A Type System for Unstructured Locking that Guarantees Deadlock Freedom without Imposing a Lock Ordering

Prodromos Gerakios  Nikolaos Papaspyrou  Konstantinos Sagonas
School of Electrical and Computer Engineering, National Technical University of Athens, Greece
{pgerakios,nickie,kostis}@softlab.ntua.gr

Deadlocks occur in concurrent programs as a consequence of cyclic resource acquisition between threads. In this paper we present a novel type system that guarantees deadlock freedom for a language with references, unstructured locking primitives, and locks which are implicitly associated with references. The proposed type system does not impose a strict lock acquisition order and thus increases programming language expressiveness.

1 Introduction

Lock-based synchronization may give rise to deadlocks. Two or more threads are deadlocked when each of them is waiting for a lock that is acquired by another thread. According to Coffman et al. [4], 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.

Coffman has identified three strategies that guarantee deadlock-freedom by denying at least one of the above conditions before or during program execution:

- **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 information that is computed in advance regarding thread resource allocation, determine 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.

Several type systems have been proposed that guarantee deadlock freedom, the majority of which is based on the first two strategies. In the deadlock prevention category, one finds type and effect systems that guarantee deadlock freedom by statically enforcing a global lock acquisition order that must be respected by all threads [6, 2, 9, 11, 12]. In this setting, lock handles are associated with type-level lock names via the use of singleton types. Thus, handle $lk_i$ is of type $lk(t)$. 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, as it unnecessarily rejects many correct programs. 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:

\((\text{lock } x \text{ in } \ldots \text{ lock } y \text{ in } \ldots) \mid\mid (\text{lock } y \text{ in } \ldots \text{ lock } x \text{ in } \ldots)\)

In a few words, there are two parallel threads which acquire two different 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 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. Thus, the program will be rejected, both in the system of Flanagan and Abadi which requires annotations \([5]\) and in the system of Kobayashi which employs inference \([9]\) as there is no single lock order for both threads. Similar considerations apply to the more recent works of Suenaga \([11]\) and Vasconcelos et al. \([12]\) dealing with non lexically-scoped locks.

Our work follows the third strategy (deadlock avoidance). It is based on an idea put forward recently by Boudol, who proposed a type system for deadlock avoidance that is more permissive than existing approaches \([1]\). However, his system is suitable for programs that use exclusively lexically-scoped locking primitives. In this paper we present a simple language with functions, mutable references, explicit (de-)allocation constructs and unstructured (i.e., non lexically-scoped) locking primitives. Our approach ensures deadlock freedom for the proposed language by preserving exact information about the order of events, both statically and dynamically.

In the next section, we informally describe Boudol’s idea and present an informal overview of our type and effect system. In Section 3 we formally define the syntax of our language, its operational semantics and the type and effect system. In Section 4 we reason about the soundness of our system and the paper ends with a few concluding remarks.

2 Deadlock Avoidance

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 runtime system utilizes the inserted annotations so that each lock operation can only proceed when its “future” lockset is unlocked. 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:

\((\text{lock}_{(y)} x \text{ in } \ldots \text{ lock}_{(x)} y \text{ in } \ldots) \mid\mid (\text{lock}_{(x)} y \text{ in } \ldots \text{ lock}_{(y)} x \text{ in } \ldots)\)

This program is accepted by Boudol’s type system which, in general, allows locks to be acquired in any order. At runtime, 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 \(x\) (and symmetrically for the second thread). The second lock operations need not ensure anything special, as the future locksets are empty.

The main disadvantage of Boudol’s work is that locking operations have to be lexically-scoped. Even if his language had \texttt{lock/unlock} constructs, instead of \texttt{lock...in...}, Boudol’s type system is not sufficient to guarantee deadlock freedom. The example program in Figure 1(a) will help us see why: It updates the values of three shared variables, \(x, y\) and \(z\), making sure at each step that only the strictly necessary locks are held\(^1\)

\(^1\)To simplify presentation, we assume here that there is one implicit lock per variable, which has the same name. This is
In our naïvely extended (and broken, as will be shown) version of Boudol’s type and effect system, the program in Figure 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 \( \text{lock} \) and \( \text{unlock} \) pair.) Now, function \( f \) is used by instantiating both \( x \) and \( y \) with the same variable \( a \), and instantiating \( z \) with a different variable \( b \). The result of this substitution is shown in Figure 1(b). The first thing to notice is that, if we want this program to work in this case, 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.

Even with re-entrant locks, however, it is easy to see that the program in Figure 1(b) does not type check with the present annotations. The first \( \text{lock} \) for \( a \) now matches with the last (and not the first) \( \text{unlock} \); 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 \( \text{lock} \) should contain \( b \), but it does not. (The annotation of the second \( \text{lock} \) 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 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.

The type and effect system proposed in this paper supports unstructured locking, by preserving more information at the effect level. Instead of treating effects as unordered collections of locks, our type system precisely tracks effects as an order of \( \text{lock} \) and \( \text{unlock} \) operations, without enforcing a strict lock-acquisition order. The continuation effect of a term represents the effect of the function code succeeding that term. In our approach, lock operations are annotated with a continuation effect. When a \( \text{lock} \) operation is evaluated, the future lockset is calculated by inspecting its continuation effect. The \( \text{lock} \) operation succeeds only when both the lock and the future lockset are available.

Figure 1 illustrates the same program as in Figure 1 except that locking operations are now annotated with continuation effects. For example, the annotation \([y+, \ x-, \ z+, \ z-, \ y-]\) at the first \( \text{lock} \) operation means that in the future (i.e., after this lock operation) \( y \) will be acquired, then \( x \) will be released, and so on.\(^2\) If \( x \) and \( y \) were different, the runtime system would deduce that between this \( \text{lock} \) operation on \( x \)

\[ \text{let } f = \lambda x. \lambda y. \lambda z. \begin{array}{l} \text{lock}_{[y]} x; \quad x := x + 1; \\ \text{lock}_{[z]} y; \quad y := y + x; \\ \text{unlock} x; \\ \text{lock}_{[a]} z; \quad z := z + y; \\ \text{unlock} z; \\ \text{unlock} y \\ \end{array} \begin{array}{l} \text{lock}_{[a]} a; \quad a := a + 1; \\ \text{lock}_{[a]} b; \quad b := b + a; \\ \text{unlock} a; \\ \text{unlock} b; \\ \text{unlock} a \\ \end{array} \]

(a) before substitution  
(b) after substitution

Figure 1: An example program, which is well typed before substitution (a) but not after (b).

\(^2\)In the examples of this section, a simplified version of effects is used, to make presentation easier. In the formalism of Section 3, the plus and minus signs would be encoded as differences in lock counts, e.g., \( y+ \) would be encoded by a \( y^{1,0} \) (an unlocked \( y \)) followed in time by a \( y^{1,1} \) (a locked \( y \)).

more or less consistent with our formalization in Section 3. However, on a different track, locks could be associated with larger memory regions, as in our previous work \( ^2 \)
and the corresponding unlock operation, only \( y \) is locked, so the future lockset in Boudol’s sense would be \( \{ y \} \). On the other hand, if \( x \) and \( y \) are instantiated with the same \( a \), the annotation becomes \( \{ a+, a-, b+, b-, a- \} \) and the future lockset that is calculated is now the correct \( \{ a, b \} \). In a real implementation, there are several optimizations that can be performed (e.g., pre-calculation of effects) but we do not deal with them in this paper.

There are three issues that must be faced, before we can apply this approach to a full programming language. First, we need to consider continuation effects in an interprocedural manner: it is possible that a lock operation in the body of function \( f \) matches with an unlock operation in the body of function \( g \) after the point where \( f \) was called, directly or indirectly. In this case, the future lockset for the lock operation may contain locks that are not visible in the body of \( f \). We choose to compute function effects intraprocedurally and to annotate each application term with a continuation effect, which represents the effect of the code succeeding the application term in the calling function’s body. A runtime mechanism pushes information about continuation effects on the stack and, if necessary, uses this information to correctly calculate future locksets, taking into account the continuation effects of the enclosing contexts.

Second, we need to support conditional statements. The tricky part here is that, even in a simple conditional statement such as

\[
\text{if } c \text{ then } (\text{lock } x; \ldots \text{unlock } x) \text{ else } (\text{lock } y; \ldots \text{unlock } y)
\]

the two branches have different effects: \( \{ x+, x- \} \) and \( \{ y+, y- \} \), respectively. A typical type and effect system would have to reject this program, but this would be very restrictive in our case. We resolve this issue by requiring that the overall effect of both alternatives is the same. This (very roughly) means that, after the plus and minus signs cancel each other out, we have equal numbers of plus or minus signs for each lock in both alternatives. Furthermore, we assign the combined effect of the two alternatives to the conditional statement, thus keeping track of the effect of both branches; in the example above, the combined effect is denoted by \( \{ x+, x- \} \cap \{ y+, y- \} \).

The third and most complicated issue that we need to face is support for recursive functions. Again, consider a simple recursive function of the form

\[
\text{fix } f. \lambda x. \text{if } c \text{ then } (\ldots f(y) \ldots) \text{ else } \ldots
\]

Let us call \( \gamma_f \) the effect of \( f \) and \( \gamma_b \) the computed effect for the body of \( f \). It is easy to see that \( \gamma_b \) must contain \( \gamma_f \) and, if any lock/unlock operations are present in the body of \( f \), \( \gamma_b \) will be strictly larger than \( \gamma_f \). Again, a typical type and effect system would require that \( \gamma_b = \gamma_f \) and reject this function definition. We resolve this issue by computing a summary of \( \gamma_b \) and requiring that the summary is equal to \( \gamma_f \). In computing the summary, we can make several simplifications that preserve the calculation of future locksets for operations residing outside function \( f \). For instance, we are not interested whether a lock is acquired and released many times or just once, we are not interested in the exact order in which lock/unlock pairs occur, and we can flatten branches.

\[
\begin{align*}
\text{let } & f = \lambda x. \lambda y. \lambda z. \text{lock}_{[x+, x-]} x; \quad x := x + 1; \\
& \text{lock}_{[y+, y-]} y; \quad y := y + x; \\
& \text{unlock } x; \\
& \text{lock}_{[z+, z-]} z; \quad z := z + y; \\
& \text{unlock } z; \\
& \text{unlock } y
\end{align*}
\]

(a) before substitution

\[
\begin{align*}
\text{let } & f = \lambda x. \lambda y. \lambda z. \text{lock}_{[x+, x-]} x; \quad x := x + 1; \\
& \text{lock}_{[a+, a-]} a; \quad a := a + 1; \\
& \text{unlock } a; \\
& \text{lock}_{[b+, b-]} b; \quad b := b + a; \\
& \text{unlock } b; \\
& \text{unlock } a
\end{align*}
\]

(b) after substitution

Figure 2: The program of Figure 1 with continuation effect annotations; now well typed in both cases.
Expression \( e ::= x | f | (e e^f) | (e)[r] | e := e \)  
| deref \( e \) | let \( \rho, x = \text{ref} \in e \) 
| share \( e \) | release \( e \) | lock \( e \) 
| unlock \( e \) | ( ) | pop \( e \) | loc \( e \) 
| if \( e \) then \( e \) else \( e \) | true | false 

Value \( \nu ::= f | () | \text{loc}, \text{true} | \text{false} \)

Function \( f ::= \lambda x.e \ as r \rightarrow \tau \ | \ A_{\rho}.f \ | \ \text{fix} x:r.f \)

Type \( \tau ::= \langle \rangle | \tau \rightarrow \tau | \forall \rho.\tau \)

Location \( r ::= \rho | @n | \rho @ n \)

Calling mode \( \xi ::= \text{seq}(\gamma) | \text{par} \)

Capability \( \kappa ::= n, n | \pi_n \)

Effect \( \gamma ::= \emptyset | \gamma, r^e | \gamma, \gamma \gamma \)

Figure 3: Language syntax.

3 Formalism

The syntax of our language is illustrated in Figure 3, where \( x \) and \( \rho \) range over term and “region” variables, respectively. Similarly to our previous work [7, 8], an array is thought of as a memory unit that can be shared between threads and whose contents can be atomically locked. In this paper, we make the simplistic assumption that there is a one-to-one correspondence between regions and memory cells (locations), but this is of course not necessary.

