it attempts to acquire a non-existing lock, or when it attempts to release a lock whose count has already reached the value zero, and so on.

Definition 1 (Thread Effect Consistency). *The following rules define* effect-consistent *threads.*

$$\frac{\iota; n_1 \vdash_{ok} \gamma \qquad \epsilon_3 = \mathsf{run}(\gamma, \iota, n_1) \qquad n;\gamma \vdash S \qquad \epsilon_1 \cap \epsilon_3 \subseteq \epsilon_2}{n;\gamma \vdash S, \iota \mapsto n; n_1; \epsilon_1; \epsilon_2}$$

$$\frac{\iota; 0 \vdash_{ok} \gamma \qquad n \neq n_1 \qquad n;\gamma \vdash S}{n;\gamma \vdash S, \iota \mapsto n_1; n_2; \epsilon_1; \epsilon_2} \qquad \frac{}{n;\gamma \vdash \emptyset}$$

Thread effect consistency (denoted by $n;\gamma \vdash S$) ensures that any lock acquired by thread n will be released before thread n terminates. Furthermore, it establishes an exact correspondence between locks in γ and S. In particular, for each lock ι in the domain of γ, $\iota; n_1 \vdash_{ok} \gamma$ must hold, where n_1 must equal the reference count of ι in S for each thread n. Notice that only one thread can have a positive reference count for ι. It also establishes that the future lockset of an acquired lock at any program point (ϵ_3 — modulo the locations that have been created *after* the lock was initially acquired) is *always* a subset of the future lockset computed when the lock was initially acquired (ϵ_2).

[7] A full formalization of our language and complete proofs are given in the Appendix.

Definition 2 (Thread Typing). *The following rules define* well typed *threads.*

$$\frac{}{S; M \vdash \emptyset} \qquad \frac{M;\emptyset;\emptyset \vdash e : \langle\rangle \,\&\, (\emptyset;\gamma) \qquad S; M \vdash T \qquad n \notin \mathsf{dom}(T) \qquad n;\gamma \vdash S}{S; M \vdash T, n : e}$$

A collection of threads T is well typed w.r.t. a lock store S and a set of lock identifiers M, if for each thread $n : e$, expression e is well-typed with an empty input effect and some output effect γ and the lock store is consistent w.r.t. n and γ.

Definition 3 (Configuration Typing). *A configuration $S; T$ is* well typed *w.r.t. M (we denote this by $M \vdash S; T$) when $S; M \vdash T$ and $M = \mathsf{dom}(S)$.*

Definition 4 (Deadlocked State). *A configuration has reached a* deadlocked state *when there exist a set of threads $n_0, \ldots, n_k$, for $k > 0$, and a set of locks $\ell_0, \ldots, \ell_k$, such that each thread n_ι has acquired lock $\ell_{(\iota+1) \bmod (k+1)}$ and is waiting for lock ℓ_ι.*

Definition 5 (Not Stuck). *A configuration $S; T$ is* not stuck *when each thread in T can take one of the evaluation steps in Fig. 6 or is waiting for a lock held by some other thread.*

Given these definitions, we can now present the main results of this paper. The *progress*, *deadlock freedom* and *preservation* lemmata are formalized at the *program* level, i.e., for all concurrently executed threads. Let expression e be the initial program. The initial program configuration $S_0; T_0$ is defined by taking $S_0 = \emptyset$, and $T_0 = \{\emptyset; e\}$.

Lemma 3 (Deadlock Freedom). *If the initial configuration takes n steps, where each step is well-typed for some M, then the resulting configuration has not reached a deadlocked state.*

Proof sketch. We assume that a cyclic set of threads exists, in the sense of Def. 4. Let m be the thread that first acquires its lock ℓ_k. When thread k subsequently acquires its lock ℓ_o, we know that ℓ_k does not belong to the corresponding future lockset (otherwise the lock could not have been acquired). We show that this is a contradiction, using the effect consistency of the store and the threads' typing.

Lemma 4 (Progress). *If $S; T$ is a closed well-typed configuration with $M \vdash S; T$, then $S; T$ is not stuck.*

Proof sketch. Let $n : e$ be a thread in T. It suffices to show that e can take a step or is waiting for a lock held by some other thread. As $S; T$ is well-typed, we know that e is well typed with type $\langle\rangle$. If it is a value, the proof is trivial. Otherwise, e is of the form $E[u]$ where u is a redex, and we proceed by a case analysis on u. In each case, based on what the typing derivation gives us, we can deduce that either u can take a step, or that it is blocked.

Lemma 5 (Preservation). *Let $S; T$ be a well-typed configuration with $M \vdash S; T$. If the operational semantics takes a step $S; T \leadsto S'; T'$, then there exists $M' \supseteq M$ such that the resulting configuration is well-typed with $M' \vdash S'; T'$.*

Proof sketch. By induction on the thread evaluation relation.

LEMMA 6 (Multi-step Preservation). *Let* $S_0; T_0$ *be a* closed well-typed configuration *for some* M_0 *and assume that* $S_0; T_0$ *evaluates to* $S_n; T_n$ *in n steps. Then for all* $\imath \in [0, n]$ $M_\imath \vdash S_\imath; T_\imath$ *holds.*

Proof. Proof by induction on the number of steps n using Lemma 5.

THEOREM 1 (Type Safety). *If the initial configuration* $S_0; T_0$ *is closed and well-typed* ($\emptyset \vdash S_0; T_0$) *and the operational semantics takes any number of steps* $S_0; T_0 \leadsto^n S_n; T_n$*, then the resulting configuration* $S_n; T_n$ *is not stuck and* T_n *has not reached a deadlocked state.*

Proof. The application of Lemma 6 to the typing derivation of $S_0; T_0$ implies that for all steps from zero to n there exists an $M_\imath$ such that $M_\imath \vdash S_\imath; T_\imath$. Therefore, Lemma 3 implies that T_n has not reached a deadlocked state and Lemma 4 implies $S_n; T_n$ is not stuck.

Using an empty typing context for typing the initial configuration $S_0; T_0$ guarantees that all functions in the program are closed and that no explicit lock values ($\mathtt{lk}_\imath$) are used in the source of the original program.

6. Prototype Implementation

We have implemented our approach for programs written in C using the pthreads library. Our tool uses CIL [15] to parse and analyze C source code, as well as some modules from the implementation of RELAY [22] that perform pointer analyses.[8]

As in the formal semantics, our approach guarantees deadlock freedom by combining static analysis and dynamic checks. Therefore, our tool performs a source-to-source transformation that instruments the original C code with meta-data representing future lock usage. The instrumented C program is then linked with a run-time library that provides replacements for some pthreads functions. We provide a very brief description of our tool below.

Static Analysis Our tool performs a bottom-up traversal of the program call graph and computes the effect of each function with a standard forward intra-procedural effect analysis. Effects flowing from back edges of a node must be equivalent (with respect to the unmatched lock and unlock operations) to effects flowing from front edges in the same node. Therefore, loops and goto statements can perform arbitrary locking operations, but they must not surprise their environment. Indirect calls (i.e., function pointers) are treated by computing the set of all possible aliases for each function pointer and assigning a new join effect, whose branches represent the effects of all aliased functions. The effect of recursive calls is computed in a manner similar to that described earlier for the formal language. Currently, the effect inference component of the tool is incomplete in the sense that it does not handle all C/pthreads programs. It cannot deal with non-local jumps, dynamically allocated data structures containing locks and rejects programs with arithmetic on pointers (including arrays) that contain or point to locks. Stack-allocated lock handles are also not supported. Lock handle variables cannot be directly used in the program (e.g., parameter passing or assignment) but only through the use of pointers. Lifting these limitations of the analysis is the target of future work.

Code Generation and Run-time System Our main goal for the run-time system was to minimize the overhead induced by "effect accounting". A naïve implementation of the formal semantics would simply allocate and initialize effect frames for each function call and this would be unacceptable in terms of performance.

[8] Our tool and the benchmark programs we use in Sect. 7 are available from `http://www.softlab.ntua.gr/~pgerakios/deadlocks/`.

The code generation phase statically creates a *single* block of initialization code for the effect of each function and inserts effect index update instructions (i.e., a single assignment) before each call and lock operation. Therefore, the overhead imposed for such operations is minimal. Each function is also instrumented with instructions for pushing and popping effects from the run-time stack at function entry and exit points respectively. This imposes a constant cost to function calls independently of the effect's size. The run-time system extends the standard implementation of locking functions such as the pthreads functions `pthread_mutex_lock` and `pthread_cond_wait`. If a lock is already held by the requesting thread then the lock's count is simply incremented. Otherwise, the run-time system computes the future lockset of the requested lock from the current effect and verifies that all locks in the future lockset are available when the lock is acquired.

7. Performance Evaluation

In this section we describe our experimental results, aiming to demonstrate that our approach can achieve deadlock freedom with relatively low run-time overhead. The experiments were performed on a multiprocessor machine with four Intel Xeon E7340 CPUs (2.40 GHz), having a total of 16 cores and 16 GB of RAM. In the benchmarks involving network interaction, a second machine was also used with two Intel Xeon CPUs (2.80 GHz), having a total of 4 cores and 4 GB of RAM. Both machines were running Linux 2.6.26-2-amd64 and GCC 4.3.2.

We used a total of six benchmark programs of varying complexity: two written by us (these are programs from the literature which are known to exhibit deadlocks) and four which are real, publicly available applications. Except for the first program, whose only purpose is to show that our approach avoids deadlocks, the rest of the benchmarks are used to evaluate scalability as the number of threads increases, and to demonstrate that our approach not only avoids deadlocks but, in some cases, may result in better parallelism (dining philosophers).

bank transactions: a small multithreaded program simulating repeated concurrent circular transactions between two accounts that may deadlock. The program is based on the example used in the introduction of Boudol's paper [1]. Each transaction consists of a withdrawal step from account A and a deposit step to account B, each of which is protected by a lock. In addition, the whole transaction is protected by the lock account A; this creates a necessity for recursive (reentrant) locks and makes the program prone to deadlocks. The original program deadlocks with probability very close to 100%.

dining philosophers: a program implementing the obvious and deadlock-prone attempt to solve the classic multi-process synchronization problem. Each philosopher first picks up the stick on his left, then the stick on his right. The original program deadlocks with a probability that decreases as the number of philosophers increases (for five philosophers, the probability for deadlock was roughly 70%) but increases again when the number of philosophers exceeds the number of available cores. For the performance comparison that we discuss below, we only used the deadlock-free runs of the original program.

The performance of the original and the instrumented versions are shown in Fig. 8. For a given elapsed time (2 secs) we measured the total number of times that the philosophers ate (using a per-thread random number generator that was identical in both versions). It is interesting to see that in the instrumented program, the number grows linearly with the number of philosophers, i.e., each philosopher eats for a (more or less) constant number of times during the 2 secs (this number is determined by

n	original	instrumented	improvement
5	126,536	126,961	0.34%
10	224,536	230,981	2.87%
15	284,150	298,563	5.07%
31	536,889	587,051	9.34%
63	1,080,322	1,193,509	10.48%
127	1,603,880	2,219,022	38.35%
255	1,480,183	4,220,603	185.14%

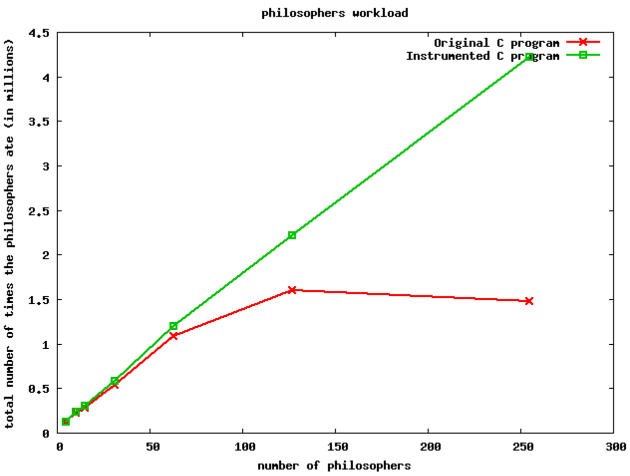

Figure 8. Performance comparison for the *dining philosophers*. We measure the total number of times that the n philosophers ate.

n	original			instrumented			overhead
	U	S	e	U	S	e	
1	7.91	7.18	14.30	8.35	7.40	14.66	2.52%
2	8.08	7.30	9.71	9.53	8.02	9.51	−2.06%
4	8.48	7.69	7.19	7.32	7.33	6.39	−11.13%
8	9.21	9.09	5.16	10.04	9.58	5.32	3.10%
16	12.02	10.42	4.94	14.48	12.42	5.45	10.32%
32	12.72	11.23	5.32	13.84	12.89	6.34	19.17%
64	12.71	11.14	5.34	13.56	12.32	5.66	5.99%

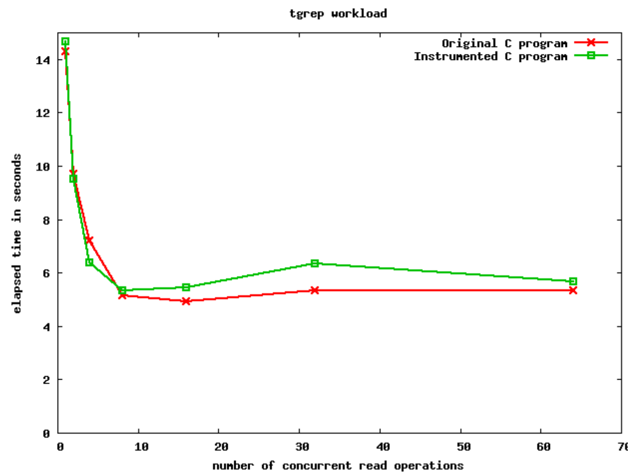

Figure 9. Performance comparison for the *tgrep* utility.

the ratio of eating time versus sleeping time, which was chosen to be 0.1 in both programs). On the other hand, in the original program, the linear growth seems to last only as long as the number of philosophers is small and we do not run out of cores.

In the original program, it is frequent that a philosopher holds his left stick while waiting to acquire his right stick. This is far less frequent in the instrumented program, which checks that the right stick is available before granting the left stick (if the right neighbour is fast enough, he can still get to it first but this is rather very improbable). This results in a much better degree of parallelism, which clearly shows in Fig. 8.

