Structural Lock Correlation with Ownership Types

Yi Lu, John Potter, and Jingling Xue

Programming Languages and Compilers Group
School of Computer Science and Engineering
University of New South Wales, Sydney, NSW 2052, Australia
{ylu,potter,jingling}@cse.unsw.edu.au

Abstract. Concurrent object-oriented programming languages coordinate conflicting memory accesses through locking, which relies on programmer discipline and suffers from a lack of modularity and compile-time support. Programmers typically work with large libraries of code whose locking behaviours are not formally and precisely specified; thus understanding and writing concurrent programs is notoriously difficult and error-prone. This paper proposes *structural lock correlation*, a new model for establishing structural connections between locks and the memory locations they protect, in an ownership-based type and effect system. Structural lock correlation enables modular specification of locking. It offers a compiler-checkable lock abstraction with an enforceable contract at interface boundaries, leading to improved safety, understandability and composability of concurrent program components.

1 Introduction

Despite the progress in modern multicore architectures, it remains a challenge to develop better programming languages for concurrent programming. This is especially so for concurrent object oriented programming, where the combination of shared object memory and the endemic use of object aliasing pose special challenges. Data races are a common problem, which occur when two concurrent computations can access the same memory location without synchronisation and one of those accesses is a write. Races often imply violations of program invariants; achieving race freedom is crucial for the safety of concurrent programs.

Most concurrent object-oriented languages use mutual exclusion locks to synchronise concurrent memory accesses to avoid data races. But programming with locks is not easy: too little locking may not preserve program safety, while too much locking compromises concurrency and increases the chances of deadlock. A fundamental difficulty is that locking is a whole program requirement, which is hard to localise to a single class or module. All code that accesses shared memory, regardless of who developed it or where it is deployed, must be coordinated. Unfortunately the programmer typically works with large libraries whose locking behaviours are not precisely specified and checked. Understanding and writing properly synchronised code is notoriously difficult and error-prone.

Type systems for safe locking [9, 10, 3, 2, 8] (sometimes called *lock types*) have been used to enforce a fixed locking discipline across all code—all accesses to a shared object (or its fields) must hold a programmer-specified lock—this "memory guarded-by lock" relationship is called *lock correlation*. While quite useful, fixed lock correlations are often less flexible. For example, in lock type systems, once lock correlations are specified, those locks must be acquired even in sequential code where they are clearly unnecessary. This highlights the lack of context sensitivity in lock types. Moreover, as part of interface specification, locks required in lock correlations must be published unnecessarily; this tends to break the abstraction and information hiding principles underlying good software design, or may inhibit the use of fine-grained locks.

In this paper we present an effect-based approach for modular reasoning about locking behaviours, by specifying lock correlations as computational effects, called *lock effects*, or simply *effects*. Furthermore, by adopting an effect system based on *ownership types* [23, 7, 5], we are able to exploit the ownership tree structure to track lock correlations, even when we hide details of the locks and effects in lock effect abstractions. Memory side-effects are modelled as collections of subtrees in the object ownership tree. For every computation, we capture an approximation of the actual memory side-effects together with any locks that may be held while those effects are occurring. Ownership permits precise local descriptions of effects, which may depend on localised, private data. We can also easily abstract the details of effects, by approximating the actual effect using owners of the objects involved. This allows coarser, but still potentially useful information about local computations to be exported to a broader setting.

Lock effects help programmers choose locks as needed, depending on context. But modular specification of lock correlations is not easy: previous approaches are either not modular or encourage the breakdown of program abstractions. The problem of how to abstract lock details remains. Our solution is a new concept, *structural lock correlation*, where the side-effects are entirely within the ownership subtree rooted at the lock. In structural lock correlation, *the lock owns the correlated side-effects*. The major benefit of this concept is that it allows lock abstraction that preserves structural lock correlation. To reason about conflicting effects, the only detail about the lock that needs to be preserved is its ownership depth, or *rank* as we call it.

A trivial example illustrates some of these ideas; we also highlight another benefit of structural lock correlations in checking race freedom for concurrent computations. For simplicity, we use `par` as a lexically scoped parallel task construct [12, 1] to introduce concurrency, and a sync expression analogous to Java's synchronised statements, typical of related work in this area:

```
par {
  sync (l1) { o1.f = ... };
  sync (l2) { o2.f = ... };
}
```

Each of the two tasks synchronises on a lock before updating an object. The two lock correlations are: `<l1::o1>` and `<l2::o2>`. To show that the tasks cannot

conflict (race), it is sufficient to know either (i) `o1` and `o2` *must not alias*, implying that they are distinct objects, or (ii) `l1` and `l2` *must alias*, implying that the two object accesses are made mutually exclusive by a common lock.

With ownership there is a third way to show safety of the two tasks. Suppose we know that `l1` owns `o1` and `l2` owns `o2`. Then we can conclude that this code is safe: if `l1` and `l2` are not aliases, then `o1` and `o2` are not, because they are in distinct ownership subtrees. if `l1` and `l2` are aliases, then two tasks are correctly synchronised. Essentially this just relies on the ownership structure; we do not need object encapsulation or reference confinement, often enforced in ownership type systems. This third way of showing safety is why structural lock correlation is useful, especially when we combine it with a form of lock abstraction.

Such lock abstraction enables better support for modular specification of lock correlations, facilitating understanding about locking requirements. Programmers, or compilers, can reason about where locking is needed or not. Lock effects provide an enforceable contract at interface boundaries, contributing to improved safety and composability for concurrent program components, at least for those which admit a structural locking policy.

