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

Abstract

Deadlocks occur in multi-threaded 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. Several type systems have been proposed [5, 2, 8, 10, 11] that prevent deadlocks by imposing a strict (non-cyclic) lock-acquisition order that must be respected throughout the entire program. This approach greatly limits programming language expressiveness as many correct programs are rejected unnecessarily. Boudol has recently proposed a type system that avoids deadlocks and is more permissive than existing approaches [1]. However, his system can only deal with programs that use lexically-scoped locking primitives.

In this paper we sketch a simple language with functions, mutable references, explicit (de-)allocation constructs and unstructured (i.e., non lexically-scoped) locking primitives. To avoid deadlocks, we propose a type system for this language based on Boudol's idea. We argue that the addition of unstructured locking primitives makes Boudol's system unsound and show that it is possible to regain soundness by preserving more information about the order of events both statically and dynamically.

Our work is part of a more general effort to design a language for systems programming [6, 7] that guarantees memory safety, race freedom and definite release of resources such as memory and locks.

2 Deadlock Freedom and Related Work

We start by providing a concrete definition of deadlocks and compare our work with existing static approaches to deadlock freedom. 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.

Therefore, deadlock freedom can be guaranteed by denying at least one of the above conditions *before* or *during* program execution. Coffman has identified three strategies that guarantee deadlock-freedom:

- *Deadlock prevention*: At each point of execution, *ensure* that at least one of the above conditions is not satisfied. Thus, programs that fall into this category are correct by design.
- *Deadlock detection and recovery*: A dedicated observer thread *determines* whether the above conditions are satisfied and preempts some of the deadlocked threads, releasing (some of) their locks, so that the remaining threads can make progress.

- *Deadlock avoidance*: Using advance information 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.

The majority of literature for language-based deadlock freedom falls under the first two strategies. In the deadlock prevention category, one finds type and effect systems [5, 2, 8, 10, 11] that guarantee deadlock freedom by statically enforcing a global lock-acquisition ordering that must be respected by all threads. In this setting, starting with the work of Flanagan and Abadi [5], lock handles are associated with type-level lock names via the use of singleton types. Thus, handle lk_ι is of type $\texttt{lk}(\iota)$. The same applies to lock handle variables. The effect system tracks the order of lock operations on handles or variables and determines whether all threads acquire locks in the same order.

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

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

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 [8] as there is no single lock order for *both* threads. Similar considerations apply to the more recent works of Suanaga [10] and Vasconcelos *et al.* [11] dealing with non lexically-scoped locks.

Recently, Boudol developed a type and effect system for deadlock freedom [1], which is based on *deadlock avoidance*. The effect system calculates for each expression the set of acquired locks and annotates lock operations with the "future" lockset. The run-time system utilizes the inserted annotations so that each lock operation can only proceed when its "future" lockset is 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:

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

This program is accepted by Boudol's type system which, in general, allows locks to be acquired in *any* order. At run-time, the first lock operation of the first thread must ensure that y has not been acquired by the second (or any other) thread, before granting x (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. As it will be shown, his type and effect system cannot guarantee deadlock freedom for unscoped locking operations. In the section that follows, we discuss a novel type system for a simple language with mutable references, that is intended to guard against deadlocks and, taking advantage of our previous work [6], against race conditions and memory violations as well.

3 Type System Overview

In this section, we sketch a type system that guarantees absence of deadlocks in a language supporting non lexically-scoped locking operations. As mentioned earlier, Boudol's proposal does not support unstructured locking; even if his language had `lock`/`unlock` constructs, instead of `lock`...`in`..., Boudol's

```
let  f = λx.λy.λz.  lock_{y} x;    x := x+1;     |  lock_{a} a;    a := a+1;
                    lock_{z} y;    y := y+x;     |  lock_{b} a;    a := a+a;
                    unlock x;                    |  unlock a;
                    lock_∅ z;      z := z+y;     |  lock_∅ b;      b := b+a;
                    unlock z;                    |  unlock b;
                    unlock y                     |  unlock a
in   f a a b
                                (a)              |              (b)
```

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

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.

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 `lock` and `unlock` pair.) Now, function f is used by instantiating both x and y with the same variable a, and instantiating z with a 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 `lock` for a now matches with the *last* (and not the first) `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 `lock` should contain b, but it does not. (The annotation of the second `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 calculating an unordered set of locks, the type system precisely tracks the order of `lock` and `unlock` operations, without enforcing a strict lock-acquisition order. As in Boudol's system, `lock` operations are annotated with the "future" effect (our "ordered future" lockset). Function application terms are explicitly annotated with a *continuation effect*, representing the effect of the code succeeding the application term. At run-time, when a function application redex is evaluated, its annotation is pushed on the stack. When a `lock` operation is evaluated, the "future" lockset is calculated by inspecting the annotation and (if necessary) the lookup proceeds with the continuation effects of the enclosing context that are found on the stack. The `lock` operation succeeds only when both the lock and the "future" lockset are available.

Figure 2 illustrates the same program as in Figure 1, except that locking operations are now annotated with the "ordered future" lockset. For example, the annotation $[y+, x-, z+, z-, y-]$ at the first `lock` operation means that in the future (i.e., after this lock operation) y will be acquired, then x will be

$$
\begin{array}{lll|ll}
\texttt{let}\ \ f = \lambda x.\lambda y.\lambda z. & \texttt{lock}_{[y+,x-,z+,z-,y-]}\ x; & x := x+1; & \texttt{lock}_{[a+,a-,b+,b-,a-]}\ a; & a := a+1; \\
 & \texttt{lock}_{[x-,z+,z-,y-]}\ y; & y := y+x; & \texttt{lock}_{[a-,b+,b-,a-]}\ a; & a := a+a; \\
 & \texttt{unlock}\ x; & & \texttt{unlock}\ a; & \\
 & \texttt{lock}_{[z-,y-]}\ z; & z := z+y; & \texttt{lock}_{[b-,a-]}\ b; & b := b+a; \\
 & \texttt{unlock}\ z; & & \texttt{unlock}\ b; & \\
 & \texttt{unlock}\ y & & \texttt{unlock}\ a & \\
\texttt{in}\ \ f\ a\ a\ b & & & & \\
 & \text{(a)} & & \text{(b)} &
\end{array}
$$

Figure 2: The example program of Figure 1, with "ordered future" lockset annotations, now well typed both before (a) and after substitution (b).

$$
\begin{array}{llll}
\textbf{Expression} & e & ::= & x \mid \mathsf{c} \mid f \mid (e\ e)^{\xi} \mid (e)[r] \mid e := e \\
 & & \mid & \texttt{deref}\ e \mid \texttt{let}\ \rho, x = \texttt{ref}\ e\ \texttt{in}\ e \\
 & & \mid & \texttt{share}\ e \mid \texttt{release}\ e \mid \texttt{lock}_{\gamma}\ e \\
 & & \mid & \texttt{unlock}\ e \mid () \mid \mathsf{pop}_{\gamma}\ e \mid \texttt{loc}_{\iota} \\
\textbf{Value} & v & ::= & f \mid \mathsf{c} \mid \texttt{loc}_{\iota} \\
\textbf{Function} & f & ::= & \lambda x.e\ \mathsf{as}\ \tau \xrightarrow{\gamma} \tau \mid \Lambda\rho.f
\end{array}
\qquad
\begin{array}{llll}
\textbf{Type} & \tau & ::= & b \mid \langle\rangle \mid \tau \xrightarrow{\gamma} \tau \mid \forall\rho.\tau \mid \texttt{ref}(\tau, r) \\
\textbf{Location} & r & ::= & \rho \mid \iota@n \\
\textbf{Calling mode} & \xi & ::= & \mathsf{seq}(\gamma) \mid \mathsf{par} \\
\textbf{Capability} & \kappa & ::= & n, n \mid \overline{n, n} \\
\textbf{Effect} & \gamma & ::= & \emptyset \mid \gamma, r^{\kappa}
\end{array}
$$

Figure 3: Language syntax.

released, and so on. If x and y were different, the run-time system would deduce that between this `lock` operation on x 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.

4 Formalism

4.1 Language Description

The syntax of our language is illustrated in Figure 3, where x and ρ range over term and "region" variables, respectively. Similarly to our previous work [6, 7], a region 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, but this is of course not necessary.

The language core comprises of variables (x), constants (c), functions, and function application. Functions can be region polymorphic ($\Lambda\rho.f$) and region 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* (ξ), which is $\mathsf{seq}(\gamma)$ for normal (sequential) application and par for parallel application. Notice that sequential application terms are annotated with γ, 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. Term $\mathsf{pop}_{\gamma}\ e$ encloses a function body e and can only appear during evaluation. The same applies to constant regions $\iota@n$, which cannot exist at the source-level. The construct `let` ρ, x = `ref` e_1 `in` e_2 allocates a fresh cell, initializes it to e_1, and associates it with variables ρ and x within expression e_2. As in other approaches, we use ρ as the type-level representation of the new cell. The type of reference variables x is the singleton type $\texttt{ref}(\rho,\tau)$, where τ is the type of the cell's contents. This allows the type system to connect x and ρ 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 $\texttt{loc}_\iota$ represents a reference to a location ι and is introduced during evaluation. Source programs cannot contain $\texttt{loc}_\iota$.

At any given program point, each cell is associated with a *capability* (κ). 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. 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 our equivalent of thread-local data.) Capabilities can be either *pure* (n_1,n_2) or *impure* $(\overline{n_1,n_2})$. 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. Capability counts determine the validity of operations on cells. Similarly with *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 $\texttt{share}\,e$, $\texttt{release}\,e$, $\texttt{lock}_\gamma\,e$ and $\texttt{unlock}\,e$ operate on reference e. The first two constructs *increment* and *decrement* the cell reference count of e respectively. Similarly, the latter two constructs *increment* and *decrement* the lock count of e. As mentioned earlier, the run-time system inspects the lock annotation γ to determine whether it is safe to lock e.

4.2 Operational Semantics

We define a *small-step* operational semantics for our language in Figure 5.[1] The evaluation relation transforms *configurations*. A configuration C (see Figure 4) consists of an abstract *store* S and a thread map T.[2] A store S maps constant locations (ι) to values (v). A thread map T associates thread identifiers to expressions (i.e., threads) and access lists. An *access list* θ, maps location identifiers to *reference* and *lock* counts.

A *frame* F, (Figure 4) is an expression with a *hole*, represented as $\Box$. The hole indicates the position where the next reduction step can take place. A *thread evaluation context* E, (Figure 4) is defined as a list of 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 [9]. At each step, a *random* thread (n) is chosen from the thread list for evaluation (Figure 5). 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. Notice, that θ is divided into two lists θ_1 and θ_2 using the new thread's effect γ_a as a reference for consuming the correct number of counts from θ.

The sequential function application (*E-A*) rule reduces an application redex to an *pop* expression, which contains the body of the function and is annotated with the same effect as the application term. Rule *E-AS* requires that the location (ℓ) being accessed, is both live and accessible and no other thread has access to ℓ. Rule *E-NG* appends a fresh location ι (with initial value v) and the dynamic count $1,1$ to S and θ respectively.

[1]A full formalization and the semantics of our language are given in the Appendix.

