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, 10, 12, 13]. In this setting, lock handles are associated with type-level lock names via the use of singleton types. Thus, handle lk_ι is of type $\mathtt{lk}(\iota)$. The same applies to lock handle variables. The effect system tracks the order of lock operations on handles or variables and determines whether all threads acquire locks in the same order.

K. Honda and A. Mycroft (Eds.): Programming Language Approaches to Concurrency and communication-cEntric Software 2010 (PLACES'10)
EPTCS 69, 2011, pp. 44–58, doi:10.4204/EPTCS.69.4

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:

(lock x in ... lock y in ...) || (lock y in ... lock x in ...)

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 [10] as there is no single lock order for *both* threads. Similar considerations apply to the more recent works of Suenaga [12] and Vasconcelos *et al.* [13] 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. It also forms the basis for a much simpler approach to providing deadlock freedom, following a quite different path, that is easier to program and amenable to type inference, which has been implemented for C/pthreads [9].

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:

(lock$_{\{y\}}$ x in ... lock$_{\emptyset}$ y in ...) || (lock$_{\{x\}}$ y in ... lock$_{\emptyset}$ x in ...)

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. In fact, as we show here, even if Boudol's language had lock/unlock constructs, instead of lock...in..., his type system is not sufficient to guarantee deadlock freedom. The example program in Figure 1(a) will

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

(a) before substitution

```
lock_{a} a;    a := a + 1;
lock_{b} a;    a := a + a;
unlock a;
lock_∅ b;      b := b + a;
unlock b;
unlock a
```

(b) after substitution

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

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]

In our naïvely extended (and broken, as will be shown) version of Boudol's 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 is easily verified by observing the lock operations between a specific `lock`/`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, 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 of Boudol's system also fails for another reason: it is based on the assumption that calling a function cannot affect the set of locks held by a thread. This is obviously not true, if non lexically-scoped locking is 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 `lock` and `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 `lock` operation is evaluated, the future lockset is calculated by inspecting its continuation effect. 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 continuation effects. For example, the annotation $[y+, x-, z+, z-, y-]$ at the first `lock` operation means that in the future (i.e., after this lock operation) y will be acquired, then x will be released, and so on.[2] If x and y were different, the runtime system would deduce that between this `lock` operation on x

[1] To simplify presentation, we assume here that there is one implicit lock per variable, which has the same name. This is more or less consistent with our formalization in Section 3.

[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).

```
let  f = λx.λy.λz.  lock[y+,x−,z+,z−,y−] x;   x := x + 1;
                    lock[x−,z+,z−,y−] y;      y := y + x;
                    unlock x;
                    lock[z−,y−] z;            z := z + y;
                    unlock z;
                    unlock y
in   f a a b
```

(a) before substitution

```
lock[a+,a−,b+,b−,a−] a;   a := a + 1;
lock[a−,b+,b−,a−] a;      a := a + a;
unlock a;
lock[b−,a−] b;            b := b + a;
unlock b;
unlock a
```

(b) after substitution

Figure 2: The program of Figure 1 with continuation effect annotations; now well typed in both cases.

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

`if` c `then` (`lock` x; ... `unlock` x) `else` (`lock` y; ... `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 $[x+, x-]?[y+, y-]$.

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

`fix` $f.\ \lambda x.$ `if` c `then` (... $f(y)$ ...) `else` ...

Let us call γ_f the effect of f and γ_b the computed effect for the body of f. It is easy to see that γ_b must *contain* γ_f and, if any lock/unlock operations are present in the body of f, γ_b will be strictly larger than γ_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 γ_b and requiring that the summary is equal to γ_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.

Expression	e	::=	$x \mid f \mid (e\ e)^{\xi} \mid (e)[r] \mid e := e$
		$\mid$	$\texttt{deref}\ e \mid \texttt{let}\ \rho, x = \texttt{ref}\ e\ \texttt{in}\ e$
		$\mid$	$\texttt{share}\ e \mid \texttt{release}\ e \mid \texttt{lock}_\gamma\ e$
		$\mid$	$\texttt{unlock}\ e \mid () \mid \texttt{pop}_\gamma\ e \mid \texttt{loc}_\iota$
		$\mid$	$\texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{true} \mid \texttt{false}$
Value	v	::=	$f \mid () \mid \texttt{loc}_\iota \mid \texttt{true} \mid \texttt{false}$
Function	f	::=	$\lambda x.e\ \texttt{as}\ \tau \xrightarrow{\gamma} \tau \mid \Lambda\rho.\, f \mid \texttt{fix}\ x : \tau.\, f$

Type	τ	::=	$\langle\rangle \mid \tau \xrightarrow{\gamma} \tau \mid \forall\rho.\, \tau$
		$\mid$	$\texttt{ref}(\tau, r) \mid \texttt{bool}$
Location	r	::=	$\rho \mid \iota@n \mid \rho@n$
Calling mode	ξ	::=	$\mathsf{seq}(\gamma) \mid \mathsf{par}$
Capability	κ	::=	$n, n \mid \overline{n, n}$
Effect	γ	::=	$\emptyset \mid \gamma, r^\kappa \mid \gamma, \gamma\,?\,\gamma$

Figure 3: Language syntax.

3 Formalism

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 [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 (locations), but this is of course not necessary.