The paper is organised as follows. Section 2 discusses background and informally introduces our model. Section 3 explains the model with examples. Section 4 formally presents a core language and associated type and effect system. Section 5 presents a dynamic semantics and some properties of the type system. Section 6 discusses related work. Section 7 concludes the paper.

2 A Model of Structural Lock Correlation

We briefly review relevant concepts from ownership types and effects, before we introduce our model for structural lock correlation.

2.1 Ownership-based Effects

Ownership types provide static information about object structures. Effect systems provide various kinds of behavioural abstractions, most commonly dealing with memory access. Ownership-based effects use ownership trees to specify the extent of effects. Considerable work has been published on ownership-based effects with various applications, for example [23, 7, 5, 20, 3, 6, 4, 16, 15, 17, 1]. Unlike early papers on ownership types [23, 7, 5, 20], we do not use ownership to encapsulate objects, with owners as access monitors for the objects they own. Rather, we use ownership here purely to establish structural relationships between objects; every object has a fixed owner, being another object or world, the top of the ownership tree. As usual for ownership type systems, object types specify their ownership context k which may be world, a class owner parameter p or a final expression including this (later, we provide a detailed syntax in the formal system of Section 4). Direct ownership is the covering relation for object containment: we write $k_1 \preceq k_2$ to say that k_1 is inside k_2, so that $\preceq$ is the reflexive, transitive closure of the (acyclic) ownership relation.

We adopt simple defaults to reduce annotation requirements. Class definitions with a single ownership parameter may omit the parameter and use the keyword owner within the class in its place. In type declarations, omitted owners default to owner; thus, by default, objects refer to other objects as peers, having the same owner. So, by default, we attain relatively flat ownership structures, with explicit use of this as owner to specify containment.

Side-effects ε are similar to those in JOE [6]; they are *writes* by default, with *reads* optional. Side-effects are expressed as regions π which are levelled forests of ownership trees. Used as a region, the ownership context k denotes the whole subtree of objects (reflexively) owned by k. Similarly, the region $k{+}n$ denotes the levelled forest of all ownership trees at depth n below k, that is, the set of all objects whose n'th owner is inside k. For convenience, we use the keyword peer to abbreviate region owner+1, which contains all objects with the same owner as this. Intuitively, based on the underlying object sets, we have the following subregion relations: this $\sqsubseteq$ peer $\sqsubseteq$ owner as illustrated on the left of Fig. 1.

Ownership allows us to summarise effects that occur inside an object using a single identifier k to denote the whole set of objects which would not otherwise be statically expressible (conventional region-based effect systems use regions for the same purpose). When we cannot name an ownership context because it is out of scope, we do not want to lose any effects specified with it. The role of $k{+}n$ is to provide ownership abstraction so that we can lift effect specifications to a wider scope, basing the specification on a context higher up in the ownership tree. Regions are just sets of objects, so the notion of subregion follows naturally, which then leads to a standard notion of side-effect abstraction.

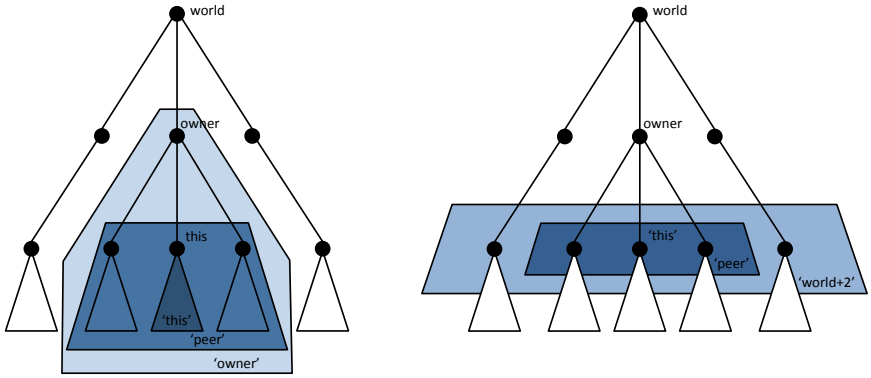

Fig. 1. Comparison of ownership effect abstraction and lock abstraction. Nodes are objects; upward edges link objects to their owners in the ownership tree. On the left, shaded polygons are side-effect regions named in quotation marks. On the right, shaded polygons are lock contexts that bound an existentially abstracted lock.

2.2 Structural Lock Correlation

Our lock effects $L::\varepsilon$ denote lock correlations where the lockset L guards the side-effect ε. We have two kinds of lock correlations: *arbitrary* and *structural*. *Arbitrary lock correlations* are conventional, being analogous to lock correlations used in lock types. They require a fixed, concrete lock or lockset to guard the effect. They can be abstracted by taking a subset of the lockset and/or an abstracted side-effect. To maintain useful information about the lock guards, we do not want to remove them from the specification, but that is the only way locks can be abstracted for arbitrary lock correlations. This is the problem with the conventional kind of lock correlation: locks cannot (usefully) be abstracted.

In *structural lock correlations* the lock must own all of the associated side-effects. A structural lock is bracketed to distinguish it from an arbitrary lock. For example, the lock effect $[\omega]::\varepsilon$ indicates that the structural lock ω is held when side-effect ε occurs, where ω contains ε. Two structural lock correlations with structural locks at the same rank (depth in the ownership tree) are guaranteed to be correctly synchronised. Two locks at the same rank in the ownership tree are either aliased, in which case, mutual exclusion is provided, or not, in which case, the objects they own cannot be aliased.