[2]The order of elements in comma-separated lists, e.g. in a store S or in a list of threads T, is unimportant; we consider all list permutations as equivalent.

$$\begin{array}{llll}
\textbf{Dynamic Counts} & \theta & ::= & \emptyset \mid \theta, \iota \mapsto n_1, n_2 \\
\textbf{Store} & S & ::= & \emptyset \mid S, \iota \mapsto v \\
\textbf{Threads} & T & ::= & \emptyset \mid T, n : \theta; e \\
\textbf{Configuration} & C & ::= & S ; T \\
\textbf{Locations} & \epsilon & ::= & \emptyset \mid \epsilon, \iota
\end{array}
\qquad
\begin{array}{lll}
E & ::= & \square \mid E[F] \\
F & ::= & (\square\ e)^{\xi} \mid (v\ \square)^{\xi} \mid (\square)[r] \mid \texttt{let}\,\rho, x = \texttt{ref}\,\square\,\texttt{in}\,e \\
 & \mid & \texttt{deref}\,\square \mid \square := e \mid v := \square \mid \texttt{share}\square \mid \texttt{release}\square \\
 & \mid & \texttt{lock}_{\gamma_1}\,\square \mid \texttt{unlock}\square \mid \texttt{pop}_{\gamma}\,\square
\end{array}$$

Figure 4: Operational semantics syntax and evaluation context.

$$\frac{v' \equiv \lambda x. e_1 \text{ as } \tau_1 \xrightarrow{\gamma_a} \tau_2 \qquad \text{fresh } n' \qquad \gamma_a \vdash \theta = \theta_1 \oplus \theta_2}{S ; T, n : \theta; E[(v'\ v)^{\mathsf{par}}] \rightsquigarrow S ; T, n : \theta_1 ; E[()], n' : \theta_2 ; \square[(v'\ v)^{\mathsf{seq}(\emptyset)}]}\ \textit{(E-SN)}$$

$$\frac{v' \equiv \lambda x. e_1 \text{ as } \tau_1 \xrightarrow{\gamma_a} \tau_2}{S ; T, n : \theta; E[(v'\ v)^{\mathsf{seq}(\gamma_b)}] \rightsquigarrow S ; T, n : \theta; E[\mathsf{pop}_{\gamma_b}\ e_1[v/x]]}\ \textit{(E-A)} \qquad \frac{\theta(\iota) \geq (1,1) \qquad \iota \notin \mathsf{locked}(T)}{S ; T, n : \theta; E[\texttt{loc}_\iota := v] \rightsquigarrow S\,[\iota \mapsto v]; T, n : \theta; E[()]}\ \textit{(E-AS)}$$

$$\frac{\text{fresh } \iota @ n_1 \qquad S' = S, \iota \mapsto v \qquad \theta' = \theta, \iota \mapsto 1, 1}{S ; T, n : \theta; E[\texttt{let}\,\rho, x = \texttt{ref}\ v\ \texttt{in}\ e_2] \rightsquigarrow S' ; T, n : \theta' ; E[e_2[\iota @ n_1/\rho][\texttt{loc}_\iota/x]]}\ \textit{(E-NG)}$$

$$\frac{\begin{array}{c}\theta' = \theta[\iota \mapsto (n_1, n_2 + 1)] \qquad E[\mathsf{pop}_{\gamma_1}\ \square]; \iota; 1 \vdash \epsilon \\ \theta(\iota) = (n_1, n_2) \qquad n_1 \geq 1 \qquad n_2 = 0 \Rightarrow \mathsf{locked}(T) \cap (\epsilon \cup \{\iota\}) = \emptyset\end{array}}{S ; T, n : \theta; E[\texttt{lock}_{\gamma_1}\ \texttt{loc}_\iota] \rightsquigarrow S ; T, n : \theta' ; E[()]}\ \textit{(E-LK)}$$

Figure 5: Selected operational rules.

The most interesting rule is *E-LK* that dynamically computes the "future" lockset (ϵ) by inspecting the preceding stack frames (E) as well as the lock annotation (γ_1). The lockset is a list of locations (and thus locks). *E-LK* requires that the reference being locked (ι) is live and checks that no other thread holds ι nor the references specified in ϵ. If it succeeds, the lock count of ι is incremented by one. Notice, that this check occurs only when ι is unlocked.

4.3 Static Semantics

We briefly discuss the most interesting parts of our type and effect system. Effects are used to statically track the capability of each cell. An effect (γ) is an *ordered list* of elements of the form r^{κ}, denoting that cell r is associated with capability κ.

The syntax of types in Figure 3 (on page 4) is more or less standard. Atomic types consist of the base (b) and the unit ($\langle\rangle$) type. The reference type $\texttt{ref}(\tau, r)$ is associated with a type-level cell name r. Monomorphic function types carry an *effect*. The input to the typing relation is an expression e, the typing context $M;\Delta;\Gamma$ and an input effect γ. M is a mapping of constant locations to types. Δ is a set of cell variables. Γ is a mapping of term variables to types. The output of the typing relation is the type τ assigned to expression e as well as an output effect γ'. We denote this by $M;\Delta;\Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$. As mentioned, each lock operation must be annotated with the "future" lockset. This requirement imposes the restriction that effects must flow backwards. Thus, the input effect to an expression e represents the operations that follow after e is evaluated. Another requirement is that effects must reflect the exact order of cell operations. Thus, the typing relation does not modify the input effect, but rather appends to it. Therefore, the input effect is *always* a prefix of the output effect.

A few selected typing rules are given in Figure 6. The typing rule for function application (*T-A*) splits and joins capabilities to input effect γ by utilizing information from the function effect (γ_a). Notice, that γ_a contains the entire history of events occurring in the function body. Thus, it contains the

$$\frac{\begin{array}{c} M;\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_a} \tau_2 \,\&\, (\gamma_3;\gamma') \qquad \xi \vdash \gamma_a \qquad \gamma_2 = \gamma \oplus \gamma_a \\ M;\Delta;\Gamma \vdash e_2 : \tau_1 \,\&\, (\gamma_2;\gamma_3) \qquad \xi = \mathsf{seq}(\gamma) \vee (\xi = \mathsf{par} \wedge \tau_2 = \langle\rangle) \end{array}}{M;\Delta;\Gamma \vdash (e_1\ e_2)^{\xi} : \tau_2 \,\&\, (\gamma;\gamma')} \ \textit{(T-A)} \qquad \frac{\begin{array}{c} M;\Delta;\Gamma \vdash e_1 : \mathtt{ref}(\tau,r) \,\&\, (\gamma_1;\gamma') \\ M;\Delta;\Gamma \vdash e_2 : \tau \,\&\, (\gamma;\gamma_1) \qquad \gamma(r) \geq (1,1) \end{array}}{M;\Delta;\Gamma \vdash e_1 := e_2 : \langle\rangle \,\&\, (\gamma;\gamma')} \ \textit{(T-AS)}$$

$$\frac{\begin{array}{c} M;\Delta;\Gamma \vdash e_1 : \tau_1 \,\&\, (\gamma_2 \setminus \rho;\gamma') \qquad \gamma_1 = \gamma_2, \rho^{1,1} \\ M;\Delta \vdash \tau \qquad M;\Delta,\rho;\Gamma, x : \mathtt{ref}(\tau_1,\rho) \vdash e_2 : \tau \,\&\, (\gamma, \rho^{0,0};\gamma_1) \end{array}}{M;\Delta;\Gamma \vdash \mathtt{let}\, \rho, x = \mathtt{ref}\, e_1\ \mathtt{in}\, e_2 : \tau \,\&\, (\gamma;\gamma')} \ \textit{(T-NG)} \qquad \frac{\begin{array}{c} M;\Delta;\Gamma \vdash e : \mathtt{ref}(\tau,r) \,\&\, (\gamma, r^{\kappa-(0,1)};\gamma') \\ \kappa \geq (1,1) \qquad \gamma(r) = \kappa \end{array}}{M;\Delta;\Gamma \vdash \mathtt{lock}_{\gamma}\, e : \langle\rangle \,\&\, (\gamma;\gamma')} \ \textit{(T-LK)}$$

Figure 6: Selected typing rules.

effects expected by the environment (precondition) as well as the effect returned to the environment (postcondition). $\gamma_2 = \gamma \oplus \gamma_a$ subtracts the effects specified in the precondition of γ_a from γ and adds the resulting effect back to the postcondition of γ_a. The resulting effect γ_2 is passed as the input to e_2 and similarly, the output effect of e_2, γ_3, is passed as the input effect to e_3. The effect for the application term is γ'. In the case of parallel application par, rule *T-A* also requires that the return type is unit, whereas for sequential application $\mathsf{seq}(\gamma')$, the input effect γ must be equal to γ'. $\xi \vdash \gamma_a$ ensures that pure capabilities are not aliased. It also enforces the invariant that all impure capabilities that appear in the postcondition of γ_a have a zero lock count and all regions and locks must be released, when parallel application takes place.

Similar considerations apply to rule *T-AS*, where the input effect γ becomes γ_1 once e_2 is type checked and γ_1 becomes γ' after e_1 is type checked. Notice that r is type-level name of the cell referenced by e_1. *T-A* checks that the capability assigned to the most recent operation on r in effect γ (i.e., $\gamma(r)$) has a positive reference and lock count. In other words, r must be live and locked once e_1 and e_2 are evaluated. Rule *T-LK* type checks the locking operator. The annotation γ must match the input effect. The rule also tells us that if r is the type-level name, then $\gamma(r)$ must be live and locked. The input effect to e_1 extends γ with $r^{\kappa-(0,1)}$, which implies that just after e_1 is evaluated the lock count of r gets incremented by one. The rule for creating new cells (*T-NG*) passes the input effect γ to the body of `let` and appends $\rho^{0,0}$ to γ. This tells us that ρ must be consumed within e_2. The output effect of e_2 is $\gamma_2, \rho^{1,1}$, which implies that when e_2 is evaluated for the first time, ρ is live and locked and there is no other event preceding $\rho^{1,1}$. The ρ is removed from remaining effect γ_2 and is passed to e_1, the initializer expression of the new cell. The output effect of e_1 is the output effect of `let`.

4.4 Type Safety

In this section we discuss the fundamental theorems that prove type safety of our language.[3] The type safety formulation is based on proving the *preservation* and *progress* lemmata. Informally, a program written in our language is safe when for each thread of execution an evaluation step can be performed or that thread is waiting for a lock (*blocked*). In addition, there must be at least one thread that is not blocked for all execution states. As discussed in Section 4.2, a thread may become stuck when it accesses a location that is not live or accessible. Of course, a thread may become stuck when it performs a non well-typed operation.

Definition 1 (Thread Typing)**.** Let T be a collection of threads. Let M be a mapping from location identifiers to types. The relation $M;\Delta;\Gamma \vdash_t \theta; E[e] : \langle\rangle \,\&\, (\gamma;\gamma')$, types the evaluation context E, the expression e and establishes a correspondence between the *access list* θ and the static effect γ'. That is, for each thread m and for each location ι owned by m, the dynamic reference and lock counts of ι are identical

[3] A proof sketch and a full formalization of our language are given in the Appendix.

to the static counts of ι deduced by the type system for the evaluation context of m. The following rules define *well-typed* threads.

$$\frac{}{M;\emptyset \vdash \emptyset} \qquad \frac{\begin{array}{c} M;\emptyset;\emptyset \vdash_t \theta;E[e] : \langle\rangle\,\&\,(\gamma;\gamma') \\ M \vdash T \qquad n \notin \mathsf{dom}(T) \end{array}}{M \vdash T, n:\theta;E[e]}$$

Definition 2 (Store Typing)**.** A store S is *well-typed* with respect to M (we denote this by $M \vdash S$) when the following conditions are met:

- the domain of S equals the domain of M and
- for each location ι, the stored value $S(\iota)$ is closed and has type $M(\iota)$ with empty effects, i.e., $M;\emptyset;\emptyset \vdash S(\iota) : M(\iota)\,\&\,(\emptyset;\emptyset)$.

Definition 3 (Configuration Typing)**.** A configuration $S;T$ is *well-typed* with respect to M (we denote this by $M \vdash S;T$) when the collection of threads T and the store S are well-typed with respect to M, and locks are acquired by at most one thread (i.e., $\mathsf{mutex}(T)$ holds).

Definition 4 (Deadlocked State)**.** A set of threads $n_0,\ldots,n_k$, where k is greater than zero, has reached a *deadlocked state*, when each thread n_ι, has acquired the lock of the succeeding thread $\ell_{(\iota+1)\,\mathtt{mod}\,k+1}$ and is waiting for lock ℓ_ι.