The language core comprises of variables (x), constants (the unit value, `true` and `false`), functions (f), and function application. Functions can be location polymorphic ($\Lambda\rho.\, f$) 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* (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 parameters have been evaluated and substituted, the function's body 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 $\texttt{pop}_\gamma\ e$ encloses a function body e and can only appear during evaluation. The same applies to constant locations $\iota@n$, which cannot exist at the source-level. The construct $\texttt{let}\ \rho, x = \texttt{ref}\ e_1\ \texttt{in}\ e_2$ 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's location. The reference variable x has 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. 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* $(\overline{n_1, n_2})$. 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. 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 ($\texttt{share}\ e$, $\texttt{release}\ e$, $\texttt{lock}_\gamma\ e$ and $\texttt{unlock}\ e$) operate on a reference e. The first two constructs *increment* and *decrement* the cell reference count of e respectively.

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

$$\begin{array}{llll} \textbf{Stack} & E & ::= & \Box \mid E[F] \\ \textbf{Frame} & F & ::= & (\Box\ e)^{\xi} \mid (v\ \Box)^{\xi} \mid (\Box)[r] \mid \texttt{let}\ \rho, x = \texttt{ref}\ \Box\ \texttt{in}\ e \\ & & \mid & \texttt{deref}\ \Box \mid \Box := e \mid v := \Box \mid \texttt{share}\ \Box \mid \texttt{release}\ \Box \\ & & \mid & \texttt{lock}_{\gamma}\ \Box \mid \texttt{unlock}\ \Box \mid \texttt{pop}_{\gamma}\ \Box \\ & & \mid & \texttt{if}\ \Box\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \end{array}$$

Reduction relation $\boxed{C \rightsquigarrow C'}$

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

$$\frac{\forall \iota.\theta(\iota) = (0,0)}{S;T,n{:}\theta;() \rightsquigarrow S;T}\ \textit{(E-T)}$$

$$\frac{v' \equiv \lambda x.e_1\ \texttt{as}\ \tau'}{S;T,n{:}\theta;E[(v'\ v)^{\mathsf{seq}(\gamma_b)}] \rightsquigarrow S;T,n{:}\theta;E[\texttt{pop}_{\gamma_b}\ e_1[v/x]]}\ \textit{(E-A)}$$

$$\frac{}{S;T,n{:}\theta;E[\texttt{pop}_{\gamma}\ v] \rightsquigarrow S;T,n{:}\theta;E[v]}\ \textit{(E-PP)}$$

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

$$\frac{}{S;T,n{:}\theta;E[(\texttt{fix}\ x:\tau.\ f\ v)^{\mathsf{seq}(\gamma_a)}] \rightsquigarrow S;T,n{:}\theta;E[(f[\texttt{fix}\ x:\tau.\ f/x]\ v)^{\mathsf{seq}(\gamma_a)}]}\ \textit{(E-FX)}$$

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

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

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

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

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

$$\frac{\theta(\iota) \geq (1,0) \qquad \theta' = \theta +_{\iota} (1,0)}{S;T,n{:}\theta;E[\texttt{share}\ \texttt{loc}_{\iota}] \rightsquigarrow S;T,n{:}\theta';E[()]}\ \textit{(E-SH)}$$

$$\frac{\theta(\iota) \geq (1,0) \qquad \theta(\iota) = (n_1, n_2) \qquad n_1 = 1 \Rightarrow n_2 = 0 \qquad \theta' = \theta +_{\iota} (-1,0)}{S;T,n{:}\theta;E[\texttt{release}\ \texttt{loc}_{\iota}] \rightsquigarrow S;T,n{:}\theta';E[()]}\ \textit{(E-RL)}$$

$$\frac{\epsilon = \mathsf{lockset}(\iota, 1, E[\texttt{pop}_{\gamma_1}\ \Box]) \qquad \theta = \theta'', \iota \mapsto n_1;0;\epsilon_1;\epsilon_2 \qquad \theta' = \theta'', \iota \mapsto n_1;1;\mathsf{dom}(S);\epsilon \qquad n_1 \geq 1 \qquad \mathsf{locked}(T) \cap \epsilon = \emptyset}{S;T,n{:}\theta;E[\texttt{lock}_{\gamma_1}\ \texttt{loc}_{\iota}] \rightsquigarrow S;T,n{:}\theta';E[()]}\ \textit{(E-LK0)}$$

$$\frac{\theta(\iota) \geq (1,1) \qquad \theta' = \theta +_{\iota} (0,1)}{S;T,n{:}\theta;E[\texttt{lock}_{\gamma_1}\ \texttt{loc}_{\iota}] \rightsquigarrow S;T,n{:}\theta';E[()]}\ \textit{(E-LK1)}$$

$$\frac{\theta(\iota) \geq (1,1) \qquad \theta' = \theta +_{\iota} (0,-1)}{S;T,n{:}\theta;E[\texttt{unlock}\ \texttt{loc}_{\iota}] \rightsquigarrow S;T,n{:}\theta';E[()]}\ \textit{(E-UL)}$$

Figure 4: Operational semantics.

Similarly, the latter two constructs *increment* and *decrement* the lock count of e. As mentioned earlier, the runtime system inspects the lock annotation γ to determine whether it is safe to lock e.

3.1 Operational Semantics

We define a *small-step* operational semantics for our language in Figure 4.[3] The evaluation relation transforms *configurations*. A configuration C consists of an abstract *store* S and a thread map T.[4] 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.

[3] 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 are given in the Appendix. A full formalization is given in the companion technical report [8].

[4] 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.