It follows that it is safe to abstract structural locks to a superset of locks at the same level. We overload the $k{+}n$ syntax to denote a lock context comprising all locks (objects) whose n'th owner is k. The right-hand part of Fig. 1 illustrates structural lock abstraction, $\ll$, capturing the nesting of lock contexts at the same rank; it is defined formally in Section 4. Recall that `peer` abbreviates `owner`+1.

Structural lock correlations allow knowledge of the fixed correlation between the lockset and side-effect to be retained when structural locks and side-effects are abstracted. This allows us to lift lock effects to scopes where we cannot precisely name the actual locks or side-effects. It is the key to achieving modularity in our specifications of lock effects.

3 Examples

We illustrate structural lock correlation with a simple bank account with a balance field and customers with a collection of accounts in Fig. 2 (see also [10, 3]). Customers can deposit given amounts into an account with a given index. We provide a variety of deposit methods to illustrate varying granularities of locking: the `depositA` method provides no synchronisation, `depositB` synchronises on `this`, and `depositC` synchronises on the account to be modified.

The Customer class has different locking policies for its different methods. We would like to specify lock effects so that clients will know how to use the different methods safely. In this paper we focus on lock effects rather than the underlying concurrency model. In order to demonstrate concurrent executions in the client code above, we use a simple `par` construct, which supports fork-join style parallelism [12, 1]. In each of the par blocks above, the two customers, `c` and `d`, attempt to deposit concurrently.

```
class Account { int balance = 0; }

class Customer {
  private final Account[] accounts;
  ...
  void depositA(int i, int x) {
    Account acct = account[i];
    acct.balance += x;
  }
  void depositB(int i, int x) {
    Account acct = account[i];
    sync (this) acct.balance += x;
  }
  void depositC(int i, int x) {
    final Account acct = account[i];
    sync (acct) acct.balance += x;
} }
```

```
// CLIENT CODE
Customer c, d; int i, j, x, y;
...
// case SeqA
c.depositA(i, x);  d.depositA(j, y);

// case ParA
par { c.depositA(i, x);
      d.depositA(j, y); }

// case ParB
par { c.depositB(i, x);
      d.depositB(j, y); }

// case ParC
par { c.depositC(i, x);
      d.depositC(j, y); }
```

Fig. 2. A bank accounts example with different granularities of locking.

Note that the client code in case ParA is unsafe, because the operation `+=` is not atomic and there is no synchronisation. If `c` and `d` are aliased, referring to the same customer, and if the account indices `i` and `j` are the same, then there is a possible data race on the account balance. Any attempt to check correct synchronisation should be able to detect this error.

In lock types, all shared variables must be guarded by a specified lock, which yields a fixed lock correlation. Case SeqA may be erroneously flagged as an error even if it is single threaded, because the `balance` is not guarded. This highlights the lack of context sensitivity in lock types. Either ParB or ParC may be accepted by lock types when a corresponding lock (either the customer or the account) is specified to guard `balance`. But the lock correlation specification is fixed, so lock types cannot accept both ParB and ParC simultaneously. We demonstrate how lock effects with structural lock correlation can deal with all these cases.

Structural Lock Correlation without Ownership We first illustrate our lock effects without any real ownership structure, where (by default), all objects are owned by world. The method effects for the three methods are:

```
depositA has effect <peer>
depositB has effect <this::peer>
depositC has effect <[peer]::peer>
```

Note that we do not consider the read effect on field `accounts` in method effects because the field is read-only and causes no conflict.

The effect for `depositA` indicates no lockset. The actual effect of the method could be expressed more precisely, within the method, as `<acct>`, but such an effect is not visible outside the method, as the local variable is out of scope. To export the effect from the method body, we abstract it to `<peer>`, since both the customer and the account objects are owned by the same (default) owner.

The effect for `depositB` indicates we are locking `this`, and the side-effect is the same as for `depositA`. When a client makes a call on a customer's `depositB` method, `this` is replaced by the actual customer, so the client knows that the call will acquire a lock on the customer object to guard its effect. This demonstrates the use of an arbitrary lock; there is no structural correlation, because the effect on the account is not contained within the lock of the customer.

The effect of `depositC` demonstrates a structural lock correlation. Inside the body of `depositC`, the lock effect is simply `<acct::acct>`. Because `acct` is inside `acct` ($\preceq$ is reflexive), we are permitted to abstract the arbitrary lock `acct` to a structural lock `[acct]`. So, by the same effect abstraction as above, the method effect becomes `<[peer]::peer>`. Despite the fact that the lock effects are no longer precise, the structural lock correlation is preserved. This abstract form of specification states that a method call will protect any of its effects (which must be within the peer forest) by acquiring some lock within the band of peers.

For the client, case SeqA is accepted, because the calls are made sequentially. Case ParA is rejected as it should be, because the side effects of the calls may conflict. In case ParB, `c` and `d` are the actual locks that are acquired. We also see that the effects of the two calls are the same, so they may conflict. Unfortunately the customers `c` and `d` may be different, this case must be rejected conservatively.

Case ParC is OK, perhaps surprisingly at the first glance. The reason is simple: we know both calls acquire a lock within the band of peers—the locks may be the same, or not. But we also know that the effect of each call is contained within the respective locks (that's the structural correlation at work). With the flat ownership of this example, that actually implies that the effect is the same as the lock (as we saw before we abstracted the method body effect above). There are two cases. If the locks are the same object, then they prevent the two calls from running concurrently—so there's no conflict. On the other hand, if the locks are not the same, then the effects are not the same—so no conflict.

For simplicity, when the structural lock is the same as the side-effect in a lock effect, we often omit the side-effect. For example, we write `<[peer]>` as a shorthand for `<[peer]::peer>`.