Definition 5 (Not stuck)**.** A configuration $S;T$ is *not stuck* when each thread in T can take one of the evaluation steps in Figure 5 or it is waiting for a lock held by some other thread. Additionally, threads in T must *not* have reached a deadlocked state.

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

Lemma 1 (Progress — Program)**.** Let $S;T$ be a closed well-typed configuration with $M \vdash S;T$ such that threads in T are not deadlocked. Then $S;T$ is not stuck.

Lemma 2 (Preservation — Program)**.** Let $S;T$ be a well-typed configuration with $M \vdash S;T$. If the operational semantics takes a step $S;T \rightsquigarrow S';T'$, then there exist $M' \supseteq M$ such that the resulting configuration is well-typed with $M' \vdash S';T'$.

The *type safety* theorem is a direct consequence of Lemmata 1 and 2. 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 $T_0 = \{0 : \emptyset; e\}$.

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

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

References

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

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

[3] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis: Proceedings of the 10th International Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

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

[5] C. Flanagan and M. Abadi. Object types against races. In J. C. M. Baeten and S. Mauw, editors, *Concurrency Theory: Proceedings of the 10th International Conference*, volume 1664 of *LNCS*, pages 288–303. Springer, 1999.

[6] P. Gerakios, N. Papaspyrou, and K. Sagonas. A concurrent language with a uniform treatment of regions and locks. In *Proceedings of the Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, 2009. An extended version will appear in the post-proceedings published by *Electronic Proceedings in Theoretical Computer Science*, 2010.

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

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

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

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

[11] V. Vasconcelos, F. Martin, and T. Cogumbreiro. Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In *Proceedings of the Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, 2009. An extended version will appear in the post-proceedings published by *Electronic Proceedings in Theoretical Computer Science*, 2010.

Appendix

Language Syntax & Substitution Relation

$$\begin{array}{llll}
\textbf{Value} & v & ::= & f \mid \mathtt{c} \mid \mathtt{loc}_\iota \\
\textbf{Expression} & e & ::= & x \mid \mathtt{c} \mid f \mid (e\ e)^\xi \mid (e)[r] \mid e := e \\
 & & \mid & \mathtt{deref}\, e \mid \mathtt{let}\, \rho, x = \mathtt{ref}\, e\, \mathtt{in}\, e \\
 & & \mid & \mathtt{share}\, e \mid \mathtt{release}\, e \mid \mathtt{lock}_\gamma\, e \\
 & & \mid & \mathtt{unlock}\, e \mid () \mid \mathtt{pop}_\gamma\, e \mid \mathtt{loc}_\iota \\
\textbf{Function} & f & ::= & \lambda x.e\, \mathsf{as}\, \tau \xrightarrow{\gamma} \tau \mid \Lambda\rho.f \\
\textbf{Type} & \tau & ::= & b \mid \langle\rangle \mid \tau \xrightarrow{\gamma} \tau \mid \forall\rho.\tau \mid \mathtt{ref}(\tau, r) \\
\textbf{Location} & r & ::= & \rho \mid \iota@n \\
\textbf{Calling mode} & \xi & ::= & \mathsf{seq}(\gamma) \mid \mathsf{par} \\
\textbf{Capability} & \kappa & ::= & n,n \mid \overline{n,n} \\
\textbf{Effect} & \gamma & ::= & \emptyset \mid \gamma, r^\kappa
\end{array}$$

$$\begin{array}{llll}
x_1[v/x] & = & v & x_1 \equiv x \\
 & \mid & x_1 & \textit{otherwise} \\
e[v/x] & = & x_1[v/x] \mid \mathtt{c} \mid \mathtt{loc}_\iota \mid \mathtt{pop}_\gamma\, e[v/x] & \\
 & \mid & \mathtt{share}\, e[v/x] \mid \mathtt{release}\, e[v/x] \mid \mathtt{lock}_{\gamma_1}\, e[v/x] & \\
 & \mid & \mathtt{unlock}\, e[v/x] \mid \mathtt{deref}\, e_1[v/x] & \\
 & \mid & e_1[v/x] := e_2[v/x] \mid f[v/x] \mid (e_1[v/x]\ e_2[v/x])^\xi & \\
 & \mid & (e_1[v/x])[r] \mid \mathtt{let}\, \rho, y = \mathtt{ref}\, e_1[v/x]\, \mathtt{in}\, e_2[v/x] & y \not\equiv x \\
f[v/x] & = & \lambda y.e[x/v]\, \mathsf{as}\, \tau \xrightarrow{\gamma_1} \tau \mid \Lambda\rho.f[x/v] & \\
r_1[r/\rho] & = & r & r_1 \equiv \rho \\
 & \mid & r_1 & \textit{otherwise} \\
f[r/\rho] & = & \lambda x.e[r/\rho]\, \mathsf{as}\, \tau_1[r/\rho] \xrightarrow{\gamma_1[r/\rho]} \tau_2[r/\rho] & \\
 & \mid & \Lambda\rho'.f[r/\rho] & \\
e[r/\rho] & = & x \mid \mathtt{c} \mid \mathtt{pop}_{\gamma[r/\rho]}\, e[r/\rho] \mid \mathtt{share}\, e[r/\rho] & \\
 & \mid & \mathtt{release}\, e[r/\rho] \mid \mathtt{lock}_{\gamma_1[r/\rho]}\, e[r/\rho] & \\
 & \mid & \mathtt{unlock}\, e[r/\rho] \mid (e_1[r/\rho]\ e_2[r/\rho])^{\xi[r/\rho]} & \\
 & \mid & \mathtt{deref}\, e_1[r/\rho] \mid e_1[r/\rho] := e_2[r/\rho] \mid \mathtt{loc}_\iota & \\
 & \mid & (f)[r/\rho] \mid (e_1[r/\rho])[r_1[r/\rho]] & \\
 & \mid & \mathtt{let}\, \rho', x = \mathtt{ref}\, e_1[r/\rho]\, \mathtt{in}\, e_2[r/\rho] & \\
\tau[r/\rho] & = & b \mid \langle\rangle \mid \mathtt{ref}(\tau[r/\rho], r[r/\rho]) & \\
 & \mid & \tau_1[r/\rho] \xrightarrow{\gamma[r/\rho]} \tau_2[r/\rho] & \\
 & \mid & \forall\rho'.\tau[r/\rho] & \\
\xi[r/\rho] & = & \mathsf{seq}(\gamma[r/\rho]) \mid \mathsf{par} & \\
\gamma[r/\rho] & = & \emptyset \mid \gamma_1[r/\rho], r_1[r/\rho]^\kappa &
\end{array}$$

Operational Semantics

Syntax & Evaluation Context

$$\begin{array}{llll}
\textbf{Dynamic Counts} & \theta & ::= & \emptyset \mid \theta, \iota \mapsto n_1, n_2 \\
\textbf{Store} & S & ::= & \emptyset \mid S, \iota \mapsto v \\
\textbf{Threads} & T & ::= & \emptyset \mid T, n : \theta; e \\
\textbf{Configuration} & C & ::= & S; T \\
\textbf{Locations} & \epsilon & ::= & \emptyset \mid \epsilon, \iota
\end{array}$$

$$\begin{array}{llll}
 & E & ::= & \square \mid E[F] \\
 & F & ::= & (\square\ e)^\xi \mid (v\ \square)^\xi \mid (\square)[r] \mid \mathtt{let}\, \rho, x = \mathtt{ref}\, \square\, \mathtt{in}\, e \\
 & & \mid & \mathtt{deref}\, \square \mid \square := e \mid v := \square \mid \mathtt{share}\, \square \mid \mathtt{release}\, \square \\
 & & \mid & \mathtt{lock}_{\gamma_1}\, \square \mid \mathtt{unlock}\, \square \mid \mathtt{pop}_\gamma\, \square \\
\textbf{Redex} & u & ::= & (v'\ v)^\xi \mid (f)[r] \mid \mathtt{lock}_{\gamma_1}\, \mathtt{loc}_\iota \mid \mathtt{unlock}\, \mathtt{loc}_\iota \mid \mathtt{share}\, \mathtt{loc}_\iota \\
 & & \mid & \mathtt{release}\, \mathtt{loc}_\iota \mid \mathtt{deref}\, \mathtt{loc}_\iota \mid \mathtt{loc}_\iota := v \mid \mathtt{let}\, \rho, x = \mathtt{ref}\, v\, \mathtt{in}\, e_2 \\
 & & \mid & \mathtt{pop}_\gamma\, v
\end{array}$$

Helper Rules & Predicates

$\mathsf{locked}(T) = \{\iota \mid \theta(\iota) \geq (1,1) \wedge (n : \theta; e) \in T\}$

$\mathsf{locked}(\gamma) = \{\iota \mid \gamma = (\gamma_1, \iota@{n_1}^{\kappa'}) :: (\gamma_2, \iota@{n_2}^{\kappa}) :: \gamma_3 \wedge (\forall n_3.(\iota@n_3)^{\kappa_1} \in \gamma_2 \Rightarrow \kappa - \kappa_1 = (n_1, 0)) \wedge \kappa - \kappa' = (n_1, -1)\}$

$$\frac{\gamma_b \subseteq_{max} \gamma_a \qquad \gamma_b = \theta_1 + \theta_2}{\gamma_a = \theta_1 \oplus \theta_2}\;\textit{(B0)} \qquad \frac{\begin{array}{c}\gamma \vdash \theta = \theta_1 + \theta_2 \qquad n_5 = n_1 + n_3 \\ n_6 = n_2 + n_4 \qquad \kappa_1 = (n_3, n_4)\end{array}}{\gamma, (\iota@n_5)^{\kappa_1} \vdash \theta, \iota \mapsto n_5, n_6 = \theta_1, \iota \mapsto n_1, n_2 + \theta_2, \iota \mapsto n_3, n_4}\;\textit{(A1)} \qquad \frac{}{\emptyset \vdash \theta = \theta + \emptyset}\;\textit{(A2)}$$

$$\frac{}{\iota; 0; \gamma \vdash \emptyset; 0}\;\textit{(W1)} \qquad \frac{\begin{array}{c} n_1 > 0 \qquad \gamma = \gamma_2, (\iota@n_2)^{\kappa} \qquad \epsilon = \mathsf{locked}(\gamma_2) \\ \forall (\iota@n_2)^{\kappa_1} \in \gamma_2 . \kappa - \kappa_1 = (n_1, 0)\end{array}}{\iota; n_1; \gamma \vdash \epsilon; n_1}\;\textit{(W2)}$$

$$\frac{\begin{array}{c} n_1 > 0 \qquad r = \iota@n_4 \qquad \gamma = (\gamma_1, r^{\kappa'}) :: (\gamma_2, r^{\kappa}) \qquad \epsilon = \mathsf{locked}(\gamma_2) \qquad n_2 > 0 \\ \forall r^{\kappa_1} \in \gamma_2 . \kappa - \kappa_1 = (n_1, 0) \qquad \kappa - \kappa' = (0, n_2) \qquad \iota; n_1 - n_2; \gamma_1, r^{\kappa_1} \vdash \epsilon'; n_3 \end{array}}{\iota; n_1; \gamma \vdash \epsilon \cup \epsilon'; n_3}\;\textit{(W3)}$$

$$\frac{}{E; \iota; 0 \vdash \emptyset}\;\textit{(L0)} \qquad \frac{n_1 > 0 \qquad E; \iota; n_2 \vdash \epsilon' \qquad \iota; n_1; \gamma \vdash \epsilon; n_2}{E[\mathsf{pop}_\gamma\, \square]; \iota; n_1 \vdash \epsilon \cup \epsilon'}\;\textit{(L2)} \qquad \frac{F \neq \mathsf{pop}_\gamma\, \square \qquad E; \iota; n_1 \vdash \epsilon \qquad n_1 > 0}{E[F]; \iota; n_1 \vdash \epsilon}\;\textit{(L3)}$$