The language core comprises of variables \( (x) \), constants (the unit value, \text{true} and \text{false}), functions \((f)\), and function application. Functions can be either polymorphic \((A_{\rho}.f)\) and location application is explicit \((e[\rho])\). Monomorphic functions \((\lambda x.e)\) must be annotated with their type. The application of monomorphic functions is annotated with a calling mode \((\xi)\), which is \text{seq}(\gamma)\) for normal (sequential) application and \text{par} for parallel application. Notice that sequential application terms are annotated with \( \gamma \), the continuation effect as mentioned earlier. The semantics of parallel application is that once the application term is evaluated to a redex, then 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. The term \( \text{pop}_\gamma e \) encloses a function body \( e \) and can only appear during evaluation. The same applies to constant locations \( i @ n \), which cannot exist at the source-level. The construct \( \text{let} \rho, x = \text{ref} \in e_1 \text{ in } e_2 \) allocates a fresh cell, initializes it to \( e_1 \), and associates it with variables \( \rho \) and \( x \) within expression \( e_2 \). As in other approaches, we use \( \rho \) as the type-level representation of the new cell’s location. The reference variable \( x \) has the singleton type \( \text{ref}(\rho, \tau) \), where \( \tau \) is the type of the cell’s contents. This allows the type system to connect \( x \) and \( \rho \) and thus to statically track uses of the new cell. As will be explained later, the cell can be consumed either by deallocation or by transferring its ownership to another thread. Assignment and dereference operators are standard. The value \( \text{loc}_i \), represents a reference to a location \( i \) and is introduced during evaluation. Source programs cannot contain \( \text{loc}_i \).

At any given program point, each cell is associated with a capability \( \kappa \). Capabilities consist of two natural numbers, the capability counts: the cell reference count, which denotes whether the cell is live, and the lock count, which denotes whether the cell has been locked to provide the current thread with exclusive access to its contents. Capability counts determine the validity of operations on cells. When first allocated, a cell starts with capability \((1, 1)\), meaning that it is live and locked, which provides exclusive access to the thread which allocated it. (This is our equivalent of thread-local data.) Capabilities can be either pure \((n_1, n_2)\) or impure \((\pi_1, n_2)\). In both cases, it is implied that the current thread can decrement the cell reference count \( n_1 \) times and the lock count \( n_2 \) times. Similarly to fractional permissions [3], impure capabilities denote that a location may be aliased. Our type system requires aliasing information so as to determine whether it is safe to pass lock capabilities to new threads.

The remaining language constructs (share \( e \), release \( e \), lock \( e \) and unlock \( e \)) operate on a reference \( e \). The first two constructs increment and decrement the cell reference count of \( e \) respectively.
Similarly, the latter two constructs \textit{increment} and \textit{decrement} the lock count of \( e \). As mentioned earlier, the runtime system inspects the lock annotation \( \gamma \) to determine whether it is safe to lock \( e \).

### 3.1 Operational Semantics

We define a \textit{small-step} operational semantics for our language in Figure 4\footnote{Due to space limitations, some of the functions and judgements that are used by the operational and (later) the static semantics are not formally defined in this paper. Verbal descriptions and a full formalization are given in the Appendix.}. The evaluation relation transforms \textit{configurations}. A configuration \( C \) consists of an abstract \textit{store} \( S \) and a thread map \( T \). A store \( S \) maps constant locations \( (i) \) to values \( (v) \). A thread map \( T \) associates thread identifiers to expressions (i.e., threads) and access lists. An \textit{access list} \( \theta \) maps location identifiers to \textit{reference} and \textit{lock} counts.

A \textit{frame} \( F \) is an expression with a \textit{hole}, represented as \( \square \). The hole indicates the position where

\[ C \leadsto C' \]

Figure 4: Operational semantics.
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 \([10]\). 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 a 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. The continuation effect associated with the sequential annotation equals the resulting effect of the function being applied (i.e., \( \min(\gamma_a) \)). Notice, that \( \theta \) is divided into two lists \( \theta_1 \) and \( \theta_2 \) using the new thread’s initial effect \( \max(\gamma_a) \) as a reference for consuming the appropriate number of counts from \( \theta \). On the other hand, when evaluation of a thread reduces to a unit value, the thread is removed from the thread list (rule \( E-T \)). This is successfully only if the thread has previously released all of its resources.

The rule for sequential function application (\( E-A \)) 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. Evaluation propagates through pop expressions (rule \( E-PP \)), which are only useful for calculating future locksets in rule \( E-LK0 \). 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 \( E-IF \)).

The rules for reference allocation, assignment and dereference are straightforward. Rule \( E-NG \) appends a fresh location \( i \) (with initial value \( v \)) and the dynamic count \((1,1)\) to \( S \) and \( \theta \) respectively. Rules \( E-AS \) and \( E-D \) require that the location \((i)\) being accessed is both live and accessible and no other thread has access to \( i \). Therefore dangling memory location accesses as well as unsynchronized accesses cause the evaluation to get stuck. Furthermore, the rules \( E-SH \), \( E-RL \) and \( E-UL \) manipulate a cell’s reference or lock count. They are also straightforward, simply checking that the cell is live and (in the case of \( E-UL \)) locked. Rule \( E-RL \) makes sure that a cell is unlocked before its reference count can be decremented to zero.

The most interesting rule is \( E-LK0 \), which applies when the reference being locked \((i)\) is initially unlocked. The future lockset \((\epsilon)\) is dynamically computed, by inspecting the preceding stack frames \((E)\) as well as the lock annotation \((\gamma_1)\). The lockset \( \epsilon \) is a list of locations (and thus locks). The reference \( i \) must be live and no other thread must hold either \( i \) or any of the locations in \( \epsilon \). Upon success, the lock count of \( i \) is incremented by one. On the other hand, rule \( E-LK1 \) applies when \( i \) has already been locked by the current thread (that tries to lock it again). This immediately succeeds and the lock count is incremented by one.

### 3.2 Static Semantics

In this section we present our type and effect system and discuss the most interesting parts. Effects are used to statically track the capability of each cell. An effect \( (\gamma) \) is an ordered list of elements of the form \( r^\gamma \) and summarizes the sequence of operations (e.g., locking or sharing) on references. The syntax of types in Figure \( 3 \) (on page 5) is more or less standard: Atomic types consist of base types (the unit type, denoted by \( \langle \rangle \), and bool); reference types \( \text{ref}(\tau, r) \) are associated with a type-level cell name \( r \) and monomorphic function types carry an effect. Figure \( 5 \) contains the typing rules. The typing relation is denoted by \( M;\Delta;\Gamma \vdash e : \tau & (\gamma;\gamma') \), where \( M;\Delta;\Gamma \) is the typing context, \( e \) is an expression, \( \tau \) is the type attributed to \( e \), \( \gamma \) is the input effect, and \( \gamma' \) is the output effect. In the typing context, \( M \) is a mapping of constant locations to types, \( \Delta \) is a set of cell variables, and \( \Gamma \) is a mapping of term variables to types.
Lock operations and sequential application terms are annotated with the continuation effect. This imposes the restriction that effects must flow backwards. The input effect γ to an expression e is indeed the continuation effect; it represents the operations that follow the evaluation of e. On the other hand, the output effect γ' represents the combined operations of e and its continuation. The typing relation guarantees that the input effect is always a prefix of the output effect.

The typing rules T-U, T-TR, T-FL, T-V, T-L, T-RF and T-RP are almost standard, except for the occasional premise τ = τ' which allows the type system to ignore the identifiers used for location aliasing and, for example, treat the types t@n1 and t@n2 as equal. The typing rule T-F checks that, if the effect γ1 that is annotated in the function’s type is well formed, it is indeed the effect of the function’s body. On the other hand, the typing rule T-A for function application has a lot more work to do. It joins the input effect γ (i.e., the continuation effect) and the function’s effect γa, which contains the entire history of events occurring in the function body; this is performed by the premise ξ ⊢ γ2 = γ ⊕ γa, which performs

Figure 5: Typing rules.
all the necessary checks to ensure that all the capabilities required in the function’s effect \( \gamma_a \) are available, that pure capabilities are not aliased, and, in the case of parallel application, that no lock capabilities are split and that the resulting capability of each location is zero. Rule T-PP works as a bridge between the body of a function that is being executed and its calling environment. Rule T-FX uses the function summary to summarize the effect of the function’s body and to check that the type annotation indeed contains the right summary. The effect summary is conservatively computed as the set of locks that are acquired within the function body; the unmatched lock/unlock operations are also taken into account.

Rule T-NG for creating new cells passes the input effect \( \gamma \) to \( e_2 \), the body of \( \texttt{let} \), augmented by \( \rho^{0.0} \). This means that, upon termination of \( e_2 \), both references and locks of \( \rho \) must have been consumed. The output effect of \( e_2 \) is a \( \gamma_1 \) such that \( \rho \) has capability \((1,1)\), which implies that when \( e_2 \) starts being evaluated \( \rho \) is live and locked. The input effect of the cell initializer expression \( e_1 \) is equal to the output effect of \( e_2 \) without any occurrences of \( \rho \). Rules T-AS and T-D check that, before dereferencing or assigning to cells, a capability of at least \((1,1)\) is held. Rules T-SH, T-RL, T-LK and T-UL are the ones that modify cell capabilities. In each rule, \( \kappa \) is the capability after the operation has been executed. In the case of T-RL, if the reference count for a cell is decremented to zero, then all locks must have previously been released. The last rule in Figure 5 and probably the least intuitive, is T-IF. Suppose \( \gamma \) is the input (continuation) effect to a conditional expression. Then \( \gamma \) is passed as the input effect to both branches. We know that the outputs of both branches will have \( \gamma \) as a common prefix; if \( \gamma_2 \) and \( \gamma_3 \) are the suffixes, respectively, then \( \gamma_2 ? \gamma_3 \) is the combined suffix, which is passed as the input effect to the condition \( e_1 \).

4 Type Safety

In this section we present proof sketches for the fundamental theorems that prove type safety of our language\(^5\). The type safety formulation is based on proving progress, deadlock freedom and preservation lemmata. Informally, a program written in our language is safe when for each thread of execution either an evaluation step can be performed, or the thread is waiting to acquire a lock (blocked). In addition, there must not exist any threads that have reached a deadlocked state. As discussed in Section 3.1, a thread may become stuck when it performs an illegal operation, or when it references a location that has been deallocated, or when it accesses a location that has not been locked.

**Thread Typing.** Let \( E[e] \) be the body of a thread and let \( \theta \) be the thread’s access list. Thread typing is defined by the rule:

\[
M; \Delta; \Gamma \vdash e : \tau & (\gamma_a;\gamma_b) \\
\forall t^x \in \gamma_1, \kappa = (0,0) \quad \text{counts} \_ \text{ok}(E[\text{pop}_{\gamma_b} \square], \theta) \quad \text{lockset} \_ \text{ok}(E[\text{pop}_{\gamma_b} \square], \theta) \\
M; \Delta; \Gamma \vdash E : \tau \xrightarrow{\gamma_a;\gamma_b} \langle \rangle \& (\gamma_1;\gamma_2) \\
(EA)
\]

First of all, thread typing implies the typing of \( E[e] \).

Secondly, thread typing establishes an exact correspondence between counts of the access list \( \theta \) and counts of pop expression annotations that reside in the evaluation context \( E[\text{pop}_{\gamma_b} \square] \) (i.e., \text{counts} \_ \text{ok} (\( E[\text{pop}_{\gamma_b} \square], \theta \)). The typing derivations of \( e \) and \( E \) establish an exact correspondence between the annotations of pop expressions and static effects. Therefore, for each location \( i \) in \( \theta \), the dynamic reference and lock counts of \( i \) are identical to the static counts of \( i \) deduced by the type system.

Thirdly, thread typing enforces the invariant that the future lockset of an acquired lock at any program point is always a subset of the future lockset computed when the lock was initially acquired (i.e.,

\(^5\)The complete proofs are given in the Appendix.
lockset\_ok(E[\text{pop}_{\gamma_i}], \theta)). This invariant is essential for establishing deadlock freedom. Finally, all locations must be deallocated and released when a thread terminates ($\forall r^k \in \gamma_1, \kappa = (0,0)$).

**Process Typing.** A collection of threads $T$ is well typed if each thread in $T$ is well typed and thread identifiers are distinct:

$$
\frac{M \vdash \emptyset}{M \vdash \emptyset; \emptyset \vdash \emptyset; \theta; e : () \& (\gamma;\gamma') \quad M \vdash T \quad n \notin \text{dom}(T)}
$$

**Store Typing.** A store $S$ is well typed if there is a one-to-one correspondence between $S$ and $M$ and all stored values are closed and well typed:

$$
\frac{\text{dom}(M) = \text{dom}(S) \quad \forall (t \mapsto \tau) \in M.M; \emptyset; \emptyset \vdash S(t) : \tau \& (\emptyset;\emptyset)}{M \vdash S}
$$

**Configuration Typing.** A configuration $S;T$ is **well typed** when both $T$ and $S$ are well typed, and locks are acquired by at most one thread (i.e., mutex($T$) holds).

$$
\frac{M \vdash T \quad M \vdash S \quad \text{mutex}(T)}{M \vdash S;T}
$$

**Deadlocked State.** A set of threads $n_0, \ldots, n_k$, where $k > 0$, has reached a **deadlocked state**, when each thread $n_i$ has acquired lock $\ell_{i(i+1)\mod(k+1)}$ and is waiting for lock $\ell_{i'}$.