Structural Lock Correlation with Ownership Now we can fix case ParB by using ownership types to establish a structural correlation between a lock (the customer) and its effects (the customer's accounts). Within the `Customer` class, we declare that all the accounts are owned by this customer:

```
private final Account<this>[] accounts;
```

The new method effects for the three methods are:

```
depositA has effect <this+1>
depositB has effect <[this]::this+1>
depositC has effect <[this+1]>
```

The effect for `depositA` indicates the account object is owned by the customer. For the same reason, the effect for `depositB` can now be written as a structural

$$
\begin{array}{lrl}
\text{Programs} & P & ::= \overline{C}\ e \\
\text{Classes} & C & ::= \mathsf{class}\ c\langle\overline{p}\rangle\ [\mathsf{extends}\ t]_{opt}\ \{\overline{[\mathsf{final}]_{opt}\ t\ f};\ \overline{M}\} \\
\text{Types} & t & ::= c\langle\overline{k}\rangle \\
\text{Methods} & M & ::= t\ m(t\ x)\ \varphi\ \{e\} \\
\text{Expressions} & e & ::= z \mid \mathsf{new}\ t \mid e.f \mid e.f = e \mid e.m(e) \mid \mathsf{sync}\ e\ e \mid \mathsf{par}\ e\ e \\
\text{Variables} & z & ::= x \mid \mathsf{this} \\
\text{Effects} & \varphi & ::= \emptyset \mid \varepsilon \mid L{::}\varphi \mid \varphi \cup \varphi \\
\text{Side-effects} & \varepsilon & ::= \mathsf{rd}\ \pi \mid \pi \\
\text{Regions} & \pi & ::= k \mid k{+}n \mid e{\rightarrow}f \\
\text{Owner contexts} & k & ::= \mathsf{world} \mid p \mid e \\
\text{Locksets} & L & ::= \emptyset \mid e \mid [\omega] \mid L \cup L \\
\text{Lock contexts} & \omega & ::= k \mid k{+}n \\
\text{Environments} & \Gamma & ::= \emptyset \mid \Gamma, p \mid \Gamma, z : t \\
\text{Identifiers} & & c, p, f, m, x \\
\text{Ownership depth} & & n \in \mathbb{N}
\end{array}
$$

$$
k{+}0 \equiv k \qquad \emptyset{::}\varphi \equiv \varphi \qquad L{::}\emptyset \equiv \emptyset
$$
$$
L{::}L'{::}\varphi \equiv (L \cup L'){::}\varphi \qquad L{::}(\varphi \cup \varphi') \equiv L{::}\varphi \cup L{::}\varphi'
$$

Fig. 3. Abstract syntax, with syntactic equivalences ($\equiv$).

lock correlation `<[this]::this+1>`. Similarly, the effect of `depositC` can be written as `<[this+1]::this+1>` or the shorthand `<[this+1]>`.

The same analysis results for SeqA and ParA are obtained as before. However, both ParB and Par C are now accepted as being correctly synchronised. In ParB, we know that the calls will acquire `c` and `d` as locks; they may or may not be aliased. If they are aliased, then both calls attempt to acquire the same lock. Now if they are not aliased, the two calls may run concurrently. The effect of each call is contained within the associated lock. We know that both customers are at the same ownership rank, and so, because they are distinct, the structure of ownership effects ensures that the two effects do not overlap, as required for safety. ParC can be argued similarly, but now the lock is some account object inside the customer, its owner. In brief, our motto is: *each lock protects its own.*

4 The Type System for Structural Lock Correlation

We present a small Java-like language, similar to those used in other ownership type systems [23, 7, 6, 4, 16, 15, 17], and incorporate lock effects.

The syntax for the core language is given is in Fig. 3. The identifiers name classes, ownership parameters, fields, methods and formal arguments of methods respectively. A program is a collection of classes with a main expression. Each class has a list of ownership parameters, an optional super type, and a list of field declarations and method definitions. Fields may be optionally declared final, as in Java. Types are simply classes with actual ownership bindings. Methods declare a return type and a single argument (for simplicity) and a lock effect corresponding to the method body. Expressions are straightforward, with reading of variables and fields, field assignments and method calls. There is no explicit sequential construct $(e; e)$—assignment and method calls already demonstrate sequential

evaluation of subexpressions. For simplicity, like most formal type and effect systems, we omit conditionals and loops. The parallel construct is used to present formal properties about data races in a concurrent setting; formally it is simpler to illustrate soundness for our techniques using lexically scoped parallelism as provided by par, but this choice is not fundamental to our approach. A $\mathsf{sync}\ e_1\ e_2$ expression synchronises on the object referenced by e_1 to guard against other concurrent execution while evaluating e_2.

The lock effects φ record the correlation $L::\epsilon$ between the possible side-effects on memory and the set of locks (if any) guarding those memory accesses. The syntactic equivalences defined in Fig. 3 ensure that lock effects can always be normalised as a set of $L::\epsilon$ lockset/side-effect correlations. In the lockset syntax L, e is an arbitrary lock, while $[\omega]$ is the existential abstraction of a lock participating in a structural lock correlation. Side-effects are captured as read or write effects on a region of memory. A region π is an ownership subtree k, a levelled forest $k{+}n$, or a named object field $e{\rightarrow}f$. Named object fields allow fine-grained effects to be identified; as a side-effect they represent access to a single named field. Ownership contexts k have an ownership relation; they are either object-valued expressions, ownership parameters, or the fixed ownership root world. Every expression/object/type has an associated owner. The ownership-based region k denotes the tree of objects owned by the context k, directly or indirectly. The forest $k{+}n$ denotes all trees rooted at ownership depth n inside k, which we can describe in terms of iterated ownership, as $\{e \mid \mathit{owner}^i_\Gamma(e) = k \text{ for } i \geq n\}$. The *owner* of e is formally defined in the next section, and is dependent on the typing environment Γ for e. We merge cases, by noting that ownership trees $k{+}0$ and k are identical regions.