Operational Rules

$$\frac{v' \equiv \lambda x. e_1 \;\mathtt{as}\; \tau_1 \xrightarrow{\gamma_a} \tau_2 \qquad \text{fresh } n' \qquad \gamma_a \vdash \theta = \theta_1 \oplus \theta_2}{S; T, n{:}\theta; E[(v' \;\; v)^{\mathsf{par}}] \rightsquigarrow S; T, n{:}\theta_1; E[()], n'{:}\theta_2; \square[(v' \;\; v)^{\mathsf{seq}(\emptyset)}]}\;\textit{(E-SN)}$$

$$\frac{\forall (\iota \mapsto n_1, n_2) \in \theta . n_1 = n_2 = 0}{S; T, n{:}\theta; () \rightsquigarrow S; T}\;\textit{(E-T)} \qquad \frac{v' \equiv \lambda x. e_1 \;\mathtt{as}\; \tau_1 \xrightarrow{\gamma_a} \tau_2}{S; T, n{:}\theta; E[(v' \;\; v)^{\mathsf{seq}(\gamma_b)}] \rightsquigarrow S; T, n{:}\theta; E[\mathsf{pop}_{\gamma_b}\, e_1[v/x]]}\;\textit{(E-A)}$$

$$\frac{\theta(\iota) \geq (1,1) \qquad \iota \notin \mathsf{locked}(T)}{S; T, n{:}\theta; E[\mathtt{loc}_\iota := v] \rightsquigarrow S[\iota \mapsto v]; T, n{:}\theta; E[()]}\;\textit{(E-AS)} \qquad \frac{\theta(\iota) \geq (1,1) \qquad \iota \notin \mathsf{locked}(T)}{S; T, n{:}\theta; E[\mathtt{deref}\;\mathtt{loc}_\iota] \rightsquigarrow S; T, n{:}\theta; E[S(\iota)]}\;\textit{(E-D)}$$

$$\frac{\text{fresh } \iota@n_1 \qquad S' = S, \iota \mapsto v \qquad \theta' = \theta, \iota \mapsto 1, 1}{S; T, n{:}\theta; E[\mathtt{let}\; \rho, x = \mathtt{ref}\; v \;\mathtt{in}\; e_2] \rightsquigarrow S'; T, n{:}\theta'; E[e_2[\iota@n_1/\rho][\mathtt{loc}_\iota/x]]}\;\textit{(E-NG)}$$

$$\frac{\theta(\iota) \geq (1,0) \qquad \theta' = \theta[\iota \mapsto \theta(\iota) + (1,0)]}{S; T, n{:}\theta; E[\mathtt{share}\;\mathtt{loc}_\iota] \rightsquigarrow S; T, n{:}\theta'; E[()]}\;\textit{(E-SH)} \qquad \frac{\begin{array}{c}\theta(\iota) \geq (1,0) \qquad \theta(\iota) = (n_1, n_2) \\ n_1 = 1 \Rightarrow n_2 = 0 \qquad \theta' = \theta[\iota \mapsto n_1 - 1, n_2]\end{array}}{S; T, n{:}\theta; E[\mathtt{release}\;\mathtt{loc}_\iota] \rightsquigarrow S; T, n{:}\theta'; E[()]}\;\textit{(E-RL)}$$

$$\frac{\begin{array}{c}\theta' = \theta[\iota \mapsto (n_1, n_2 + 1)] \qquad E[\mathsf{pop}_{\gamma_1}\, \square]; \iota; 1 \vdash \epsilon \\ \theta(\iota) = (n_1, n_2) \qquad n_1 \geq 1 \qquad n_2 = 0 \Rightarrow \mathsf{locked}(T) \cap (\epsilon \cup \{\iota\}) = \emptyset\end{array}}{S; T, n{:}\theta; E[\mathtt{lock}_{\gamma_1}\;\mathtt{loc}_\iota] \rightsquigarrow S; T, n{:}\theta'; E[()]}\;\textit{(E-LK)} \qquad \frac{\theta(\iota) \geq (1,1) \qquad \theta' = \theta[\iota \mapsto \theta(\iota) - (0,1)]}{S; T, n{:}\theta; E[\mathtt{unlock}\;\mathtt{loc}_\iota] \rightsquigarrow S; T, n{:}\theta'; E[()]}\;\textit{(E-UL)}$$

$$\frac{\text{fresh } n_2}{S; T, n{:}\theta; E[(\Lambda\rho. f)[\iota@n_1]] \rightsquigarrow S; T, n{:}\theta; E[f[\iota@n_2/\rho]]}\;\textit{(E-RP)} \qquad \frac{}{S; T, n{:}\theta; E[\mathsf{pop}_\gamma\, v] \rightsquigarrow S; T, n{:}\theta; E[v]}\;\textit{(E-PP)}$$

Static Semantics

Syntax

Type variable list	Δ	::=	$\emptyset \mid \Delta, \rho$
Memory List	M	::=	$\emptyset \mid M, \iota \mapsto \tau$
Variable list	Γ	::=	$\emptyset \mid \Gamma, x : \tau$

Typing Context Substitution Relation

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

Typing Context Well-formedness Judgements

Constraint Well-formedness

$$\frac{}{M; \Delta \vdash \emptyset} \qquad \frac{M; \Delta \vdash r \qquad M; \Delta \vdash \gamma_1}{M; \Delta \vdash \gamma_1, r^{\kappa}}$$

Region Well-formedness

$$\frac{r \in \Delta \cup \mathsf{dom}(M)}{M; \Delta \vdash r} \qquad \frac{M; \Delta \vdash \iota}{M; \Delta \vdash \iota@n}$$

Program Typing Context Well-formedness

$$\frac{\begin{array}{c}\vdash M \qquad M; \Delta \vdash \Gamma \qquad M; \Delta \vdash \gamma_1 \qquad M; \Delta \vdash \gamma_2 \\ \gamma_1 \lhd \gamma_2 \qquad \mathsf{seq}(\emptyset) \vdash \gamma_2\end{array}}{\vdash M; \Delta; \Gamma; \gamma_1; \gamma_2}$$

Type Well-formedness

$$\frac{}{M;\Delta \vdash b} \qquad \frac{M;\Delta,\rho \vdash \tau}{M;\Delta \vdash \forall\rho.\tau} \qquad \frac{M;\Delta \vdash \tau \quad M;\Delta \vdash r}{M;\Delta \vdash \texttt{ref}(\tau,r)} \qquad \frac{\begin{array}{c}\gamma_2 \subseteq_{min} \gamma_1 \\ M;\Delta \vdash \tau_1 \quad M;\Delta \vdash \gamma_1 \quad M;\Delta \vdash \tau_2\end{array}}{M;\Delta \vdash \tau_1 \xrightarrow{\gamma_1} \tau_2} \qquad \frac{}{M;\Delta \vdash \langle\rangle}$$

Γ Well-formedness **M Well-formedness**

$$\frac{}{M;\Delta \vdash \emptyset} \qquad \frac{\begin{array}{c}M;\Delta \vdash \tau_1 \quad x \notin \mathsf{dom}(\Gamma_1) \\ M;\Delta \vdash \Gamma_1\end{array}}{M;\Delta \vdash \Gamma_1, x : \tau_1} \qquad \frac{}{\vdash \emptyset} \qquad \frac{\vdash M \quad \iota \notin \mathsf{dom}(M) \quad M;\emptyset \vdash \tau}{\vdash M, \iota \mapsto \tau}$$

Predicates

$$\mathsf{is\_pure}(\kappa) = \exists n_1.\ \exists n_2.\ \kappa = n_1, n_2 \qquad \mathsf{set}(\gamma) = \forall r^\kappa, \gamma_1, \gamma_2. \gamma = (\gamma_1, r^\kappa) :: \gamma_2 \Rightarrow r \notin \mathsf{dom}(\gamma_1) \cup \mathsf{dom}(\gamma_2)$$

$$\frac{\begin{array}{c}\gamma_1 \subseteq_{min} \gamma \qquad \gamma_2 \subseteq_{max} \gamma \\ \xi = \mathsf{par} \Rightarrow \forall r^\kappa.(r^\kappa \in \gamma_1 \Rightarrow \kappa = (0,0)) \wedge (r^\kappa \in \gamma_2 \wedge \kappa = \overline{n_1, n_2} \Rightarrow n_2 = 0) \\ \forall r^{n_1,n_2}\, \gamma_a \gamma_b. \gamma = (\gamma_a, r^{n_1,n_2}) :: \gamma_b \Rightarrow \neg\exists r'. r' \neq r \wedge r^\kappa \in (\gamma_a \cup \gamma_b) \wedge r \simeq r'\end{array}}{\xi \vdash \gamma}\ \textit{(OK)} \qquad \frac{\gamma_1 \subseteq_{max} \gamma \quad \gamma_1 = (\gamma_2, r^\kappa) :: \gamma_3}{\gamma(r) = \kappa}$$

$$\frac{}{\emptyset \subseteq_{max} \emptyset} \qquad \frac{\gamma_1 \subseteq_{max} (\gamma_2 \setminus_u r)}{\gamma_1, r^\kappa \subseteq_{max} \gamma_2, r^\kappa} \qquad \frac{\gamma_2 = \gamma_1 :: \gamma_3 \quad \mathsf{dom}(\gamma_2) = \mathsf{dom}(\gamma_1) \quad \mathsf{set}(\gamma_1)}{\gamma_1 \subseteq_{min} \gamma_2} \qquad \frac{\gamma_1 = \gamma_2 :: \gamma_3}{\gamma_2 \lhd \gamma_1}$$

$$\frac{r' \simeq r \quad \gamma' = \gamma \setminus r'}{\gamma' = \gamma, r^\kappa \setminus r'}\ \textit{(M0)} \qquad \frac{\neg(r' \simeq r) \quad \gamma' = \gamma \setminus r'}{\gamma', r^\kappa = \gamma, r^\kappa \setminus r'}\ \textit{(M1)} \qquad \frac{}{\emptyset = \emptyset \setminus r}\ \textit{(M2)}$$

$$\frac{\gamma' = \gamma \setminus r}{\gamma' = \gamma, r^\kappa \setminus_u r}\ \textit{(N0)} \qquad \frac{r' \neq r \quad \gamma' = \gamma \setminus_u r'}{\gamma', r^\kappa = \gamma, r^\kappa \setminus_u r'}\ \textit{(N1)} \qquad \frac{}{\emptyset = \emptyset \setminus_u r}\ \textit{(N2)} \qquad \frac{\begin{array}{c}\gamma_2 = (\gamma_3, r^\kappa) :: \gamma_4 \\ \gamma_1 \cong (\gamma_3 :: \gamma_4)\end{array}}{\gamma_1, r^\kappa \cong \gamma_2}\ \textit{(G0)} \qquad \frac{}{\emptyset \cong \emptyset}\ \textit{(G1)}$$

Capability Manipulation Rules

$$\frac{\begin{array}{c}\gamma_{1b} \subseteq_{min} \gamma_1 \quad \gamma_{1a} \subseteq_{max} \gamma_1 \quad \gamma_a \subseteq_{max} \gamma \\ \gamma_c = \gamma_{1b} + \gamma_{r_1} \quad \gamma_a \cong \gamma_c \quad \gamma_b = \gamma_{1a} + \gamma_{r_1}\end{array}}{\gamma :: \gamma_b = \gamma \oplus \gamma_1}\ \textit{(D0)} \qquad \frac{}{\gamma = \emptyset + \gamma}\ \textit{(ES1)} \qquad \frac{\begin{array}{c}\gamma, r^{\kappa_2} = \gamma_1 + \gamma_2 \quad \kappa = \kappa_2 + \kappa_1 \\ \mathsf{is\_pure}(\kappa_1) \Rightarrow \kappa_2 = (0,0) \quad r \simeq r'\end{array}}{\gamma, r^\kappa = \gamma_1, r'^{\kappa_1} + \gamma_2}\ \textit{(ES2)}$$

