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, 9, 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, 9, 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 $\mathtt{lk}(\iota)$. The same applies to lock handle variables. The effect system tracks the order of lock operations on handles or variables and determines whether all threads acquire locks in the same order.

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

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

In a few words, there are two parallel threads which acquire two different locks, x and y, in reverse order. When trying to find a partial order $\leq$ on locks for this program, the type system or static analysis tool will deduce that $x \leq y$ must be true, because of the first thread, and that $y \leq x$ must be true, because of the second. Thus, the program will be rejected, both in the system of Flanagan and Abadi which requires annotations [5] and in the system of Kobayashi which employs inference [9] as there is no single lock order for *both* threads. Similar considerations apply to the more recent works of 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:

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

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_{z} y;      y := y + x;
                     unlock x;
                     lock_∅ z;        z := z + y;
                     unlock z;
                     unlock y
in   f a a b
```

(a)

```
lock_{a} a;      a := a + 1;
lock_{b} a;      a := a + a;
unlock a;
lock_∅ b;        b := b + a;
unlock b;
unlock 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 released, and so on. If x and y were different, the run-time system would deduce that between this

```
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)

```
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)

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

Expression	e	::=	$x \mid \mathtt{c} \mid f \mid (e\ e)^{\xi} \mid (e)[\rho] \mid e := e$
			$\mid\ \mathtt{deref}\ e \mid \mathtt{let}\ \rho, x = \mathtt{ref}\ e\ \mathtt{in}\ e$
			$\mid\ \mathtt{share}\ e \mid \mathtt{release}\ e \mid \mathtt{lock}_{\gamma}\ e$
			$\mid\ \mathtt{unlock}\ e \mid ()$
Function	f	::=	$\lambda x.e\ \mathtt{as}\ \tau \xrightarrow{\gamma} \tau \mid \Lambda\rho.\ f$

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

Figure 3: Language syntax.

`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

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 supports explicit region polymorphism. Monomorphic functions must be annotated with their type, which carries their overall effect. Application is annotated with a *calling mode* which differentiates normal (sequential) application from parallel application, i.e., the spawning of a new thread. Sequential application is further annotated with the *continuation effect*, as mentioned earlier. The construct $\mathtt{let}\ \rho, x = \mathtt{ref}\ e_1\ \mathtt{in}\ e_2$ 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 $\mathtt{ref}(\rho, \tau)$, 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. Assignment and dereference operators are standard.

At any given program point, each cell is associated with a *capability*, which roughly consists of two natural numbers: the *capability counts*. The first number is the *cell reference* count (n_1), which denotes whether the cell is live, and the second is the *cell lock* count (n_2), which denotes whether the cell has been locked by the current thread. (We use natural numbers, instead of booleans, to support sharing and re-entrant locks.) Furthermore, a capability can be *impure* (denoted by $\overline{n_1, n_2}$), which allows for cell aliasing in the same spirit as in fractional permissions [3]. This aliasing information is required to determine whether it is safe to pass lock capabilities to new threads. The remaining language constructs operate on a cell reference and modify its capability: `share` and `release` increase and decrease n_1,

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

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

$$\frac{\Delta;\Gamma \vdash e_1 : \tau_1 \,\&\, (\gamma_2 \setminus \rho;\gamma') \qquad \gamma_1 = \gamma_2, \rho^{1,1} \quad \Delta \vdash \tau \qquad \Delta, \rho;\Gamma, x : \mathtt{ref}(\tau_1,\rho) \vdash e_2 : \tau \,\&\, (\gamma, \rho^{0,0};\gamma_1)}{\Delta;\Gamma \vdash \mathtt{let}\ \rho, x = \mathtt{ref}\ e_1\ \mathtt{in}\ e_2 : \tau \,\&\, (\gamma;\gamma')} \ \textit{(T-NG)}$$

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

Figure 4: Selected typing rules.

respectively, whereas `lock` and `unlock` do the same for n_2. As mentioned in the previous section, the run-time system inspects the annotation on `lock` to determine whether it is safe to lock a cell.

We now briefly discuss the most interesting parts of our type and effect system. Effects are used to statically track cell capabilities. An effect is an *ordered list* of elements of the form ρ^{κ} and represents a sequence of operations that affect the capabilities of various cells. The typing relation is denoted by $\Delta;\Gamma \vdash e : \tau \,\&\, (\gamma;\gamma')$, where Δ and Γ form the typing context, γ is the input effect, and γ' is the output effect. The reader should bear in mind two deviations from standard practice in type and effect systems. First, as each lock operation must be annotated with the "future" lockset, *effects flow backwards* through typing: the input effect to expression e represents the operations that follow after e is evaluated, and the output effect is the combined effect of the expression *and* its future. Second, as effects must reflect the exact order of cell operations, typing rules do not update effects, but rather append to them. Therefore, the input effect is *always* a prefix of the output effect.

A few selected typing rules are given in Figure 4. The typing rule for function application (*T-A*) joins the input effect γ and the function's effect γ_a, which contains the entire history of events occurring in the function body. In the case of parallel application, the function's return type must be unit, whereas in sequential application the annotation is checked against the input effect. The premise $\xi \vdash \gamma_a$ enforces a number of soundness restrictions, e.g., that pure capabilities are not aliased. In the rule for assignment (*T-AS*), the premises ensure that the referenced cell has positive reference and lock counts; in other words, that ρ is live and locked after the evaluation of e_1 and e_2. The rule for the `lock` operator (*T-LK*) checks that the annotation matches the input effect. It also checks that the cell is locked *after* the `lock` operation and makes sure to remove one from the lock count, in the output effect. Finally, the rule for creating new cells (*T-NG*) checks that the new cell is properly released (and unlocked) in the input effect, and makes sure to initialize the new cell with capability $(1,1)$, before the evaluation of e_2 starts.

For well typed programs, the safety theorem in our system guarantees three things: *memory safety* (and definite release of memory resources), *race freedom* (and definite release of locks), and *deadlock freedom*. A full formalization for our language, containing the operational semantics and a proof sketch, are given in the companion technical report [8].

References

[1] G. Boudol. A deadlock-free semantics for shared memory concurrency. In *Proc. of the International Colloquium on Theoretical Aspects of Computing*, vol. 5684 of *LNCS*, pp. 140–154. Springer, 2009.

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

[3] J. Boyland. Checking interference with fractional permissions. In *Static Analysis: Proc. of the 10th International Symposium*, vol. 2694 of *LNCS*, pp. 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 *Concurrency Theory: Proc. of the 10th International Conference*, vol. 1664 of *LNCS*, pp. 288–303. Springer, 1999.

[6] P. Gerakios, N. Papaspyrou, and K. Sagonas. A concurrent language with a uniform treatment of regions and locks. In *Proc. of the Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, 2009. Extended version will appear in *EPTCS*, 2010.

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

[8] P. Gerakios, N. Papaspyrou, and K. Sagonas. A type system for unstructured locking that guarantees deadlock freedom without imposing a lock ordering. Technical report, National Technical University of Athens, 2010.

[9] N. Kobayashi. A new type system for deadlock-free processes. In *CONCUR 2006*, vol. 4137 of *LNCS*, pp. 233–247. Springer, 2006.

[10] K. Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In *Proc. of the Asian Symposium on Programming Languages and Systems*, vol. 5356 of *LNCS*, pp. 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 *Proc. of the Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software*, 2009. Extended version will appear in *EPTCS*, 2010.