A *frame F* is an expression with a *hole*, represented as □. The hole indicates the position where the next reduction step can take place. A *thread evaluation context E*, is defined as a stack of nested frames. Our notion of evaluation context imposes a call-by-value evaluation strategy to our language. Subexpressions are evaluated in a left-to-right order. We assume that concurrent reduction events can be totally ordered [11]. 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., $\mathsf{min}(\gamma_a)$). Notice, that θ is divided into two lists θ_1 and θ_2 using the new thread's initial effect $\mathsf{max}(\gamma_a)$ as a reference for consuming the appropriate number of counts from θ. 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 successfuly 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 ι (with initial value v) and the dynamic count $(1,1)$ to S and θ respectively. Rules *E-AS* and *E-D* require that the location (ι) being accessed is both live and accessible and no other thread has access to ι. 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 (ι) is initially unlocked. The future lockset (ϵ) is dynamically computed, by inspecting the preceding stack frames (E) as well as the lock annotation (γ_1). The lockset ϵ is a list of locations (and thus locks). The reference ι must be live and no other thread must hold either ι or any of the locations in ϵ. Upon success, the lock count of ι is incremented by one. On the other hand, rule *E-LK1* applies when ι 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

We now 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 (γ) is an *ordered list* of elements of the form r^κ and summarizes the sequence of operations (e.g., locking or sharing) on references. The syntax of types in Figure 3 (on page 48) is more or less standard: Atomic types consist of base types (the unit type, denoted by $\langle\rangle$, and `bool`); reference types $\mathtt{ref}(\tau, r)$ 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, τ is the type attributed to e, γ is the *input effect*, and γ' is the *output effect*. In the typing context, M is a mapping of constant

Typing relation $\boxed{M;\Delta;\Gamma \vdash e : \tau\,\&\,(\gamma;\gamma')}$

$$\frac{\vdash M;\Delta;\Gamma;\gamma;\gamma}{M;\Delta;\Gamma \vdash () : \langle\rangle\,\&\,(\gamma;\gamma)}\ (T\text{-}U) \qquad \frac{\vdash M;\Delta;\Gamma;\gamma;\gamma}{M;\Delta;\Gamma \vdash \mathtt{true} : \mathtt{bool}\,\&\,(\gamma;\gamma)}\ (T\text{-}TR) \qquad \frac{\vdash M;\Delta;\Gamma;\gamma;\gamma}{M;\Delta;\Gamma \vdash \mathtt{false} : \mathtt{bool}\,\&\,(\gamma;\gamma)}\ (T\text{-}FL)$$

$$\frac{\vdash M;\Delta;\Gamma;\gamma;\gamma \quad (x:\tau') \in \Gamma \quad \tau \simeq \tau'}{M;\Delta;\Gamma \vdash x : \tau\,\&\,(\gamma;\gamma)}\ (T\text{-}V) \qquad \frac{\vdash M;\Delta;\Gamma;\gamma;\gamma \quad \tau' \equiv \tau_1 \xrightarrow{\gamma_b} \tau_2 \quad M;\Delta \vdash \tau' \quad \tau \simeq \tau' \quad \mathsf{seq}(\emptyset) \vdash \gamma_b \Rightarrow M;\Delta;\Gamma, x:\tau_1 \vdash e_1 : \tau_2\,\&\,(\mathsf{min}(\gamma_b);\gamma_b)}{M;\Delta;\Gamma \vdash \lambda x.e_1\ \mathtt{as}\ \tau' : \tau\,\&\,(\gamma;\gamma)}\ (T\text{-}F)$$

$$\frac{M;\Delta,\rho;\Gamma \vdash f : \tau\,\&\,(\gamma;\gamma)}{M;\Delta;\Gamma \vdash \Lambda\rho.\, f : \forall\rho.\,\tau\,\&\,(\gamma;\gamma)}\ (T\text{-}RF) \qquad \frac{M;\Delta \vdash r \quad M;\Delta \vdash \tau[r/\rho] \quad M;\Delta;\Gamma \vdash e_1 : \forall\rho.\,\tau\,\&\,(\gamma;\gamma')}{M;\Delta;\Gamma \vdash (e_1)[r] : \tau[r/\rho]\,\&\,(\gamma;\gamma')}\ (T\text{-}RP)$$

$$\frac{M;\Delta;\Gamma \vdash e_1 : \tau_1 \xrightarrow{\gamma_a} \tau_2\,\&\,(\gamma_3;\gamma') \quad \xi \vdash \gamma_2 = \gamma \oplus \gamma_a \quad M;\Delta;\Gamma \vdash e_2 : \tau_1\,\&\,(\gamma_2;\gamma_3) \quad \xi = \mathsf{par} \Rightarrow \tau_2 = \langle\rangle}{M;\Delta;\Gamma \vdash (e_1\ e_2)^{\xi} : \tau_2\,\&\,(\gamma;\gamma')}\ (T\text{-}A)$$

$$\frac{M;\Delta;\Gamma \vdash e : \tau'\,\&\,(\mathsf{min}(\gamma_b);\gamma_b) \quad \gamma_b \simeq \gamma'_b \quad \mathsf{seq}(\gamma) \vdash \gamma' = \gamma \oplus \gamma'_b \quad \tau' \simeq \tau \quad \vdash M;\Delta;\Gamma;\gamma;\gamma'}{M;\Delta;\Gamma \vdash \mathtt{pop}_{\gamma}\ e : \tau\,\&\,(\gamma;\gamma')}\ (T\text{-}PP)$$