Type Equivalence

$$\frac{}{r \simeq r}\ \textit{(S0)} \qquad \frac{r' \simeq r}{r \simeq r'}\ \textit{(S1)} \qquad \frac{r \simeq r'}{r \simeq r' @ n_2}\ \textit{(S2)} \qquad \frac{}{\emptyset \simeq \emptyset}\ \textit{(S3)} \qquad \frac{r_1 \simeq r_2 \quad \gamma_1 \simeq \gamma_2}{\gamma_1, {r_1}^\kappa \simeq \gamma_2, {r_2}^\kappa}\ \textit{(S4)}$$

$$\frac{}{\tau \simeq \tau}\ \textit{(S6)} \qquad \frac{\tau_3 \simeq \tau_4 \quad r_1 \simeq r_2}{\texttt{ref}(\tau_3, r_1) \simeq \texttt{ref}(\tau_4, r_2)}\ \textit{(S7)} \qquad \frac{\begin{array}{c}\text{fresh } \rho_1 @ n \\ \tau_1[\rho_1 @ n/\rho] \simeq \tau_2[\rho_1 @ n/\rho']\end{array}}{\forall\rho.\tau_1 \simeq \forall\rho'.\tau_2}\ \textit{(S8)} \qquad \frac{\begin{array}{c}\tau_1 \simeq \tau_3 \quad \tau_2 \simeq \tau_4 \\ \gamma_1 \simeq \gamma_3 \quad \gamma_2 \simeq \gamma_4\end{array}}{\tau_1 \xrightarrow{\gamma_1} \tau_2 \simeq \tau_3 \xrightarrow{\gamma_3} \tau_4}\ \textit{(S9)}$$

Typing Rules

$$\frac{\begin{array}{c}\vdash M;\Delta;\Gamma;\gamma;\gamma \\ (x : \tau') \in \Gamma \quad \tau \simeq \tau'\end{array}}{M;\Delta;\Gamma \vdash x : \tau \,\&\, (\gamma;\gamma)}\ \textit{(T-V)} \qquad \frac{\vdash M;\Delta;\Gamma;\gamma;\gamma}{M;\Delta;\Gamma \vdash \mathsf{c} : b \,\&\, (\gamma;\gamma)}\ \textit{(T-I)} \qquad \frac{M;\Delta,\rho;\Gamma \vdash f : \tau \,\&\, (\gamma;\gamma)}{M;\Delta;\Gamma \vdash \Lambda\rho.f : \forall\rho.\tau \,\&\, (\gamma;\gamma)}\ \textit{(T-RF)}$$

$$\frac{\vdash_= M;\Delta;\Gamma;\gamma;\gamma}{M;\Delta;\Gamma \vdash () : \langle\rangle \,\&\, (\gamma;\gamma)}\ \textit{(T-U)} \qquad \frac{M;\Delta;\Gamma \vdash e_1 : \forall\rho.\tau \,\&\, (\gamma;\gamma')}{M;\Delta;\Gamma \vdash (e_1)[r] : \tau' \,\&\, (\gamma;\gamma')}\ \textit{(T-RP)} \qquad \frac{\begin{array}{c}M;\Delta;\Gamma \vdash e : \tau \,\&\, (\gamma_1;\gamma_2) \quad \gamma_2 \lhd \gamma_3 \\ M;\Delta \vdash \gamma_3 \quad \vdash M;\Delta;\Gamma;\gamma;\gamma' \quad \gamma' = \gamma \oplus \gamma_3\end{array}}{M;\Delta;\Gamma \vdash \mathsf{pop}_\gamma\ e : \tau \,\&\, (\gamma;\gamma')}\ \textit{(T-PP)}$$

$$\frac{\begin{array}{c}\vdash M;\Delta;\Gamma;\gamma;\gamma \\ (\iota \mapsto \tau') \in M \qquad \tau' \simeq \tau\end{array}}{M;\Delta;\Gamma \vdash \mathtt{loc}_\iota : \mathtt{ref}(\tau,\iota)\,\&\,(\gamma;\gamma)}\ \textit{(T-L)} \qquad \frac{\begin{array}{c}\vdash_= M;\Delta;\Gamma;\gamma;\gamma \qquad \tau' \equiv \tau_1 \xrightarrow{\gamma_b} \tau_2 \qquad M;\Delta \vdash \tau' \qquad \tau \simeq \tau' \\ \mathsf{seq}(\emptyset) \vdash \gamma_b \Rightarrow M;\Delta;\Gamma, x:\tau_1 \vdash e_1 : \tau_2\,\&\,(\gamma_a;\gamma_b) \wedge \gamma_a \subseteq_{min} \gamma_b\end{array}}{M;\Delta;\Gamma \vdash \lambda x.e_1\ \mathsf{as}\ \tau' : \tau\,\&\,(\gamma;\gamma)}\ \textit{(T-F)}$$

$$\frac{\begin{array}{c}M;\Delta;\Gamma \vdash e : \mathtt{ref}(\tau,r)\,\&\,(\gamma, r^{\kappa-(1,0)};\gamma') \\ \kappa \geq (2,0) \qquad \gamma(r) = \kappa\end{array}}{M;\Delta;\Gamma \vdash \mathsf{share}\, e : \langle\rangle\,\&\,(\gamma;\gamma')}\ \textit{(T-SH)} \qquad \frac{\begin{array}{c}M;\Delta;\Gamma \vdash e : \mathtt{ref}(\tau,r)\,\&\,(\gamma, r^{\kappa+(1,0)};\gamma') \\ \kappa = (n_1,n_2) \qquad n_1 = 0 \Rightarrow n_2 = 0 \qquad \gamma(r) = \kappa\end{array}}{M;\Delta;\Gamma \vdash \mathtt{release}\, e : \langle\rangle\,\&\,(\gamma;\gamma')}\ \textit{(T-RL)}$$

$$\frac{\begin{array}{c}M;\Delta;\Gamma \vdash e : \mathtt{ref}(\tau,r)\,\&\,(\gamma, r^{\kappa-(0,1)};\gamma') \\ \kappa \geq (1,1) \qquad \gamma(r) = \kappa\end{array}}{M;\Delta;\Gamma \vdash \mathtt{lock}_\gamma\, e : \langle\rangle\,\&\,(\gamma;\gamma')}\ \textit{(T-LK)} \qquad \frac{\begin{array}{c}M;\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_a} \tau_2\,\&\,(\gamma_3;\gamma') \qquad \xi \vdash \gamma_a \qquad \gamma_2 = \gamma \oplus \gamma_a \\ M;\Delta;\Gamma \vdash e_2 : \tau_1\,\&\,(\gamma_2;\gamma_3) \qquad \xi = \mathsf{seq}(\gamma) \vee (\xi = \mathsf{par} \wedge \tau_2 = \langle\rangle)\end{array}}{M;\Delta;\Gamma \vdash (e_1\ e_2)^\xi : \tau_2\,\&\,(\gamma;\gamma')}\ \textit{(T-A)}$$

$$\frac{\begin{array}{c}M;\Delta;\Gamma \vdash e : \mathtt{ref}(\tau,r)\,\&\,(\gamma, r^{\kappa+(0,1)};\gamma') \\ \kappa \geq (1,0) \qquad \gamma(r) = \kappa\end{array}}{M;\Delta;\Gamma \vdash \mathtt{unlock}\, e : \langle\rangle\,\&\,(\gamma;\gamma')}\ \textit{(T-UL)} \qquad \frac{\begin{array}{c}M;\Delta;\Gamma \vdash e_1 : \tau_1\,\&\,(\gamma_2 \setminus \rho;\gamma') \qquad \gamma_1 = \gamma_2, \rho^{1,1} \\ M;\Delta \vdash \tau \qquad M;\Delta,\rho;\Gamma, x : \mathtt{ref}(\tau_1,\rho) \vdash e_2 : \tau\,\&\,(\gamma, \rho^{0,0};\gamma_1)\end{array}}{M;\Delta;\Gamma \vdash \mathtt{let}\ \rho, x = \mathtt{ref}\ e_1\ \mathtt{in}\ e_2 : \tau\,\&\,(\gamma;\gamma')}\ \textit{(T-NG)}$$

$$\frac{\begin{array}{c}M;\Delta;\Gamma \vdash e_1 : \mathtt{ref}(\tau,r)\,\&\,(\gamma_1;\gamma') \\ M;\Delta;\Gamma \vdash e_2 : \tau\,\&\,(\gamma;\gamma_1) \qquad \gamma(r) \geq (1,1)\end{array}}{M;\Delta;\Gamma \vdash e_1 := e_2 : \langle\rangle\,\&\,(\gamma;\gamma')}\ \textit{(T-AS)} \qquad \frac{\begin{array}{c}\gamma(r) \geq (1,1) \\ M;\Delta;\Gamma \vdash e_1 : \mathtt{ref}(\tau,r)\,\&\,(\gamma;\gamma')\end{array}}{M;\Delta;\Gamma \vdash \mathtt{deref}\ e_1 : \tau\,\&\,(\gamma;\gamma')}\ \textit{(T-D)}$$

Type Safety

Evaluation Context Typing

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

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

$$\frac{\begin{array}{c}\vdash M;\Delta;\Gamma;\gamma_1;\gamma_4 \qquad \gamma_2=\gamma_1\oplus\gamma_a \qquad \gamma_3\lhd\gamma_4\\ M;\Delta;\Gamma\vdash e_2:\tau_1\,\&\,(\gamma_2;\gamma_3) \qquad M;\Delta\vdash\tau_1\xrightarrow{\gamma_a}\tau_2\\ \xi\vdash\gamma_a \qquad \xi=\mathsf{seq}(\gamma_1)\vee(\xi=\mathsf{par}\wedge\tau_2=\langle\rangle)\end{array}}{M;\Delta;\Gamma\vdash(\Box\;e_2)^{\xi}:(\tau_1\xrightarrow{\gamma_a}\tau_2)\xrightarrow{\gamma_3;\gamma_4}\tau_2\,\&\,(\gamma_1;\gamma_4)}\;\textit{(F1)}$$

$$\frac{\begin{array}{c}\vdash M;\Delta;\Gamma;\gamma_1;\gamma_3 \qquad \gamma_2=\gamma_1\oplus\gamma_a \qquad \gamma_2\lhd\gamma_3\\ M;\Delta;\Gamma\vdash v_1:\tau_1\xrightarrow{\gamma_a}\tau_2\,\&\,(\gamma_3;\gamma_3) \qquad M;\Delta\vdash\tau_1\xrightarrow{\gamma_3}\tau_2\\ \xi\vdash\gamma_a \qquad \xi=\mathsf{seq}(\gamma_1)\vee(\xi=\mathsf{par}\wedge\tau_2=\langle\rangle)\end{array}}{M;\Delta;\Gamma\vdash(v_1\;\Box)^{\xi}:\tau_1\xrightarrow{\gamma_2;\gamma_3}\tau_2\,\&\,(\gamma_1;\gamma_3)}\;\textit{(F2)}$$

$$\frac{\begin{array}{c}\vdash M;\Delta;\Gamma;\gamma;\gamma' \qquad \vdash M;\Delta;\Gamma;\gamma_1;\gamma_3\\ \gamma'=\gamma\oplus\gamma_3 \qquad M;\Delta\vdash\tau \qquad \gamma_2\lhd\gamma_3\end{array}}{M;\Delta;\Gamma\vdash\mathsf{pop}_{\gamma}\,\Box:\tau\xrightarrow{\gamma_1;\gamma_2}\tau\,\&\,(\gamma;\gamma')}\;\textit{(F3)}$$