**Not Stuck.** A configuration $S;T$ is **not stuck** when each thread in $T$ can take one of the evaluation steps in Figure 4 or it is trying to acquire a lock which (either itself or its future lockset) is unavailable (i.e., blocked($T,n$) holds).

Given these definitions, we can now present the main results of this paper. **Progress, deadlock freedom and preservation** are formalized at the **program** level, i.e., for all concurrently executed threads.

**Lemma 1 (Deadlock Freedom)** If the initial configuration takes $n$ steps, where each step is well typed, then the resulting configuration has not reached a deadlocked state.

**Proof.** Let us assume that $z$ threads have reached a deadlocked state and let $m \in [0,z-1]$, $k = (m + 1) \mod z$ and $\sigma = (k + 1) \mod z$. According to definition of **deadlocked state**, thread $m$ acquires lock $\ell_k$ and waits for lock $\ell_m$, whereas thread $k$ acquires lock $\ell_o$ and waits for lock $\ell_k$. Assume that $m$ is the first of the $z$ threads that acquires a lock so it acquires lock $\ell_k$ before thread $k$ acquires lock $\ell_o$.

Let us assume that $S_{z};T_y$ is the configuration once $\ell_o$ is acquired by thread $k$ for the first time, $e_{1y}$ is the corresponding lockset of $\ell_o$ ($e_{1y} = \text{lockset}(\ell_o,1,E[\text{pop}_{\gamma}],\emptyset)$) and $e_{2y}$ is the set of all heap locations ($e_{2y} = \text{dom}(S_{z})$) at the time $\ell_o$ is acquired. Then, $\ell_o$ does not belong to $e_{1y}$, otherwise thread $k$ would have been blocked at the lock request of $\ell_o$ as $\ell_o$ is already owned by thread $m$.

Let us assume that when thread $k$ attempts to acquire $\ell_k$, the configuration is of the form $S_{\chi};T_x$. According to the assumption of this lemma that all configurations are well typed so $S_{\chi};T_x$ is well-typed as well. By inversion of the typing derivation of $S_{\chi};T_x$, we obtain the typing derivation of thread $n_k : \theta_k;E_k[\text{lock}_{\gamma'},1\text{loc}_{\gamma'}];\text{lock}_{\gamma'}1\text{loc}_{\gamma'}$ is well-typed with input-output effect $(\gamma';\gamma'')$, where $\kappa = \gamma'_k(\ell_k \& n')$, $\kappa \geq (1,1)$, $\gamma'' = \gamma'_k(\ell_k \& n')^{-(1,0)}$ and lockset\_ok($E_k[\text{pop}_{\gamma'}],\emptyset_k$) holds, where $\emptyset_k$ is the access list of thread $k$. lockset\_ok($E_k[\text{pop}_{\gamma'}],\emptyset_k$) implies lockset($\ell_o,n_2,E_k[\text{pop}_{\gamma'}],\emptyset$) $\cap e_1 \subseteq e_2$, where $\emptyset_k = \emptyset'_k, \ell_o \mapsto n_1n_2; e_1; e_2$ (notice that $n_2$ is positive, $e_2 = e_{1y}$ and $e_1 = e_{2y}$ — this is immediate by the operational steps from $S_{z};T_y$ to $S_{\chi};T_x$ and rule E-LK0).

We have assumed that $m$ is the first thread to lock $\ell_k$ at some step before $S_{\chi};T_y$, thus $t_k \in \text{dom}(S_{z})$ (the store can only grow — this is immediate by observing the operational semantics rules). By the definition of lockset function and the definition of $\gamma''$ we have that $t_k \in \text{lockset}(\ell_o,n_2,E_k[\text{pop}_{\gamma'}],\emptyset$). Therefore, $t_k \in \text{lockset}(\ell_o,n_2,E_k[\text{pop}_{\gamma'}],\emptyset$ $\cap \text{dom}(S_{z}) \subseteq e_{1y}$, which is a contradiction. \hfill \Box
Lemma 2 (Progress) If S; T is a well-typed configuration, then S; T is not stuck.

Proof. It suffices to show that for any thread in T, a step can be performed or block predicate holds for it. Let n be an arbitrary thread in T such that T = T_1, n; e for some T_1. By inversion of the typing derivation of S; T we have that M; ∅; 0 ⊢_T; e : ⟨⟩ & (γ; γ'), mutex(T), and M ⊢ S.

If e is a value then by inversion of M; ∅; 0 ⊢_T; e : ⟨⟩ & (γ; γ'), we obtain that γ = γ', E[e] = [0] and ∀t.θ(t) = (0, 0), as a consequence of ∀v^k ∈ γ.x = (0, 0) and countsOk(∅[pop_γ □], θ). Thus, rule E-T can be applied.

If e is not a value then it can be trivially shown (by induction on the typing derivation of e) that there exists a redex u and an evaluation context E such that e = E[u]. By inversion of the typing derivation for e we obtain that M; ∅; 0 ⊢ u : τ & (γ_a; γ_b), M; ∅; 0 ⊢ E : τ −→_1 (γ & (γ; γ')). countsOk(E[pop_γ □], θ) hold.

Then, we proceed by performing a case analysis on u (we only consider the most interesting cases):

Case (λx. e' as τ v par): it suffices to show that (θ_1, θ_2) = split(θ, max(γ_c)) is defined, where γ_c is the notation of type τ. If max(γ_c) is empty, then the proof is immediate from the base case of split function. Otherwise, we must show that for all i, the count θ(i) is greater than or equal to the sum of all (i@n)^k in max(γ_c). This can be shown by considering par ⊢ γ_b = γ_a ⊕ γ_c (i.e., the max counts in γ_c are less than or equal to the max counts in γ_b), which can be obtained by inversion of the typing derivation of (λx. e' as τ v par), and the exact correspondence between static (γ_b) and dynamic counts (i.e, countsOk(E[pop_γ □], θ)). Thus, rule E-SN can be applied to perform a single step.

Case share loc_c: countsOk(E[pop_γ_a □], θ) establishes an exact correspondence between dynamic and static counts. The typing derivation implies that γ_a(i@n_1) ≥ (2, 0), for some n_1 existentially bound in the premise of the derivation. Therefore, θ(i) ≥ (1, 0). It is possible to perform a single step using rule E-SH. The cases for release loc_c and unlock loc_c can be shown in a similar manner.

Case lock_γ_a loc_c: similarly to the case we can show that θ(i) = (n_1, n_2) and n_1 is positive. If n_2 is positive, rule E-LK1 can be applied. Otherwise, n_2 is zero. Let e be equal to locked(T_1) ∩ lockset(i, 1, E[pop_γ_a □]). If e is empty then rule E-LK0 can be applied in order to perform a single step. Otherwise, blocked(T, n) predicate holds and the configuration is not stuck.

Case deref loc_c: it can be trivially shown (as in the previous case of share that we proved θ(i) ≥ (1, 0)), that θ(i) ≥ (1, 1) and since mutex(T_1, n; θ; E[deref loc_c]) holds, then i ∉ locked(T_1) and thus rule E-D can be used to perform a step. The case of loc_c := v can be shown in a similar manner.

Lemma 3 (Preservation) Let S; T be a well-typed configuration with M ⊢ S; T. If the operational semantics takes a step S; T ⊢ S'; T', then there exists M' ⊢ M such that the resulting configuration is well-typed with M' ⊢ S'; T'.

Proof. We proceed by case analysis on the thread evaluation relation (we only consider a few cases due to space limitations):

Case E-A: Rule E-A implies S' = S, T' = T, n; θ; E[pop_y_a e_1[v/x]] and e = (λx. e_1 as τ_1 → τ_2 v) in (γ_c_q). By inversion of the configuration typing assumption we have that mutex(T, n; θ; E[e]) and M; ∅; 0 ⊢_T; e : ⟨⟩ & (γ; γ') hold. It suffices to show that mutex(T, n; θ; E[pop_y_a e_1[v/x]]) and M; ∅; 0 ⊢_T; e : ⟨⟩ & (γ; γ') hold. The former is immediate from mutex(T, n; θ; E[e]) as no new locks are acquired. Now we proceed with the latter, which can be shown by proving that M; ∅; 0 ⊢_T; e : ⟨⟩ & (γ; γ')
Theorem 1 (Type Safety) Let expression $e$ be the initial program and let the initial typing context $M_0$ and the initial program configuration $S_0; T_0$ be defined as follows: $M_0 = \emptyset$, $S_0 = \emptyset$, and $T_0 = \{0 : \emptyset; e\}$. If $S_0; T_0$ is well-typed in $M_0$ and the operational semantics takes any number of steps $S_0; T_0 \leadsto^* S_n; T_n$, 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 4 to the typing derivation of $S_0; T_0$ implies that for all steps from zero to $n$ there exists an $M_i$ such that $M_i \vdash S_i; T_i$. Therefore, Lemma 1 implies that $\neg$deadlocked($T_n$) and Lemma 2 implies $S_n; T_n$ is not stuck. □

Typing the initial configuration $S_0; T_0$ with the empty typing context $M_0$ guarantees that all functions in the program are closed and that no explicit location values ($\text{loc}_i$) are used in the source of the original program.

5 Concluding Remarks

The main contribution of this work is type-based deadlock avoidance for a language with unstructured locking primitives and the meta-theory for the proposed semantics. The type system presented in this paper guarantees that well-typed programs will not deadlock at execution time. This is possible by statically verifying that program annotations reflect the order of future lock operations and using the annotations at execution time to avoid deadlocks. The main advantage over purely static approaches to deadlock freedom is that our type system accepts a wider class of programs as it does not enforce a total order on lock acquisition. The main disadvantages of our approach is that it imposes an 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. For example, when a thread locks $x$ and executes a lengthy computation (without acquiring other locks) before releasing $x$, it would be safe to allow another thread to lock $y$ even if $x$ is in its future lockset.

We have shown that this is a non-trivial extension for existing type systems based on deadlock avoidance. There are three significant sources of complexity: (i) lock acquisition and release operations may not be properly nested, (ii) lock-unlock pairs may span multiple contexts: function calls that contain lock operations may not always increase the size of lockset, but instead limit the lockset size. In addition, future locksets must be computed in a context-sensitive manner (stack traversal in our case), and (iii) in the presence of location (lock) polymorphism and aliasing, it is very difficult for a static type system even to detect the previous two sources of complexity. To address lock aliasing without imposing restrictions statically, we defer lockset resolution until run-time.

References


Appendix

A.1 Formalism Summary: Operational Semantics

locked(T) this function takes a list of threads T and returns a set of locations é locked by threads in T.

θ +i (n1,n2) this function updates the map θ so that the reference and lock counts of θ(i) are incremented by n1 and n2 respectively.

θ(i) this function returns the reference and lock counts of θ(i).

(θ1, θ2) = split(θ, max(γa)) this judgment takes γa (the effect of a new thread) and θ and returns θ1 and θ2, such that the sum of the counts of each location in θ1 and θ2 equals the counts of the same location in θ.

lockset(i, n, E) the lockset computation function traverses the evaluation context E, and returns the union of the application frame_lockset function to each frame the form popγ □. The traversal ends when E is empty or n is zero.

A.2 Formalism Summary: Static Semantics

M;Δ ⊢ τ well-formedness judgement within a typing context M;Δ for type τ.

M;Δ ⊢ r well-formedness judgement within a typing context M;Δ for location r.

⊢ M;Δ;Γ;γ1;γ2 well-formedness judgement for typing context M;Δ;Γ and effect (γ1;γ2).

ξ + γ ensures that pure capabilities are not aliased within γ. In the case of parallel application (i.e., ξ = par), the ending capability of each location must be zero, whereas the starting capability of each location must have a zero lock count when that capability is impure.

γ(r) this function returns the most recent (i.e., rightmost) occurrence of r within effect γ.
\textbf{max}(γ') returns a subset of γ', say γ' such that no duplicate locations or branches exist, the domain of γ' equals the domain of γ and each element of γ is equal to γ(r) for any r in the domain of γ.