$$\frac{\tau \equiv \tau_1 \xrightarrow{\gamma_b} \tau_2 \quad \tau' \equiv \tau'_1 \xrightarrow{\gamma'_a} \tau'_2 \quad \tau \simeq \tau' \quad \gamma_a \simeq \gamma'_a \quad M;\Delta;\Gamma, x:\tau \vdash f : \tau'\,\&\,(\gamma;\gamma) \quad \gamma_b = \mathsf{summary}(\gamma_a)}{M;\Delta;\Gamma \vdash \mathtt{fix}\ x:\tau.\, f : \tau\,\&\,(\gamma;\gamma)}\ (T\text{-}FX) \qquad \frac{\vdash M;\Delta;\Gamma;\gamma;\gamma \quad (\iota \mapsto \tau') \in M \quad \tau \simeq \mathtt{ref}(\tau',\iota)}{M;\Delta;\Gamma \vdash \mathtt{loc}_{\iota} : \tau\,\&\,(\gamma;\gamma)}\ (T\text{-}L)$$

$$\frac{M;\Delta;\Gamma \vdash e_1 : \tau_1\,\&\,(\gamma_1 \setminus \rho;\gamma') \quad \gamma_1(\rho) = (1,1) \quad M;\Delta \vdash \tau \quad M;\Delta,\rho;\Gamma, x:\mathtt{ref}(\tau_1,\rho) \vdash e_2 : \tau\,\&\,(\gamma,\rho^{0,0};\gamma_1)}{M;\Delta;\Gamma \vdash \mathtt{let}\ \rho, x = \mathtt{ref}\ e_1\ \mathtt{in}\ e_2 : \tau\,\&\,(\gamma;\gamma')}\ (T\text{-}NG)$$

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

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

$$\frac{M;\Delta;\Gamma \vdash e : \mathtt{ref}(\tau,r)\,\&\,(\gamma, r^{\kappa-(0,1)};\gamma') \quad \kappa \geq (1,1) \quad \gamma(r) = \kappa}{M;\Delta;\Gamma \vdash \mathtt{lock}_{\gamma}\ e : \langle\rangle\,\&\,(\gamma;\gamma')}\ (T\text{-}LK) \qquad \frac{M;\Delta;\Gamma \vdash e : \mathtt{ref}(\tau,r)\,\&\,(\gamma, r^{\kappa+(0,1)};\gamma') \quad \kappa \geq (1,0) \quad \gamma(r) = \kappa}{M;\Delta;\Gamma \vdash \mathtt{unlock}\ e : \langle\rangle\,\&\,(\gamma;\gamma')}\ (T\text{-}UL)$$

$$\frac{M;\Delta;\Gamma \vdash e_1 : \mathtt{bool}\,\&\,(\gamma, \gamma_2\,?\,\gamma_3;\gamma') \quad \mathsf{max}(\gamma :: \gamma_2) = \mathsf{max}(\gamma :: \gamma_3) \quad M;\Delta;\Gamma \vdash e_2 : \tau\,\&\,(\gamma;\gamma :: \gamma_2) \quad M;\Delta;\Gamma \vdash e_3 : \tau\,\&\,(\gamma;\gamma :: \gamma_3)}{M;\Delta;\Gamma \vdash \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3 : \tau\,\&\,(\gamma;\gamma')}\ (T\text{-}IF)$$

Figure 5: Typing rules.

locations to types, Δ is a set of cell variables, and Γ 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 $\tau \simeq \tau'$ which allows the type system to ignore the identifiers used for location aliasing and, for example, treat the types $\iota@n_1$ and $\iota@n_2$ as equal. The typing rule *T-F* checks that, if the effect γ_b 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 γ_a, which contains the entire history of

events occurring in the function body; this is performed by the premise $\xi \vdash \gamma_2 = \gamma \oplus \gamma_a$, which performs all the necessary checks to ensure that all the capabilities required in the function's effect γ_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 γ to e_2, the body of `let`, augmented by $\rho^{0,0}$. This means that, upon termination of e_2, both references and locks of ρ must have been consumed. The output effect of e_2 is a γ_1 such that ρ has capability $(1,1)$, which implies that when e_2 starts being evaluated ρ 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 ρ. 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, κ 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 γ is the input (continuation) effect to a conditional expression. Then γ is passed as the input effect to both branches. We know that the outputs of both branches will have γ as a common prefix; if γ_2 and γ_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 θ be the thread's *access list*. Thread typing is defined by the rule:

$$\frac{\begin{array}{cc} M;\Delta;\Gamma \vdash e : \tau \,\&\, (\gamma_a;\gamma_b) & M;\Delta;\Gamma \vdash E : \tau \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \,\&\, (\gamma_1;\gamma_2) \\ \forall r^{\kappa} \in \gamma_1 . \kappa = (0,0) \quad \mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta) & \mathsf{lockset\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta) \end{array}}{M;\Delta;\Gamma \vdash_t \theta; E[e] : \langle\rangle \,\&\, (\gamma_1;\gamma_2)} \; (EA)$$

First of all, thread typing implies the typing of $E[e]$.

Secondly, thread typing establishes an exact correspondence between counts of the access list θ and counts of pop expression annotations that reside in the evaluation context $E[\mathsf{pop}_{\gamma_b}\ \square]$ (i.e., $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta)$). The typing derivations of e and E establish an exact correspondence between the annotations of pop expressions and static effects. Therefore, for each location ι in θ, the dynamic reference and lock counts of ι are identical to the static counts of ι 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 companion technical report [8].

$\mathsf{lockset\_ok}(E[\mathsf{pop}_{\gamma_b}\ \square], \theta)$). This invariant is essential for establishing deadlock freedom. Finally, all locations must be deallocated and released when a thread terminates ($\forall r^{\kappa} \in \gamma_1.\, \kappa = (0,0)$).

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

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

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:

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

Configuration Typing. 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., $\mathsf{mutex}(T)$ holds).