The ownership type system is largely standard, except that we do not enforce encapsulation[23, 7, 5, 20, 3, 6]. Alternatively, existential ownership, based on our earlier scheme [16, 15] could provide for more liberal reference to owned objects. We could also extend our system with constraints for disjointness and rank equivalence on ownership parameters for extra expressiveness. We omit such extensions in this paper to focus on the key novelty: structural lock correlation.

Well-formed Program, Class and Method $\quad \vdash P \quad \vdash C \quad \Gamma \vdash M$

$$\frac{\vdash \overline{C} \qquad \emptyset \vdash e : t\ !\ \varphi}{\vdash \overline{C}\ e}\ [\text{PROGRAM}] \qquad \frac{\Gamma = \overline{p}, \mathsf{this} : c\langle\overline{p}\rangle \qquad \Gamma \vdash \overline{M} \qquad \Gamma \vdash t, \overline{t} \qquad \mathit{owner}(c\langle\overline{p}\rangle) = \mathit{owner}(t)}{\vdash \mathsf{class}\ c\langle\overline{p}\rangle\ [\mathsf{extends}\ t]_{opt}\ \{[\overline{\mathsf{final}}]_{opt}\ t\ f;\ \overline{M}\}}\ [\text{CLASS}]$$

$$\frac{\Gamma \vdash t, t' \qquad \Gamma, x : t' \vdash e : t\ !\ \varphi \qquad \mathit{defin}(\Gamma(\mathsf{this}), \mathsf{this}) = \ldots\ [\mathsf{extends}\ t'']_{opt} \qquad \mathit{method}(m, t'', \mathsf{this}, x) = t_0\ t'_0\ \varphi' \ldots \qquad t \leq t_0 \qquad t'_0 \leq t' \qquad \Gamma \vdash \varphi \sqsubseteq \varphi'}{\Gamma \vdash t\ m(t'\ x)\ \varphi\ \{e\}}\ [\text{METHOD}]$$

Our type and effects system starts with top-level constructs. The judgements are given for a fixed program P. By [PROGRAM], a well-formed program has well-formed classes and a well-typed main expression. In [CLASS], a class checks its supertype, method definitions and field types in an environment containing the

class's ownership parameters and a self-type binding for this. As is standard in ownership types, the owner of the class is the same as in its supertype; this prevents ownership information from being lost in type subsumption. With a slight abuse of notation, we omit those parts of the rule's antecedent that rely on the optional supertype when it is not present. Rule [METHOD] uses the auxiliary definitions to look up method definitions with appropriate bindings. The rule checks that a method body has the declared return type and effect, given the argument type declarations. A method is covariant in its return type and effect, and contravariant in its arguments; this is checked relative to the inherited version of the method, if any, instantiated with the self-type and method arguments.

$$\frac{L = \mathsf{class}\ c\langle\overline{p}\rangle\ ...}{\mathit{defin}(c\langle\overline{k}\rangle, e) = [\overline{k/p}, e/\mathsf{this}]L}\ [\text{LOOKUP-DEFIN}] \qquad \frac{\mathit{defin}(t,e) = ...\ \mathsf{extends}\ t'\ \{\overline{[\mathsf{final}]_{opt}\ t\ f}\ ...\ \}}{\mathit{fields}(t,e) = \overline{[\mathsf{final}]_{opt}\ t\ f}, \mathit{fields}(t',e)}\ [\text{LOOKUP-FIELDS}]$$

$$\frac{\mathit{defin}(t,e) = ...\ t'\ m(t''\ x)\ \varphi\ \{e''\}\ ...}{\mathit{method}(m,t,e,e') = t'\ t''\ [e'/x]\varphi\ [e'/x]e''}\ [\text{LOOKUP-METHOD}] \qquad \frac{\Gamma \vdash e : t\ !\ \varphi}{\mathit{owner}_\Gamma(e) = \mathit{owner}(t)}\ [\text{LOOKUP-OWNER}]$$

$$\frac{\mathit{defin}(t,e) = ...\ \mathsf{extends}\ t'\ \{\ ...\ ;\overline{M}\}\qquad (...\ m(...)\ ...) \notin \overline{M}}{\mathit{method}(m,t,e,e') = \mathit{method}(m,t',e,e')}\ [\text{LOOKUP-METHOD-EXT}] \qquad \frac{}{\mathit{owner}(c\langle k\ ...\rangle) = k}\ [\text{LOOKUP-OWNER-TYPE}]$$

The auxiliary lookup functions resolve type instances for fields and methods selected in a target application, by binding this and its type appropriately. The recursive definition [LOOKUP-FIELDS] unwinds inherited field definitions, terminating when a top-level class with no supertype is reached. Method lookup is split into separate recursive and base cases. The first context parameter of a type is its owner [LOOKUP-OWNER-TYPE]; the owner of an object denoted by an arbitrary expression is determined from the type of the expression [LOOKUP-OWNER].