The four remaining programs are applications whose source code is publicly available on the internet. The first two contain only one lock (therefore they cannot possibly deadlock) while the last two have multiple locks. In all programs, independently of whether deadlocks are possible or not, we were primarily concerned with measuring and comparing the performance of the original and the instrumented versions of the programs, in order to evaluate the overhead imposed by our approach.

thrhttp: a multithreaded HTTP server (implemented in 500 lines), using a single lock for synchronizing various counters which are concurrently accessed [20]. We measured the server's performance at varying loads (number of responses over number of requests) using *httperf*, an open source tool for measuring web server performance. The results were almost identical for the original and the instrumented version of *thrhttp*.

flam3: a multithreaded program which creates "cosmic recursive fractal flames", i.e., (animations consisting of) algorithmically generated images based on fractals [5]. A single lock is used to synchronize access to a shared bucket accumulator that merges computations of distinct threads. We measured the time required to generate a long sequence of fractal images, varying the number of threads that were dedicated to the task. The results again were almost identical for the original and the instrumented version of *flam3-render*.

tgrep: a multithreaded version of the utility program `grep` which is part of the SUNWdev suite of Solaris 10 [19]. The program achieves speedup by splitting the search space across threads, using multiple locks for implementing thread-safe queues, logging and counters. In our experiment, we looked for an occurrence of a six-letter word in a directory tree containing 100,000 files, with a varying number of threads dedicated to the task. The results are shown in Fig. 9. The performance difference between the original and the instrumented program is roughly between −10% and 20%. The instrumented program consistently performs better for a few working cores (2 and 4) and worse for more working cores (≥ 16).

sshfs: a filesystem client based on the SSH File Transfer Protocol [17]. It creates threads on demand so as to serve concurrent read and write requests to the filesystem, using multiple locks to synchronize data structures, logging and access to non thread-safe functions. In our experiment, we mount a remote directory over *sshfs* and start n concurrent threads, each of which is trying to download a number of large files. The total volume of data that is copied over *sshfs* is linear w.r.t. n, and this is of course reflected in our measured results in Fig. 10. Again, we notice a small improvement in the performance of the instrumented program w.r.t. the original one, which we attribute to a slightly better degree of parallelism (as in the case of the dining philosophers).

8. Further Comparison with Related Work

In Sect. 2 we mentioned type-based approaches to preventing deadlocks. All works in this category prevent deadlocks using a type

n	original			instrumented			improvement
	U	S	e	U	S	e	
1	0.00	0.48	0.57	0.00	0.53	0.57	0.00%
2	0.04	2.02	1.46	0.01	1.96	1.44	1.37%
4	0.07	6.07	2.59	0.03	5.85	2.58	0.39%
8	0.13	18.70	4.48	0.14	17.71	4.42	1.34%
16	0.27	122.53	11.97	0.27	103.75	10.95	8.52%
32	0.45	422.57	31.52	0.50	412.08	30.75	2.44%
64	0.90	1050.64	71.83	1.05	1029.45	70.20	2.27%

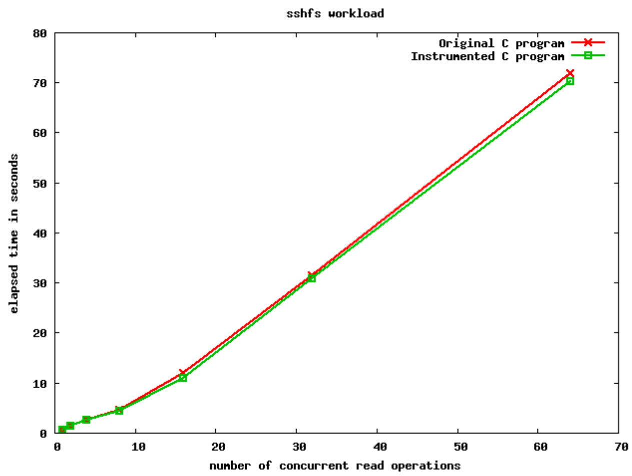

Figure 10. Performance comparison for the *sshfs*.

system which computes a partial order of all locks in the program and checks statically that all threads adhere to this order. In most such systems [6, 13, 18, 21], this partial order is statically fixed and can not be changed at runtime. A notable exception is the type system of Boyapati *et al.* [2] which allows for some form of dynamism. Namely, it allows programmers to partition the locks into a fixed number of equivalence classes (lock levels), use recursive tree-based data structures to describe their partial order, and also perform a limited set of mutations to these data structures which can change the partial order of locks *within* a given lock level at runtime. Even in this system though, to guarantee soundness, the partial order between lock levels is fixed statically. In contrast, our system does not impose any partial order on locks at compile time, but instead naturally grants locks of different threads during runtime based on the actual program needs and lock contention.

In the rest of this section, we compare our work with other techniques and tools that deal with deadlock detection and avoidance.

Purely static approaches to deadlock detection employ flow-sensitive static analysis [4] and theorem proving [7] to identify places in the code where programs do not adhere to some global lock acquisition order for all threads. In theory, such static approaches are attractive because they do not incur run-time overhead. In practice however, adhering to a strict lock acquisition order is rarely easy and seems unsuitable for systems programming. Even in simpler application domains, experience has shown that a global lock ordering is inflexible and difficult to enforce in complex, multi-layered software written by large teams of programmers. More importantly, because purely static approaches are by definition conservative, they often reject programs unnecessarily or result in a large number of false alarms.

On the other end of the spectrum, dynamic approaches to deadlock detection [12, 16] do not suffer from false positives, but they are often inflexible because when a deadlock is detected it is quite often too late to react on or recover from it. (The programs may have already performed some irrevocable operations such as I/O.)

From approaches that combine static and dynamic techniques, besides Boudol's proposal for deadlock avoidance, a tool that is quite similar to ours is Gadara [23]. Gadara employs whole program analysis to model programs and discrete control theory to synthesize a concurrent logic that avoids deadlocks at run time [24]. Like our work, Gadara targets C/pthreads programs and is claimed to avoid deadlocks quite efficiently because it performs the majority of its deadlock avoidance computations offline. (The tool is not publicly available.) Similarly to our future locksets, Gadara uses the notion of *control places* to decide whether it is safe to admit a lock acquisition. More precisely, a lock acquisition can only proceed when all the control places associated with the lock are available. The mostly static approach followed by Gadara, as well as the lack of alias analysis, results in an over-approximation of the set of run-time locks associated with a control place.

9. Concluding Remarks

Locks are here to stay as a language construct either for programmers or for compiler writers. Deadlocks are an important problem especially for languages that employ non block-structured locking.

In this paper, we have presented a novel technique that dynamically avoids deadlock states for a lock-polymorphic lambda calculus with unstructured locking primitives. The key idea is to utilize statically computed information regarding lock usage at execution time in order to avoid deadlocks. This approach accepts a wider class of programs compared to purely static approaches based on deadlock prevention. The main drawback is the additional run-time overhead induced by the future lockset computation and blocking time (i.e., both the requested lock and its future lockset must be available). Additionally, in some cases threads may unnecessarily block because our type and effect system is conservative.

We have presented a semantics for the proposed language, a sound type and effect system that guarantees that well-typed programs cannot reach a deadlocked state, and a proof sketch for the type safety theorem and related lemmata. Most importantly, we have also shown the promise of our approach by implementing a tool for C/pthreads and evaluating it on a number of programs. Our benchmark evaluation suggests that our approach imposes only a modest run-time overhead on real applications and, in some cases, it produces remarkably increased throughput.

Acknowledgement

This research is partially funded by the programme for supporting basic research (ΠΕΒΕ 2010) of the National Technical University of Athens under a project titled "Safety properties for concurrent programming languages."

References

[1] G. Boudol. A deadlock-free semantics for shared memory concurrency. In M. Leucker and C. Morgan, editors, *Proceedings of the International Colloquium on Theoretical Aspects of Computing*, volume 5684 of *LNCS*, pages 140–154. Springer, 2009.

[2] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230, New York, NY, USA, Nov. 2002. ACM Press.

[3] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[4] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Proceedings of ACM Symposium on*

Operating Systems Principles, pages 237–252, New York, NY, USA, 2003. ACM.

[5] flam3.com. Cosmic recursive fractal flames. http://flam3.com/.

[6] C. Flanagan and M. Abadi. Types for safe locking. In *Programming Language and Systems: Proceedings of the European Symposium on Programming*, number 1576 in LNCS, pages 91–108. Springer, 1999.

[7] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

[8] P. Gerakios, N. Papaspyrou, and K. Sagonas. Race-free and memory-safe multithreading: Design and implementation in Cyclone. In *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, pages 15–26, New York, NY, USA, 2010. ACM Press.

[9] P. Gerakios, N. Papaspyrou, and K. Sagonas. A type system for unstructured locking that guarantees deadlock freedom without imposing a lock ordering. In *Pre-proceedings of the Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software (PLACES)*, 2010. An extended version of this paper is available from http://www.softlab.ntua.gr/~pgerakios/deadlocks/.

[10] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 282–293, New York, NY, USA, 2002. ACM Press.

[11] M. Hicks, J. S. Foster, and P. Pratikakis. Lock inference for atomic sections. In *Proceedings of the First ACM SIGPLAN Workshop on Languages Compilers, and Hardware Support for Transactional Computing*, June 2006.

[12] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: Enabling systems to defend against deadlocks. In R. Draves and R. van Renesse, editors, *Symposium on Operating Systems Design and Implementation*, pages 295–308. USENIX Association, 2008.

[13] N. Kobayashi. A new type system for deadlock-free processes. In C. Baier and H. Hermanns, editors, *CONCUR 2006*, volume 4137 of *LNCS*, pages 233–247. Springer, 2006.

[14] L. Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Prog. Lang. Syst.*, 1(1):84–97, 1979.

[15] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction: Proceedings of the International Conference*, volume 2304 of *LNCS*, pages 213–228. Springer, 2002.

[16] F. Qin, J. Tucek, Y. Zhou, and J. Sundaresan. Rx: Treating bugs as allergies — a safe method to survive software failures. *ACM Trans. Comput. Syst.*, 25(3):7/2, 2007.

[17] SSH FileSystem. http://fuse.sourceforge.net/sshfs.html.

[18] K. Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In G. Ramalingam, editor, *Asian Symposium on Programming Languages and Systems*, volume 5356 of *LNCS*, pages 155–170. Springer, 2008.

[19] Multithreaded `grep`. Part of Sun Microsystems' *Multithreaded Programming Guide*, available at http://docs.sun.com/app/docs/doc/806-5257.

[20] Multithreaded HTTP server. http://www.xmailserver.org/thrhttp.c.

[21] V. Vasconcelos, F. Martin, and T. Cogumbreiro. Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In A. R. Beresford and S. Gay, editors, *Post-proceedings of the Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software (PLACES 2009)*, volume 17 of *EPTCS*, pages 95–109, 2010.

[22] J. W. Voung, R. Jhala, and S. Lerner. RELAY: static race detection on millions of lines of code. In *Proceedings of the joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 205–214, New York, NY, USA, 2007. ACM.

[23] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. Mahlke. Gadara: Dynamic deadlock avoidance for multithreaded programs. In R. Draves and R. van Renesse, editors, *Symposium on Operating Systems Design and Implementation*, pages 281–294. USENIX Association, 2008.

[24] Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke. The theory of deadlock avoidance via discrete control. In *Conference Record of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 252–263, New York, NY, USA, 2009. ACM.

Appendix

1.1 Language Syntax & Substitution Relation

$$
\begin{array}{llll}
\textbf{Value} & v & ::= & () \mid \texttt{true} \mid \texttt{false} \mid f \mid \texttt{lk}_\iota \\
\textbf{Expression} & e & ::= & x \mid v \mid (e\ e)^\xi \mid (e)\,[r] \mid \texttt{pop}_\gamma\ e \\
 & & \mid & \texttt{newlock}\,\rho, x\,\texttt{in}\,e \mid \texttt{lock}_\gamma\,e \mid \texttt{unlock}\,e \\
 & & \mid & \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \\
\textbf{Function} & f & ::= & \lambda x.\,e \mid \Lambda\rho.\,f \mid \texttt{fix}\,x.\,f \\
\textbf{Type} & \tau & ::= & \langle\rangle \mid \texttt{Bool} \mid \texttt{Lk}(r) \mid \tau \xrightarrow{\gamma} \tau \mid \forall\rho.\,\tau \\
\textbf{Lock} & r & ::= & \rho \mid \iota \\
\textbf{Calling mode} & \xi & ::= & \mathsf{seq}(\gamma) \mid \mathsf{par} \\
\textbf{Operation} & \kappa & ::= & + \mid - \\
\textbf{Effect} & \gamma & ::= & \emptyset \mid r^\kappa, \gamma \mid \gamma\,?\,\gamma, \gamma
\end{array}
$$

$$
\begin{array}{lcll}
x_1[v/x] & = & v & x_1 \equiv x \\
 & \mid & x_1 & \textit{otherwise} \\
e[v/x] & = & x_1[v/x] \mid f[v/x] \mid \texttt{lk}_\iota \mid \texttt{true} \mid \texttt{false} \mid () & \\
 & \mid & \texttt{pop}_\gamma\ e[v/x] \mid \texttt{lock}_\gamma\,e[v/x] & \\
 & \mid & \texttt{unlock}\,e[v/x] \mid (e_1[v/x]\ \ e_2[v/x])^\xi & \\
 & \mid & (e_1[v/x])\,[r] \mid \texttt{newlock}\,\rho, y\,\texttt{in}\,e_1[v/x] & \\
 & \mid & \texttt{if}\ e_1[v/x]\ \texttt{then}\ e_2[v/x]\ \texttt{else}\ e_3[v/x] & \\
f[v/x] & = & \lambda y.\,e[v/x] \mid \Lambda\rho.\,f[v/x] & \\
 & \mid & \texttt{fix}\,y.\,e_1[v/x] & \\
r_1[r/\rho] & = & r & r_1 \equiv \rho \\
 & \mid & r_1 & \textit{otherwise} \\
f[r/\rho] & = & \lambda x.\,e[r/\rho] \mid \Lambda\rho'.\,f[r/\rho] & \\
 & \mid & \texttt{fix}\,x.\,f[r/\rho] & \\
e[r/\rho] & = & x \mid f[r/\rho] \mid \texttt{lk}_\iota \mid \texttt{true} \mid \texttt{false} \mid () & \\
 & \mid & \texttt{pop}_{\gamma[r/\rho]}\ e[r/\rho] \mid \texttt{lock}_{\gamma_1[r/\rho]}\,e[r/\rho] & \\
 & \mid & \texttt{unlock}\,e[r/\rho] \mid (e_1[r/\rho]\ \ e_2[r/\rho])^{\xi[r/\rho]} & \\
 & \mid & (e_1[r/\rho])\,[r_1[r/\rho]] \mid \texttt{newlock}\,\rho', x\,\texttt{in}\,e_1[r/\rho] & \\
 & \mid & \texttt{if}\ e_1[r/\rho]\ \texttt{then}\ e_2[r/\rho]\ \texttt{else}\ e_3[r/\rho] & \\
\tau[r/\rho] & = & b \mid \langle\rangle \mid \tau_1[r/\rho] \xrightarrow{\gamma[r/\rho]} \tau_2[r/\rho] \mid \forall\rho'.\,\tau[r/\rho] & \\
 & \mid & \texttt{Lk}(r[r/\rho]) & \\
\xi[r/\rho] & = & \mathsf{seq}(\gamma[r/\rho]) \mid \mathsf{par} & \\
\gamma[r/\rho] & = & \emptyset \mid \gamma_1[r/\rho], \gamma_2[r/\rho]\,?\,\gamma_3[r/\rho] \mid \gamma_1[r/\rho], r_1[r/\rho]^\kappa &
\end{array}
$$