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

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

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., $\mathsf{blocked}(T, n)$ 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 $m \in [0, z-1]$, $k = (m+1) \bmod z$ and $o = (k+1) \bmod z$. According to definition of *deadlocked state*, thread m acquires lock ι_k and waits for lock ι_m, whereas thread k acquires lock ι_o and waits for lock ι_k. Assume that m is the first of the z threads that acquires a lock so it acquires lock ι_k, before thread k acquires lock ι_o.

Let us assume that $S_y;T_y$ is the configuration once ι_o is acquired by thread k for the first time, ϵ_{1y} is the corresponding lockset of ι_o ($\epsilon_{1y} = \mathsf{lockset}(\iota_o, 1, E[\mathsf{pop}_{\gamma_y}\ \square])$) and ϵ_{2y} is the set of all heap locations ($\epsilon_{2y} = \mathsf{dom}(S_y)$) at the time ι_o is acquired. Then, ι_k does not belong to ϵ_{1y}, otherwise thread k would have been blocked at the lock request of ι_o as ι_k is already owned by thread m.

Let us assume that when thread k attempts to acquire ι_k, the configuration is of the form $S_x;T_x$. According to the assumption of this lemma that all configurations are well typed so $S_x;T_x$ is well-typed as well. By inversion of the typing derivation of $S_x;T_x$, we obtain the typing derivation of thread $n_k : \theta_k; E_k[\mathtt{lock}_{\gamma'_k}\ \mathtt{loc}_{\iota_k}]$: $\mathtt{lock}_{\gamma'_k}\ \mathtt{loc}_{\iota_k}$ is well-typed with input-output effect $(\gamma'_k;\gamma''_k)$, where $\kappa = \gamma'_k(\iota_k@n')$, $\kappa \geq (1,1)$, $\gamma'' = \gamma'_k, (\iota_k@n')^{\kappa-(1,0)}$, and $\mathsf{lockset\_ok}(E_k[\mathsf{pop}_{\gamma''_k}\ \square], \theta_k)$ holds, where θ_k is the access list of thread k. $\mathsf{lockset\_ok}(E_k[\mathsf{pop}_{\gamma''_k}\ \square], \theta_k)$ implies $\mathsf{lockset}(\iota_o, n_2, E_k[\mathsf{pop}_{\gamma''_k}\ \square]) \cap \epsilon_1 \subseteq \epsilon_2$, where $\theta_k = \theta'_k, \iota_o \mapsto n_1; n_2; \epsilon_1; \epsilon_2$ (notice that n_2 is positive, $\epsilon_2 = \epsilon_{1y}$ and $\epsilon_1 = \epsilon_{2y}$ — this is immediate by the operational steps from $S_y;T_y$ to $S_x;T_x$ and rule *E-LK0*).

We have assumed that m is the first thread to lock ι_k at some step before $S_y;T_y$, thus $\iota_k \in \mathsf{dom}(S_y)$ (the store can only grow — this is immediate by observing the operational semantics rules). By the definition of *lockset* function and the definition of γ''_k we have that $\iota_k \in \mathsf{lockset}(\iota_o, n_2, E_k[\mathsf{pop}_{\gamma''_k}\ \square])$. Therefore, $\iota_k \in \mathsf{lockset}(\iota_o, n_2, E_k[\mathsf{pop}_{\gamma''_k}\ \square]) \cap \mathsf{dom}(S_y) \subseteq \epsilon_{1y}$, which is a contradiction. □

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 $T = T_1, n:\theta; e$ for some T_1. By inversion of the typing derivation of $S;T$ we have that $M;\emptyset;\emptyset \vdash_t \theta; e : \langle\rangle \& (\gamma;\gamma')$, $\mathsf{mutex}(T)$, and $M \vdash S$.

If e is a *value* then by inversion of $M;\emptyset;\emptyset \vdash_t \theta; e : \langle\rangle \& (\gamma;\gamma')$, we obtain that $\gamma = \gamma'$, $E[e] = \Box[()]$ and $\forall \iota.\theta(\iota) = (0,0)$, as a consequence of $\forall r^\kappa \in \gamma.\kappa = (0,0)$ and $\mathsf{counts\_ok}(\Box[\mathsf{pop}_\gamma\ \Box], \theta)$. 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 thread typing derivation for e we obtain that $M;\emptyset;\emptyset \vdash u : \tau \& (\gamma_a;\gamma_b)$, $M;\emptyset;\emptyset \vdash E : \tau \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \& (\gamma;\gamma')$, $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\ \Box], \theta)$ hold.

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

Case $(\lambda x.e'\ \mathtt{as}\ \tau\ v)^{\mathsf{par}}$: it suffices to show that $(\theta_1, \theta_2) = \mathsf{split}(\theta,\ \mathsf{max}(\gamma_c))$ is defined, where γ_c is the nnotation of type τ. If $\mathsf{max}(\gamma_c)$ is empty, then the proof is immediate from the base case of split function. Otherwise, we must show that for all ι, the count $\theta(\iota)$ is greater than or equal to the sum of all $(\iota@n)^\kappa$ in $\mathsf{max}(\gamma_c)$. This can be shown by considering $\mathsf{par} \vdash \gamma_b = \gamma_a \oplus \gamma_c$ (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 $(\lambda x.e'\ \mathtt{as}\ \tau\ v)^{\mathsf{par}}$, and the exact correspondence between static (γ_b) and dynamic counts (i.e, $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\ \Box], \theta)$). Thus, rule *E-SN* can be applied to perform a single step.