$$\frac{\begin{array}{c}\gamma_3=\gamma_2\setminus\rho \qquad \gamma_3\lhd\gamma' \qquad \gamma_1=\gamma_2,\rho^{1,1} \qquad M;\Delta\vdash\tau_1 \qquad M;\Delta\vdash\tau\\ \vdash M;\Delta;\Gamma;\gamma;\gamma' \qquad M;\Delta,\rho;\Gamma,x:\mathtt{ref}(\tau_1,\rho)\vdash e_2:\tau\,\&\,(\gamma,\rho^{0,0};\gamma_1)\end{array}}{M;\Delta;\Gamma\vdash\mathtt{let}\,\rho,x=\mathtt{ref}\,\Box\,\mathtt{in}\,e_2:\tau_1\xrightarrow{\gamma_3;\gamma'}\tau\,\&\,(\gamma;\gamma')}\;\textit{(F4)}$$

$$\frac{\begin{array}{c}\vdash M;\Delta;\Gamma;\gamma;\gamma' \qquad M;\Delta\vdash\mathtt{ref}(\tau,r)\\ M;\Delta;\Gamma\vdash e_2:\tau\,\&\,(\gamma;\gamma_1) \qquad \gamma(r)\geq(1,1)\end{array}}{M;\Delta;\Gamma\vdash\Box:=e_2:\mathtt{ref}(\tau,r)\xrightarrow{\gamma_1;\gamma'}\langle\rangle\,\&\,(\gamma;\gamma')}\;\textit{(F5)}$$

$$\frac{\begin{array}{c}\vdash M;\Delta;\Gamma;\gamma;\gamma' \qquad \gamma(r)\geq(1,1)\\ M;\Delta;\Gamma\vdash\mathtt{loc}_{\iota}:\mathtt{ref}(\tau,r)\,\&\,(\gamma';\gamma')\end{array}}{M;\Delta;\Gamma\vdash\mathtt{loc}_{\iota}:=\Box:\tau\xrightarrow{\gamma;\gamma'}\langle\rangle\,\&\,(\gamma;\gamma')}\;\textit{(F6)}$$

$$\frac{\vdash M;\Delta;\Gamma;\gamma;\gamma' \qquad \gamma(r)\geq(1,1) \qquad M;\Delta\vdash\mathtt{ref}(\tau,r)}{M;\Delta;\Gamma\vdash\mathtt{deref}\,\Box:\mathtt{ref}(\tau,r)\xrightarrow{\gamma;\gamma'}\tau\,\&\,(\gamma;\gamma')}\;\textit{(F7)}$$

$$\frac{\begin{array}{c}\vdash M;\Delta;\Gamma;\gamma;\gamma' \qquad M;\Delta\vdash\mathtt{ref}(\tau,r)\\ \kappa\geq(2,0) \qquad \gamma(r)=\kappa \qquad \gamma_1=\gamma,r^{\kappa-(1,0)} \qquad \gamma_1\lhd\gamma'\end{array}}{M;\Delta;\Gamma\vdash\mathtt{share}\Box:\mathtt{ref}(\tau,r)\xrightarrow{\gamma_1;\gamma'}\langle\rangle\,\&\,(\gamma;\gamma')}\;\textit{(F8)}$$

$$\frac{\begin{array}{c}\vdash M;\Delta;\Gamma;\gamma;\gamma' \qquad M;\Delta\vdash\mathtt{ref}(\tau,r) \qquad n_1=0\Rightarrow n_2=0\\ \kappa=(n_1,n_2) \qquad \gamma(r)=\kappa \qquad \gamma_1=\gamma,r^{\kappa+(1,0)} \qquad \gamma_1\lhd\gamma'\end{array}}{M;\Delta;\Gamma\vdash\mathtt{release}\Box:\mathtt{ref}(\tau,r)\xrightarrow{\gamma_1;\gamma'}\langle\rangle\,\&\,(\gamma;\gamma')}\;\textit{(F9)}$$

$$\frac{\begin{array}{c}\vdash M;\Delta;\Gamma;\gamma;\gamma' \qquad M;\Delta\vdash\mathtt{ref}(\tau,r)\\ \kappa\geq(1,0) \qquad \gamma(r)=\kappa \qquad \gamma_1=\gamma,r^{\kappa+(0,1)} \qquad \gamma_1\lhd\gamma'\end{array}}{M;\Delta;\Gamma\vdash\mathtt{unlock}\Box:\mathtt{ref}(\tau,r)\xrightarrow{\gamma_1;\gamma'}\langle\rangle\,\&\,(\gamma;\gamma')}\;\textit{(F10)}$$

$$\frac{\begin{array}{c}\vdash M;\Delta;\Gamma;\gamma;\gamma' \qquad M;\Delta\vdash\mathtt{ref}(\tau,r)\\ \kappa\geq(1,1) \qquad \gamma(r)=\kappa \qquad \gamma_1=\gamma,r^{\kappa-(0,1)} \qquad \gamma_1\lhd\gamma'\end{array}}{M;\Delta;\Gamma\vdash\mathtt{lock}_{\gamma}\Box:\mathtt{ref}(\tau,r)\xrightarrow{\gamma_1;\gamma'}\langle\rangle\,\&\,(\gamma;\gamma')}\;\textit{(F11)}$$

$$\frac{M;\Delta;\Gamma\vdash e:\tau\,\&\,(\gamma_a;\gamma_b) \qquad M;\Delta;\Gamma\vdash E:\tau\xrightarrow{\gamma_a;\gamma_b}\tau'\,\&\,(\gamma_1;\gamma_2)}{M;\Delta;\Gamma\vdash E[e]:\tau'\,\&\,(\gamma_1;\gamma_2)}\;\textit{(EA0)}$$

$$\frac{\begin{array}{c}M;\Delta;\Gamma\vdash u:\tau\,\&\,(\gamma_a;\gamma_b) \qquad M;\Delta;\Gamma\vdash E:\tau\xrightarrow{\gamma_a;\gamma_b}\langle\rangle\,\&\,(\gamma_1;\gamma_2)\\ \gamma_2'\subseteq_{min}\gamma_2 \qquad \forall r^{\kappa}\in\gamma_2'.\kappa=(0,0) \qquad E;\gamma_c;\gamma_c\vdash\theta\\ \gamma_c=\text{if }u\neq\mathsf{pop}_{\gamma_a}\,v\text{ then }\gamma_b\text{ else }\emptyset\end{array}}{M;\Delta;\Gamma\vdash_t\theta;E[u]:\langle\rangle\,\&\,(\gamma_1;\gamma_2)}\;\textit{(EA1)}$$

$$\frac{\begin{array}{c}M;\Delta;\Gamma\vdash():\langle\rangle\,\&\,(\gamma_1;\gamma_1) \qquad M;\Delta;\Gamma\vdash\Box:\langle\rangle\xrightarrow{\gamma_1;\gamma_1}\langle\rangle\,\&\,(\gamma_1;\gamma_1)\\ \gamma_1'\subseteq_{min}\gamma_1 \qquad \forall r^{\kappa}\in\gamma_1'.\kappa=(0,0) \qquad \Box;\gamma_1';\gamma_1'\vdash\theta\end{array}}{M;\Delta;\Gamma\vdash_t\theta;\Box[()]:\langle\rangle\,\&\,(\gamma_1;\gamma_1)}\;\textit{(EA2)}$$

Dynamic Count Typing

$$\frac{\begin{array}{c}\theta;\gamma\vdash\theta',\iota\mapsto n_1,n_2 \qquad \kappa=\overline{n_3,n_4}\\ (n_1,n_2)\geq(n_3,n_4) \qquad \iota\simeq r\end{array}}{\theta;\gamma,r^{\kappa}\vdash\theta',\iota\mapsto n_1-n_3,n_2-n_4}\;\textit{(B0)}$$

$$\frac{\theta;\gamma\vdash\theta',\iota\mapsto n_1,n_2 \qquad \kappa=n_1,n_2 \qquad \iota\simeq r}{\theta;\gamma,r^{n_1,n_2}\vdash\theta',\iota\mapsto 0,0}\;\textit{(B1)}$$

$$\frac{}{\theta;\emptyset\vdash\theta}\;\textit{(B2)}$$

$$\frac{\begin{array}{c}\gamma''\subseteq_{max}\gamma' \qquad \theta;\gamma''\vdash\theta'\\ \forall(n\mapsto n_1,n_2)\in\theta'.n_1=n_2=0\end{array}}{\Box;\gamma;\gamma'\vdash\theta}\;\textit{(C0)}$$

$$\frac{\begin{array}{c}\gamma_{1b}\subseteq_{min}\gamma_1 \qquad \gamma_a\subseteq_{max}\gamma\\ \gamma_c=\gamma_{1b}+\gamma_r \qquad \gamma_a\cong\gamma_c\\ \theta;\gamma_r\vdash\theta' \qquad E;\gamma;\gamma_2\vdash\theta'\end{array}}{E[\mathsf{pop}_{\gamma}\,\Box];\gamma_1;\gamma_2\vdash\theta}\;\textit{(C1)}$$

$$\frac{F\neq\mathsf{pop}_{\gamma'}\,\Box \qquad E;\gamma;\gamma''\vdash\theta}{E[F];\gamma;\gamma''\vdash\theta}\;\textit{(C2)}$$

Configuration Typing

$\mathsf{mutex}(T) \quad \equiv \forall T_1, n:\theta; E[e]. T = T_1, n:\theta; E[e] \Rightarrow \forall \iota. \theta(\iota) \geq (1,1) \Rightarrow \iota \notin \mathsf{locked}(T_1)$

$\mathsf{deadlocked}(T) \quad \equiv T \supseteq T_1, n_0:\theta_0; E[\mathtt{lock}_{\gamma_0}\ \mathtt{loc}_{\iota_0}], \dots n_k:\theta_k; E_k[\mathtt{lock}_{\gamma_k}\ \mathtt{loc}_{\iota_k}] \wedge k > 0 \Rightarrow \forall m_1 \in [0,k]. m_2 = (m_1+1) \bmod (k+1) \wedge \theta_{m_1}(\iota_{m_2}) \geq (1,1$

$\mathrm{blocked}(T,n) \quad \equiv T = T_1, n_1:\theta; E[\mathtt{lock}_{\gamma_2}\ \mathtt{loc}_{\iota}] \wedge \theta(\iota) \geq (1,0) \wedge E;\iota;1 \vdash \epsilon \wedge \mathsf{locked}(T_1) \cap (\epsilon \cup \{\iota\}) \neq \emptyset$

Store Typing

$$\frac{\mathsf{dom}(M) = \mathsf{dom}(S) \qquad \forall(\iota \mapsto \tau) \in M. M;\emptyset;\emptyset \vdash S(\iota) : \tau \,\&\, (\emptyset;\emptyset)}{M \vdash S}$$

Configuration Typing

$$\frac{M \vdash T \qquad M \vdash S \qquad \mathsf{mutex}(T)}{M \vdash S;T}$$

Thread Typing

$$\frac{}{M;\emptyset \vdash \emptyset} \qquad \frac{M;\emptyset;\emptyset \vdash_t \theta; E[e] : \langle\rangle \,\&\, (\gamma;\gamma') \qquad M \vdash T \qquad n \notin \mathsf{dom}(T)}{M \vdash T, n:\theta; E[e]}$$

Not Stuck

$$\frac{\neg\mathsf{deadlocked}(T) \qquad \forall n:\theta; e \in T (S;T \leadsto_n S';T') \vee \mathsf{blocked}(T,n)}{\vdash S;T}$$

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

Main Theorems/Lemmas

Safety:	$S_0;T_0 \equiv \emptyset;0:\emptyset;e \wedge \emptyset \vdash S_0;T_0 \wedge S_0;T_0 \leadsto^n S';T' \Rightarrow \vdash S';T'$
Preservation:	$M \vdash S;T \wedge S;T \leadsto S';T' \Rightarrow \exists M' \supseteq M.\ M' \vdash S';T'$
Progress:	$\neg\mathsf{deadlocked}(T) \wedge M \vdash S;T \Rightarrow \vdash S;T$