1.2 Operational Semantics: Syntax & Evaluation Context

$$
\begin{array}{llll}
\textbf{Lock Store} & S & ::= & \emptyset \mid S, \iota \mapsto n; n; \epsilon; \epsilon \\
\textbf{Threads} & T & ::= & \emptyset \mid T, n : e \\
\textbf{Configuration} & C & ::= & S\,;T \\
\textbf{Lockset} & \epsilon & ::= & \emptyset \mid \epsilon, \iota
\end{array}
$$

$$
\begin{array}{llll}
\textbf{Context} & E & ::= & \square \mid E[F] \\
\textbf{Frame} & F & ::= & (\square\ e)^\xi \mid (v\ \square)^\xi \mid (\square)\,[r] \mid \texttt{pop}_\gamma\ \square \\
 & & \mid & \texttt{lock}_{\gamma_1}\,\square \mid \texttt{unlock}\,\square \mid \texttt{if}\ \square\ \texttt{then}\ e\ \texttt{else}\ e \\
\textbf{Redex} & u & ::= & (v'\ \ v)^\xi \mid (f)\,[r] \mid \texttt{lock}_{\gamma_1}\,v \mid \texttt{unlock}\,v \\
 & & \mid & \texttt{newlock}\,\rho, x\,\texttt{in}\,e_1 \mid \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \texttt{pop}_\gamma\ v
\end{array}
$$

1.3 Operational Semantics: Helper Relations

$$
\mathsf{run}(\gamma, \iota, n) = \begin{cases}
\emptyset & \text{if } n = 0 \\
\mathsf{run}(\gamma', \iota, n+1) & \text{if } \gamma = \iota^+, \gamma' \text{ and } n > 0 \\
\mathsf{run}(\gamma', \iota, n-1) & \text{if } \gamma = \iota^-, \gamma' \text{ and } n > 0 \\
\mathsf{run}(\gamma', \iota, n) \cup \{\mathtt{j}\} & \text{if } \gamma = \mathtt{j}^+, \gamma' \text{ and } n > 0 \\
\mathsf{run}(\gamma', \iota, n) & \text{if } \gamma = \mathtt{j}^-, \gamma' \text{ and } n > 0 \\
\mathsf{run}((\gamma_1 :: \gamma'), \iota, n) \cup \mathsf{run}((\gamma_2 :: \gamma'), \iota, n) & \text{if } \gamma = \gamma_1\,?\,\gamma_2, \gamma' \text{ and } n > 0
\end{cases}
$$

$$
\mathsf{stack}(E) = \begin{cases}
\emptyset & \text{if } E = \square \\
\mathsf{stack}(E') & \text{if } E = E'[F] \text{ and } F \neq \texttt{pop}_{\gamma'}\ \square \\
\gamma' :: \mathsf{stack}(E') & \text{if } E = E'[\texttt{pop}_{\gamma'}\ \square]
\end{cases}
$$

$$
\mathsf{available}(S, n) = \begin{cases}
\emptyset & \text{if } S = \emptyset \\
\mathsf{available}(S', n) \cup \{\iota\} & \text{if } S = S', \iota \mapsto n_1; n_2; \epsilon_1; \epsilon_2 \text{ and } n_1 = n \text{ or } n_2 = 0 \\
\mathsf{available}(S', n) & \text{if } S = S', \iota \mapsto n_1, n_2; \epsilon_1; \epsilon_2 \text{ and } n_1 \neq n \text{ and } n_2 > 0
\end{cases}
$$

$$
\mathsf{dom}(S) = \{\iota \mid \iota \mapsto n_1; n_2; \epsilon_a; \epsilon_b \in S\}
$$

1.4 Operational Semantics: Reduction Relation

$$\frac{}{S;T,n:E[\texttt{if true then } e_1 \texttt{ else } e_2] \rightsquigarrow S;T,n:E[e_1]}\ \textit{(E-IT)}$$

$$\frac{}{S;T,n:E[\texttt{if false then } e_1 \texttt{ else } e_2] \rightsquigarrow S;T,n:E[e_2]}\ \textit{(E-IF)}$$

$$\frac{v' = \texttt{fix}\ x.\,f}{S;T,n:E[(v'\ \ v)^{\mathsf{seq}(\gamma)}] \rightsquigarrow S;T,n:E[(f[v'/x]\ \ v)^{\mathsf{seq}(\gamma)}]}\ \textit{(E-FX)}$$

$$\frac{\text{fresh } n'}{S;T,n:E[(v'\ \ v)^{\mathsf{par}}] \rightsquigarrow S;T,n:E[()],n':\square[(v'\ \ v)^{\mathsf{seq}(\emptyset)}]}\ \textit{(E-SN)}$$

$$\frac{}{S;T,n:\square[()] \rightsquigarrow S;T}\ \textit{(E-T)} \qquad \frac{}{S;T,n:E[((\lambda x.\,e_1)\ \ v)^{\mathsf{seq}(\gamma)}] \rightsquigarrow S;T,n:E[\mathsf{pop}_\gamma\ e_1[v/x]]}\ \textit{(E-A)}$$

$$\frac{}{S;T,n:E[(\Lambda\rho.\,f)[\iota]] \rightsquigarrow S;T,n:E[f[\iota/\rho]]}\ \textit{(E-RP)} \qquad \frac{}{S;T,n:E[\mathsf{pop}_\gamma\ v] \rightsquigarrow S;T,n:E[v]}\ \textit{(E-PP)}$$

$$\frac{\text{fresh } \iota \qquad S' = S,\iota \mapsto n;0;\emptyset;\emptyset}{S;T,n:E[\texttt{newlock}\ \rho,x\ \texttt{in}\ e_1] \rightsquigarrow S';T,n:E[e_1[\iota/\rho][\texttt{lk}_\iota/x]]}\ \textit{(E-NG)}$$

$$\frac{\begin{array}{cc} S(\iota) = n_1;0;\epsilon_1;\epsilon_2 & S' = S[\iota \mapsto n;1;\mathsf{dom}(S);\epsilon] \\ \epsilon = \mathsf{run}(\mathsf{stack}(E[\mathsf{pop}_{\gamma_1}\ \square]),\iota,1) & \epsilon \cup \{\iota\} \subseteq \mathsf{available}(S,n) \end{array}}{S;T,n:E[\texttt{lock}_{\gamma_1}\ \texttt{lk}_\iota] \rightsquigarrow S';T,n:E[()]}\ \textit{(E-LK0)}$$

$$\frac{S(\iota) = n;n_2;\epsilon_1;\epsilon_2 \qquad n_2 > 0 \qquad S' = S[\iota \mapsto n;n_2+1;\epsilon_1;\epsilon_2]}{S;T,n:E[\texttt{lock}_{\gamma_1}\ \texttt{lk}_\iota] \rightsquigarrow S';T,n:E[()]}\ \textit{(E-LK1)}$$

$$\frac{S(\iota) = n;n_2;\epsilon_1;\epsilon_2 \qquad n_2 > 0 \qquad S' = S[\iota \mapsto n;n_2-1;\epsilon_1;\epsilon_2]}{S;T,n:E[\texttt{unlock}\ \texttt{lk}_\iota] \rightsquigarrow S';T,n:E[()]}\ \textit{(E-UL)}$$

1.5 Static Semantics: Syntax and Typing Context Substitution Relation

Type variable list $\Delta \ ::= \ \emptyset \mid \Delta,\rho$

Memory List $M \ ::= \ \emptyset \mid M,\iota$

Variable list $\Gamma \ ::= \ \emptyset \mid \Gamma, x:\tau$

$\Gamma[r/\rho] \ ::= \ \emptyset \mid \Gamma_1[r/\rho], x:\tau[r/\rho]$

1.6 Static Semantics: Well Formedness Relation

Constraint Well-formedness

$$\frac{}{M;\Delta \vdash \emptyset} \qquad \frac{r \in M \cup \Delta \qquad M;\Delta \vdash \gamma_1}{M;\Delta \vdash \gamma_1, r^\kappa} \qquad \frac{M;\Delta \vdash \gamma_1 \qquad M;\Delta \vdash \gamma_2 \qquad M;\Delta \vdash \gamma_3}{M;\Delta \vdash \gamma_1,\gamma_2\,?\,\gamma_3}$$

Type Well-formedness

$$\frac{}{M;\Delta \vdash \texttt{Bool}} \qquad \frac{M;\Delta,\rho \vdash \tau}{M;\Delta \vdash \forall\rho.\,\tau} \qquad \frac{r \in M \cup \Delta}{M;\Delta \vdash \texttt{Lk}(r)} \qquad \frac{M;\Delta \vdash \tau_1 \qquad M;\Delta \vdash \gamma_1 \qquad M;\Delta \vdash \tau_2}{M;\Delta \vdash \tau_1 \xrightarrow{\gamma_1} \tau_2} \qquad \frac{}{M;\Delta \vdash \langle\rangle}$$

Γ Well-formedness

$$\frac{}{M;\Delta \vdash \emptyset} \qquad \frac{M;\Delta \vdash \tau_1 \qquad x \notin \mathsf{dom}(\Gamma_1) \qquad M;\Delta \vdash \Gamma_1}{M;\Delta \vdash \Gamma_1, x:\tau_1}$$

1.7 Static Semantics: Typing Rules

$$\frac{\begin{array}{c} x : \tau \in \Gamma \\ M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma \end{array}}{M;\Delta;\Gamma \vdash x : \tau \,\&\, (\gamma;\gamma)}\ \textit{(T-V)} \qquad \frac{M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma}{M;\Delta;\Gamma \vdash \texttt{true} : \texttt{Bool} \,\&\, (\gamma;\gamma)}\ \textit{(T-T)} \qquad \frac{M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma}{M;\Delta;\Gamma \vdash \texttt{false} : \texttt{Bool} \,\&\, (\gamma;\gamma)}\ \textit{(T-F)}$$

$$\frac{M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma}{M;\Delta;\Gamma \vdash () : \langle\rangle \,\&\, (\gamma;\gamma)}\ \textit{(T-U)} \qquad \frac{M;\Delta,\rho;\Gamma \vdash f : \tau \,\&\, (\gamma;\gamma)}{M;\Delta;\Gamma \vdash \Lambda\rho.\, f : \forall\rho.\,\tau \,\&\, (\gamma;\gamma)}\ \textit{(T-RF)} \qquad \frac{M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma \quad \iota \in M}{M;\Delta;\Gamma \vdash \texttt{lk}_\iota : \texttt{Lk}(\iota) \,\&\, (\gamma;\gamma)}\ \textit{(T-L)}$$

$$\frac{\begin{array}{c} M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma \quad \tau \equiv \tau_1 \xrightarrow{\gamma_b} \tau_2 \\ M;\Delta \vdash \tau \quad M;\Delta;\Gamma, x : \tau_1 \vdash e_1 : \tau_2 \,\&\, (\emptyset;\gamma_b) \end{array}}{M;\Delta;\Gamma \vdash \lambda x.\, e_1 : \tau \,\&\, (\gamma;\gamma)}\ \textit{(T-FN)} \qquad \frac{\begin{array}{c} M;\Delta;\Gamma, x : \tau \vdash f : \tau' \,\&\, (\gamma;\gamma) \quad \tau' \equiv \tau_1 \xrightarrow{\gamma_b} \tau_2 \\ \tau \equiv \tau_1 \xrightarrow{\gamma_a} \tau_2 \quad \gamma_a = \mathsf{summary}(\gamma_b) \end{array}}{M;\Delta;\Gamma \vdash \texttt{fix}\ x.\, f : \tau \,\&\, (\gamma;\gamma)}\ \textit{(T-FX)}$$

$$\frac{M;\Delta;\Gamma \vdash e : \texttt{Lk}(r) \,\&\, (r^+, \gamma;\gamma')}{M;\Delta;\Gamma \vdash \texttt{lock}_\gamma\, e : \langle\rangle \,\&\, (\gamma;\gamma')}\ \textit{(T-LK)} \qquad \frac{M;\Delta;\Gamma \vdash e : \texttt{Lk}(r) \,\&\, (r^-, \gamma;\gamma')}{M;\Delta;\Gamma \vdash \texttt{unlock}\, e : \langle\rangle \,\&\, (\gamma;\gamma')}\ \textit{(T-UL)}$$

$$\frac{\begin{array}{c} M;\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_a} \tau_2 \,\&\, (\gamma_1;\gamma') \\ M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_a :: \gamma;\gamma_1) \end{array}}{M;\Delta;\Gamma \vdash (e_1\ e_2)^{\mathsf{seq}(\gamma)} : \tau_2 \,\&\, (\gamma;\gamma')}\ \textit{(T-SA)}$$

$$\frac{\begin{array}{c} \forall r \in \mathsf{dom}(\gamma_a).\ r;0 \vdash_{ok} \gamma_a \\ M;\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_a} \langle\rangle \,\&\, (\gamma_1;\gamma') \quad M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma;\gamma_1) \end{array}}{M;\Delta;\Gamma \vdash (e_1\ e_2)^{\mathsf{par}} : \langle\rangle \,\&\, (\gamma;\gamma')}\ \textit{(T-PA)}$$

$$\frac{r \in M \cup \Delta \quad M;\Delta;\Gamma \vdash e_1 : \forall\rho.\,\tau \,\&\, (\gamma;\gamma')}{M;\Delta;\Gamma \vdash (e_1)\,[r] : \tau[r/\rho] \,\&\, (\gamma;\gamma')}\ \textit{(T-RP)} \qquad \frac{\begin{array}{c} M;\Delta \vdash \tau \quad \rho \notin \mathsf{dom}(\gamma) \quad \rho;0 \vdash_{ok} \gamma' \\ M;\Delta,\rho;\Gamma, x : \texttt{Lk}(\rho) \vdash e_1 : \tau \,\&\, (\gamma;\gamma') \end{array}}{M;\Delta;\Gamma \vdash \texttt{newlock}\,\rho, x\ \texttt{in}\ e_1 : \tau \,\&\, (\gamma;\gamma' \setminus \rho)}\ \textit{(T-NG)}$$

$$\frac{M;\Delta \vdash \gamma \quad M;\Delta;\Gamma \vdash e : \tau \,\&\, (\emptyset;\gamma')}{M;\Delta;\Gamma \vdash \mathsf{pop}_\gamma\ e : \tau \,\&\, (\gamma;\gamma' :: \gamma)}\ \textit{(T-PP)} \qquad \frac{\begin{array}{c} M;\Delta;\Gamma \vdash e_1 : \texttt{Bool} \,\&\, (\gamma_1 \,?\, \gamma_2, \gamma;\gamma') \\ M;\Delta;\Gamma \vdash e_2 : \tau \,\&\, (\gamma;\gamma_1 :: \gamma) \quad M;\Delta;\Gamma \vdash e_3 : \tau \,\&\, (\gamma;\gamma_2 :: \gamma) \end{array}}{M;\Delta;\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau \,\&\, (\gamma;\gamma')}\ \textit{(T-IF)}$$

where $\mathsf{summary}(\gamma_a) = \gamma_1 :: \gamma_2 :: \gamma_3$ if $\mathsf{rsummary}(\gamma_a) = \gamma_1; \gamma_2; \gamma_3$