Case $\mathtt{share}\ \mathtt{loc}_\iota$: $\mathsf{counts\_ok}(E[\mathsf{pop}_{\gamma_b}\ \Box], \theta)$ establishes an exact 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*. The cases for $\mathtt{release}\ \mathtt{loc}_\iota$ and $\mathtt{unlock}\ \mathtt{loc}_\iota$ can be shown in a similar manner.

Case $\mathtt{lock}_{\gamma_a}\ \mathtt{loc}_\iota$: similarly to the case we can show that $\theta(\iota) = (n_1, n_2)$ and n_1 is positive. If n_2 is positive, rule *E-LK1* can be applied. Otherwise, n_2 is zero. Let ϵ be equal to $\mathsf{locked}(T_1) \cap \mathsf{lockset}(\iota, 1, E[\mathsf{pop}_{\gamma_a}\ \Box])$. If ϵ is empty then rule *E-LK0* can be applied in order to perform a single step. Otherwise, $\mathsf{blocked}(T, n)$ predicate holds and the configuration is not stuck.

Case $\mathtt{deref}\ \mathtt{loc}_\iota$: 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. The case of $\mathtt{loc}_\iota := v$ can be shown in a similar manner. □

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

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 $S' = S$, $T' = T, n:\theta; E[\mathsf{pop}_{\gamma_a}\ e_1[v/x]]$ and $e = (\lambda x.e_1\ \mathtt{as}\ \tau_1 \xrightarrow{\gamma_c} \tau_2\ v)^{\mathsf{seq}(\gamma_a)}$. By inversion of the configuration typing assumption we have that $\mathsf{mutex}(T, n:\theta; E[e])$ and $M;\emptyset;\emptyset \vdash_t \theta; E[e] : \langle\rangle \& (\gamma;\gamma')$ hold. It suffices to show that $\mathsf{mutex}(T, n:\theta; E[\mathsf{pop}_{\gamma_a}\ e_1[v/x]])$ and $M;\emptyset;\emptyset \vdash_t \theta; E[\mathsf{pop}_{\gamma_a}\ e_1[v/x]] : \langle\rangle \& (\gamma;\gamma')$ hold. The former is immediate from $\mathsf{mutex}(T, n:\theta; E[e])$ as no new locks are acquired. Now we proceed with the latter, which can be shown by proving that $M;\emptyset;\emptyset \vdash$

$\mathtt{pop}_{\gamma_a}\ e_1[v/x] : \tau_2' \& (\gamma_a;\gamma_b)$ holds. By inversion on the thread typing derivation $E[e]$ we have $M;\emptyset;\emptyset \vdash v : \tau_1' \& (\gamma_b;\gamma_b)$, $\mathsf{seq}(\gamma_a) \vdash \gamma_b = \gamma_a \oplus \gamma_c'$ and $M;\emptyset;\emptyset \vdash \lambda x.e_1\ \mathtt{as}\ \tau_1 \xrightarrow{\gamma_c} \tau_2 : \tau_1' \xrightarrow{\gamma_c'} \tau_2' \& (\gamma_b;\gamma_b)$, where $\tau_1' \xrightarrow{\gamma_c'} \tau_2' \simeq \tau_1 \xrightarrow{\gamma_c} \tau_2$. We can use proof by induction on the expression typing relation to show that if v is well typed with τ_1', then it is also well typed with τ_1 provided that $\tau_1 \simeq \tau_1'$. Therefore, $M;\emptyset;\emptyset \vdash v : \tau_1 \& (\gamma_b;\gamma_b)$ holds. By inversion of the function typing derivation we obtain that $\mathsf{seq}(\emptyset) \vdash \gamma_c \Rightarrow M;\emptyset;\emptyset, x : \tau_1 \vdash e_1 : \tau_2 \& (\mathsf{min}(\gamma_c);\gamma_c)$. $\mathsf{seq}(\emptyset) \vdash \gamma_c'$ (premise of $\mathsf{seq}(\gamma_a) \vdash \gamma_b = \gamma_a \oplus \gamma_c'$) and $\gamma_c \simeq \gamma_c'$ imply that $\mathsf{seq}(\emptyset) \vdash \gamma_c$ holds, thus $M;\emptyset;\emptyset, x : \tau_1 \vdash e_1 : \tau_2 \& (\mathsf{min}(\gamma_c);\gamma_c)$ holds. By applying the standard value substitution lemma on the new typing derivation of v we obtain that $M;\emptyset;\emptyset \vdash e_1[v/x] : \tau_2 \& (\mathsf{min}(\gamma_c);\gamma_c)$ holds. The application of rule *T-PP* implies that $M;\emptyset;\emptyset \vdash \mathtt{pop}_{\gamma_a}\ e_1[v/x] : \tau_2' \& (\gamma_a;\gamma_b)$ holds.

Case *E-LK0*, *E-LK1*, *E-UL*, *E-SH* and *E-RL*: these rules generate side-effects as they modify the reference/lock count of location ι. 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: $\mathtt{lock}_{\gamma_1}\ \mathtt{loc}_\iota$, $\mathtt{unlock}\ \mathtt{loc}_\iota$ $\mathtt{share}\ \mathtt{loc}_\iota$ or $\mathtt{release}\ \mathtt{loc}_\iota$. 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 θ differs with respect to θ' only in the one of the counts of ι (i.e., $\theta' = \theta[\iota \mapsto \theta(\iota) + (n_1, n_2)]$ and $\gamma_a(r) - \kappa = (n_1, n_2)$ — γ_a is the input effect of $E[u]$).