Proof Sketch

Theorem 1 (Type safety)**.** *If the initial configuration $S_0;T_0$ (defined in page 8) is well-typed with $\emptyset \vdash S_0;T_0$ and the operational semantics takes any number of steps $S_0;T_0 \leadsto^n S_n;T_n$, then the resulting configuration $S_n;T_n$ is not stuck.*

Proof. The application of lemma 1 to the assumption implies that $\forall \iota \in [0,n]. \exists M_\iota. M_\iota \vdash S_\iota;T_\iota$. Therefore, $S_n;T_n$ is well-typed for some M_n. The application of lemma 2 to $\forall \iota \in [0,n]. \exists M_\iota. M_\iota \vdash S_\iota;T_\iota$ and $\emptyset;0:\emptyset;e \leadsto^n S_n;T_n$ implies that $\neg\mathsf{deadlocked}(T_n)$. The application of lemma 14 to the latter facts 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 *such that $M_0 \vdash S_0;T_0$ for some M_0. If the operational semantics evaluates $S_0;T_0$ to $S_n;T_n$ in n steps, then $\forall \iota \in [0,n]. \exists M_\iota. M_\iota \vdash S_\iota;T_\iota$*

Proof. Proof by induction on the number of steps n. When no steps are performed (i.e., $n = 0$) the proof is immediate from the assumption. When some steps are performed (i.e., $n > 0$), we have that $S_0;T_0 \leadsto^n S_n;T_n$ or $S_0;T_0 \leadsto^{n-1} S_{n-1};T_{n-1}$ and $S_{n-1};T_{n-1} \leadsto S_n;T_n$. 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 $\forall \iota \in [0,n-1]. \exists M_\iota. M_\iota \vdash S_\iota;T_\iota$. Thus, $M_{n-1} \vdash S_{n-1};T_{n-1}$ holds. The application of lemma 4 to $M_{n-1} \vdash S_{n-1};T_{n-1}$ and $S_{n-1};T_{n-1} \leadsto S_n;T_n$. implies that $M_n \vdash S_n;T_n$. Therefore, $\forall \iota \in [0,n]. \exists M_\iota. M_\iota \vdash S_\iota;T_\iota$. □

Lemma 2 (Deadlock Freedom)**.** *if $\emptyset;\emptyset,0:\emptyset;e \leadsto^n S_n;T_n$ and $\forall \iota \in [0,n]. \exists M_\iota. M_\iota \vdash S_\iota;T_\iota$ then $\neg\mathsf{deadlocked}(T_n)$.*

Proof. Assume that $\mathsf{deadlocked}(T_x)$ holds for some $x \in [0,n]$ and the first deadlock occuring in the program is in T_x (i.e. $\forall \iota. \iota < x \Rightarrow \neg\mathsf{deadlocked}(T_\iota)$). Then, the following hold:

- $T_x = T, n_0:\theta_0; E_0[\mathtt{lock}_{\gamma_0}\ \mathtt{loc}_{\iota_0}], \dots n_k:\theta_k; E_k[\mathtt{lock}_{\gamma_k}\ \mathtt{loc}_{\iota_k}]$: we abbreviate each lock request operation as $(n, \epsilon_{n'}, \iota_{n'})$, where the first element of the tuple is the thread identifier n, the second is the lockset ($\epsilon_{n'}$) inferred by the run-time system (where $\epsilon_{n'}$ is given by $E_n[\mathsf{pop}_{\gamma_{n'}}\ \Box]; \iota_{n'}; 1 \vdash \epsilon_{n'}$) and the third element is a lock identifier $\iota_{n'}$. Here, locks are numbered from 0 to k. We also use the notation $eps(n,n')$, to extract the second element of the tuple $(n, \epsilon_{n'}, \iota_{n'})$ for some $\epsilon_{n'}$. We use the abbreviation $locked(n,n')$ when thread n has locked $\iota_{n'}$. In addition, we define a partial order over this relation: $locked(n,n') < locked(n'',n''')$, which implies that thread n locked $\iota_{n'}$ before thread n'' locked $\iota_{n'''}$.
- $k > 0$ and $\forall m_1 \in [0,k]. m_2 = (m_1+1) \bmod (k+1) \wedge \theta_{m_1}(\iota_{m_2}) \geq (1,1)$: we define function $succ(n) = (n+1) \bmod (k+1)$ and function $pred(n) = (k+n) \bmod (k+1)$.

Let m be the thread that acquires the *first* of the $k+1$ locks that cause the deadlock, namely $\mathsf{succ}(m)$ (given the definition of T_x). Then, the following holds: $locked(m, \mathsf{succ}(m)) < locked(\mathsf{succ}(m), \mathsf{succ}(\mathsf{succ}(m)))$. Notice that, $\mathsf{succ}(m) \notin \mathsf{eps}(\mathsf{succ}(m), \mathsf{succ}(\mathsf{succ}(m)))$ holds. Otherwise, thread $\mathsf{succ}(m)$ would not have acquired lock $\mathsf{succ}(\mathsf{succ}(m))$ (the semantics would get stuck) as lock $\mathsf{succ}(m)$ would have been locked by thread m.

According to lemma 3, $\mathsf{succ}(m) \notin \mathsf{eps}(\mathsf{succ}(m), \mathsf{succ}(\mathsf{succ}(m)))$ holds when $\iota_{\mathsf{succ}(m)}$ is *allocated* at thread $\mathsf{succ}(m)$ once lock $\iota_{\mathsf{succ}(\mathsf{succ}(m))}$ has been acquired. This contradicts the assumption that thread m is the first of the deadlocked threads that acquired a lock as thread $\mathsf{succ}(m)$ locks $\iota_{\mathsf{succ}(\mathsf{succ}(m))}$ before creating and sharing the lock $\iota_{\mathsf{succ}(m)}$. □

Lemma 3 (Locking prior to allocation)**.** *If the following hold*

- $S_0;T,k:\theta_0;E_0[\mathtt{lock}_\gamma\,\mathtt{loc}_\iota] \rightsquigarrow^m S_m;T',k:\theta_{n-1};E_{n-1}[\mathtt{lock}_{\gamma'}\,\mathtt{loc}_{\iota'}]$,
- *each configuration* $S_\iota;T_\iota,k:\theta_\iota;E_\iota[e]$, *where* $\iota \in [0,m]$, *is well-typed in* M_ι,
- $\theta_0(\iota) = (n_1,n_2)$, $n_1 > 0$, $n_2 = 0$,
- $\theta_1(\iota) = (n_1,n_2)$, $n_1 > 0$, $n_2 = 1$,
- $\theta_{n-1}(\iota') = (n_1,n_2)$, $n_1 > 0$, $n_2 = 0$,
- *for all* $z \in [1,n-1].\theta_z(\iota) = (n_1,n_2) \wedge n_1 > 0 \wedge n_2 \geq 1$
- $\iota' \notin \epsilon$, *where* ϵ *is defined in* $E_0[\mathsf{pop}_\gamma\,\square];\iota;1 \vdash \epsilon$

then $\iota' \notin \mathsf{dom}(\theta_0)$

Proof. Let us assume that $\iota' \in \mathsf{dom}(\theta_0)$. Let us assume that ι' is locked within the same function context as ι, then the typing relation for thread k at step 0 and the fact that ι' remains locked until step m imply that locking event of ι' exists in γ. Hence $\iota' \in \epsilon$, where ϵ is defined in $E_0[\mathsf{pop}_\gamma\,\square];\iota;1 \vdash \epsilon$ (operational rules *L2,W2,W3*). This contradicts the assumption of this lemma, thus it must the case that ι' must be locked outside the function context. This is a contradiction as well, using the same reasoning about outer effects γ_x and rules *L0-L3*, *W1-W3*. Thus, the initial assumption that $\iota' \in \mathsf{dom}(\theta_0)$ leads to a contradiction. □

Lemma 4 (Preservation — Program)**.** *Let* $S;T$ *be a well-typed configuration with* $M \vdash S;T$. *If the operational semantics takes a step* $S;T \rightsquigarrow S';T'$, *then there exist an* $M' \supseteq M$ *such that the resulting configuration is well-typed with* $M' \vdash S';T'$.

Proof. By case analysis on the thread evaluation relation:

Case *E-T*: Rule *E-T* implies that $\theta;E[e] = \theta;\square[()]$, $S' = S$ and $T' = T$, $\forall \iota \mapsto n_1,n_2 \in \theta.n_1 = n_2 = 0$. By inversion of the configuration typing assumption we have that:

- $M \vdash T,n:\theta;\square[()]$: by inversion of this derivation we have that:
  * $M;\emptyset;\emptyset \vdash_t \theta;\square[()] : \langle\rangle\,\&\,(\gamma;\gamma)$
  * $M \vdash T$
  * $n \notin \mathsf{dom}(T)$
- $M \vdash S$
- $\mathsf{mutex}(T,n:\theta;\square[()])$: implies that $\mathsf{mutex}(T)$.

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

Case *E-PP* : Rule *E-PP* implies the following facts:

- $S' = S$, $T' = T,n:\theta;E[v]$ and $e = \mathsf{pop}_{\gamma_a}\,v$.

By inversion of the configuration typing assumption we have that:

- $M \vdash S$
- $\mathsf{mutex}(T,n:\theta;E[\mathsf{pop}_{\gamma_a}\,v])$: no new locks are acquired. Thus, $\mathsf{mutex}(T,n:\theta;E[\mathsf{pop}_{\gamma_a}\,v])$ holds.
- $M \vdash T,n:\theta;E[\mathsf{pop}_{\gamma_a}\,v]$: by inversion of this derivation we have that:
  * $M \vdash T$
  * $n \notin \mathsf{dom}(T)$
  * $M;\emptyset;\emptyset \vdash_t \theta;E[\mathsf{pop}_{\gamma_a}\,v] : \langle\rangle\,\&\,(\gamma;\gamma')$: by inversion of this derivation we have that $E;\emptyset;\emptyset \vdash \theta$, $M;\emptyset;\emptyset \vdash E : \tau_2 \xrightarrow{\gamma_a;\gamma_b} \langle\rangle\,\&\,(\gamma;\gamma')$, $M;\emptyset;\emptyset \vdash \mathsf{pop}_{\gamma_a}\,v : \tau_2\ \&\,(\gamma_a;\gamma_b)$. By inversion of the latter derivation we have that $M;\emptyset;\emptyset \vdash v : \tau_2\ \&\,(\gamma_d;\gamma_d)$, $\vdash M;\Delta;\Gamma;\gamma_a;\gamma_b$ and $\gamma_b = \gamma_a \oplus \gamma_e$, where $\gamma_d \lhd \gamma_e$. The application of lemma 6 to the former two derivations implies that $M;\emptyset;\emptyset \vdash v : \tau_2\,\&\,(\gamma_a;\gamma_a)$. The application of lemma 8 to $M;\emptyset;\emptyset \vdash E : \tau_2 \xrightarrow{\gamma_a;\gamma_b} \langle\rangle\,\&\,(\gamma;\gamma')$ and $\gamma_a \lhd \gamma_b$ implies that there exists a γ'' such that $\gamma'' \lhd \gamma'$ and $M;\emptyset;\emptyset \vdash E : \tau_2 \xrightarrow{\gamma_a;\gamma_a} \langle\rangle\,\&\,(\gamma;\gamma'')$. Thus, $M;\emptyset;\emptyset \vdash E[v] : \langle\rangle\,\&\,(\gamma;\gamma'')$, by the application of rule *EA0*.
    If $E = \square$, then rule *EA2* implies that $M;\emptyset;\emptyset \vdash_t \theta;\square[()] : \langle\rangle\,\&\,(\gamma;\gamma)$. Otherwise, lemma 7 implies that there exists an E' and u' such that $E[v] = E'[u']$ and $M;\emptyset;\emptyset \vdash E'[u'] : \langle\rangle\,\&\,(\gamma;\gamma'')$. Since, we have assumed that E is not equal to □ and given that v is a value, then $E = E'[F]$ and $u' = F[v]$ for some frame F. By inversion of the latter derivation we obtain that $M;\emptyset;\emptyset \vdash u' : \tau_a\,\&\,(\gamma'_a;\gamma'_b)$ and $M;\emptyset;\emptyset \vdash E' : \tau_a \xrightarrow{\gamma'_a;\gamma'_b} \langle\rangle\,\&\,(\gamma;\gamma'')$. It can be shown that $E';\gamma'_c;\gamma'_c \vdash \theta$, where $\gamma'_c =$ if $u' \neq \mathsf{pop}_{\gamma'_a}\,v'$ then γ'_b else $\emptyset$, using the above facts and $E;\emptyset;\emptyset \vdash \theta$. Thus, $M;\emptyset;\emptyset \vdash_t \theta;E'[u'] : \langle\rangle\,\&\,(\gamma;\gamma'')$.