1.8 Lock Removal

$$\frac{\gamma' = \gamma \setminus r}{\gamma' = r^\kappa, \gamma \setminus r}\ \textit{(M0)} \qquad \frac{r' \neq r \quad \gamma' = \gamma \setminus r'}{r^\kappa, \gamma' = r^\kappa, \gamma \setminus r'}\ \textit{(M1)} \qquad \frac{}{\emptyset = \emptyset \setminus r}\ \textit{(M3)}$$

$$\frac{\gamma_1' = \gamma_1 \setminus r' \quad \gamma_2' = \gamma_2 \setminus r' \quad \gamma_3' = \gamma_3 \setminus r'}{\gamma_2' \,?\, \gamma_3', \gamma_1' = \gamma_2 \,?\, \gamma_3, \gamma_1 \setminus r}\ \textit{(M4)}$$

1.9 Effect Validation

$$\frac{0 \le n \quad r;n+1 \vdash_{ok} \gamma}{r;n \vdash_{ok} r^+, \gamma}\ \textit{(OK1)} \qquad \frac{r;n-1 \vdash_{ok} \gamma \quad n > 0}{r;n \vdash_{ok} r^-, \gamma}\ \textit{(OK2)} \qquad \frac{0 \le n \quad r;n \vdash_{ok} \gamma \quad r \neq r'}{r;n \vdash_{ok} r'^\kappa, \gamma}\ \textit{(OK3)}$$

$$\frac{}{r;0 \vdash_{ok} \emptyset}\ \textit{(OK4)} \qquad \frac{0 \le n \quad r;n \vdash_{ok} \gamma_1 :: \gamma \quad r;n \vdash_{ok} \gamma_2 :: \gamma}{r;n \vdash_{ok} \gamma_1 \,?\, \gamma_2, \gamma}\ \textit{(OK5)}$$

1.10 Summary of Recursive Effect

$$\frac{\begin{array}{c} r;n_a \vdash_{ok} \gamma_a :: (r^-)^{n_b} \quad r \in \mathsf{dom}(\gamma_a) \quad \forall n_c.\, \neg\,(r;n_a - 1 \vdash_{ok} \gamma_a :: (r^-)^{n_c}) \\ \gamma_3 = \{r^+, r^- \mid r^+ \in \gamma_a\} \quad \mathsf{rsummary}(\gamma_a \setminus r) = \gamma_2; \gamma_1; \gamma_0 \end{array}}{\mathsf{rsummary}(\gamma_a) = (r^+)^{n_b} :: \gamma_2; \gamma_3 :: \gamma_1; (r^-)^{n_a} :: \gamma_0}\ \textit{(PX0)} \qquad \frac{}{\mathsf{rsummary}(\emptyset) = \emptyset; \emptyset; \emptyset}\ \textit{(PX1)}$$

1.11 Type Safety: Evaluation Context Typing

Sub-effect $\gamma \lhd \gamma' \equiv \exists \gamma''.\gamma' = \gamma'' :: \gamma$

$$\frac{M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma_1 \quad M;\Delta \vdash \gamma_2 \quad M;\Delta \vdash \tau}{M;\Delta;\Gamma \vdash \square : \tau \xrightarrow{\gamma_1;\gamma_2} \tau \,\&\, (\gamma_1;\gamma_2)}\ \textit{(E0)}$$

$$\frac{\begin{array}{c} M;\Delta;\Gamma \vdash E : \tau_2 \xrightarrow{\gamma_5;\gamma_6} \tau_3 \,\&\, (\gamma_1;\gamma_2) \\ M;\Delta;\Gamma \vdash F : \tau_1 \xrightarrow{\gamma_3;\gamma_4} \tau_2 \,\&\, (\gamma_5;\gamma_6) \end{array}}{M;\Delta;\Gamma \vdash E[F] : \tau_1 \xrightarrow{\gamma_3;\gamma_4} \tau_3 \,\&\, (\gamma_1;\gamma_2)}\ \textit{(E1)}$$

$$\frac{\begin{array}{c} M;\Delta \vdash \gamma_2 \quad \gamma_1 \lhd \gamma_2 \quad M;\Delta \vdash \tau_2 \\ M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_a :: \gamma;\gamma_1) \end{array}}{M;\Delta;\Gamma \vdash (\square\ e_2)^{\mathsf{seq}(\gamma)} : (\tau_1 \xrightarrow{\gamma_a} \tau_2) \xrightarrow{\gamma_1;\gamma_2} \tau_2 \,\&\, (\gamma;\gamma_2)}\ \textit{(F1)}$$

$$\frac{\begin{array}{c} \gamma_2 \lhd \gamma_3 \quad M;\Delta \vdash \tau_1 \xrightarrow{\gamma_a} \langle\rangle \\ M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_1;\gamma_2) \end{array}}{M;\Delta;\Gamma \vdash (\square\ e_2)^{\mathsf{par}} : (\tau_1 \xrightarrow{\gamma_a} \langle\rangle) \xrightarrow{\gamma_2;\gamma_3} \langle\rangle \,\&\, (\gamma_1;\gamma_3)}\ \textit{(F2)}$$

$$\frac{\begin{array}{c} \gamma_2 = \gamma_1 :: \gamma_a \quad M;\Delta \vdash \gamma_3 \quad \gamma_2 \lhd \gamma_3 \\ M;\Delta;\Gamma \vdash v_1 : \tau_1 \xrightarrow{\gamma_a} \tau_2 \,\&\, (\gamma_3;\gamma_3) \end{array}}{M;\Delta;\Gamma \vdash (v_1\ \square)^{\mathsf{seq}(\gamma_1)} : \tau_1 \xrightarrow{\gamma_2;\gamma_3} \tau_2 \,\&\, (\gamma_1;\gamma_3)}\ \textit{(F3)}$$

$$\frac{\begin{array}{c} M;\Delta \vdash \gamma_2 \quad \gamma_1 \lhd \gamma_2 \\ M;\Delta;\Gamma \vdash v_1 : \tau_1 \xrightarrow{\gamma_a} \langle\rangle \,\&\, (\gamma_2;\gamma_2) \end{array}}{M;\Delta;\Gamma \vdash (v_1\ \square)^{\mathsf{par}} : \tau_1 \xrightarrow{\gamma_1;\gamma_2} \langle\rangle \,\&\, (\gamma_1;\gamma_2)}\ \textit{(F4)}$$

$$\frac{M;\Delta \vdash \tau \quad M;\Delta \vdash \gamma' \quad \gamma' = \gamma_2 :: \gamma}{M;\Delta;\Gamma \vdash \mathsf{pop}_\gamma\ \square : \tau \xrightarrow{\emptyset;\gamma_2} \tau \,\&\, (\gamma;\gamma')}\ \textit{(F5)}$$

$$\frac{\begin{array}{c} M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma' \\ M;\Delta \vdash \mathtt{Lk}(r) \quad \gamma_1 = r^-, \gamma \quad \gamma_1 \lhd \gamma' \end{array}}{M;\Delta;\Gamma \vdash \mathtt{unlock}\ \square : \mathtt{Lk}(r) \xrightarrow{\gamma_1;\gamma'} \langle\rangle \,\&\, (\gamma;\gamma')}\ \textit{(F6)}$$

$$\frac{\begin{array}{c} M;\Delta \vdash \Gamma \quad M;\Delta \vdash \gamma' \\ M;\Delta \vdash \mathtt{Lk}(r) \quad \gamma_1 = r^+, \gamma \quad \gamma_1 \lhd \gamma' \end{array}}{M;\Delta;\Gamma \vdash \mathtt{lock}_\gamma\ \square : \mathtt{Lk}(r) \xrightarrow{\gamma_1;\gamma'} \langle\rangle \,\&\, (\gamma;\gamma')}\ \textit{(F7)}$$

$$\frac{\begin{array}{c} \gamma_3 = \gamma_1\, ?\, \gamma_2, \gamma \quad M;\Delta \vdash \gamma' \quad \gamma_3 \lhd \gamma' \\ M;\Delta;\Gamma \vdash e_2 : \tau \,\&\, (\gamma;\gamma_1 :: \gamma) \quad M;\Delta;\Gamma \vdash e_3 : \tau \,\&\, (\gamma;\gamma_2 :: \gamma) \end{array}}{M;\Delta;\Gamma \vdash \mathtt{if}\ \square\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \mathtt{Bool} \xrightarrow{\gamma_3;\gamma'} \tau \,\&\, (\gamma;\gamma')}\ \textit{(F8)}$$

$$\frac{M;\Delta \vdash \gamma' \quad M;\Delta \vdash \forall \rho.\tau \quad r \in M \cup \Delta \quad \gamma \lhd \gamma'}{M;\Delta;\Gamma \vdash (\square)\,[r] : \forall \rho.\tau \xrightarrow{\gamma;\gamma'} \tau[r/\rho] \,\&\, (\gamma;\gamma')}\ \textit{(F9)}$$

1.12 Type Safety: Configuration Typing

$$\mathsf{locks}(S,\iota,n) = \begin{cases} n_2 & \text{if } S(\iota) = (n; n_2; \epsilon_1; \epsilon_2) \\ 0 & \text{if } S(\iota) = (n_1; n_2; \epsilon_1; \epsilon_2) \wedge n_1 \neq n \end{cases}$$

$$\begin{array}{rl} \mathsf{deadlocked}(T) \equiv & T \supseteq T_1, n_0 : E[\mathtt{lock}_{\gamma_0}\ \mathtt{lk}_{\iota_0}], \ldots, n_k : E_k[\mathtt{lock}_{\gamma_k}\ \mathtt{lk}_{\iota_k}] \ \wedge\ k > 0\ \wedge \\ & \forall m_1 \in [0,k].\, m_2 = (m_1 + 1) \bmod (k+1)\ \wedge\ \mathsf{locks}(S, \iota_{m_2}, m_1) > 0 \end{array}$$

$$\mathsf{dom}(\gamma) = \begin{cases} \emptyset & \text{if } \gamma = \emptyset \\ \mathsf{dom}(\gamma') \cup \mathsf{dom}(\gamma_1) \cup \mathsf{dom}(\gamma_2) & \text{if } \gamma = \gamma_1\, ?\, \gamma_2, \gamma' \\ \mathsf{dom}(\gamma') \cup \{r\} & \text{if } \gamma = r^\kappa, \gamma' \end{cases}$$

$$\mathsf{dom}(T) = \{n \mid n : e \in T\}$$

Configuration Typing

$$\frac{S\,;M \vdash T \quad M = \mathsf{dom}(S)}{M \vdash S\,;T}$$

Thread Effect Consistency

$$\frac{\begin{array}{c} \iota; n_1 \vdash_{ok} \gamma \quad \epsilon_3 = \mathsf{run}(\gamma,\iota,n_1) \\ n;\gamma \vdash S \quad \epsilon_1 \cap \epsilon_3 \subseteq \epsilon_2 \end{array}}{n;\gamma \vdash S, \iota \mapsto n; n_1; \epsilon_1; \epsilon_2} \qquad \frac{\iota; 0 \vdash_{ok} \gamma \quad n \neq n_1 \quad n;\gamma \vdash S}{n;\gamma \vdash S, \iota \mapsto n_1; n_2; \epsilon_1; \epsilon_2} \qquad \frac{}{n;\gamma \vdash \emptyset}$$

Thread Typing

$$\frac{}{S\,;M \vdash \emptyset} \qquad \frac{\begin{array}{c} M;\emptyset;\emptyset \vdash e : \langle\rangle \,\&\, (\emptyset;\gamma) \quad S\,;M \vdash T \\ n \notin \mathsf{dom}(T) \quad n;\gamma \vdash S \end{array}}{S\,;M \vdash T, n : e}$$

Not Stuck

$$\frac{}{\vdash S\,;\emptyset} \qquad \frac{\vdash S\,;T \quad S;T,n:e \rightsquigarrow S';T' \quad T \subseteq T'}{\vdash S\,;T,n:e} \qquad \frac{\begin{array}{c} \vdash S\,;T \quad \mathsf{locks}(S,\iota,n) = 0 \\ \mathsf{run}(\mathsf{stack}(E[\mathsf{pop}_\gamma\ \square]), \iota; 1) = \epsilon \quad \epsilon \cup \{\iota\} \supset \mathsf{available}(S,n) \end{array}}{\vdash S\,;T, n : E[\mathtt{lock}_\gamma\ \mathtt{lk}_\iota]}$$

1.13 Type Safety: Multi-step Evaluation

$$\frac{n > 0 \qquad S;T \rightsquigarrow^{n-1} S_{n-1};T_{n-1} \qquad S_{n-1};T_{n-1} \rightsquigarrow S_n;T_n}{S;T \rightsquigarrow^{n} S_n;T_n}\ \textit{(E-M1)} \qquad \frac{}{S;T \rightsquigarrow^{0} S;T}\ \textit{(E-M2)}$$

1.14 Type Safety: Main Theorems

Safety

$S_0;T_0 \equiv \emptyset;0:e \wedge \emptyset \vdash S_0;T_0 \ \wedge\ S_0;T_0 \rightsquigarrow^n S';T' \Rightarrow \vdash S';T' \ \wedge\ \neg\mathsf{deadlocked}(T')$

Preservation

$M \vdash S;T \ \wedge\ \ S;T \rightsquigarrow S';T' \Rightarrow \exists M' \supseteq M.\ \ M' \vdash S';T'$

Progress

$M \vdash S;T \Rightarrow \vdash S;T$

Deadlock Freedom

$\emptyset;0:e \rightsquigarrow^n S_n;T_n \ \wedge\ \forall\iota \in [0,n].\exists M_\iota.\, M_\iota \vdash S_\iota;T_\iota \ \Rightarrow\ \neg\mathsf{deadlocked}(T_n)$.