\textbf{min}(γ') this function takes γ' and returns a prefix γ' of γ such that no duplicate locations or branches exist and the domain of γ' equals the domain of γ.

γ \setminus r this function takes γ and r and removes all occurrences of r' from γ such that r' is identical to r modulo the tags of constant locations.

ξ ⊢ γ' = γ ⊕ γ this function takes as an input γ representing the environment effect before a function call, the function effect γ_1 and yields the environment effect γ' representing the environment effect after the function call. γ is a prefix of γ' and the suffix of γ' is an adjusted version of γ_1: the order of locations is the same as in γ_1 but the counts may be greater than the ones in γ_1 as some counts may have been abstracted withing the scope of the function. It also enforces ξ ⊢ γ.

κ ≥ κ' this predicate is true if both counts of κ are no smaller than the corresponding counts of κ'.

κ + κ', κ − κ' these functions calculate the sum and difference of two capabilities (considered here as two-dimensional vectors).

\textbf{summary}(γ) the effect summarization function, used primarily in the definition of recursive functions.

τ ≡ τ' this predicate is true when τ and τ' are structurally equivalent after removing @n annotations from locations.

M; Δ; Γ ⊢ E : τ \xrightarrow{γ a; γ b} τ' & (γ_1; γ_2) the evaluation typing context judgement that takes as an input the typing context M; Δ; Γ, the evaluation context E, the expected effect (γ_a; γ_b) and the expected type τ (for the innermost hole in E), the input effect γ_1 and returns the type τ' and the effect γ_2 that will be returned by E when it is filled with an expression of type τ and effect (γ_a; γ_b).

\section*{A.3 Formalism Summary: Type Safety}

\textbf{blocked}(T,n) this predicate is true when thread n of thread list T is in a blocked (i.e., waiting for a lock) state.

\textbf{mutex}(T) this predicate is true when each lock is held by at most one thread of T.

\textbf{counts_ok}(E, θ) this predicate takes as an evaluation context E, and an access list θ and holds when the sum of all pop expression annotations in E equal the counts of the access list θ. This judgement establishes an exact correspondence between dynamic and static counts.

\textbf{lockset_ok}(E, θ) this predicate takes as an evaluation context E, and an access list θ and holds when the future lockset (lockset function) of an acquired lock at any program point is always a subset of the future lockset computed when the lock was initially acquired.
A.4 Language Syntax & Substitution Relation

Value \( v ::= \) \( f() \mid \text{loc} \mid \text{true} \mid \text{false} \)
\( x'[/x] = v \quad x' = x \) otherwise

Expression \( e ::= x \mid f \mid (e)e \mid (e)[r] \mid e := e \)
\( \text{deref} e \mid \text{let} \rho.x = \text{ref} e \in e \)
\( \text{share} e \mid \text{release} e \mid \text{lock} e \)
\( \text{unlock} e \mid () \mid \text{pop} e \mid \text{loc} \)
\( \text{if} e \) then \( e \) else \( e \) \true \false

Function \( f ::= \lambda x.e \) as \( \tau \rightarrow \gamma \) \( \Lambda pf \mid \text{fix} x : \tau. f \)

Type \( \tau ::= \) \( \emptyset \mid \tau \rightarrow \tau \mid \forall \rho. \tau \)
\( \text{ref}(\tau, r) \mid \text{bool} \)

Location \( r ::= \rho \mid \iota[n] \mid \rho @ n \)
\( \tau'[r/p] = r \quad r' = \rho \)

Calling mode \( \xi ::= \) \( \text{seq}(\gamma) \mid \text{par} \)
\( \tau\rho \mid \iota[n] \) \( r' = \rho @ n \)

Capability \( \kappa ::= \) \( n.n \mid \overline{n.n} \)
\( \tau \rightarrow \gamma \)

Effect \( \gamma ::= \) \( \emptyset \mid \gamma, r^k \mid \gamma, y?\gamma \)
\( f[r/p] = \) \( \lambda x.e[r/p] \) as \( \tau_1[r/p] \rightarrow \tau_2[r/p] \) \( \Lambda pf[r/p] \)
\( \text{fix} x : \tau[r/p]. f[r/p] \)
\( e[r/p] = \) \( x \mid \text{loc} \mid \text{true} \mid \text{false} \mid () \mid \text{pop}_{\gamma[r/p]} e[r/p] \)
\( \text{share} e[r/p] \mid \text{release} e[r/p] \mid \text{lock}_{\gamma[r/p]} e[r/p] \)
\( \text{unlock} e[r/p] \mid (e[r/p])'[r/p] \mid f[r/p] \)
\( \text{deref} e[r/p] \mid e'[r/p] \mid (e[r/p])'[r/p] \mid f[r/p] \)
\( \text{let} \rho'. x = \text{ref} e[r/p] \) in \( e'[r/p] \)
\( \text{if} e[r/p] \) then \( e'[r/p] \) else \( e''[r/p] \)

\( \tau[r/p] = \) \( \emptyset \mid \tau_1[r/p] \rightarrow \tau_2[r/p] \) \( \text{bool} \)
\( \forall \rho'. \tau[r/p] \mid \text{ref}(\tau[r/p], r'[r/p]) \)
\( \xi[r/p] = \) \( \text{seq}(\gamma[r/p]) \) \( \text{par} \)
\( \gamma[r/p] = \) \( \emptyset \mid \gamma'[r/p], r'[r/p]^k \)
\( \gamma'[r/p], \gamma''[r/p], \gamma'''[r/p] \)

A.5 Operational Semantics: Syntax & Evaluation Context

Access Lists \( \theta ::= \emptyset \mid \theta, t \mapsto n; n; e; e \)

Stack \( E ::= \emptyset \mid E[F] \)

Store \( S ::= \emptyset \mid S, t \mapsto v \)

Frame \( F ::= (\emptyset e)^k \mid (v \emptyset)^k \mid (\emptyset)[r] \mid \text{let} \rho, x = \text{ref} \emptyset \) in \( e \)
\( \text{deref} \emptyset \mid \emptyset := e \mid v := \emptyset \mid \text{share} \emptyset \mid \text{release} \emptyset \)

Threads \( T ::= \emptyset \mid T, t : \theta, e \)

Configuration \( C ::= S ; T \)

Locations \( e ::= \emptyset \mid e, t \)

Redex \( u ::= (\lambda x.e \) as \( \tau v)^k \mid \text{lock}_{\gamma_1} \text{loc} \mid \text{unlock} \text{loc} \mid \text{share} \text{loc} \)
\( \text{release} \text{loc} \mid \text{deref} \text{loc} \mid \text{loc} = v \mid \text{let} \rho, x = \text{ref} \emptyset \) in \( e_2 \)
\( \text{pop} v \mid \text{if} v \) then \( e_1 \) else \( e_2 \) \( (\Lambda pf)[r] \) \( \text{fix} x : \tau. f \) \( v^{\text{seq}(\gamma_2)} \)

A.6 Generic Predicates and Functions

\( \text{set}(\gamma) = \forall x, y_1, y_2. \gamma = (y_1, \alpha) :: y_2 \implies \alpha = r^k \land r \notin \text{dom}(\gamma_1) \cup \text{dom}(\gamma_2) \)
### A.7 Operational Semantics: Helper Rules & Predicates

\[ \begin{align*}
\text{locked}(T) & = \{ i \mid (n; t; e) \in T \land \theta(i) \geq (1, 1) \} \\
\text{lk}(k) & = n_2 \quad \text{if } k = (n_1, n_2) \\
\theta + (n_1, n_2) & = \theta[i \mapsto n_1 + n_2; n_2 + n_4; e_1; e_2] \quad \text{if } (i \mapsto n_3; n_4; e_1; e_2) \in \theta \\
\theta(i) & = (n_1, n_2) \quad \text{if } (i \mapsto n_1; n_2; e_1; e_2) \in \theta
\end{align*} \]

\[ \begin{align*}
\kappa & = (n_1, n_2) \\
(\theta_a, \theta_b) & = \text{split}(\theta, \gamma) \\
(\theta_a, \theta_d) & = \text{split}(\theta, \gamma, (\epsilon @ n_0)^\gamma) \\
(\theta_c, \theta_d) & = \text{split}(\theta, \gamma, (\epsilon @ n_0)^\gamma) \\
(\theta, \emptyset) & = \text{split}(\theta, \emptyset) \\
\emptyset & = \text{split}(\theta, \emptyset)
\end{align*} \]

\[ \begin{align*}
\text{(0, n)} & = \text{frame_lockset}(i, n, \emptyset) \\
(0, 0) & = \text{frame_lockset}(i, 0, \emptyset) \\
(\epsilon, n_2) & = \text{frame_lockset}(i, n_1, \gamma_1 :: \gamma_2) \\
(\epsilon, n_2) & = \text{frame_lockset}(i, n_1, \gamma_1 :: \gamma_2) \\
(\epsilon @ n_0, n_1) & = \text{frame_lockset}(i, n_1, \gamma_1 :: \gamma_2) \\
(\epsilon @ n_0, n_1) & = \text{frame_lockset}(i, n_1, \gamma_1 :: \gamma_2) \\
(\epsilon @ n_0, n_1) & = \text{frame_lockset}(i, n_1, \gamma_1 :: \gamma_2) \\
(\epsilon @ n_0, n_1) & = \text{frame_lockset}(i, n_1, \gamma_1 :: \gamma_2)
\end{align*} \]

\[ \begin{align*}
\text{(0, n)} & = \text{frame_lockset}(i, n, \emptyset) \\
\text{lk}(k) & = n_2 \quad \text{if } k = (n_1, n_2) \\
\theta + (n_1, n_2) & = \theta[i \mapsto n_1 + n_2; n_2 + n_4; e_1; e_2] \quad \text{if } (i \mapsto n_3; n_4; e_1; e_2) \in \theta \\
\theta(i) & = (n_1, n_2) \quad \text{if } (i \mapsto n_1; n_2; e_1; e_2) \in \theta
\end{align*} \]

\[ \begin{align*}
\kappa & = (n_1, n_2) \\
(\theta_a, \theta_b) & = \text{split}(\theta, \gamma) \\
(\theta_a, \theta_d) & = \text{split}(\theta, \gamma, (\epsilon @ n_0)^\gamma) \\
(\theta_c, \theta_d) & = \text{split}(\theta, \gamma, (\epsilon @ n_0)^\gamma) \\
(\theta, \emptyset) & = \text{split}(\theta, \emptyset) \\
\emptyset & = \text{split}(\theta, \emptyset)
\end{align*} \]

\[ \begin{align*}
\text{(0, n)} & = \text{frame_lockset}(i, n, \emptyset) \\
(0, 0) & = \text{frame_lockset}(i, 0, \emptyset) \\
(\epsilon, n_2) & = \text{frame_lockset}(i, n_1, \gamma_1 :: \gamma_2) \\
(\epsilon, n_2) & = \text{frame_lockset}(i, n_1, \gamma_1 :: \gamma_2) \\
(\epsilon @ n_0, n_1) & = \text{frame_lockset}(i, n_1, \gamma_1 :: \gamma_2) \\
(\epsilon @ n_0, n_1) & = \text{frame_lockset}(i, n_1, \gamma_1 :: \gamma_2) \\
(\epsilon @ n_0, n_1) & = \text{frame_lockset}(i, n_1, \gamma_1 :: \gamma_2)
\end{align*} \]
A.8 Operational Semantics: Reduction Relation