Case *E-D*,*E-AS*, *E-A*,*E-RP*,*E-NG*, *E-SH*, *E-RL*,*E-LK*,*E-UL*,*E-SN*: similar to the previous case. In the case of rule *E-RP*, the proof requires the use of lemma 11. In the case of rules *E-AS* and *E-NG* the use of lemma 9 will also be required. The proof of rule *E-NG* also requires lemmata 12 and 13. For rules *E-SH*, *E-RL*, *E-LK*, *E-UL* it must be shown that the correspondence between static and dynamic counts is *preserved*. This is easy to show given the typing derivation of $\theta;E[u]$ and the input effects of the typing rules *T-SH*, *T-RL*, *T-LK*, *T-UL* respectively. In the case of *E-SN*, θ is divided into θ_1 and θ_2 for threads n and n' respectively. Thus, it is shown that the effects of the remaining computation of thread n match θ_1, whereas the effect of the new thread n' matches θ_2.

□

Lemma 5 (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 γ_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 γ_a is the annotation of the abstraction type (i.e. the type of e_1) implies that $\gamma_0 \lhd \gamma_1$. $\vdash M;\Delta;\Gamma;\gamma_1;\gamma_2$ and $\vdash M;\Delta;\Gamma;\gamma_2;\gamma_3$ imply that $\gamma_1 \lhd \gamma_2$, $\gamma_2 \lhd \gamma_3$. Thus, $\gamma_0 \lhd \gamma_3$. They also imply that $\mathsf{seq}(\emptyset) \vdash \gamma_3$, $\vdash M$, $M;\Delta \vdash \Gamma$ and $M;\Delta \vdash \gamma_3$. The latter fact and the fact that $\gamma_0 \lhd \gamma_3$ imply that $M;\Delta \vdash \gamma_0$. Thus, $\vdash M;\Delta;\Gamma;\gamma_0;\gamma_3$ holds. □

Lemma 6 (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 $\vdash M;\Delta;\Gamma;\gamma_1;\gamma_1$ and also $\vdash M;\Delta;\Gamma;\gamma_2;\gamma_2$ 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-I*,*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 $(\gamma_1;\gamma_1)$ and $(\gamma_2;\gamma_2)$ respectively. The case for rule *T-RF* can be shown trivially by induction (the base case is the same as for rule *T-F*). □

Lemma 7 (Redex)**.** *If $M;\Delta;\Gamma \vdash E[e] : \tau\,\&\,(\gamma_1;\gamma_2)$ and $E[e]$ is* not *a value then $M;\Delta;\Gamma \vdash E'[u] : \tau\,\&\,(\gamma_1;\gamma_2)$ such that $E'[u] = E[e]$.*

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

Lemma 8 (Evaluation Context Variable Substitution)**.** *If $M;\Delta;\Gamma \vdash E : \tau \xrightarrow{\gamma_1;\gamma_2} \tau'\,\&\,(\gamma_3;\gamma_4)$ and $\gamma_3 \lhd \gamma_5 \lhd \gamma_4$, then $M;\Delta;\Gamma \vdash E : \tau \xrightarrow{\gamma_1;\gamma_5} \tau'\,\&\,(\gamma_3;\gamma_6)$ and $\gamma_6 \lhd \gamma_4$*

Proof. Straightforward induction on the evaluation context typing relation. □

Lemma 9 (Variable Substitution)**.** $M;\Delta;\Gamma,x:\tau_1 \vdash e:\tau_2\,\&\,(\gamma_1;\gamma_2) \wedge M;\emptyset;\emptyset \vdash v:\tau_1\,\&\,(\gamma;\gamma) \Rightarrow M;\Delta;\Gamma \vdash e[v/x]:\tau_2\,\&\,(\gamma_1;\gamma_2)$

Proof. Straightforward induction on the expression typing derivation. □

Lemma 10 (Type Well-formedness)**.** $M;\Delta;\Gamma \vdash e:\tau\,\&\,(\gamma;\gamma') \Rightarrow M;\Delta \vdash \tau$

Proof. Straightforward induction on the typing rules. □

Lemma 11 (Location Substitution)**.** *If the following hold:*

- $M,\iota \mapsto \tau';\Delta,\rho;\Gamma \vdash e:\tau\,\&\,(\gamma;\gamma')$
- fresh n
- $\mathsf{seq}(\emptyset) \vdash \gamma'[\iota@n/\rho]$.

then $M,\iota \mapsto \tau';\Delta;\Gamma[\iota@n/\rho] \vdash e[\iota@n/\rho] : \tau[\iota@n/\rho]\,\&\,(\gamma[\iota@n/\rho];\gamma'[\iota@n/\rho])$.

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 $\iota \notin \mathsf{dom}(M)$ then $M,\iota \mapsto \tau';\Delta;\Gamma \vdash e:\tau\,\&\,(\gamma;\gamma')$.*

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

Lemma 13 (Evaluation Context Typing Weakening)**.** *$M;\Delta;\Gamma \vdash E : \tau \xrightarrow{\gamma_1;\gamma_2} \tau'\,\&\,(\gamma;\gamma')$, $M;\emptyset \vdash \tau'$ and $\iota \notin \mathsf{dom}(M)$ then $M,\iota \mapsto \tau';\Delta;\Gamma \vdash E : \tau \xrightarrow{\gamma_1;\gamma_2} \tau'\,\&\,(\gamma;\gamma')$.*

Proof. Proof by induction on the derivation of E. □

Lemma 14 (Progress — Program)**.** *Let $S;T$ be a closed well-typed configuration with $M \vdash S;T$ and $\neg\mathsf{deadlocked}(T)$, then $S;T$ is not stuck ($\vdash S;T$).*

Proof. In order to prove that the configuration is not stuck, we need to prove that each of the executing threads can either perform a step or *block* predicate holds for it. We also need to show that there exists no deadlock in T, but this is immediate from the assumption $\neg\mathsf{deadlocked}(T)$.

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

- $n \notin \mathsf{dom}(T_1)$
- $M;\emptyset;\emptyset \vdash_t \theta; E[e] : \langle\rangle \,\&\, (\gamma;\gamma')$: if $E[e]$ is a value then by inversion of $M;\emptyset;\emptyset \vdash_t \theta; E[e] : \langle\rangle \,\&\, (\gamma;\gamma')$ (rule *EA2*), we have that $\forall \iota \mapsto n_1, n_2 \in \theta. n_1 = n_2 = 0$ ($\forall r^{\kappa} \in \gamma'_1.\kappa = (0,0)$ and $\Box;\gamma'_1;\gamma'_1 \vdash \theta$, where $\gamma'_1 \subseteq_{min} \gamma$) and that $E[e] = \Box[()]$. Thus, rule *E-T* can be applied. Otherwise, rule *EA1* applies and by inversion using *EA1* we obtain that $M;\emptyset;\emptyset \vdash u : \tau \,\&\, (\gamma_a;\gamma_b)$, $M;\emptyset;\emptyset \vdash E' : \tau \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \;\&\; \&(\gamma;\gamma')$ and $E;\gamma_g;\gamma_g \vdash \theta$, where $\gamma_g = $ if $u \neq \mathsf{pop}_{\gamma_a}\ v$ then γ_b else $\emptyset$. Then, we proceed by a case analysis on u:
  - $\mathsf{pop}_{\gamma_a}\ v$: rule *E-PP* can be applied to perform a single step.
  - $(v'\ v)^{\mathsf{seq}(\gamma_a)}$: rule *E-A* can be applied to perform a single step.
  - $(f)[r]$: rule *E-RP* can be applied to perform a single step.
  - $\mathtt{let}\,\rho, x = \mathtt{ref}\ v\ \mathtt{in}\, e_2$: rule *E-NG* can be applied to perform a single step.
  - $(v'\ v)^{\mathsf{par}}$: it suffices to show that $\gamma_c \vdash \theta = \theta_1 \oplus \theta_2$ holds, where γ_c is the annotation of function v'. If γ_d is defined by $\gamma_d \subseteq_{min} \gamma_c$, then we need to show that $\gamma_d \vdash \theta_1 + \theta_2$. The proof can be reduced to showing that the counts of each ι of θ are greater than or equal to the sum of counts of all ι in γ_d. This is immediate by $\gamma_2 = \gamma_a \oplus \gamma_c$, which can be obtained by inversion of the typing derivation of $(v'\ v)^{\mathsf{par}}$, and the strict correspondence between static and dynamic counts (i.e, $E;\gamma_b;\gamma_b \vdash \theta$; notice that γ_a is a prefix of γ_b by well-formedness of the typing relation). Thus, rule *E-SN* can be applied to perform a single step.
  - $\mathtt{share}\,\mathtt{loc}_\iota$: $E;\gamma_b;\gamma_b \vdash \theta$ establishes a strict correspondence between dynamic and static counts. The typing derivation implies that $\gamma_a(\iota@n_1) \geq (2,0)$, for some n_1 existentially bound in the premise of the derivation. Therefore, $\theta(\iota) \geq (1,0)$. It is possible to perform a single step using rule *E-SH*.
  - $\mathtt{release}\,\mathtt{loc}_\iota$: similar to previous case. It is possible to perform a single step using rule *E-RL*.
  - $\mathtt{unlock}\,\mathtt{loc}_\iota$:similar to previous case. It is possible to perform a single step using rule *E-UL*.
  - $\mathtt{lock}_{\gamma_a}\,\mathtt{loc}_\iota$: similarly to the previous cases we can show that $\theta(\iota) = (n_1, n_2)$ and n_1 is positive. Relation $E[\mathsf{pop}_{\gamma_a}\,\Box];\iota;1 \vdash \epsilon$ is derivable. The intuition behind this proof is that for each *lock* operation there exists a corresponding *unlock* operation. This is enforced by the typing rule *T-RL*. If n_2 is positive then the proof is completed using rule *E-LK* to perform a step. Otherwise, if $\mathsf{locked}(T_1) \cap (\epsilon \cup \{\iota\})$ is empty then rule *E-LK* can be used to perform a step. Otherwise, $\mathsf{blocked}(T, n)$ holds.
  - $\mathtt{deref}\,\mathtt{loc}_\iota$: it can be trivially shown (as in the previous case of *share* that we proved $\theta(\iota) \geq (1,0)$), that $\theta(\iota) \geq (1,1)$ and since $\mathsf{mutex}(T_1, n\!:\!\theta; E[\mathtt{deref}\ \mathtt{loc}_\iota])$ holds, then $\iota \notin \mathsf{locked}(T_1)$ and thus rule *E-D* can be used to perform a step.
  - $\mathtt{loc}_\iota := v$: similar to the previous case. Rule *E-AS* can be used to perform a step.

□