1.15 Type Safety Proof

Theorem 1 (Type Safety) *If the initial configuration $S_0;T_0$ is well-typed (cf. page* **??***) with $\emptyset \vdash S_0;T_0$ and the operational semantics takes any number of steps $S_0;T_0 \rightsquigarrow^n S_n;T_n$, then the resulting configuration $S_n;T_n$ is not stuck and T_n has not reached a deadlocked state.*

Proof. The application of lemma 1 to the assumption implies that $\forall\iota \in [0,n].\exists M_\iota.M_\iota \vdash S_\iota;T_\iota$. Therefore, $S_n;T_n$ is well-typed for some M_n. The application of lemma 19 to $M_n \vdash S_n;T_n$ implies $S_n;T_n$ is not stuck. The application of lemma 2 to $\forall\iota \in [0,n].\exists M_\iota.M_\iota \vdash S_\iota;T_\iota$ and $\emptyset;0:\emptyset;e \rightsquigarrow^n S_n;T_n$ implies that T_n has not reached a deadlocked state.

Lemma 1 (Multi-step Program Preservation) *Let $S_0;T_0$ be a* closed well-typed configuration *such that $M_0 \vdash S_0;T_0$ for some M_0. If the operational semantics evaluates $S_0;T_0$ to $S_n;T_n$ in n steps, then $\forall\iota \in [0,n].\exists M_\iota.\, M_\iota \vdash S_\iota;T_\iota$*

Proof. Proof by induction on the number of steps n. When no steps are performed (i.e., $n = 0$) the proof is immediate from the assumption. When some steps are performed (i.e., $n > 0$), we have that $S_0;T_0 \rightsquigarrow^n S_n;T_n$ or $S_0;T_0 \rightsquigarrow^{n-1} S_{n-1};T_{n-1}$ and $S_{n-1};T_{n-1} \rightsquigarrow S_n;T_n$. The application of the induction hypothesis to the fact that $S_0;T_0$ is well-typed implies $\forall\iota \in [0,n-1].\exists M_\iota.M_\iota \vdash S_\iota;T_\iota$. Thus, $M_{n-1} \vdash S_{n-1};T_{n-1}$ holds. The application of lemma 3 to $M_{n-1} \vdash S_{n-1};T_{n-1}$ and $S_{n-1};T_{n-1} \rightsquigarrow S_n;T_n$. implies that $M_n \vdash S_n;T_n$. Therefore, $\forall\iota \in [0,n].\exists M_\iota.M_\iota \vdash S_\iota;T_\iota$.

Lemma 2 (Deadlock Freedom) *Let the initial configuration take n steps, where each step is well-typed for some M, then the resulting configuration has not reached a deadlocked state.*

Proof. The assumptions imply that $\emptyset;0:e \rightsquigarrow^n S_n;T_n$ and $\forall\iota \in [0,n].\exists M_\iota.M_\iota \vdash S_\iota;T_\iota$. Assume that $\mathsf{deadlocked}(T_x)$ holds for some $x \in [0,n]$ and the first deadlock occuring in the program is in T_x (i.e. $\forall\iota.\iota < x \Rightarrow \neg\mathsf{deadlocked}(T_\iota)$). Then, the following hold:

- $T_x = T, n_0 : E_0[\mathtt{lock}_{\gamma_0}\ \mathtt{lk}_{\iota_0}], \ldots n_z : E_z[\mathtt{lock}_{\gamma_z}\ \mathtt{lk}_{\iota_z}]$, where threads 0 to z are in a deadlocked state.
- $z > 0$ and $\forall m_1 \in [0,z].\ \mathsf{locks}(S, \iota_{\mathsf{succ}(m_1)}, m_1) > 0$, where $succ(n) = (n+1)\ \mathrm{mod}(z+1)$.

Let m be the thread that acquires the *first* of the $z+1$ locks that cause the deadlock, namely $\iota_{\mathsf{succ}(m)}$ (given the definition of T_x). Then thread m acquired lock ι_k, where k equals $\mathsf{succ}(m)$, before thread k acquired lock $\iota_{\mathsf{succ}(\kappa)}$. Let us assume that $\epsilon_{1y} = \mathsf{run}(\mathsf{stack}(E[\mathsf{pop}_{\gamma_y}\ \Box]), \iota_{\mathsf{succ}(k)}, 1)$ and $\epsilon_{2y} = \mathsf{dom}(S_y)$, where $y < x$ such that $\iota_{\mathsf{succ}(k)}$ is acquired for the *first* time by thread k. Then, ι_k does not belong to ϵ_{1y}, otherwise thread k would have been blocked at the lock request of $\iota_{\mathsf{succ}(k)}$ as ι_k is already owned by thread m.

According to the assumption, each step is well-typed so $S_x; T_x$ is well-typed. By inversion of the typing configuration of thread $n_k : E_k[\mathtt{lock}_{\gamma'_k}\ \mathtt{lk}_{\iota_k}]$ we obtain that $n_k; \gamma_k \vdash S_k$, where γ_k is the effect assigned to expression $E_k[\mathtt{lock}_{\gamma'_k}\ \mathtt{lk}_{\iota_k}]$ by thread typing. We have that $\iota_{\mathsf{succ}(k)}$ is locked by thread k so by inversion of $n_k; \gamma_k \vdash S_k$ we have that:

- $S_k(\iota_{\mathsf{succ}(k)}) = n_k; n_1; \epsilon_1; \epsilon_2$, where n_1 is positive.
- $\iota_{\mathsf{succ}(k)}; n_1 \vdash_{ok} \gamma_k$
- $\epsilon_3 = \mathsf{run}(\gamma_k, \iota_{\mathsf{succ}(k)}, n_1)$
- $\epsilon_1 \cap \epsilon_3 \subseteq \epsilon_2$, where $\epsilon_2 = \epsilon_{1y}$ and $\epsilon_1 = \epsilon_{2y}$ (this is immediate by the operational steps from step y to x).

Thus, it suffices to prove that $\iota_k \in \epsilon_1$ and $\iota_k \in \epsilon_3$. For all evaluation steps f and g such that f less or than equal to g, $\mathsf{dom}(S_f) \subseteq \mathsf{dom}(S_g)$ holds (trivial to show by observation of the evaluation relation). We have assumed that m is the first thread to lock ι_k at some step y' (so $\iota_k \in \mathsf{dom}(S_{y'})$) prior to y so $\iota_k \in \epsilon_1$ (so $\iota_k \in \mathsf{dom}(S_{y'}) \subseteq (\mathsf{dom}(S_y) = \epsilon_{2y} = \epsilon_1)$.

The application of lemma 16 to the typing derivation of $E_k[\mathtt{lock}_{\gamma'_k}\ \mathtt{lk}_{\iota_k}]$ implies that $\mathtt{lock}_{\gamma'_k}\ \mathtt{lk}_{\iota_k}$ is well-typed with effect $(\gamma'_k; {\iota_k}^+, \gamma'_k)$ and that E_k is well-typed. Thus, $M_k; \emptyset; \emptyset \vdash E_k : \langle\rangle \xrightarrow{\gamma'_k;{\iota_k}^+,\gamma'_k} \langle\rangle \,\&\, (\emptyset;\gamma_k)$. Lemma 9 implies that $\gamma_k = {\iota_k}^+, \gamma''_k$ for some γ''_k. Thus, $\epsilon_3 = \mathsf{run}({\iota_k}^+, \gamma''_k,\ \iota_{\mathsf{succ}(k)}, n_1) = \mathsf{run}(\gamma''_k,\ \iota_{\mathsf{succ}(k)}, n_1) \cup \{\iota_k\}$ (by the definition of function *run*). Therefore $\iota_k \in \epsilon_{1y}$, which is a contradiction.

Lemma 3 (Preservation) *Let $S; T$ be a well-typed configuration with $M \vdash S; T$. If the operational semantics takes a step $S; T \rightsquigarrow S'; T'$, then there exist an $M' \supseteq M$ such that the resulting configuration is well-typed with $M' \vdash S'; T'$.*

Proof. By case analysis on the thread evaluation relation:

Case *E-T*: rule *E-T* implies that $S; T, n : \Box[()] \rightsquigarrow S; T$. By inversion of the configuration typing assumption we have that:

- $S; M \vdash T, n : \Box[()]$: by inversion of this derivation we have that:
  - $M; \emptyset; \emptyset \vdash \Box[()] : \langle\rangle \,\&\, (\emptyset;\emptyset)$
  - $n \notin \mathsf{dom}(T)$
  - $S; M \vdash T$
  - $n; \emptyset \vdash S$
- $M = \mathsf{dom}(S)$

Given the above facts, $M \vdash S; T$ holds.

Case *E-A*: rule *E-A* implies that $S; T, n : E[(v'\ \ v)^{\mathsf{seq}(\gamma_a)}] \rightsquigarrow S; T, n : E[\mathsf{pop}_{\gamma_a}\ e_1[v/x]]$, where v' is equal to $\lambda x.\, e_1$. By inversion of the configuration typing assumption we have that:

- $M = \mathsf{dom}(S)$
- $S; M \vdash T, n : E[e]$, where e is equal to $(v'\ \ v)^{\mathsf{seq}(\gamma_a)}$: by inversion of this derivation we have that:

– $S;M \vdash T$

– $n \notin \mathsf{dom}(T)$

– $n;\gamma \vdash S$

– $M;\emptyset;\emptyset \vdash E[e] : \langle\rangle \,\&\, (\emptyset;\gamma)$: lemma 16 implies that $M;\emptyset;\emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \,\&\, (\emptyset;\gamma)$ and $M;\emptyset;\emptyset \vdash e : \tau_2' \,\&\, (\gamma_a;\gamma_b)$. By inversion of the latter derivation we have that $M;\emptyset;\emptyset \vdash v : \tau_1' \,\&\, (\gamma_b;\gamma_b)$, and $M;\emptyset;\emptyset \vdash \lambda x.\, e_1 : \tau_1' \xrightarrow{\gamma_c'} \tau_2' \,\&\, (\gamma_b;\gamma_b)$, where $\gamma_b = \gamma_c' :: \gamma_a$. By inversion of the typing derivation of v' we obtain that $M;\emptyset;\emptyset, x : \tau_1' \vdash e_1 : \tau_2' \,\&\, (\emptyset;\gamma_c')$. Lemma 11 implies that $M;\emptyset;\emptyset \vdash e_1[v/x] : \tau_2' \,\&\, (\emptyset;\gamma_c')$ holds. The application of rule *T-PP* implies that $M;\emptyset;\emptyset \vdash \mathsf{pop}_{\gamma_a}\; e_1[v/x] : \tau_2' \,\&\, (\gamma_a;\gamma_b)$ holds. The application of lemma 15 implies that $M;\emptyset;\emptyset \vdash E[\mathsf{pop}_{\gamma_a}\; e_1[v/x]] : \langle\rangle \,\&\, (\emptyset;\gamma)$.

Case *E-SN*: Rule *E-SN* implies that $S;T,n:E[(v'\;\;v)^{\mathsf{par}}] \rightsquigarrow S;T,n:E[()],n':\Box[(v'\;\;v)^{\mathsf{seq}(\emptyset)}]$

By inversion of the configuration typing assumption we have that:

- $M = \mathsf{dom}(S)$

- $S;M \vdash T,n:E[e]$: by inversion of this derivation we have that:

  – $S;M \vdash T$

  – $n;\gamma \vdash S$

  – $n \notin \mathsf{dom}(T)$

  – $M;\emptyset;\emptyset \vdash E[(v'\;\;v)^{\mathsf{par}}] : \langle\rangle \,\&\, (\emptyset;\gamma)$: lemma 16 implies that $M;\emptyset;\emptyset \vdash E : \langle\rangle \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \,\&\, (\emptyset;\gamma)$ and $M;\emptyset;\emptyset \vdash (v'\;\;v)^{\mathsf{par}} : \langle\rangle \,\&\, (\gamma_a;\gamma_a)$. By inversion of the latter derivation we have that $M;\emptyset;\emptyset \vdash v : \tau_1' \,\&\, (\gamma_a;\gamma_a)$, $M;\emptyset;\emptyset \vdash v' : \tau_1' \xrightarrow{\gamma_c'} \langle\rangle \,\&\, (\gamma_a;\gamma_a)$ and $\forall r \in \mathsf{dom}(\gamma_c').\; r;0 \vdash_{ok} \gamma_c'$. The application of lemma 7 to the typing derivation of v' implies that $M;\emptyset \vdash \gamma_c'$. Lemma 8 implies that $M;\emptyset;\emptyset \vdash v : \tau_1' \,\&\, (\gamma_c';\gamma_c')$ and $M;\emptyset;\emptyset \vdash v' : \tau_1' \xrightarrow{\gamma_c'} \langle\rangle \,\&\, (\gamma_c';\gamma_c')$. Rule *T-AP* implies $M;\emptyset;\emptyset \vdash (v'\;\;v)^{\mathsf{seq}(\emptyset)} : \langle\rangle \,\&\, (\emptyset;\gamma_c')$. Therefore, lemma 15 implies that $M;\emptyset;\emptyset \vdash \Box(v'\;\;v)^{\mathsf{seq}(\emptyset)} : \langle\rangle \,\&\, (\emptyset;\gamma_c')$. The application of lemma 6 to the typing derivation of v' implies $M;\emptyset \vdash \gamma_a$. Rule *T-U* yields that $M;\emptyset;\emptyset \vdash () : \langle\rangle \,\&\, (\gamma_a;\gamma_a)$. The application of lemma 15 implies that $M;\emptyset;\emptyset \vdash E[()] : \langle\rangle \,\&\, (\emptyset;\gamma)$.