By inversion of the configuration typing assumption we have that:

- $\mathsf{mutex}(T, n : \theta; E[u])$: In the case of *E-UL*, *E-SH*, *E-LK1* and *E-RL* no new locks are acquired. Thus, $\mathsf{mutex}(T, n : \theta'; E[()])$ holds. In the case of rule *E-LK0*, a new lock ι is acquired (i.e., when the lock count of ι is zero) the precondition of *E-LK0* suggests that no other thread holds ι: $\mathsf{locked}(T) \cap \mathsf{lockset}(\iota, 1, E[\mathtt{pop}_{\gamma_a}\ \Box]) = \emptyset$. Thus, $\mathsf{mutex}(T, n : \theta'; E[()])$ holds.
- $M;\emptyset;\emptyset \vdash_t \theta; E[u] : \langle\rangle \& (\gamma;\gamma')$: By inversion we have that $M;\emptyset;\emptyset \vdash E : \langle\rangle \xrightarrow{\gamma_a;\gamma_b} \langle\rangle \& (\gamma;\gamma')$ and $M;\emptyset;\emptyset \vdash u : \langle\rangle \& (\gamma_a;\gamma_b)$, where $\gamma_b = \gamma_a, (\iota @ n')^\kappa$ for some n'. It can be trivially shown from the latter derivation that $M;\emptyset;\emptyset \vdash () : \langle\rangle \& (\gamma_a;\gamma_a)$. We can obtain from the typing derivation of E (proof by induction) that $M;\emptyset;\emptyset \vdash E : \langle\rangle \xrightarrow{\gamma_a;\gamma_a} \langle\rangle \& (\gamma;\gamma'')$, where $\gamma' = \gamma'', (\iota @ n')^\kappa$.
- $\mathsf{lockset\_ok}(E[\mathtt{pop}_{\gamma_b}\ \Box], \theta)$ and $\mathsf{counts\_ok}(E[\mathtt{pop}_{\gamma_b}\ \Box], \theta)$: By the definition of *lockset* function it can be shown that $\mathsf{lockset}(\mathtt{j}, n_b, E[\mathtt{pop}_{\gamma_a}\ \Box]) \subseteq \mathsf{lockset}(\mathtt{j}, n_b, E[\mathtt{pop}_{\gamma_b}\ \Box])$ for all $\mathtt{j} \neq \iota$ in the domain of θ' (n_b is the lock count of $\mathtt{j}$ in θ). The same applies for $\mathtt{j} = \iota$ in the case of rules *E-SH*, *E-RL* as the lock count of ι is not affected. In the case of rules *E-LK0*, *E-LK1*, *E-UL* we have $\mathsf{lockset}(\iota, n_b \pm 1, E[\mathtt{pop}_{\gamma_a}\ \Box])$, but this is identical to $\mathsf{lockset}(\iota, n_b, E[\mathtt{pop}_{\gamma_b}\ \Box])$ by the definition of *lockset*. Therefore $\mathsf{lockset\_ok}(E[\mathtt{pop}_{\gamma_a}\ \Box], \theta')$ holds. The predicate *counts_ok* $(E[\mathtt{pop}_{\gamma_b}\ \Box], \theta)$ enforces the invariant that the static counts are identical to the dynamic counts (θ) of ι. The lock count of θ is modified by ± 1 and γ_a differs with respect to γ_b by $(\iota @ n')^\kappa$. We can use this fact to show that $\mathsf{counts\_ok}(E[\mathtt{pop}_{\gamma_a}\ \Box], \theta')$. □

Lemma 4 (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 $\iota \in [0, n]$ $M_\iota \vdash S_\iota; T_\iota$ holds.*

Proof. Proof by induction on the number of steps *n* using Lemma 3. □

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 $S_0; T_0 \rightsquigarrow^n S_n; T_n$, then the resulting configuration $S_n; T_n$ is not stuck and T_n has not reached a deadlocked state.*

Proof. The application of Lemma 4 to the typing derivation of $S_0; T_0$ implies that for all steps from zero to n there exists an M_ι such that $M_\iota \vdash S_\iota; T_\iota$. Therefore, Lemma 1 implies that $\neg\mathsf{deadlocked}(T_n)$ 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 ($\mathtt{loc}_\iota$) are used in 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.

Acknowledgement

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

References

[1] Gérard Boudol (2009): *A Deadlock-Free Semantics for Shared Memory Concurrency*. In Martin Leucker & Carroll Morgan, editors: *Proceedings of the International Colloquium on Theoretical Aspects of Computing*, *LNCS* 5684, Springer, pp. 140–154, doi:10.1007/978-3-642-03466-4_9.

[2] Chandrasekhar Boyapati, Robert Lee & Martin Rinard (2002): *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*, ACM Press, New York, NY, USA, pp. 211–230, doi:10.1145/582419.582440.

[3] John Boyland (2003): *Checking Interference with Fractional Permissions*. In Radhia Cousot, editor: *Static Analysis: Proceedings of the 10th International Symposium, LNCS* 2694, Springer, pp. 55–72, doi:10.1007/3-540-44898-5_4.

[4] Edward G. Coffman, Jr., Michael J. Elphick & Arie Shoshani (1971): *System Deadlocks*. *ACM Comput. Surv.* 3(2), pp. 67–78, doi:10.1145/356586.356588.

[5] Cormac Flanagan & Martín Abadi (1999): *Object Types Against Races*. In Jos C. M. Baeten & Sjouke Mauw, editors: *International Conference on Concurrency Theory, LNCS* 1664, Springer, pp. 288–303, doi:10.1007/3-540-48320-9_21.