Expression Typing $\quad \Gamma \vdash e : t\ !\ \varphi$

$$\frac{\Gamma \vdash e : t\ !\ \varphi \qquad (t'\ f) \in \mathit{fields}(t,e)}{\Gamma \vdash e.f : t'\ !\ \varphi \cup \mathsf{rd}\ e{\rightarrow}f}\ [\text{SELECTION}] \qquad \frac{\Gamma \vdash e : t\ !\ \varphi \qquad \Gamma \vdash e' : t'\ !\ \varphi' \qquad (t'\ f) \in \mathit{fields}(t,e) \qquad \Gamma \vdash t'}{\Gamma \vdash e.f = e' : t'\ !\ \varphi \cup \varphi' \cup e{\rightarrow}f}\ [\text{UPDATE}]$$

$$\frac{\Gamma \vdash e : t\ !\ \varphi \qquad (\mathsf{final}\ t'\ f) \in \mathit{fields}(t,e)}{\Gamma \vdash e.f : t'\ !\ \varphi}\ [\text{FINAL}]$$

$$\frac{\Gamma \vdash e : t\ !\ \varphi \qquad \Gamma \vdash e' : t'\ !\ \varphi' \qquad \Gamma \vdash t' \qquad \mathit{method}(m,t,e,e') = t''\ t'\ \varphi''\ ...}{\Gamma \vdash e.m(e') : t''\ !\ \varphi \cup \varphi' \cup \varphi''}\ [\text{CALL}] \qquad \frac{z : t \in \Gamma}{\Gamma \vdash z : t\ !\ \emptyset}\ [\text{VARIABLE}] \qquad \frac{\Gamma \vdash t}{\Gamma \vdash \mathsf{new}\ t : t\ !\ \emptyset}\ [\text{NEW}]$$

$$\frac{\Gamma \vdash e : t\ !\ \varphi \qquad \Gamma \vdash e' : t'\ !\ \varphi'}{\Gamma \vdash \varphi\ \#\ \varphi'}\ [\text{PARALLEL}] \qquad \frac{\Gamma \vdash e \qquad \Gamma \vdash e' : t\ !\ \varphi}{\Gamma \vdash \mathsf{sync}\ e\ e' : t\ !\ e{::}\varphi}\ [\text{SYNCHRONISATION}] \qquad \frac{\Gamma \vdash e : t'\ !\ \varphi' \qquad \vdash t' \leq t}{\Gamma \vdash \varphi' \sqsubseteq \varphi}\ [\text{SUBSUMPTION}]$$

The form of judgment for expressions is more or less standard for effect systems, where each rule checks both the type and the behavioural effect for expressions. The type judgements are unsurprising. [SYNCHRONISATION] states that the type of sync $e\ e'$ is the same as e', its guard (lock) expression e must be final thus having

no effect itself, and the lock e is correlated with the effect of e'. [SYNCHRONISATION] is the only rule that affects the lock part of an effect. [PARALLEL] also yields the type of the second expression, in line with our (arbitrary) choice of the value of a par pair.

In general, the effect of an expression takes the union of its subexpression effects, with any additional effect of the particular kind of expression. Only object field access yields a side-effect: a read effect on the field in [SELECTION] and a write in [UPDATE]; these are the most specific side-effects, and can be abstracted via [SUBSUMPTION] which allows a subeffect to be replaced by a super effect in a judgment, thus losing precision, but possibly extending the visibility of the effect to a broader scope. In [CALL] the target binding of the method definition determines the additional lock effect φ'' of the call execution. Finally [PARALLEL] asserts that a par effect is simply the union of the subeffects of the parallel subexpressions. However, this judgment is only valid if the two subeffects do not conflict: $\varphi \# \varphi'$, as discussed in the next section.

Well-formed Types and Subtyping $\quad \Gamma \vdash t \quad \vdash t \leq t$

$$\frac{\Gamma \vdash \overline{k}}{\Gamma \vdash c\langle\overline{k}\rangle}\ [\text{TYPE}] \qquad \frac{\mathit{defin}(c\langle\overline{k}\rangle, \_) = \ldots \,\mathsf{extends}\ t}{\vdash c\langle\overline{k}\rangle \leq t}\ [\text{SUBTYPE-EXT}] \qquad \frac{}{\vdash t \leq t}\ [\text{SUBTYPE-REFL}] \qquad \frac{\vdash t \leq t'' \quad \vdash t'' \leq t'}{\vdash t \leq t'}\ [\text{SUBTYPE-TRANS}]$$

The rules for well-formed types and subtypes are standard for ownership types. Types can only be formed from well-formed contexts, and class inheritance and parameter substitution provides the base for the subtype relation. Well-formedness of types is checked wherever type declarations are explicit (in [CLASS] and [METHOD]) and wherever object references are created or bound (in [NEW], [UPDATE] and [CALL]).

Well-formed Contexts and Final Expressions $\quad \Gamma \vdash k \quad \Gamma \vdash_{\mathsf{final}} e : t$

$$\frac{p \in \Gamma}{\Gamma \vdash p}\ [\text{CONTEXT-FORMAL}] \qquad \frac{}{\Gamma \vdash \mathsf{world}}\ [\text{CONTEXT-WORLD}] \qquad \frac{\Gamma \vdash_{\mathsf{final}} e : t}{\Gamma \vdash e}\ [\text{CONTEXT-FINAL}]$$

$$\frac{z : t \in \Gamma}{\Gamma \vdash_{\mathsf{final}} z : t}\ [\text{FINAL-VARIABLE}] \qquad \frac{\Gamma \vdash_{\mathsf{final}} e : t \qquad (\mathsf{final}\ t'\ f) \in \mathit{fields}(t, e)}{\Gamma \vdash_{\mathsf{final}} e.f : t'}\ [\text{FINAL-FIELD}]$$

Only final expressions are allowed as well-formed contexts by [CONTEXT-FINAL]. Variables (read-only method parameters and this) are final, as are final field expressions, where the target object is accessed via another final expression.