  – $n';\gamma_c' \vdash S$: this is immediate from $\forall r \in \mathsf{dom}(\gamma_c').\; r;0 \vdash_{ok} \gamma_c'$ and the fact that $\mathsf{locks}(S,\iota,n') = 0$ for all ι.

Case *E-IT*: rule *E-IT* implies that $S;T,n:E[\texttt{if true then } e_1 \texttt{ else } e_2] \rightsquigarrow S;T,n:E[e_1]$. By inversion of the configuration typing assumption we have that:

- $M = \mathsf{dom}(S)$

- $S;M \vdash T,n:E[e]$, where e is equal to $\texttt{if true then } e_1 \texttt{ else } e_2$: by inversion of this derivation we have that:

  – $S;M \vdash T$

  – $n \notin \mathsf{dom}(T)$

  – $M;\emptyset;\emptyset \vdash E[e] : \langle\rangle \,\&\, (\emptyset;\gamma)$: lemma 16 implies that $M;\emptyset;\emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \,\&\, (\emptyset;\gamma)$ and $M;\emptyset;\emptyset \vdash e : \tau_2' \,\&\, (\gamma_a;\gamma_b)$. By inversion of the latter derivation we have that $M;\emptyset;\emptyset \vdash e_1 : \tau_2' \,\&\, (\gamma_a;\gamma_{b_1} :: \gamma_a)$, $M;\emptyset;\emptyset \vdash e_2 : \tau_2' \,\&\, (\gamma_a;\gamma_{b_2} :: \gamma_a)$ and $\gamma_b = \gamma_{b_1} \,?\, \gamma_{b_2}, \gamma_a$. Lemma 9 implies that $M;\emptyset;\emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \,\&\, (\emptyset;\gamma')$ and $M;\emptyset \vdash \gamma$, where $\gamma = \gamma_{b_1} \,?\, \gamma_{b_2}, \gamma'$. Thus, $M;\emptyset \vdash \gamma_{b_1} :: \gamma'$ holds. The application of lemma 9 to the latter fact, $M;\emptyset;\emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \,\&\, (\emptyset;\gamma')$ implies that $M;\emptyset;\emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_{b_1}::\gamma_a} \langle\rangle \,\&\, (\emptyset;\gamma_{b_1} :: \gamma')$. Lemma 15 implies that $M;\emptyset;\emptyset \vdash E[e_1] : \langle\rangle \,\&\, (\emptyset;\gamma_{b_1} :: \gamma')$.

- $n;\gamma \vdash S$: Let us assume that $S = S', \iota_1 \mapsto n_a; n_b; \epsilon_a; \epsilon_b$, for any ι_1 in $\mathsf{dom}(S)$. Given $n;\gamma \vdash \iota_1 \mapsto n_a; n_b; \epsilon_a; \epsilon_b$, where $\gamma = \gamma_{b_1} ? \gamma_{b_2}, \gamma'$, it suffices to prove $n; \gamma_{b_1} :: \gamma' \vdash \iota_1 \mapsto n_a; n_b; \epsilon_a; \epsilon_b$. If $n_a \neq n$, then it suffices to prove $\iota_1; 0 \vdash_{ok} \gamma_{b_1} :: \gamma'$, which immediate by $\iota_1; 0 \vdash_{ok} \gamma_{b_1} ? \gamma_{b_2}, \gamma'$ (by inversion of $n;\gamma \vdash \iota_1 \mapsto n_a; n_b; \epsilon_a; \epsilon_b$). If $n_a \neq n$, then it suffices to prove $\iota_1; n_a \vdash_{ok} \gamma_{b_1} :: \gamma'$ and $\mathsf{run}((\gamma_{b_1} :: \gamma'), \iota_1, n_a) \cap \epsilon_a \subseteq \epsilon_b$. The proof for the former is identical to the proof where $n_a \neq n$. By inversion of $n;\gamma \vdash \iota_1 \mapsto n_a; n_b; \epsilon_a; \epsilon_b$ we obtain that $\mathsf{run}(\gamma, \iota_1, n_a) \cap \epsilon_a \subseteq \epsilon_b$. Thus, it suffices to show that $\mathsf{run}(\gamma, \iota_1, n_a) \supseteq \mathsf{run}((\gamma_{b_1} :: \gamma'), \iota_1, n_a)$, which is immediate by the definition of *run*.

Case *E-IF*: similar to the previous case.

Case *E-FX*: Rule *E-FX* implies $S; T, n : E[(\mathtt{fix}\, x. f\ \ v)^{\mathsf{seq}(\gamma_a)}] \leadsto S; T, n : E[(f[\mathtt{fix}\, x. f/x]\ \ v)^{\mathsf{seq}(\gamma_a)}]$ holds. By inversion of the configuration typing assumption we have that:

- $M = \mathsf{dom}(S)$
- $S; M \vdash T, n : E[e]$, where e is equal to $(\mathtt{fix}\, x. f\ \ v)^{\mathsf{seq}(\gamma_a)}$: by inversion of this derivation we have that:
  - $S; M \vdash T$
  - $n \notin \mathsf{dom}(T)$
  - $M; \emptyset; \emptyset \vdash E[e] : \langle\rangle \,\&\, (\emptyset;\gamma)$: lemma 16 implies that $M; \emptyset; \emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \,\&\, (\emptyset;\gamma)$ and $M; \emptyset; \emptyset \vdash e : \tau_2' \,\&\, (\gamma_a;\gamma_b)$. By inversion of the latter derivation we have that $M; \emptyset; \emptyset \vdash v : \tau_1' \,\&\, (\gamma_b;\gamma_b)$, and $M; \emptyset; \emptyset \vdash \mathtt{fix}\, x. f : \tau_1' \xrightarrow{\gamma_c'} \tau_2' \,\&\, (\gamma_b;\gamma_b)$, where $\gamma_b = \gamma_c' :: \gamma_a$. By inversion of the typing derivation of $\mathtt{fix}\, x. f$ we obtain that $M; \emptyset; \emptyset, x : \tau_1' \xrightarrow{\gamma_c'} \tau_2' \vdash f : \tau_1' \xrightarrow{\gamma_c''} \tau_2' \,\&\, (\gamma_b;\gamma_b)$, $\mathsf{summary}(\gamma_c'') = \gamma_c' = \gamma_{x1} :: \gamma_y :: \gamma_{x2}$ such that $\mathsf{rsummary}(\gamma_c'') = \gamma_{x1}; \gamma_y; \gamma_{x2}$. Lemma 11 implies that $M; \emptyset; \emptyset \vdash f[\mathtt{fix}\, x. f/x] : \tau_1' \xrightarrow{\gamma_c''} \tau_2' \,\&\, (\gamma_b;\gamma_b)$ holds. Lemma 9 implies that $M; \emptyset; \emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \,\&\, (\emptyset;\gamma')$ and $M; \emptyset \vdash \gamma$, where $\gamma = \gamma_c' :: \gamma'$. Thus, $M; \emptyset \vdash \gamma_c'' :: \gamma'$ holds ($M; \emptyset \vdash \gamma_c''$ holds by the application of lemma 7 to the typing derivation $f[\mathtt{fix}\, x. f/x]$). The application of lemma 9 to the latter fact, $M; \emptyset; \emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \,\&\, (\emptyset;\gamma')$ implies that $M; \emptyset; \emptyset \vdash E : \tau_2' \xrightarrow{\gamma_a;\gamma_c''::\gamma_a} \langle\rangle \,\&\, (\emptyset;\gamma_c'' :: \gamma')$. Lemma 8 implies that $M; \emptyset; \emptyset \vdash f[\mathtt{fix}\, x. f/x] : \tau_1' \xrightarrow{\gamma_c''} \tau_2' \,\&\, (\gamma_c'' :: \gamma_a;\gamma_c'' :: \gamma_a)$ and $M; \emptyset; \emptyset \vdash v : \tau_1' \,\&\, (\gamma_c'' :: \gamma_a;\gamma_c'' :: \gamma_a)$. Therefore, $M; \emptyset; \emptyset \vdash (f[\mathtt{fix}\, x. f/x]\ \ v)^{\mathsf{seq}(\gamma_a)} : \tau_2' \,\&\, (\gamma_a;\gamma_c'' :: \gamma_a)$. The application of lemma 15 implies that $M; \emptyset; \emptyset \vdash E[(f[\mathtt{fix}\, x. f/x]\ \ v)^{\mathsf{seq}(\gamma_a)}] : \langle\rangle \,\&\, (\emptyset;\gamma_c'' :: \gamma')$.
  - $n;\gamma \vdash S$: $n; \gamma_c'' :: \gamma' \vdash S$ is immediate by lemma 4.

Case *E-RP* and *E-PP*: these rules are side-effect free and therefore we provide a single proof for all cases. Hence, we are assuming here that u (i.e. in $E[u]$) has one of the following forms: $(\Lambda\rho. f)\,[\iota]$ or $\mathsf{pop}_\gamma\ v$. Rules E-RP and *E-PP* imply that $S' = S$, $T' = T, n : E[v]$, where v is the value that replaces u in context E. By inversion of the configuration typing assumption we have that:

- $M = \mathsf{dom}(S)$
- $S; M \vdash T, n : E[u]$: by inversion of this derivation we have that:
  - $S; M \vdash T$
  - $n \notin \mathsf{dom}(T)$
  - $M; \emptyset; \emptyset \vdash E[u] : \langle\rangle \,\&\, (\emptyset;\gamma)$: in the case of rule *E-PP*, v is well-typed by inversion of the typing derivation of u. We need to apply lemma 8 so as to change the effect of v from $\emptyset$ to γ. In the case of rule *E-RP*, v is obtained by substituting ι in the body of function f (i.e. the initial term is $(\Lambda\rho. f)\,[\iota]$). This is immediate by lemma 12.

– $n;\gamma \vdash S$

Case *E-NG*: rule *E-NG* implies that $S\,;T,n:E[\texttt{newlock}\,\rho,x\,\texttt{in}\,e_1] \rightsquigarrow S,\iota\mapsto n;0;\emptyset;\emptyset;T,n:E[e_1[\iota/\rho][\texttt{lk}_\iota/x]]$. By inversion of the configuration typing assumption we have that:

- $M = \mathsf{dom}(S)$: $M,\iota = \mathsf{dom}(S,\iota\mapsto n;0;\emptyset;\emptyset)$ is immediate.
- $S\,;M \vdash T,n:E[\texttt{newlock}\,\rho,x\,\texttt{in}\,e_1]$: by inversion of this derivation we have that:
  - $S\,;M \vdash T$: $\quad S,\iota\mapsto n;0;\emptyset;\emptyset;M,\iota \vdash T$ trivially holds by using lemma 13 to obtain that threads of T are well-typed in the extended context M,ι; ι is *fresh* (it does not exist in the effects or stack of other threads) so the invariant $n;\gamma_{n'} \vdash S,\iota\mapsto n;0;\emptyset;\emptyset$ trivially holds from $n;\gamma_{n'} \vdash S$, where $\gamma_{n'}$ is the effect of thread n' (other than n).
  - $n \notin \mathsf{dom}(T)$
  - $M;\emptyset;\emptyset \vdash E[\texttt{newlock}\,\rho,x\,\texttt{in}\,e_1] : \langle\rangle\,\&\,(\emptyset;\gamma)$: lemma 16 implies that $M;\emptyset;\emptyset \vdash E : \tau \xrightarrow{\gamma_a;\gamma_b} \langle\rangle\,\&\,(\emptyset;\gamma)$ and $M;\emptyset;\emptyset \vdash \texttt{newlock}\,\rho,x\,\texttt{in}\,e_1 : \tau\,\&\,(\gamma_a;\gamma_b)$. By inversion of the latter derivation we obtain that $M;\Delta \vdash \tau,\rho;0 \vdash_{ok} \gamma_c$, $M;\emptyset,\rho;\emptyset,x:\mathtt{Lk}(\rho) \vdash e_1 : \tau\,\&\,(\gamma_a;\gamma_c)$ and $\gamma_b = \gamma_c\backslash\rho = (\gamma_e\backslash\rho) :: \gamma_a$ for some γ_e. Lemma 13 implies that $M,\iota;\emptyset,\rho;\emptyset,x:\mathtt{Lk}(\rho) \vdash e_1 : \tau\,\&\,(\gamma_a;\gamma_c)$ holds. Lemma 12 implies that $M,\iota;\emptyset;\emptyset,x:\mathtt{Lk}(\iota) \vdash e_1[\iota/\rho] : \tau[\iota/\rho]\,\&\,(\gamma_a[\iota/\rho];\gamma_c[\iota/\rho])$ holds. γ_a and τ do not any contain occurences of ρ so the above derivation can be further simplified to $M,\iota;\emptyset;\emptyset,x:\mathtt{Lk}(\iota) \vdash e_1[\iota/\rho] : \tau\,\&\,(\gamma_a;\gamma_c[\iota/\rho])$ holds. Lemma 11 and $M,\iota;\emptyset;\emptyset \vdash \texttt{lk}_\iota : \mathtt{Lk}(\iota)\,\&\,(\emptyset;\emptyset)$ imply that $M,\iota;\emptyset;\emptyset \vdash e_1[\iota/\rho][\texttt{lk}_\iota/x] : \tau\,\&\,(\gamma_a;\gamma_c[\iota/\rho])$ holds. Lemma 9 implies that $M;\emptyset;\emptyset \vdash E : \tau \xrightarrow{\gamma_a;\gamma_a} \langle\rangle\,\&\,(\emptyset;\gamma')$ and $M;\emptyset \vdash \gamma$, where $\gamma = (\gamma_e\setminus\rho) :: \gamma'$. The application of lemma 6 to the typing derivation of $e_1[\iota/\rho][\texttt{lk}_\iota/x]$ implies that $M;\emptyset \vdash \gamma_c[\iota/\rho]$. Lemma 14 implies that $M,\iota;\emptyset;\emptyset \vdash E : \tau \xrightarrow{\gamma_a;\gamma_a} \langle\rangle\,\&\,(\emptyset;\gamma')$. The application of lemma 9 to the latter facts imply that $M,\iota;\emptyset;\emptyset \vdash E : \tau \xrightarrow{\gamma_a;\gamma_c[\iota/\rho]} \langle\rangle\,\&\,(\emptyset;\gamma'')$, where $\gamma'' = \gamma_e[\iota/\rho] :: \gamma'$. Lemma 15 implies that $M,\iota;\emptyset;\emptyset \vdash E[e_1[\iota/\rho][\texttt{lk}_\iota/\rho]] : \langle\rangle\,\&\,(\emptyset;\gamma_e[\iota/\rho] :: \gamma')$.
  - $n;\gamma \vdash S$: we need to prove that $n;\gamma_e[\iota/\rho] :: \gamma' \vdash S,\iota\mapsto n;0;\emptyset;\emptyset$. It suffices to show that $\iota;0 \vdash_{ok} \gamma_e[\iota/\rho] :: \gamma'$, $\mathsf{run}(\gamma_e[\iota/\rho] :: \gamma',\iota,0)$ is defined, $\mathsf{run}(\gamma_e[\iota/\rho] :: \gamma',\iota,0) \cap \emptyset \subseteq \emptyset$ (immediate), and $n;\gamma_e[\iota/\rho] :: \gamma' \vdash S$. $\iota;0 \vdash_{ok} \gamma_e[\iota/\rho] :: \gamma'$ holds as a consequence of the following facts:
    - $\rho;0 \vdash_{ok} \gamma_e :: \gamma_a$ holds by the typing rule *T-NG*
    - ρ does not occur in γ_a nor γ'
    - ι does not occur anywhere
    - thus $\iota;0 \vdash_{ok} \gamma_e[\iota/\rho] :: \gamma'$ holds.