[6] Cormac Flanagan & Martín Abadi (1999): *Types for Safe Locking*. In: *Programming Language and Systems: Proceedings of the European Symposium on Programming, LNCS* 1576, Springer, pp. 91–108, doi:10.1007/3-540-49099-X_7.

[7] Prodromos Gerakios, Nikolaos Papaspyrou & Konstantinos Sagonas (2010): *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*, ACM Press, New York, NY, USA, pp. 15–26, doi:10.1145/1708016.1708020.

[8] Prodromos Gerakios, Nikolaos Papaspyrou & Konstantinos Sagonas (2010): *A Type System for Unstructured Locking that Guarantees Deadlock Freedom without Imposing a Lock Ordering*. Technical Report, National Technical University of Athens. Available at http://softlab.ntua.gr/~pgerakios/papers/reglock_deadlock_techrep10.pdf.

[9] Prodromos Gerakios, Nikolaos Papaspyrou & Konstantinos Sagonas (2011): *A Type and Effect System for Deadlock Avoidance in Low-level Languages*. In: *Proceedings of the ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, ACM Press, New York, NY, USA, pp. 15–28, doi:10.1145/1929553.1929558.

[10] Naoki Kobayashi (2006): *A New Type System for Deadlock-Free Processes*. In C. Baier & H. Hermanns, editors: *International Conference on Concurrency Theory, LNCS* 4137, Springer, pp. 233–247, doi:10.1007/11817949_16.

[11] Leslie Lamport (1979): *A New Approach to Proving the Correctness of Multiprocess Programs*. *ACM Transactions on Programming Languages and Systems* 1(1), pp. 84–97, doi:10.1145/357062.357068.

[12] Kohei Suenaga (2008): *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, LNCS* 5356, Springer, pp. 155–170, doi:10.1007/978-3-540-89330-1_12.

[13] Vasco Vasconcelos, Francisco Martin & Tiago Cogumbreiro (2010): *Type Inference for Deadlock Detection in a Multithreaded Polymorphic Typed Assembly Language*. In Alastair R. Beresford & Simon Gay, editors: *Proceedings of the Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, EPTCS* 17, pp. 95–109, doi:10.4204/EPTCS.17.8.

Appendix

A.1 Formalism Summary: Operational Semantics

$\mathsf{locked}(T)$	takes a list of threads T and returns a set of locations locked by threads in T.
$\theta +_\iota (n_1, n_2)$	updates the map θ so that the reference and lock counts of $\theta(\iota)$ are incremented by n_1 and n_2 respectively.
$\theta(\iota)$	returns the reference and lock counts of $\theta(\iota)$.
$(\theta_1, \theta_2) = \mathsf{split}(\theta, \mathsf{max}(\gamma_a))$	takes γ_a (the effect of a new thread) and θ and returns θ_1 and θ_2, such that the sum of the counts of each location in θ_1 and θ_2 equals the counts of the same location in θ.

$\mathsf{lockset}(\iota, n, E)$	traverses the evaluation context E and returns the future lockset for ι acquired n times, only examining frames of the form $\mathsf{pop}_\gamma\ \square$. The traversal ends when E is empty or n is zero.

A.2 Formalism Summary: Static Semantics

$M;\Delta \vdash \tau$	well-formedness judgement within a typing context $M;\Delta$ for type τ.
$M;\Delta \vdash r$	well-formedness judgement within a typing context $M;\Delta$ for location r.
$\vdash M;\Delta;\Gamma;\gamma_1;\gamma_2$	well-formedness judgement for typing context $M;\Delta;\Gamma$ and effect $(\gamma_1;\gamma_2)$.
$\xi \vdash \gamma$	ensures that pure capabilities are not aliased within γ. In the case of parallel application (i.e., $\xi = \mathsf{par}$), 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.
$\gamma(r)$	returns the most recent (i.e., rightmost) occurence of r within effect γ.
$\mathsf{max}(\gamma')$	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 $\gamma(r)$ for any r in the domain of γ.
$\mathsf{min}(\gamma')$	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.
$\gamma \setminus r$	takes γ and r and removes all occurences of r' from γ such that r' is identical to r modulo the tags of constant locations.
$\xi \vdash \gamma' = \gamma \oplus \gamma_1$	takes 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 $\xi \vdash \gamma$.
$\kappa \geq \kappa'$	true if both counts of κ are no smaller than the correspoding counts of κ'.
$\kappa + \kappa'$, $\kappa - \kappa'$	calculate the sum and difference of two capabilities (considered here as two-dimensional vectors).
$\mathsf{summary}(\gamma)$	used primarily for calculating the summarized effects of recursive functions.
$\tau \simeq \tau'$	true when τ and τ' are structurally equivalent after removing $@n$ annotations from locations.
$M;\Delta;\Gamma \vdash E : \tau \xrightarrow{\gamma_a;\gamma_b} \tau' \,\&\, (\gamma_1;\gamma_2)$	the evaluation typing context judgement that takes the typing context $M;\Delta;\Gamma$, the evaluation context E, the expected effect $(\gamma_a;\gamma_b)$ 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 $(\gamma_a;\gamma_b)$.

A.3 Formalism Summary: Type Safety

$\mathsf{blocked}(T, n)$	true when thread n of thread list T is in a blocked (i.e, waiting for a lock) state.
$\mathsf{mutex}(T)$	true when each lock is held by at most one thread of T.
$\mathsf{counts\_ok}(E, \theta)$	takes 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 θ. It establishes an exact correspondence between dynamic and static counts.
$\mathsf{lockset\_ok}(E, \theta)$	takes 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.