Nonconflict $\quad \Gamma \vdash \varphi \# \varphi$

$$\frac{\Gamma \vdash \varepsilon \# \varepsilon'}{\Gamma \vdash L{::}\varepsilon \# L'{::}\varepsilon'}\ [\text{NONCONF-SIDEEFF}] \qquad \frac{\Gamma \vdash e}{\Gamma \vdash e{::}\varepsilon \# e{::}\varepsilon'}\ [\text{NONCONF-ARBITRARY}] \qquad \frac{\Gamma \vdash \omega \approx \omega'}{\Gamma \vdash [\omega]{::}\varepsilon \# [\omega']{::}\varepsilon'}\ [\text{NONCONF-STRUCTURAL}]$$

$$\frac{\Gamma \vdash L'{::}\varepsilon \# \varphi \quad L' \subseteq L}{\Gamma \vdash L{::}\varepsilon \# \varphi}\ [\text{NONCONF-LOCKSET}] \qquad \frac{}{\Gamma \vdash \emptyset \# \varphi}\ [\text{NONCONF-}\emptyset] \qquad \frac{\Gamma \vdash \varphi \# \varphi' \quad \Gamma \vdash \varphi \# \varphi''}{\Gamma \vdash \varphi \# \varphi' \cup \varphi''}\ [\text{NONCONF-UNION}] \qquad \frac{\Gamma \vdash \varphi' \# \varphi}{\Gamma \vdash \varphi \# \varphi'}\ [\text{NONCONF-SYM}]$$

Nonconflicting lock effects offer race-free concurrency. We presume that lock effects have been normalised to a set of lockset/side-effect correlations. The first three rules handle base cases, dealing with nonconflict between a pair of correlations, in three distinct ways: (i) by [NONCONF-SIDEEFF], if the side-effects are nonconflicting; (ii) by [NONCONF-ARBITRARY], if the same arbitrary lock exists in the lockset; and (iii) by [NONCONF-STRUCTURAL], if there are two structural locks with the same rank. Case (i) lifts nonconflict for side-effects to nonconflict for lock effects. Case (ii) is the standard notion that concurrent memory access must be protected by a common lock. Case (iii) is the key novelty. The structural lock correlation tells us that both side-effects are each protected by their own lock at the same ownership rank. There are two possibilities: either both existential locks are the same, in which case we are back to case (ii), or the two existential locks are distinct, in which case the corresponding ownership trees do not overlap. But being structural locks, each tree contains their respective side-effects, and so we are back to case (i). So there are no conflicts for both possibilities. By [NONCONF-UNION], all correlation pairs in the cross product of two lock effects must be nonconflicting in order for the two effects to be nonconflicting.

Subeffecting $\Gamma \vdash \varphi \sqsubseteq \varphi$

$$\frac{\Gamma \vdash \varepsilon \sqsubseteq e}{\Gamma \vdash e{::}\varepsilon \sqsubseteq [e]{::}\varepsilon}\ [\text{SUBEFF-STRUCTURAL}] \qquad \frac{\Gamma \vdash \omega \ll \omega'}{\Gamma \vdash [\omega]{::}\varepsilon \sqsubseteq [\omega']{::}\varepsilon}\ [\text{SUBEFF-ABSTRACT}] \qquad \frac{L' \subseteq L \qquad \Gamma \vdash \varepsilon \sqsubseteq \varepsilon'}{\Gamma \vdash L{::}\varepsilon \sqsubseteq L'{::}\varepsilon'}\ [\text{SUBEFF-LOCKSET}]$$

$$\frac{\Gamma \vdash \varphi \sqsubseteq \varphi'' \qquad \Gamma \vdash \varphi'' \sqsubseteq \varphi'}{\Gamma \vdash \varphi \sqsubseteq \varphi'}\ [\text{SUBEFF-TRANS}] \qquad \frac{\Gamma \vdash \varphi \sqsubseteq \varphi'' \qquad \Gamma \vdash \varphi' \sqsubseteq \varphi'''}{\Gamma \vdash \varphi \cup \varphi' \sqsubseteq \varphi'' \cup \varphi'''}\ [\text{SUBEFF-UNION}]$$

The rule [SUBEFF-STRUCTURAL] asserts that e is acting as a structural lock in the lock effect: it owns its correlated effect. Such a lock can be replaced by the existential form $[e]$. Of itself, this is of little use, but by [SUBEFF-ABSTRACT] these structural locks may be abstracted, by moving up the ownership tree (see [RANK-OWNER] as discussed below), while preserving the rank where the actual lock exists; again, this abstraction increases the scope where the effects are visible. These two rules are the only means by which abstract structural correlations are introduced. Such abstraction preserves lock correlation and rank equivalence of the locks in a structural lock (see Lemmas 2 and 3 in Section 5). By [SUBEFF-LOCKSET] we can move to a supereffect by removing parts of the lock (because it is safe to lose locking information in supereffects), or by abstracting the side-effect.

Lock Abstraction and Rank Equivalence $\Gamma \vdash \omega \ll \omega \quad \Gamma \vdash \omega \approx \omega$

$$\frac{owner_\Gamma^{n-m}(k) = k' \qquad m \leq n}{\Gamma \vdash k{+}m \ll k'{+}n}\ [\text{LOCK-ABSTRACT}] \qquad \frac{\exists i \cdot owner_\Gamma^{i-m}(k) = owner_\Gamma^{i-n}(k')}{\Gamma \vdash k{+}m \approx k'{+}n}\ [\text{RANK-OWNER}]$$

Recall that when $k{+}n$ is a region π, it denotes the forest of subtrees at ownership rank n below k. However, when $k{+}n$ is a lock context ω, it is used to assert that a structural lock exists at a rank n below k. In other words, allowed contexts

for the lock are the roots of the region's forest. That is, the ω interpretation of $k{+}n$ is $\{e \mid owner^n_\Gamma(e) = k\}$.