    Now, $\mathsf{run}(\gamma_e[\iota/\rho] :: \gamma',\iota,0)$ is defined as $\iota;0 \vdash_{ok} \gamma_e[\iota/\rho] :: \gamma'$ holds (by simply observing that *ok* is defined then so is *run*). Finally, $n;\gamma_e[\iota/\rho] :: \gamma' \vdash S$ holds as a consequence of the following facts:
    - for all $\mathtt{j} \neq \iota$ $\gamma_e[\iota/\rho]$ contains that same order of $+$ and $-$ operations as $\gamma_e\setminus\rho$,
    - $n;(\gamma_e\setminus\rho) :: \gamma' \vdash S$ holds (by inversion of $n;\gamma \vdash S$) and
    - for all $\mathtt{j}$ such that $S(\mathtt{j}) = (n_x;n_y;\epsilon_x;\epsilon_y)$, $\iota \notin \epsilon_x$ as ι is *fresh.*

Case *E-UL*: this rule creates side-effects as it modifies the count of lock ι. rule *E-UL* implies that $T' = T,n:E[()]$, where $()$ replaces u ($u = \texttt{unlock}\,\texttt{lk}_\iota$) in context E. The rule also implies that $S(\iota) = (n;n_2;\epsilon_1;\epsilon_2)$, $n_2 > 0$ and $S' = S[\iota\mapsto n;n_2-1;\epsilon_1;\epsilon_2]$. By inversion of the configuration typing assumption we have that:

- $M = \mathsf{dom}(S)$: ι is already contained in S so $M = \mathsf{dom}(S')$ trivially holds.
- $S\,;M \vdash T,n:E[u]$: by inversion of this derivation we have that:

– $S;M \vdash T$: we must prove that $S';M \vdash T$. It suffices to prove $n';\gamma_{n'} \vdash S'$ given that $n';\gamma_{n'} \vdash S$ holds, where $\gamma_{n'}$ is the effect of thread n'. This is immediate for all locks $\mathtt{j}$ other than ι as they remain unchanged. The invariant holds for the updated ι as *only* the reference count of lock ι is modified and therefore $\mathsf{locks}(S',\iota,n') = 0$ for all $n' \neq n$.

– $n \notin \mathsf{dom}(T)$

– $M;\emptyset;\emptyset \vdash E[u] : \langle\rangle\,\&\,(\emptyset;\gamma)$: lemma 16 implies that $M;\emptyset;\emptyset \vdash E : \langle\rangle \xrightarrow{\gamma_a;\iota^-,\gamma_a} \langle\rangle\,\&\,(\emptyset;\gamma)$ and $M;\emptyset;\emptyset \vdash u : \langle\rangle\,\&\,(\gamma_a;\iota,^-\gamma_a)$. Lemma 9 implies that $M;\emptyset;\emptyset \vdash E : \langle\rangle \xrightarrow{\gamma_a;\gamma_a} \langle\rangle\,\&\,(\emptyset;\gamma')$ and $\gamma = \iota^-,\gamma'$. The application of lemma 7 to the typing derivation of u implies that $M;\emptyset \vdash \gamma_a$. Thus, rule *T-U* implies $M;\emptyset;\emptyset \vdash () : \langle\rangle\,\&\,(\gamma_a;\gamma_a)$. $M;\emptyset;\emptyset \vdash E[()] : \langle\rangle\,\&\,(\emptyset;\gamma')$.

– $n;\iota^-,\gamma' \vdash S$: $S = S'',\iota \mapsto n;n_2;\epsilon_1;\epsilon_2$, where n_2 is positive, and $S' = S'',\iota \mapsto n;n_2-1;\epsilon_1;\epsilon_2$. The thread identifier of ι is unchanged in S' so it suffices to prove the following:

  * $\iota;n_2-1 \vdash_{ok} \gamma'$: by inversion of $n;\iota^-,\gamma' \vdash S$ we obtain $\iota;n_2 \vdash_{ok} \iota^-,\gamma'$. By inversion (rule *OK2*) of the latter fact we have that $\iota;n_2-1 \vdash_{ok} \gamma'$.
  * $\epsilon_3 = \mathsf{run}(\gamma',\iota,n_2-1)$ is defined: by inversion of $n;\iota^-,\gamma' \vdash S$ we obtain $\mathsf{run}((\iota^-,\gamma'),\iota,n_2)$. By unfolding the definition of *run* $\mathsf{run}((\iota^-,\gamma'),\iota,n_2)$ becomes $\mathsf{run}(\iota,\gamma',\,n_2-1)$.
  * $\epsilon_1 \cap \epsilon_3 \subseteq \epsilon_2$: trivially holds from the above.
  * $n;\gamma' \vdash S''$: we have that $n;\iota^-,\gamma' \vdash S''$ by inversion of $n;\iota^-,\gamma' \vdash S$. $n;\iota^-,\gamma' \vdash S''$ implies that for all $\mathtt{j}$ in $\mathsf{dom}(S'')$ such that $S''(\mathtt{j}) = (n_{1\mathtt{j}};n_{2\mathtt{j}};\epsilon_{1\mathtt{j}};\epsilon_{2\mathtt{j}})$ the following hold: $\mathtt{j};n_{2\mathtt{j}} \vdash_{ok} \iota^-,\gamma'$ and $\mathsf{run}((\iota^-,\gamma'),\mathtt{j},n_{2\mathtt{j}}) \cap \epsilon_{1\mathtt{j}} \subseteq \epsilon_{2\mathtt{j}}$. By inversion of $\mathtt{j};n_{2\mathtt{j}} \vdash_{ok} \iota^-,\gamma'$ (rule *OK3*) we have that $\mathtt{j};n_{2\mathtt{j}} \vdash_{ok} \gamma'$. If we unfold $\mathsf{run}((\iota^-,\gamma'),\mathtt{j},n_{2\mathtt{j}})$ once then we obtain $\mathsf{run}((\iota^-,\gamma'),\mathtt{j},n_{2\mathtt{j}})$ is equal to $\mathsf{run}((\gamma'),\mathtt{j},n_{2\mathtt{j}})$.

Case *E-LK1*: the proof is identical to the previous case. In the case of proving $n;\gamma' \vdash S''$ is more interesting: $\mathsf{run}((\iota^+,\gamma'),\mathtt{j},n_{2\mathtt{j}})$ is equal to $\mathsf{run}((\gamma'),\mathtt{j},n_{2\mathtt{j}}) \cup \{\iota\}$. Thus $\mathsf{run}((\gamma'),\mathtt{j},n_{2\mathtt{j}}) \subseteq \mathsf{run}((\iota^+,\gamma'),\mathtt{j},n_{2\mathtt{j}})$ and therefore, $\mathsf{run}((\gamma'),\mathtt{j},n_{2\mathtt{j}}) \cap \epsilon_{1\mathtt{j}} \subseteq \epsilon_{2\mathtt{j}}$ holds.

Case *E-LK0*: rule *E-LK0* implies that $T' = T, n : E[()]$, where $()$ replaces u ($u = \mathtt{lock}_{\gamma_a}\ \mathtt{lk}_\iota$) in context E. It also implies that $\epsilon = \mathsf{run}(\mathsf{stack}(E[\mathsf{pop}_{\gamma_a}\ \square]),\iota,1)$, $\epsilon \cup \{\iota\} \subseteq \mathsf{available}(S,n)$, $S(\iota) = (n_a;0;\epsilon_a;\epsilon_b)$ and $S = S[\iota \mapsto n;1;\mathsf{dom}(S);\epsilon]$. By inversion of the configuration typing assumption we have that:

- $M = \mathsf{dom}(S)$: in both cases ι is already contained in S so $M = \mathsf{dom}(S')$ trivially holds.
- $S;M \vdash T, n : E[u]$: by inversion of this derivation we have that:
  – $S;M \vdash T$: we must prove that $S';M \vdash T$. It suffices to prove $n';\gamma_{n'} \vdash S'$ given that $n';\gamma_{n'} \vdash S$ holds, where $\gamma_{n'}$ is the effect of thread n'. This is immediate for all locks $\mathtt{j}$ other than ι as S' differs from S in respect to lock ι. It also holds for ι as $\mathsf{locks}(S',\iota,n') = 0$ for all $n' \neq n$.
  – $n \notin \mathsf{dom}(T)$
  – $M;\emptyset;\emptyset \vdash E[u] : \langle\rangle\,\&\,(\emptyset;\gamma)$: lemma 16 implies that $M;\emptyset;\emptyset \vdash E : \langle\rangle \xrightarrow{\gamma_a;\iota^+,\gamma_a} \langle\rangle\,\&\,(\emptyset;\gamma)$ and $M;\emptyset;\emptyset \vdash u : \langle\rangle\,\&\,(\gamma_a;\iota^+,\gamma_a)$. Lemma 9 implies that $M;\emptyset;\emptyset \vdash E : \langle\rangle \xrightarrow{\gamma_a;\gamma_a} \langle\rangle\,\&\,(\emptyset;\gamma')$ and $\gamma = \iota^+,\gamma'$. The application of lemma 7 to the typing derivation of u implies that $M;\emptyset \vdash \gamma_a$. Thus, rule *T-U* implies $M;\emptyset;\emptyset \vdash () : \langle\rangle\,\&\,(\gamma_a;\gamma_a)$. $M;\emptyset;\emptyset \vdash E[()] : \langle\rangle\,\&\,(\emptyset;\gamma')$.
  – $n;\gamma \vdash S$: $S = S'',\iota \mapsto n_1;0;\epsilon_a;\epsilon_b$, and $S' = S'',\iota \mapsto n;1;\mathsf{dom}(S);\epsilon$. It suffices to prove the following:
    * $\iota;1 \vdash_{ok} \gamma'$: by inversion of $n;\iota^+,\gamma' \vdash S$ we obtain $\iota;0 \vdash_{ok} \iota^+,\gamma'$. By inversion (rule *OK1*) of the latter fact we have that $\iota;1 \vdash_{ok} \gamma'$.
    * $\epsilon_3 = \mathsf{run}(\gamma',\iota,1)$ is defined: by inversion of $n;\iota^+,\gamma' \vdash S$ we obtain $\mathsf{run}((\iota^+,\gamma'),\iota,0)$. By unfolding the definition of *run* $\mathsf{run}((\iota^+,\gamma'),\iota,0)$ becomes $\mathsf{run}(\gamma',\iota,1)$.
    * $\epsilon \cap \mathsf{dom}(S) \subseteq \epsilon$: trivially holds. The typing implies that $\mathsf{dom}(S) = M$ and ϵ is derived from γ' which is well-typed in the context of M.

- $* \; n;\gamma' \vdash S''$: we have that $n;\iota^+,\gamma' \vdash S''$ by inversion of $n;\iota^+,\gamma' \vdash S$. $n;\iota^+,\gamma' \vdash S''$ implies that for all $\mathtt{j}$ in $\mathsf{dom}(S'')$ such that $S''(\mathtt{j}) = (n_{1\mathtt{j}}; n_{2\mathtt{j}}; \epsilon_{1\mathtt{j}}; \epsilon_{2\mathtt{j}})$ the following hold: $\mathtt{j}; n_{2\mathtt{j}} \vdash_{ok} \iota^+,\gamma'$ and $\mathsf{run}((\iota^+,\gamma'),\mathtt{j},n_{2\mathtt{j}}) \cap \epsilon_{1\mathtt{j}} \subseteq \epsilon_{2\mathtt{j}}$. By inversion of $\mathtt{j}; n_{2\mathtt{j}} \vdash_{ok} \iota^+,\gamma'$ (rule *OK3*) we have that $\mathtt{j}; n_{2\mathtt{j}} \vdash_{ok} \gamma'$. If we unfold $\mathsf{run}((\iota^+,\gamma'),\mathtt{j},n_{2\mathtt{j}})$ once then we obtain $\mathsf{run}((\iota^+,\gamma'),\mathtt{j},n_{2\mathtt{j}})$ is equal to $\mathsf{run}((\gamma'),\mathtt{j},n_{2\mathtt{j}}) \cup \{\iota\}$. Thus $\mathsf{run}((\gamma'),\mathtt{j},n_{2\mathtt{j}}) \subseteq \mathsf{run}((\iota^+,\gamma'),\mathtt{j},n_{2\mathtt{j}})$ and therefore, $\mathsf{run}((\gamma'),\mathtt{j},n_{2\mathtt{j}}) \cap \epsilon_{1\mathtt{j}} \subseteq \epsilon_{2\mathtt{j}}$ holds.

Lemma 4 (Thread Lock Typing Preservation — Recursion) *If* $n;\gamma \vdash S$, $\gamma = \gamma_{x1} :: \gamma_{x2} :: \gamma_{x3} :: \gamma'$ *and* $\mathsf{rsummary}(\gamma_x) = \gamma_{x1};\gamma_{x2};\gamma_{x3}$ *then* $n;\gamma_x :: \gamma' \vdash S$.