\[
\begin{align*}
\frac{\nu' \equiv \lambda x.e_1 \text{ as } \tau_1 \rightarrow \nu_2}{\text{fresh } n' \quad (\theta_1, \theta_2) = \text{split}(\theta, \max(\gamma_0))} & \quad (E-\text{SN}) \\
\frac{S; T; n; \theta; E[v'(v' \triangleright_{\text{seq}})] \twoheadrightarrow S; T; n; \theta; E[\mathbf{0}] [v'[v] / x]}{\forall \theta, (i) \geq (0, 0)} & \quad (E-T) \\
\frac{S; T; n; \theta; E[v'(v' \triangleright_{\text{seq}})] \twoheadrightarrow S; T; n; \theta; E[\mathbf{0}] [v'[v] / x]}{\forall \theta, (i) \geq (1, 1)} & \quad (E-A) \\
\frac{S; T; n; \theta; E[v'(v' \triangleright_{\text{seq}})] \twoheadrightarrow S; T; n; \theta; E[\mathbf{0}] [v'[v] / x]}{\forall \theta, i \in \text{locked}(S)} & \quad (E-D) \\
\frac{S; T; n; \theta; E[\text{lock}(c)] \twoheadrightarrow S; T; n; \theta; E[\mathbf{0}]}{\forall \theta, (i) \geq (1, 1)} & \quad (E-\text{AS}) \\
\frac{S; T; n; \theta; E[\text{deref lock}_1] \twoheadrightarrow S; T; n; \theta; E[S(i) \theta]}{\forall \theta, i \in \text{locked}(T)} & \quad (E-E) \\
\frac{\text{fresh } n_1\; S' = S, i \mapsto v \; \theta' = \theta, i \mapsto 1; 1; \emptyset}{\forall \theta, (i) \geq (1, 0)} & \quad (E-\text{NG}) \\
\frac{S; T; n; \theta; E[\text{let } \rho, x = \text{ref } v \text{ in } e_2] \twoheadrightarrow S'; T; n; \theta'; E[e_2[i \mapsto n_1]; [\text{loc}_i / x]]}{\forall \theta, (i) \geq (1, 0) \quad \theta' = \theta + (1, 0)} & \quad (E-\text{SH}) \\
\frac{S; T; n; \theta; E[\text{share } \text{loc}_1] \twoheadrightarrow S; T; n; \theta; E[\mathbf{0}]}{\forall \theta, (i) \geq (1, 0) \quad \theta' = \theta + (0, -1)} & \quad (E-\text{RP}) \\
\frac{S; T; n; \theta; E[\text{if true then } e_1 \text{ else } e_2] \twoheadrightarrow S; T; n; \theta; E[e_1]}{\forall \theta, (i) \geq (1, 1) \quad \theta' = \theta + (0, -1)} & \quad (E-\text{IT}) \\
\frac{S; T; n; \theta; E[\text{unlock } \text{loc}_1] \twoheadrightarrow S; T; n; \theta; E[\mathbf{0}]}{\forall \theta, (i) \geq (1, 1) \quad \theta' = \theta + (0, 1)} & \quad (E-\text{UL}) \\
\frac{S; T; n; \theta; E[\text{deref } \text{loc}_1] \twoheadrightarrow S; T; n; \theta; E[\mathbf{0}]}{\forall \theta, (i) \geq (1, 1) \quad \theta' = \theta + (0, 1)} & \quad (E-\text{IF}) \\
\frac{S; T; n; \theta; E[\text{lock}_{\gamma_1} \text{loc}_1] \twoheadrightarrow S; T; n; \theta; E[\mathbf{0}]}{\forall \theta, (i) \geq (1, 1) \quad \theta' = \theta + (0, 1)} & \quad (E-LK0) \\
\frac{S; T; n; \theta; E[\text{fix } x : \tau.f \triangleright_{\text{seq}}] \twoheadrightarrow S; T; n; \theta; E[\mathbf{0}] [f[x/v] / x]}{\forall \theta, (i) \geq (1, 1) \quad \theta' = \theta + (0, 1)} & \quad (E-LK1) \\
\frac{S; T; n; \theta; E[\text{fix } x : \tau.f \triangleright_{\text{seq}}] \twoheadrightarrow S; T; n; \theta; E[\mathbf{0}] [f[x/v] / x]}{\forall \theta, (i) \geq (1, 1) \quad \theta' = \theta + (0, 1)} & \quad (E-FX)
\end{align*}
\]

A.9 Static Semantics: Syntax and Typing Context Substitution Relation

| Type variable list | \( A \) ::= \( \emptyset \mid A, \rho \) |
| Memory List | \( M \) ::= \( \emptyset \mid M, i \mapsto \tau \) |
| Variable list | \( \Gamma \) ::= \( \emptyset \mid \Gamma, x : \tau \) |

A.10 Static Semantics: Type Equivalence

\[
\begin{align*}
\frac{r \equiv r}{r' \equiv r} & \quad (S0) \\
\frac{r \equiv r'}{r \equiv r'} & \quad (S1) \\
\frac{r \equiv r \quad r \equiv r'}{r \equiv r \odot n_2} & \quad (S2) \\
\frac{\emptyset \equiv \emptyset}{r_1 \equiv r_2 \quad r_1 \equiv r_2} & \quad (S3) \\
\frac{\gamma_1 \equiv \gamma_2}{\gamma_1 \odot \gamma_2 \equiv \gamma_2 \odot \gamma_2} & \quad (S4) \\
\frac{\gamma_1 \equiv \gamma_4 \quad \gamma_2 \equiv \gamma_5 \quad \gamma_3 \equiv \gamma_6}{\gamma_1 \odot \gamma_2 \odot \gamma_3 \equiv \gamma_4 \odot \gamma_5 \odot \gamma_6} & \quad (S5) \\
\end{align*}
\]

\[
\begin{align*}
\frac{\tau \equiv \tau}{\tau_1 \equiv \tau_4 \\
\frac{\text{fresh } \rho_1 \odot n}{\tau_1 \equiv \tau_4 \\
\frac{\tau_1 \equiv \tau_4 \quad \rho_1 \equiv \rho_2}{\tau_1[\rho_1 \odot n] \equiv \tau_2[\rho_1 \odot n/\rho'] \quad \forall \rho, \tau_1 \equiv \forall \rho', \tau_2} & \quad (S7) \\
\frac{\tau_1 \equiv \tau_4}{\tau_1[\rho_1 \odot n] \equiv \tau_2[\rho_1 \odot n/\rho'] \quad \forall \rho, \tau_1 \equiv \forall \rho', \tau_2} & \quad (S8) \\
\frac{\tau_1 \equiv \tau_4}{\tau_1 \equiv \tau_4 \\
\frac{\gamma_1 \equiv \gamma_3 \quad \gamma_2 \equiv \gamma_4 \quad \gamma_3 \equiv \gamma_6}{\gamma_1 \equiv \gamma_3 \\
\frac{\tau_1 \equiv \tau_4}{\tau_1 \equiv \tau_4}} & \quad (S9)
\end{align*}
\]
A.11 Static Semantics: Capability Related Rules

\[ \text{is\_pure}(\kappa) = \exists n_1, \exists n_2, \kappa = n_1, n_2 \]

\[
\forall r^x \in \gamma, \text{is\_pure}(\kappa) \Rightarrow \forall r'^{x'} \in \gamma, r' \neq r \Rightarrow -(r \approx r')
\]

\[
\xi = \text{par} \Rightarrow \forall r^x, (r^x \in \text{min}(\gamma) \land \kappa = (0,0)) \land (r^x \in \text{max}(\gamma) \land -\text{is\_pure}(\kappa) \Rightarrow \text{lk}(\kappa) = 0)
\]

(OK)

\[
\frac{r' = r}{r' = r', r' = r'} (M0) \quad \frac{\neg(r' = r)}{\neg(r' = r), r' = r'} (M1) \quad \frac{\emptyset = \emptyset / r}{\emptyset = \emptyset / r} (M2)
\]

\[\text{is\_pure}(k_1) \Rightarrow k_2 = (0,0) \land \text{is\_pure}(k_1) \land \text{is\_pure}(k_2)\]

\[
\begin{align*}
\gamma_1 &= \text{add}(\gamma, \gamma') \quad &\gamma_1 &= \text{add}(\gamma, \gamma') \\
\gamma_2 &= \text{subtract}(\gamma, r^x, \gamma_1) \quad &\kappa &= k_2 + k_1 \\
\gamma_2 &= \text{subtract}(\gamma, r^x, r_1, r_2) \quad &\gamma_2 &= \text{subtract}(\gamma, r^x, \gamma_1, r_2) \\
\gamma_2 &= \text{min}(\gamma_2) \vdash \gamma_3 \quad &\text{seq}(\emptyset) + \gamma
\end{align*}
\]

(D0)

(X1)

\[\text{lk}(\kappa) = 0 \Rightarrow \text{locked}(\gamma, r^x, 0, r) \quad \text{lk}(\kappa) > 0 \Rightarrow \text{locked}(\gamma, r^x, 1, r)
\]

(X2)

\[\text{locked}(\gamma :: \gamma_1, n, r) \lor \text{locked}(\gamma :: \gamma_2, n, r) \quad \text{locked}(\gamma, \gamma_1 ? \gamma_2, n, r)
\]

(X4)

L0)

A.12 Static Semantics: Well Formedness Relation

Constraint Well-formedness

\[
\begin{align*}
M; \Delta \vdash 0 & \quad \vdash M; \Delta \vdash r \\
M; \Delta \vdash r^x & \quad M; \Delta \vdash \gamma_1 \\
M; \Delta \vdash \gamma_1 \land \gamma_2 & \quad M; \Delta \vdash \gamma_3 \\
M; \Delta \vdash \gamma_1, \gamma_2 ? \gamma_3 & \quad M; \Delta \vdash \gamma_3
\end{align*}
\]

Region Well-formedness

\[
\begin{align*}
M; \Delta \vdash r & \quad M; \Delta \vdash \gamma_1 \\
M; \Delta \vdash \gamma_1, r^x & \quad M; \Delta \vdash \gamma_2 \\
M; \Delta \vdash \gamma_1, \gamma_2 & \quad M; \Delta \vdash \gamma_3
\end{align*}
\]

Program Typing Context Well-formedness

\[
\begin{align*}
\vdash M & \quad M; \Delta \vdash \Gamma \\
M; \Delta \vdash \gamma_1 & \quad M; \Delta \vdash \gamma_2 \\
M; \Delta \vdash \gamma_2 & \quad \text{seq}(\emptyset) + \gamma_2
\end{align*}
\]

\[
\begin{align*}
\vdash M; \Delta \vdash \Gamma & \quad M; \Delta \vdash \gamma_1, \gamma_2 \\
\vdash M; \Delta \vdash \gamma_1 & \quad \text{seq}(\emptyset) + \gamma_2
\end{align*}
\]

Type Well-formedness

\[
\begin{align*}
M; \Delta \vdash \text{bool} & \quad M; \Delta \vdash \text{void} \tau \\
M; \Delta \vdash \text{ref}(\tau, r) & \quad M; \Delta \vdash \tau_1 \\
M; \Delta \vdash \tau_1 \land \tau_2 & \quad M; \Delta \vdash \langle \rangle
\end{align*}
\]

\[
\begin{align*}
M; \Delta \vdash \text{bool} & \quad M; \Delta \vdash \text{void} \tau \\
M; \Delta \vdash \tau_1 \land \tau_2 & \quad M; \Delta \vdash \langle \rangle
\end{align*}
\]

\[\Gamma \text{ Well-formedness} \]

\[
\begin{align*}
M; \Delta \vdash 0 & \quad x \notin \text{dom}(\Gamma) \\
M; \Delta \vdash \Gamma \\
M; \Delta \vdash \Gamma, x : \tau
\end{align*}
\]

\[M \text{ Well-formedness} \]

\[
\begin{align*}
\vdash 0 & \quad \vdash M; \Delta \vdash \tau \\
\vdash M; \Delta \vdash \tau
\end{align*}
\]
A.13 Static Semantics: Typing Rules
A.14 Type Safety: Evaluation Context Typing

\[ \vdash M; \Delta; \Gamma; \gamma_1; \xi \quad M; \Delta; \top \]
\[ \vdash M; \Delta; \Gamma; \otimes \quad \text{pop}_p \]
\[ \vdash M; \Delta; \Gamma; \gamma \quad M; \Delta; \text{ref}(r) \]
\[ \vdash M; \Delta; \Gamma; \text{lock}_r \]
\[ \vdash M; \Delta; \Gamma; \xi \quad \text{lockset}_\text{ok}(E[pop_p], \theta) \]

\[ M; \Delta; \Gamma; \rho, \Delta; \Gamma; \gamma_1; \gamma_2 \]
\[ M; \Delta; \Gamma; E : \tau_1 \rightarrow \tau_2 \]
\[ M; \Delta; \Gamma; E[F] : \tau_1 \rightarrow \tau_2 \]
\[ M; \Delta; \Gamma; E[F] : \tau_1 \rightarrow \tau_2 \quad (E0) \]

\[ \vdash M; \Delta; \Gamma; \xi \quad (F1) \]
\[ \vdash M; \Delta; \Gamma; \xi \quad (F0) \]

\[ \xi = \text{lockset}(\text{pop}_p, \theta, \gamma, \epsilon) \quad \text{lockset}_\text{ok}(E, \theta) \]

A.15 Type Safety: Access List Typing

The text is too lengthy and complex to transcribe in a natural format. It involves formal logic and type system rules, which are typically displayed using special symbols and notation specific to type theory and formal proof systems. The content includes rules for evaluating contexts, rules for type safety, and rules for handling access lists, among other things. Each rule and definition is accompanied by formal expressions and logical inferences that are typical in formal logic and type theory.
Proof. The application of Lemma 1 to the assumption implies that $\forall i \in [0, n]. \exists M_i, M_i \vdash S_i; T_i$. Therefore, $S_n; T_n$ is well-typed for some $M_n$. The application of Lemma 18 to $\forall i \in [0, n]. \exists M_i, M_i \vdash S_i; T_i$ and $\emptyset; 0; e \leadsto^e S_n; T_n$ implies $\neg \text{deadlocked}(T_n)$. The application of Lemma 18 to $M_n \vdash S_n; T_n$ implies $S_n; T_n$ is not stuck.

Lemma 1 (Multi-step Program 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 $i \in [0, n]$ $M_i \vdash S_i; T_i$ holds. Let $S_0; T_0$ be a closed well-typed configuration such that $M_0 \vdash S_0; T_0$ for some $M_0$. 

A.17 Type Safety: Multi-step Evaluation

\[
\frac{\forall t \mapsto \tau \in M.M; \emptyset; t \vdash \tau \land (\emptyset; \emptyset) \not\in \emptyset}{M \vdash S}
\]

Configuration Typing

\[
\frac{M \vdash T \quad M \vdash S \quad \text{mutex}(T)}{M \vdash S; T}
\]

Thread Typing

\[
\frac{M; \emptyset; \emptyset; t; e : () \land (y; y') \quad M \vdash T \quad n \not\in \text{dom}(T)}{M \vdash T, n; \emptyset; e}
\]

Not Stuck

\[
\forall T', n; \emptyset; e.T = T', n; \emptyset; e \Rightarrow (T' \subseteq T'' \land S; T \leadsto S'; T'') \lor \text{blocked}(T, n)
\]

\[
\vdash S; T
\]

A.18 Type Safety: Main Theorems

Safety

\[
\emptyset \vdash 0; \emptyset; e \land 0; \emptyset; e \leadsto^e S', T' \Rightarrow S', T' \land \neg \text{deadlocked}(T)
\]

Preservation

\[
M \vdash S; T \land S; T \leadsto S'; T' \Rightarrow \exists M' \supseteq M. M' \vdash S'; T'
\]

Progress

\[
M \vdash S; T \Rightarrow S; T
\]

Deadlock Freedom

\[
\emptyset; 0; e \leadsto^e S_n; T_n \land \forall i \in [0, n]. \exists M_i, M_i \vdash S_i; T_i \Rightarrow \neg \text{deadlocked}(T_n).
\]

A.19 Proof Sketch

Theorem 1 (Type Safety) Let expression $e$ be the initial program and let the initial typing context $M_0$ and the initial program configuration $S_0; T_0$ be defined as follows: $M_0 = \emptyset$, $S_0 = \emptyset$, and $T_0 = [0 : \emptyset; e]$. If $S_0; T_0$ is well-typed in $M_0$ and the operational semantics takes any number of steps $S_0; T_0 \leadsto^0 S_n; T_n$, 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 $\forall i \in [0, n]. \exists M_i, M_i \vdash S_i; T_i$. Therefore, $S_n; T_n$ is well-typed for some $M_n$. The application of Lemma 11 to $\forall i \in [0, n]. \exists M_i, M_i \vdash S_i; T_i$ and $\emptyset; 0; e \leadsto^e S_n; T_n$ implies $\neg \text{deadlocked}(T_n)$. The application of Lemma 18 to $M_n \vdash S_n; T_n$ implies $S_n; T_n$ is not stuck.

Lemma 1 (Multi-step Program 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 $i \in [0, n] M_i \vdash S_i; T_i$ holds. Let $S_0; T_0$ be a closed well-typed configuration such that $M_0 \vdash S_0; T_0$ for some $M_0$. 

\[
\frac{n > 0 \quad S; T \leadsto^0 S_{n-1}; T_{n-1} \quad S_{n-1}; T_{n-1} \leadsto S_n; T_n}{S; T \leadsto^0 S_n; T_n} (E-M1)
\]

\[
\frac{S; T \leadsto^0 S_n; T_n}{S; T \leadsto^0 S; T} (E-M2)
\]
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 \( S_0; T_0 \overset{\sigma}{\rightarrow}^n S_n; T_n \) or \( S_0; T_0 \overset{\sigma}{\rightarrow}^{n-1} S_{n-1}; T_{n-1} \) and \( S_{n-1}; T_{n-1} \overset{\sigma}{\rightarrow} S_n; T_n \). By applying the induction hypothesis on the fact that \( S_0; T_0 \) is well-typed and that \( n - 1 \) steps are performed we obtain that \( \forall t \in [0, n - 1]. \exists M_t, M_{t_1} \vdash S_{n-1}; T_{n-1} \) holds. 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 \( S_{n-1}; T_{n-1} \overset{\sigma}{\rightarrow} S_n; T_n \) implies that \( M_n \vdash S_n; T_n \). Therefore, \( \forall t \in [0, n]. \exists M_t, M_{t_1} \vdash S_t; T_t \).

□

Lemma 2 (Deadlock Freedom) If the initial configuration takes \( n \) steps, where each step is well typed, then the resulting configuration has not reached a deadlocked state.

Proof. Let us assume that \( z \) threads have reached a deadlocked state and let \( m \in [0, z - 1] \), \( k = (m + 1) \mod z \) and \( o = (k + 1) \mod z \). According to definition of deadlocked state, thread \( m \) acquires lock \( t_0 \) and waits for lock \( t_o \), whereas thread \( k \) acquires lock \( t_o \) and waits for lock \( t_k \). Assume that \( m \) is the first of the \( z \) threads that acquires a lock so it acquires lock \( t_k \), before thread \( k \) acquires lock \( t_0 \).

Let us assume that \( S_y; T_y \) is the configuration once \( t_o \) is acquired by thread \( k \) for the first time, \( e_{1_y} \) is the corresponding lockset of \( t_o \), \( (e_{1_y} = \text{lockset}(t_o, 1, E[\text{pop}, \square])) \) and \( e_{2_y} \) is the set of all heap locations \( (e_{2_y} = \text{dom}(S_y)) \) at the time \( t_o \) is acquired. Then, \( (T_o) \) does not belong to \( e_{1_y} \), otherwise thread \( k \) would have been blocked at the lock request of \( t_o \), as \( t_k \) is already owned by thread \( m \).

Let us assume that when thread \( k \) attempts to acquire \( t_k \), the configuration is of the form \( \gamma_k; E_k[\text{lock}_{\gamma_k} \text{loc}_{\gamma_k}] : \text{lock}_{\gamma_k} \text{loc}_{\gamma_k} \). \( \gamma_k \) is well-typed with \( \theta_k; E_k[\text{pop}_{\gamma_k} \square] \text{lock}_{\gamma_k} \text{loc}_{\gamma_k} \) is well-typed with \( \gamma_k \). We have assumed that \( m \) is the first thread to lock \( t_k \) at some step before \( S_y; T_y \), thus \( t_k \in \text{dom}(S_y) \) (the store can only grow — this is immediate by observing the operational semantics rules). By the definition of lockset function and the definition of \( \gamma_k \) we have that \( t_k \in \text{lockset}(t_o, n_2, E_k[\text{pop}_{\gamma_k} \square]) \). Therefore, \( t_k \in \text{lockset}(t_o, n_2, E_k[\text{pop}_{\gamma_k} \square]) \cap \text{dom}(S_y) \subseteq e_{1_y} \), 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 \( S; T \overset{\sigma}{\rightarrow} S'; T' \), 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 \( \theta; E[e] = \theta; e[\square] \) and \( S' = S \) and \( T' = T \), \( \forall t. \theta(t) = (0, 0) \). By inversion of the configuration typing assumption we have that:

- \( M \vdash T, n: \theta; e[\square] \): by inversion of this derivation we have \( M \vdash T \).

- \( M \vdash S \)

- \( \text{mutex}(T, n: \theta; e[\square]) \): implies that \( \text{mutex}(T) \).

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

Case E-A: Rule E-A implies that \( S' = S, \ T' = T, n: \theta; E[\text{pop}_{\gamma_o} e_1[v/x]] \) and \( e = (\lambda x. e_1 \text{ as } \tau_1 \rightarrow \tau_2 \ v) \).

By inversion of the configuration typing assumption we have that:

- \( M \vdash S \)
- \textit{mutex}(T, n; \theta; E[e]): no new locks are acquired. Thus, \textit{mutex}(T, n; \theta; E[\text{pop}_{y_a} e_1[v/x]]) holds.

- \textit{M} \vdash T, n; \theta; E[e]: by inversion of this derivation we have that:
  \begin{itemize}
  \item \textit{M} \vdash T
  \item \( n \notin \text{dom}(T) \)
  \item \( M; \emptyset; \emptyset; E[e] : \langle \gamma \rangle & (\gamma; \gamma') \): by inversion of this derivation we have that \textit{counts\_ok}(E[\text{pop}_{y_b} \square], \theta), \textit{lockset\_ok}(E[\text{pop}_{y_a} \square], \theta), \forall r^e \in \gamma; k = (0, 0), \) and \( M; \emptyset; \emptyset; E : \tau_2' \xrightarrow{\gamma_a; y_b} \langle \gamma \rangle & (\gamma; \gamma') \), \( M; \emptyset; \emptyset; e : \tau_2' & (\gamma_a; y_b) \). By inversion of the latter derivation we have that \( M; \emptyset; \emptyset; v : \tau_1' & (\gamma_b; y_b) \), \( \text{seq}(\gamma_a) + y_b = \gamma_a \oplus \gamma'_b \) and \( M; \emptyset; \emptyset; \lambda x. e_1 \) as \( \tau_1 \xrightarrow{\gamma_a} \tau_2 : \tau'_1 \xrightarrow{\gamma'_a} \tau'_2 & (\gamma_b; y_b) \), where \( \tau'_1 \xrightarrow{\gamma'_a} \tau'_2 \approx \tau_1 \xrightarrow{\gamma_a} \tau_2 \).
  \end{itemize}

By inversion of the function typing derivation we obtain that \( \text{seq}(\emptyset) \vdash \gamma_c \Rightarrow M; \emptyset; \emptyset, x : \tau_1 \vdash e_1 : \tau_2 & (\min(\gamma_c); \gamma_c) \). \( \text{seq}(\emptyset) \vdash \gamma'_c \) (by inversion of \( \text{seq}(\gamma_a) + y_b = \gamma_a \oplus \gamma'_b \) and \( \gamma_c = \gamma'_c \) imply that \( \text{seq}(\emptyset) \vdash \gamma_c \), hence \( M; \emptyset; \emptyset, x : \tau_1 \vdash e_1 : \tau_2 & (\min(\gamma_c); \gamma_c) \) holds. Lemma \([8]\) implies that \( M; \emptyset; \emptyset; v : \tau_1 & (\gamma_b; y_b) \). Lemma \([7]\) implies that \( M; \emptyset; \emptyset; e_1[v/x] : \tau_2 & (\min(\gamma_c); \gamma_c) \) holds. The application of rule \textit{T-PP} implies that \( M; \emptyset; \emptyset; \text{pop}_{y_a} e_1[v/x] : \tau'_2 & (\gamma_a; y_b) \) holds (\( \vdash M; \emptyset; \emptyset; \gamma_a; y_b \) can be derived by the application of lemma \([4]\) to the typing derivation of \( e \)). Thus, \( M; \emptyset; \emptyset; E[\text{pop}_{y_a} e_1[v/x]] : \langle \gamma \rangle & (\gamma; \gamma') \), by the application of rule \textit{EA} to the typing derivation of \( \text{pop}_{y_a} e_1[v/x] \), \( \forall r^e \in \gamma; k = (0, 0) \), \textit{counts\_ok}(E[\text{pop}_{y_b} \square], \theta) and \textit{lockset\_ok}(E[\text{pop}_{y_a} \square], \theta).

Case \textit{E-FX}: Rule \textit{E-FX} implies \( S; T, n; \theta; E[(\text{fix } x : \tau. f) \triangleright^{\text{seq}(\gamma_a)}] \leadsto S; T, n; \theta; E[(f[\text{fix } x : \tau. f/x] \triangleright^{\text{seq}(\gamma_a)}) \).

By inversion of the configuration typing assumption we have that:

- \textit{M} \vdash S
- \textit{mutex}(T, n; \theta; E[e]): no new locks are acquired. Thus, \textit{mutex}(T, n; \theta; E[\text{pop}_{y_a} e_1[v/x]]) holds.