Proof. Proof by induction. If S is empty the conclusion trivially holds. Otherwise S is of the form $S', \iota \mapsto n_1; n_2; \epsilon_a; \epsilon_b$ for some S'. There are two cases:

- $n_1 \neq n$: we need to prove that $\iota; 0 \vdash_{ok} \gamma_x :: \gamma'$ given that $\iota; 0 \vdash_{ok} \gamma_{x1} :: \gamma_{x2} :: \gamma_{x3} :: \gamma'$ holds. This is immediate by Lemma 5.
- $n_1 = n$: as in the previous case, lemma 5 suggests that $\iota; n_2 \vdash_{ok} \gamma_x :: \gamma'$ holds. The remaining proof obligation is $\mathsf{run}((\gamma_x :: \gamma'),\iota,n_2) \subseteq \mathsf{run}((\gamma_{x1} :: \gamma_{x2} :: \gamma_{x3} :: \gamma'),\iota,n_2)$. By observation of function *run* it suffices to prove that the lockset of $\gamma_{x1} :: \gamma_{x2} :: \gamma_{x3}$ is a superset of the lockset of γ_x. This is immediate by the definition of γ_{x1} that only contains r^+, r^- pairs for all all r in the domain of γ_x.

$n;\gamma \vdash S'$ holds by the induction hypothesis.

Lemma 5 (Implication of *ok*)
If

- $\mathsf{rsummary}(\gamma_x) = \gamma_{x1};\gamma_{x2};\gamma_{x3}$
- $r; n \vdash_{ok} \gamma_{x1} :: \gamma_{x2} :: \gamma_{x3} :: \gamma$

then $r; n \vdash_{ok} \gamma_x :: \gamma$.

Proof. The second assumption implies that r belongs in the domain of γ_x and thus by inversion of the first assumption we have that $r; n_a \vdash_{ok} \gamma_x :: (r^-)^{n_b}$ such that n_a and n_b are the number of unmatched *unlock* and *lock* operations respectively for r in γ_x (notice that $n_a \leq n$ by the second assumption). The definition of *rsumarry* also tells us that there exist exactly n_a and n_b unmatched *unlock* and *lock* operations for r in in $\gamma_{x1} :: \gamma_{x2} :: \gamma_{x3}$. Therefore, γ_x can safely replace $\gamma_{x1} :: \gamma_{x2} :: \gamma_{x3}$ and $r; n \vdash_{ok} \gamma_x :: \gamma$ holds.

Lemma 6 (Well-Formedness) *If an expression e is well-typed in the typing context* $M;\Delta;\Gamma$*, with effect* $\gamma;\gamma'$*, then* $M;\Delta \vdash \Gamma$*,* $M;\Delta \vdash \gamma$ *and* $M;\Delta \vdash \gamma'$ *hold.*

Proof. Straightforward proof by induction on the expression typing derivation.

Lemma 7 (Type Well-formedness) $M;\Delta;\Gamma \vdash e : \tau \,\&\, (\gamma;\gamma') \Rightarrow M;\Delta \vdash \tau$

Proof. Straightforward induction on the typing rules.

Lemma 8 (Value-Effect) *If value v is well-typed in the typing context* $M;\Delta;\Gamma$*, with effect* $(\gamma;\gamma)$ *and* $M;\Delta \vdash \gamma_1$ *then v is well-typed in the same typing context with effect* $(\gamma_1;\gamma_1)$*.*

Proof. The proof is trivial, but we provide the key steps behind the proof. By inversion of the typing derivation of v (for any v) we obtain the well-formedness derivation as well as some other premises (in the case of rules *T-L*, *T-V*, *T-F*, *T-RF*, *T-T*, *T-FN*, *T-U*, and *T-FX*). We may use the latter premises of value typing, which *still hold* (same typing context), along with $M;\Delta \vdash \gamma_1$ to formulate the new value typing derivations with effect $(\gamma_1;\gamma_1)$. The cases for rules *T-RF* and *T-FX* can be shown trivially by induction (the base case is the same as for rule *T-F*).

Lemma 9 (Evaluation Context Subtyping)

- $M;\Delta;\Gamma \vdash E : \tau \xrightarrow{\gamma_1;\gamma_2} \tau' \,\&\, (\gamma_3;\gamma_4)$
- $\gamma_2 = \gamma_{22} :: \gamma_{21}$ *and* $\gamma_1 \lhd \gamma_{21}$

if and only if

- $M;\Delta;\Gamma \vdash E : \tau \xrightarrow{\gamma_1;\gamma_{21}} \tau' \,\&\, (\gamma_3;\gamma_5)$
- $\gamma_4 = \gamma_{22} :: \gamma_5$ *and* $M;\Delta \vdash \gamma_4$

Proof. Straightforward induction on the evaluation context typing relation. The base case is trivial. The inductive hypothesis is trivial by lemma 10.

Lemma 10 (Frame Subtyping) *If the following conditions hold*

- $M;\Delta;\Gamma \vdash F : \tau \xrightarrow{\gamma_1;\gamma_2} \tau' \,\&\, (\gamma_3;\gamma_4)$
- $\gamma_2 = \gamma_{22} :: \gamma_{21}$ *and* $\gamma_1 \lhd \gamma_{21}$

if and only if

- $M;\Delta;\Gamma \vdash F : \tau \xrightarrow{\gamma_1;\gamma_{21}} \tau' \,\&\, (\gamma_3;\gamma_5)$
- $\gamma_4 = \gamma_{22} :: \gamma_5$ *and* $M;\Delta \vdash \gamma_4$

Proof. Straightforward case analysis on the frame typing relation.

Lemma 11 (Variable Substitution) $M;\Delta;\Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, (\gamma_1;\gamma_2) \wedge M;\emptyset;\emptyset \vdash v : \tau_1 \,\&\, (\gamma;\gamma) \Rightarrow M;\Delta;\Gamma \vdash e[v/x] : \tau_2 \,\&\, (\gamma_1;\gamma_2)$

Proof. Straightforward induction on the expression typing derivation.

Lemma 12 (Lock Substitution) *If* $M,\iota;\Delta,\rho;\Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$ *then* $M,\iota;\Delta;\Gamma[\iota/\rho] \vdash e[\iota/\rho] : \tau[\iota/\rho] \,\&\, (\gamma[\iota/\rho];\gamma'[\iota/\rho])$.

Proof. Proof by induction on the typing derivation of e.

Lemma 13 (Evaluation Typing Weakening) $M;\Delta;\Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$, $M;\emptyset \vdash \tau'$ *and* $\iota \notin \mathsf{dom}(M)$ *then* $M,\iota;\Delta;\Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$.

Proof. Proof by induction on the typing derivation of e.

Lemma 14 (Evaluation Context Typing Weakening) $M;\Delta;\Gamma \vdash E : \tau \xrightarrow{\gamma_1;\gamma_2} \tau' \,\&\, (\gamma;\gamma')$ *and* $\iota \notin \mathsf{dom}(M)$ *then* $M,\iota;\Delta;\Gamma \vdash E : \tau \xrightarrow{\gamma_1;\gamma_2} \tau' \,\&\, (\gamma;\gamma')$.

Proof. Proof by induction on the derivation of E.

Lemma 15 (Evaluation Context Composition — E) *If* $M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_a;\gamma_b)$ *and* $M; \Delta; \Gamma \vdash E : \tau \xrightarrow{\gamma_a;\gamma_b} \tau' \,\&\, (\gamma_1;\gamma_2)$*, then* $M; \Delta; \Gamma \vdash E[e] : \tau' \,\&\, (\gamma_1;\gamma_2)$.

Proof. Proof by induction on typing derivation of E. The base case is immediate as $\Box[e] = e$. The inductive case where $E = E'[F]$, the proof is immediate by inversion of the derivation of E and the application of lemma 17.

Lemma 16 (Evaluation Context Decomposition — E) *If* $M; \Delta; \Gamma \vdash E[e] : \tau' \,\&\, (\gamma_1;\gamma_2)$*, then there exists a* γ_a*,* γ_b *and* τ *such that* $M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_a;\gamma_b)$ *and* $M; \Delta; \Gamma \vdash E : \tau \xrightarrow{\gamma_a;\gamma_b} \tau' \,\&\, (\gamma_1;\gamma_2)$.

Proof. Proof by induction on the structure of E. The base case is immediate by using the well-formedness derivation for the type and typing context of e (i.e., lemmas 6 and 7) and the application rule *E0*. The inductive case, where $E[e] = E'[F][e]$ is immediate by lemma 18 and rule *E1*.

Lemma 17 (Frame Composition — F) *If* $M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_a;\gamma_b)$ *and* $M; \Delta; \Gamma \vdash F : \tau \xrightarrow{\gamma_a;\gamma_b} \tau' \,\&\, (\gamma_1;\gamma_2)$*, then* $M; \Delta; \Gamma \vdash F[e] : \tau' \,\&\, (\gamma_1;\gamma_2)$.

Proof. Proof by case analysis on typing derivation of F. The premises required to construct the typing derivation of $F[e]$ are given as premises of the typing derivation of F.

Lemma 18 (Frame Decomposition — F) *If* $M; \Delta; \Gamma \vdash F[e] : \tau' \,\&\, (\gamma_1;\gamma_2)$*, then there exists a* γ_a*,* γ_b *and* τ *such that* $M; \Delta; \Gamma \vdash e : \tau \,\&\, (\gamma_a;\gamma_b)$ *and* $M; \Delta; \Gamma \vdash F : \tau \xrightarrow{\gamma_a;\gamma_b} \tau' \,\&\, (\gamma_1;\gamma_2)$.

Proof. Proof by case analysis on the structure of F. The premises required for each case (i.e., rules *F1-F9*) are given by the premises of the typing derivation of $F[e]$.

Lemma 19 (Progress) *Let $S\,;T$ be a closed well-typed configuration with $M \vdash S\,;T$ then $S\,;T$ is not stuck ($\vdash S\,;T$).*

Proof. Without loss of generality, we choose a random thread from the thread list such that $T = T_1, n : e$ for some T_1 and show that it is either blocked or it can perform a step. By inversion of the configuration typing derivation we have that $S\,;M \vdash T_1, n : e$, and $M = \mathsf{dom}(S)$. By inversion of the former derivation we obtain that

- $n \notin \mathsf{dom}(T_1)$
- $n;\gamma \vdash S$
- $M;\emptyset;\emptyset \vdash e : \langle\rangle \,\&\, (\emptyset;\gamma)$: If e is a value then it can only be the unit value and a step can be performed using rule *E-T*. If e is not value then according to lemma 20 there exists a $E[u]$ such that $e = E[u]$. Lemma 16 implies that $M;\emptyset;\emptyset \vdash u : \tau \,\&\, (\gamma_a;\gamma_b)$, $M;\emptyset;\emptyset \vdash E' : \tau \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \,\&\, \,\&\, (\emptyset;\gamma)$. We proceed by a case analysis on u:

  Case $\mathsf{pop}_{\gamma_a}\ v$: rule *E-PP* can be applied to perform a single step.

  Case $(v'\ \ v)^{\mathsf{seq}(\gamma_a)}$: the typing derivation of v' implies that v' is of the form $\lambda x.\, e'$ or $\mathtt{fix}\ x.\, e'$. In the first case rule *E-A* can be applied, whereas in the second case rule *E-FX* can be applied.

  Case $(v'\ \ v)^{\mathsf{par}}$: rule *E-SN* can be applied to perform a single step.

  Case $(f)\,[r]$: the typing derivation of u implies that f is of form $\Lambda\rho.\, f'$. Rule *E-RP* can be applied to perform a single step.

  Case $\mathtt{newlock}\,\rho, x\ \mathtt{in}\ e_2$: rule *E-NG* can be applied to perform a single step.

  Case $\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$: the typing derivation of u implies that v is of type $\mathtt{Bool}$. Therefore v can be either $\mathtt{true}$ or $\mathtt{false}$. In the first case rule *E-IT* can be applied, whereas in the second case rule *E-IF* can be applied.

  Case $\mathtt{unlock}\,v$: the typing derivation of u implies that v is a lock handle (i.e., $v = \mathtt{lk}_\iota$). As in lemma 3, case *E-UL* we can use the typing derivation for thread n to derive $\gamma = \iota^-, \gamma'$, where γ is the effect assigned to the entire thread. By inversion of the store typing premise ($n;\gamma \vdash S$) of the derivation for thread n we have that $\iota; n_2 \vdash_{ok} \iota^-, \gamma'$, where n_2 is the reference count of lock ι. By inversion of the latter derivation (rule *OK2*) n_2 is positive. The latter fact and the store typing derivation also tell us that the thread identifier of ι is n. Therefore, a single step can be performed via rule *E-UL*.

  Case $\mathtt{lock}_{\gamma_a}\,v$: the typing derivation of u implies that v is a lock handle (i.e., $v = \mathtt{lk}_\iota$). If the reference count (n_2) of lock ι is positive then the proof is similar to the case of $\mathtt{unlock}\,v$ and a step can be performed via rule *E-LK1*. Otherwise, $n_2 = 0$. As in lemma 3, case *E-LK0* we can use the typing derivation for thread n to derive $\gamma = (\iota^+, \gamma_a) :: \gamma'$, where γ is the effect assigned to the entire thread. By inversion of the store typing premise ($n;\gamma \vdash S$) of the derivation for thread n we have that $\iota; 0 \vdash_{ok} (\iota^+, \gamma_a) :: \gamma'$ and that the thread identifier of ι is n. Therefore $\iota; 0 \vdash_{ok} (\iota^+, \gamma_a) :: \gamma'$ implies $\epsilon = \mathsf{run}(\mathsf{stack}(E[\mathsf{pop}_{\gamma_a}\ \Box]), \iota, 1)$ is defined (here we are using the fact that the typing derivation implies that $\gamma_a :: \gamma' = \mathsf{stack}(E[\mathsf{pop}_{\gamma_a}\ \Box])$ and also the fact than when *ok* is defined so is *run* — this can be trivially shown).

  Now, if $\epsilon \cup \{\iota\} \subseteq \mathsf{available}(S, n)$, then rule *E-LK0* can be applied. Otherwise, the thread is considered to be blocked *but not stuck* (see the third rule of judgement *stuck*).

Lemma 20 (Redex) *If $M;\Delta;\Gamma \vdash E[e] : \tau \,\&\, (\gamma_1;\gamma_2)$ and $E[e]$ is* not *a value then $M;\Delta;\Gamma \vdash E'[u] : \tau \,\&\, (\gamma_1;\gamma_2)$ such that $E'[u] = E[e]$.*

Proof. By induction on the shape of e. The key idea is to convert typing derivations of e, when e is not a redex, to typing derivations of the form $E'[e']$ and apply induction for e'.