- \textit{M} \vdash T, n; \theta; E[e], where \( e \) is equal to \( (\text{fix } x : \tau. f) \triangleright^{\text{seq}(\gamma_a)} \): by inversion of this derivation we have that:
  \begin{itemize}
  \item \textit{M} \vdash T
  \item \( n \notin \text{dom}(T) \)
  \item \( M; \emptyset; \emptyset; E[e] : \langle \gamma_0 \rangle & (\gamma_0; \gamma) \): lemma \([15]\) implies that \( M; \emptyset; \emptyset; E : \tau_2' \xrightarrow{\gamma_a; y_b} \langle \gamma_0 \rangle & (\gamma_0; \gamma) \) and \( M; \emptyset; \emptyset; e : \tau_2' & (\gamma_a; y_b) \). By inversion of the latter derivation we have that \( M; \emptyset; \emptyset; v : \tau_1 & (\gamma_b; y_b) \), and \( M; \emptyset; \emptyset; \text{fix } x : \tau. f : \tau & (\gamma_b; y_b) \), where \( \text{seq}(\gamma_a) + y_b = \gamma_a \oplus \gamma_b \) and \( \tau = \tau_1 \xrightarrow{\gamma_a} \tau_2 \). By inversion of the typing derivation of \( \text{fix } x : \tau. f \) we obtain that \( M; \emptyset; \emptyset, x : \tau \vdash f : \tau' & (\gamma_b; y_b) \), where \( \tau' = \tau_1 \xrightarrow{\gamma_a} \tau_2 \), \( \gamma_d = \gamma_d \), \( \tau = \tau' \) and \( \gamma_c = \text{summary}(\gamma_d) \). Lemma \([8]\) implies that \( M; \emptyset; \emptyset; f[\text{fix } x : \tau. f/x] : \tau' & (\gamma_b; y_b) \) holds. \( \text{seq}(\gamma_a) + y_a \vdash y_b = \gamma_a \oplus y_d \) is implied by \( \gamma_d = \gamma_d \). \text{seq}(\gamma_a) + y_b = \gamma_a \oplus \gamma_c \), \text{max}(\gamma_c) = \text{max}(\gamma_a) \) and \( \text{min}(\gamma_c) = \text{min}(\gamma_a) \).

Lemma \([6]\) implies that \( M; \emptyset; \emptyset; E : \tau_2' \xrightarrow{\gamma_a; y_a} \langle \gamma_0 \rangle & (\gamma_0; \gamma') \) holds (the domain of \( \gamma'_b \) is a subset of the domain of \( y_b \)). The application of lemma \([6]\) to the latter fact, \( M; \emptyset; \emptyset; E : \tau_2' \xrightarrow{\gamma_a; y_a} \langle \gamma_0 \rangle & (\gamma_0; \gamma') \) implies that \( M; \emptyset; \emptyset; E : \tau_2' \xrightarrow{\gamma_a; y_a} \langle \gamma_0 \rangle & (\gamma_0; \gamma') \).

Lemma \([8]\) implies that \( M; \emptyset; \emptyset; f[\text{fix } x : \tau. f/x] : \tau_1' \xrightarrow{\gamma'_a} \tau_2' & (\gamma_a; y_b; y_a \vdash \gamma'_b) \) and \( M; \emptyset; \emptyset; v : \tau_1' & (\gamma_a \vdash y_b' \gamma_a \vdash \gamma'_b) \). Therefore, \( M; \emptyset; \emptyset; (f[\text{fix } x : \tau. f/x] \triangleright^{\text{seq}(\gamma_a)}) : \tau_2' & (\gamma_a; y_a \vdash \gamma'_a) \).
Case E-LK0, E-LK1, E-UL, E-SH and E-RL: these rules generate side-effects as they modify the reference/count of location $t$. Hence, we are assuming here that $u$ (i.e., in $E[u]$) has one of the following forms: lock$_v$ lock, unlock lock, or release lock. Rules E-LK0, E-LK1, E-UL, E-SH and E-RL imply that $S' = S$, $T' = T, n: \theta'; E[()]$, where () replaces $u$ in context $E$ and $\theta$ differs with respect to $\theta'$ only in the one of the counts of $t$ (i.e., $\theta' = \theta[t \mapsto \theta(t) + (n_1, n_2)]$ and $\gamma_a(r) - \kappa = (n_1, n_2)$ — $\gamma_a$ is the input effect of $E[u]$ and $\kappa$ is the count of the last element of the output effect of $u$).

By inversion of the configuration typing assumption we have that:

- $\text{mutex}(T, n: \theta; E[u]):$ In the case of E-UL, E-SH, E-LK1 and E-RL no new locks are acquired. Thus, mutex$(T, n: \theta'; E[()])$ holds. In the case of rule E-LK0, a new lock $t$ is acquired (i.e., when the lock count of $t$ is zero) the precondition of E-LK0 suggests that no other thread holds $t$: locked$(T) \cap \text{lockset}(t, 1, E[\text{pop}_{\gamma_v}]) = \emptyset$. Thus, mutex$(T, n: \theta'; E[()])$ holds.

- $M; \emptyset; \theta + \gamma':$ By inversion we have that $M; \emptyset; \theta + E: ()$ and $M; \emptyset; \theta + u: ()$ for some $\gamma'_a$ and $\gamma_a$, where $\gamma_a = \gamma_a(i@n)'$ for some $n'$. It can be trivially shown from the derivation that $M; \emptyset; \theta + ()$ and $E[\text{pop}_{\gamma_a}]$.

By definition of lockset function it can be shown that lockset$(j, n_b, E[\text{pop}_{\gamma_v}]) \subseteq \text{lockset}(j, n_b, E[\text{pop}_{\gamma_v}])$ for all $j \neq t$. The same applies for $j = t$ in the case of rules E-SH, E-RL as the lock count of $t$ is not affected. In the case of rules E-LK0, E-LK1, E-UL we have lockset$(t, n_b \pm 1, E[\text{pop}_{\gamma_v}])$, but this is identical to lockset$(t, n_b, E[\text{pop}_{\gamma_v}])$ by the definition of lockset. Therefore lockset$(E[\text{pop}_{\gamma_v}], \theta')$ holds. The predicate counts.ok$(E[\text{pop}_{\gamma_v}], \theta')$ enforces the invariant that the static counts are identical to the dynamic counts $(\theta)$ of $t$. The lock count of $\theta$ is modified by $\pm 1$ and $\gamma_a$ differs with respect to $\gamma_b$ by $(i@n)'$. We can use this fact to show that counts.ok$(E[\text{pop}_{\gamma_v}], \theta')$.

Case E-D, E-AS, 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: $(A \text{p}. f)[i@n_1]$, deref lock, lock := $v$ or pop $v$. Rules E-AS, E-D, E-RP, and E-PP imply that $S' = S$, $T' = T, n: \theta; E[v]$, where $v$ is the value that replaces $u$ in context $E$. By inversion of the configuration typing assumption we have that:

- $M + S$
- $\text{mutex}(T, n: \theta; E[u]):$ no new locks are acquired. Thus, mutex$(T, n: \theta; E[v])$ holds.

- $M + T, n: \theta; E[u]:$ by inversion of this derivation we have that:
  - $M + T$
  - $n \notin \text{dom}(T)$
  - $M; \emptyset; \theta + \gamma':$ In the case of rule E-D and E-AS the value $v$ that substitutes $u$ is () and it can be trivially shown that it is well-typed. In the case rule E-PP applies, $v$ is well-typed by inversion of the typing derivation of $u$. In the case of rule E-RP, $v$ is obtained by substituting $i@n_2$ in the body of function $f$ (i.e., the initial term is...
(A\rho. f)[r@n_1]). We can obtain the typing derivation of \( v \) by applying lemma 11. Notice, that we are dealing with side-effect free rules in this case thus \( u \) has the same input and output effect.

Case E-IT: rule E-IT implies that \( S; T, n : \theta; E[u] \leadsto S; T, n : \theta; E[e] \), where \( u \) is equal to if \( \text{true} \) then \( e_1 \) else \( e_2 \).

By inversion of the configuration typing assumption we have that:
- \( M \vdash S \)
- \( \text{mutex}(T, n : \theta; E[u]): \) no new locks are acquired. Thus, \( \text{mutex}(T, n : \theta; E[v]) \) holds.
- \( M \vdash T, n : \theta; E[u] \), where \( u \) is equal to if \( \text{true} \) then \( e_1 \) else \( e_2 \): by inversion of this derivation we have that:
  - \( M \vdash T \)
  - \( n \notin \text{dom}(T) \)
  - \( M; 0; 0 \vdash \theta; E[u] : \gamma \&(\gamma'; \gamma'') \): by inversion of the typing derivation of \( \theta; E[e] \) we obtain that \( M; 0; 0 \vdash E : \tau_2^\gamma\top \gamma \land (\gamma'; \gamma'') \) and \( M; 0; 0 \vdash u : \tau_2^\gamma \land (\gamma_a; \gamma_b) \). By inversion of the latter derivation we have that \( M; 0; 0 \vdash e_1 : \tau_2^\gamma \land (\gamma_a; \gamma_b), M; 0; 0 \vdash e_2 : \tau_2^\gamma \land (\gamma_a; \gamma_b) \) and \( \gamma_b = \gamma_a, \gamma_b \). Lemma 10 implies that \( M; 0; 0 \vdash E : \tau_2^\gamma\top \gamma \land (\gamma'; \gamma'') \) and \( M; 0 \vdash \gamma' \), where \( \gamma' = \gamma', \gamma_b, \gamma_b \). Thus, \( M; 0 \vdash \gamma'' : \gamma_b \) holds. The application of lemma 3 to the latter fact, \( M; 0; 0 \vdash E : \tau_2^\gamma\top \gamma \land (\gamma'; \gamma'') \) implies that \( M; 0; 0 \vdash E : \tau_2^\gamma\top \gamma \land (\gamma'; \gamma'') \).
  - \( \text{lockset\_ok}(E[\text{pop}_{\gamma_b} \square], \theta) \) and \( \text{counts\_ok}(E[\text{pop}_{\gamma_b} \square], \theta) \): \( \text{lockset\_ok}(E[\text{pop}_{\gamma_b} \square], \theta) \) is immediate from \( \text{lockset}(j, n_b, E[\text{pop}_{\gamma_b} \square]) \subseteq \text{lockset}(j, n_b, E[\text{pop}_{\gamma_b} \square]) \) that holds for all \( j \) in the domain of \( \theta \) (\( n_b \) is the lock count of \( j \) in \( \theta \)) and rule W4. Predicate \( \text{counts\_ok}(E[\text{pop}_{\gamma_b} \square], \theta) \) holds from \( \text{counts\_ok}(E[\text{pop}_{\gamma_b} \square], \theta) \) and the fact that \( \max(\gamma_a : \gamma_b) = \max(\gamma_b) \).

Case E-IF: similar to the previous case.

Case E-NG and E-SN: similar to case E-AP. In the case of rule E-NG the use of lemma 9 will also be required in addition to lemmas 12 and 13. In the case of E-SN, \( \theta \) is divided into \( \theta_1 \) and \( \theta_2 \) for threads \( n \) and \( n' \) respectively. Thus, it is shown that the effects of the remaining computation of thread \( n \) match \( \theta_1 \), whereas the effect of the new thread \( n' \) matches \( \theta_2 \).

Lemma 4 (Well-Formedness) If an expression \( e \) is well-typed in the typing context \( M; \Delta; \Gamma, \) with effect \( \gamma; \gamma' \), then \( \vdash M; \Delta; \Gamma; \gamma; \gamma' \) holds.

Proof. Straightforward proof by induction on the expression typing derivation. The most interesting case is rule T-AP, where it needs to be shown that if \( \vdash M; \Delta; \Gamma; \gamma_1; \gamma_2 \) and \( \vdash M; \Delta; \Gamma; \gamma_2; \gamma_3 \) are the well-formedness derivations of expressions \( e_2 \) and \( e_1 \) respectively and \( \gamma_0 \) is the input effect to the application term, then \( \vdash M; \Delta; \Gamma; \gamma_0; \gamma_3 \) holds.

The premise that \( \gamma_1 = \gamma_0 \oplus \gamma_a \), where \( \gamma_a \) is the annotation of the abstraction type (i.e. the type of \( e_1 \)) implies that \( \gamma_0 \triangleleft \gamma_1 \), \( \vdash M; \Delta; \Gamma; \gamma_1; \gamma_2 \) and \( \vdash M; \Delta; \Gamma; \gamma_2; \gamma_3 \) imply that \( \gamma_1 \triangleleft \gamma_2 \triangleleft \gamma_3 \). Thus, \( \gamma_0 \triangleleft \gamma_3 \). They also imply that \( \text{seq}(\theta) \vdash \gamma_3, \vdash M; M; \Delta \vdash \Gamma \) and \( M; \Delta \vdash \gamma_3 \). The latter fact and the fact that \( \gamma_0 \triangleleft \gamma_3 \) imply that \( M; \Delta \vdash \gamma_0 \). Thus, \( \vdash M; \Delta; \Gamma; \gamma_0; \gamma_3 \) holds.

Lemma 5 (Value-Effect — Using Well-Formedness) If value \( v \) is well-typed in the typing context \( M; \Delta; \Gamma, \) with effect \( \gamma; \gamma \) and \( \vdash M; \Delta; \Gamma; \gamma_1; \gamma_2 \), then \( v \) is well-typed in the same typing context with effect \( \gamma_1; \gamma_1 \) and \( \gamma_2; \gamma_2 \).
Proof. The proof is trivial, but we provide the key steps behind the proof. The assumption implies that \( M;\Delta;\Gamma; y_1; y_2 \) and also \( M;\Delta;\Gamma; y_2; y_2 \) hold (trivial). 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 \( T-L,T-V,T-F,T-RF,T-Tf,T-Tr,T-FX,T-U \)). We may use the latter premises of value typing, which still hold (same typing context), along with the latter two well-formedness derivations to formulate the new value typing derivations with effect \( (y_1; y_1) \) and \( (y_2; y_2) \) respectively. The case for rule \( T-RF \) or rule \( T-FX \) can be trivially shown by induction (the base case is the same as for rule \( T-F \)).

Lemma 6 (Evaluation Context Subtyping)

- \( M;\Delta;\Gamma; E : \gamma_1 ; \gamma_2 \) and \( \gamma_2 = \gamma_2 \) if and only if \( \gamma_1 < \gamma_2 \)

Proof. Straightforward induction on the evaluation context typing relation. The base case is trivial. The inductive hypothesis is trivial by lemma 7.

Lemma 7 (Frame Subtyping)

If the following conditions hold

- \( M;\Delta;\Gamma; F : \gamma_1 ; \gamma_2 \) and \( \gamma_2 = \gamma_2 \) if and only if \( \gamma_1 < \gamma_2 \)

Proof. Straightforward case analysis on the frame typing relation.

Lemma 8 (Typing Equivalence)

If \( \tau' = \tau \) and \( M;\Delta;\Gamma; e : \tau \) and \( \gamma; \gamma' \) then \( M;\Delta;\Gamma; e : \tau \) and \( \gamma; \gamma' \)

Proof. Straightforward induction on the shape of \( e' \).

Lemma 9 (Variable Substitution)

\( M;\Delta;\Gamma;_{x : \tau_1} e : \tau_2 \) and \( M;\Gamma; e [v/x] : \tau_2 \) and \( \gamma; \gamma' \)

Proof. Straightforward induction on the expression typing derivation.

Lemma 10 (Type Well-formedness)

\( M;\Delta;\Gamma; e : \tau \) and \( \gamma; \gamma' \) imply \( M;\Delta;\Gamma; e : \tau \) and \( \gamma; \gamma' \)

Proof. Straightforward induction on the typing rules.

Lemma 11 (Location Substitution)

If the following hold:

- \( M;\Delta;\Gamma; e : \tau \) and \( \gamma; \gamma' \)
- fresh \( n \)

then \( M;\Delta;\Gamma; e [t @ n/\rho] : \tau [t @ n/\rho] \) and \( \gamma [t @ n/\rho]; \gamma' [t @ n/\rho] \).

Proof. Proof by induction on the typing derivation of \( e \).
Lemma 12 (Evaluation Typing Weakening) \( M; \Delta; \Gamma \vdash e : \tau \& (\gamma; \gamma') \), \( M; \emptyset \vdash \tau' \) and \( i \notin \text{dom}(M) \) then \( M; i \mapsto \tau'; \Delta; \Gamma \vdash e : \tau \& (\gamma; \gamma') \).

**Proof.** Proof by induction on the typing derivation of \( e \). \( \square \)

Lemma 13 (Evaluation Context Typing Weakening) \( M; \Delta; \Gamma \vdash E : \tau \overset{\gamma_1; \gamma_2}{\longrightarrow} \tau' \& (\gamma; \gamma') \), \( M; \emptyset \vdash \tau' \) and \( i \notin \text{dom}(M) \) then \( M; i \mapsto \tau'; \Delta; \Gamma \vdash E : \tau \overset{\gamma_1; \gamma_2}{\longrightarrow} \tau' \& (\gamma; \gamma') \).

**Proof.** Proof by induction on the derivation of \( E \). \( \square \)

Lemma 14 (Evaluation Context Composition — E) \( M; \Delta; \Gamma \vdash E[e] : \tau' \& (\gamma_1; \gamma_2) \) and \( M; \Delta; \Gamma \vdash F : \tau \overset{\gamma_a; \gamma_b}{\longrightarrow} \tau' \& (\gamma_1; \gamma_2) \), then \( M; \Delta; \Gamma \vdash E[F[e]] : \tau' \& (\gamma_1; \gamma_2) \).

**Proof.** Proof by case analysis on typing derivation of \( E \). The base case is immediate as \( \square[e] = e \). The inductive case where \( E = E'[F] \), the proof is immediate by inversion of the derivation of \( E \) and the application of lemma [16] \( \square \)

Lemma 15 (Evaluation Context Decomposition — E) \( M; \Delta; \Gamma \vdash E[e] : \tau' \& (\gamma_1; \gamma_2) \), then there exists a \( \gamma_a, \gamma_b \) and \( \tau \) such that \( M; \Delta; \Gamma \vdash e : \tau \& (\gamma_a; \gamma_b) \) and \( M; \Delta; \Gamma \vdash F : \tau \overset{\gamma_a; \gamma_b}{\longrightarrow} \tau' \& (\gamma_1; \gamma_2) \).

**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 [4] and [10] and the application rule E0). The inductive case, where \( E[e] = E'[F][e] \) is immediate by lemma [17] and rule E1 \( \square \)

Lemma 16 (Evaluation Context Composition — F) \( M; \Delta; \Gamma \vdash e : \tau \& (\gamma_a; \gamma_b) \) and \( M; \Delta; \Gamma \vdash F : \tau \overset{\gamma_a; \gamma_b}{\longrightarrow} \tau' \& (\gamma_1; \gamma_2) \), 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 \( F \). \( \square \)

Lemma 17 (Evaluation Context Decomposition — F) \( M; \Delta; \Gamma \vdash F[e] : \tau' \& (\gamma_1; \gamma_2) \), then there exists a \( \gamma_a, \gamma_b \) and \( \tau \) such that \( M; \Delta; \Gamma \vdash e : \tau \& (\gamma_a; \gamma_b) \) and \( M; \Delta; \Gamma \vdash F : \tau \overset{\gamma_a; \gamma_b}{\longrightarrow} \tau' \& (\gamma_1; \gamma_2) \).

**Proof.** Proof by case analysis on the structure of \( F \). The premises required for each case (i.e., rules F1-F10) are given by the premises of the typing derivation of \( F[e] \). \( \square \)

Lemma 18 (Progress) Let \( S; T \) be a closed well-typed configuration with \( M \vdash S; T \) then \( S; T \) is not stuck (\( \tau \vdash S; T \)).

**Proof.** It suffices to show that for any thread in \( T \), a step can be performed or block predicate holds for it. Let \( n \) be an arbitrary thread in \( T \) such that \( T = T_1, n; \theta; e \) for some \( T_1 \). By inversion of the typing derivation of \( S; T \) we have that \( M; \emptyset; \theta; e : \emptyset \& (\gamma; \gamma') \), mutex(\( T \)), and \( M \vdash S \).

If \( e \) is a value then by inversion of \( M; \emptyset; \theta; e : \emptyset \& (\gamma; \gamma') \), we obtain that \( \gamma = \gamma' \), \( E[e] = \square[()] \) and \( \forall \theta. \theta(i) = (0, 0) \), as a consequence of \( \forall \theta. \theta(i) = (0, 0) \) and counts-ok(\( \square[pop_{\gamma} \square], \theta \)). Thus, rule E-T can be applied.

If \( e \) is not a value then according to lemma [19] there exists a redex \( u \) and an evaluation context \( E \) such that \( e = E[u] \). By inversion of the thread typing derivation for \( e \) we obtain that \( M; \emptyset; \theta; u : \tau \& (\gamma_a; \gamma_b) \), \( M; \emptyset; \theta; E : \tau \overset{\gamma_a; \gamma_b}{\longrightarrow} \emptyset \& (\gamma; \gamma') \), counts-ok(\( E[pop_{\gamma}] \square, \theta \)) hold.

Then, we proceed by performing a case analysis on \( u \):
Case \((\lambda x. e') \text{ as } \tau \triangleright v\)\text{par}: it suffices to show that \((\theta_1, \theta_2) = \text{split}(\theta, \max(\gamma_c))\) is defined, where \(\gamma_c\) is the annotation of type \(\tau\). If \(\max(\gamma_c)\) is empty, then the proof is immediate from the base case of split function. Otherwise, we must show that for all \(i\), the count \(\theta(i)\) is greater than or equal to the sum of all \((i@\eta)^k\) in \(\max(\gamma_c)\). This can be shown by considering \(\text{par} \vdash \gamma_b = \gamma_a \oplus \gamma_c\) (i.e., the \(\max\) counts in \(\gamma_a\) are less than or equal to the \(\max\) counts in \(\gamma_b\)), which can be obtained by inversion of the typing derivation of \((\lambda x. e' \text{ as } \tau \triangleright v)\text{par}\), and the exact correspondence between static \((\gamma_b)\) and dynamic counts (i.e., \(\text{counts.ok}(E[\text{pop}\gamma_b \square]), \theta\)). Thus, rule \(E-SN\) can be applied to perform a single step.

Case \(\text{share loc}_c: \text{counts.ok}(E[\text{pop}\gamma_a \square], \theta)\) establishes an exact correspondence between dynamic and static counts. The typing derivation implies that \(\gamma_a(i@\eta) \geq (2, 0)\), for some \(n_1\) existentially bound in the premise of the derivation. Therefore, \(\theta(i) \geq (1, 0)\). It is possible to perform a single step using rule \(E-SH\). The cases for \(\text{release loc}_c\) and \(\text{unlock loc}_c\) can be shown in a similar manner.

Case \(\text{lock}_a \text{ loc}_c\): similarly to the case we can show that \(\theta(i) = (n_1, n_2)\) and \(n_1\) is positive. If \(n_2\) is positive, rule \(E-LKI\) can be applied. Otherwise, \(n_2\) is zero. Let \(\epsilon\) be equal to \(\text{lock}(T_1) \cap \text{lockset}(i, 1,E[\text{pop}\gamma_a \square])\). If \(\epsilon\) is empty then rule \(E-LK0\) can be applied in order to perform a single step. Otherwise, \(\text{blocked}(T, n)\) predicate holds and the configuration is not stuck.

Case \(\text{deref loc}_c\): it can be trivially shown (as in the previous case of \(\text{share}\) that we proved \(\theta(i) \geq (1, 0)\), that \(\theta(i) \geq (1, 1)\) and since \(\text{mutex}(T_1, n: \theta; E[\text{deref loc}_c])\) holds, then \(i \notin \text{lock}(T_1)\) and thus rule \(E-D\) can be used to perform a step. The case of \(\text{loc}_c := v\) can be shown in a similar manner.

Case \((\lambda x. e' \text{ as } \tau \triangleright v)\text{seq}(\gamma_a)\): a step can be taken by rule \(E-A\).

Case \(\text{pop}\gamma_b e'[v/x]\): a step can be taken by rule \(E-PP\).

Case \((\text{fix } x : \tau. f)\text{[i@n_1]}\): a step can be taken by rule \(E-RP\).

Case \((\text{fix } x : \tau. f)\text{[i@n_1]}\): a step can be taken by rule \(E-FX\).

Case \(\text{if true then } e_1\text{ else } e_2\): a step can be taken by rule \(E-IT\).

Case \(\text{if false then } e_1\text{ else } e_2\): a step can be taken by rule \(E-IF\).

**Lemma 19 (Redex)** \(\text{If } M; \Delta; \Gamma \vdash e : \tau \land (\gamma_1 \land \gamma_2)\text{ and } e\) is not a value then \(M; \Delta; \Gamma \vdash E'[u] : \tau \land (\gamma_1 \land \gamma_2)\) such that \(E'[u] = e\).

**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 \(e'\). \(\square\)