The [RANK-OWNER] rule asserts that two lock contexts are at the same rank, if they are at the same rank below a common owner. Intuitively, we can abstract a structural lock to another lock context which contains all of the possible contexts of the original lock context. This loses precision about the possible values, by allowing extra contexts. However, in order to be able to reason about nonconflict of structural locks, we must ensure that all of the contexts are at the same rank. Hence [LOCK-ABSTRACT] defines $\Gamma \vdash k{+}m \ll k'{+}n$ for $m \leq n$ so that k' is the owner of k at the rank that ensures that both lock contexts appear at rank n below k'. Because k' owns k, $k'{+}n$ contains all of $k{+}m$, as required.

Structural lock abstraction preserves structural lock correlation, as already noted. It is worth noting that lock abstraction is only possible for locksets comprising a singleton (structural) lock. Any other (arbitrary) locks must be elided before the abstraction is allowed; if those locks are actually needed to demonstrate nonconflict for lock effects, then it is pointless to elide them, and hence pointless to aim for structural lock abstraction. This impacts on the design of synchronisation policies: to achieve abstraction of the locks, it is necessary, at least in our model, to focus on the use of ownership for structural locking, and avoid mixing structural and arbitrary locking.

Side-Effects Disjointness and Subsumption $\quad \Gamma \vdash \varepsilon \sqsubseteq \varepsilon \quad \Gamma \vdash \varepsilon \# \varepsilon$

$$\frac{\Gamma \vdash \pi \sqsubseteq \pi'}{\Gamma \vdash \mathsf{rd}\ \pi \sqsubseteq \mathsf{rd}\ \pi'}\ [\text{SUB-SIDE-EFF-RD}] \qquad \frac{\Gamma \vdash \pi \sqsubseteq \pi'}{\Gamma \vdash \mathsf{rd}\ \pi \sqsubseteq \pi'}\ [\text{SUB-SIDE-EFF-RD-WR}]$$

$$\frac{}{\Gamma \vdash \mathsf{rd}\ \pi \# \mathsf{rd}\ \pi'}\ [\text{SIDE-EFF-RD}] \qquad \frac{\Gamma \vdash \pi \# \pi'}{\Gamma \vdash \mathsf{rd}\ \pi \# \pi'}\ [\text{SIDE-EFF-RD-WR}]$$

Read side-effects may be subsumed by default (write) side-effects. Read side-effects do not conflict with each other. A read (or write) side-effect will not conflict with another write side-effect if their corresponding regions are disjoint.

Region Disjointness and Subsumption $\quad \Gamma \vdash \pi \sqsubseteq \pi \quad \Gamma \vdash \pi \# \pi$

$$\frac{\exists\, i \cdot i + m \geq n \quad owner^i_\Gamma(k) = k'}{\Gamma \vdash k{+}m \sqsubseteq k'{+}n}\ [\text{SUBREG-FOREST}] \qquad \frac{\Gamma \vdash e}{\Gamma \vdash e{\to}f \sqsubseteq e}\ [\text{SUBREG-FIELD}]$$

$$\frac{}{\Gamma \vdash \pi \sqsubseteq \mathsf{world}}\ [\text{SUBREG-WORLD}] \qquad \frac{f \neq f'}{\Gamma \vdash e{\to}f \# e'{\to}f'}\ [\text{DISJOINT-FIELD}]$$

$$\frac{\Gamma \vdash e \otimes e'}{\Gamma \vdash e{\to}f \# e'{\to}f'}\ [\text{DISJOINT-FIELD-ALIAS}] \qquad \frac{\Gamma \vdash k \approx k' \quad \Gamma \vdash k \otimes k'}{\Gamma \vdash k \# k'}\ [\text{DISJOINT-RANK-EQ}]$$

$$\frac{\Gamma \vdash \pi \sqsubseteq \pi'' \quad \Gamma \vdash \pi' \sqsubseteq \pi''' \quad \Gamma \vdash \pi'' \# \pi'''}{\Gamma \vdash \pi \# \pi'}\ [\text{DISJOINT-SUBREGION}]$$

By [SUBREG-FOREST], $\Gamma \vdash k{+}m \sqsubseteq k'{+}n$ just when k is owned by k', ensuring that the k region (subtree) is a subset of the k' subtree. The rank inequality ensures that $k{+}m$ has a deeper ownership rank than $k'{+}n$, thus ensuring that $k{+}m$'s region (forest) is a subset of $k'{+}n$'s. [LOCK-ABSTRACT] is a special case of [SUBREG-FOREST] with $i = n - m$. [SUBREG-FIELD] deals with a special case where the named field is subsumed by the target expression/context it belongs to.

Two named fields are disjoint as regions just when they have distinct names, or they belong to two objects accessed via expressions which must not be aliased.

The rule [DISJOINT-SUBREGION] asserts that subregions are disjoint if corresponding super regions are (that is, disjointness is preserved by forming subregions). Intuitively this matches the ownership tree model with subregions corresponding to subsetting of subtrees. To show disjointness we can apply this rule to lift one or both subregions to super regions k and k' at the same rank. This is clearly possible for regions of the form $k{+}m$ and $e{\rightarrow}f$. Then we test for disjointness by testing for non-aliasing of the two region roots k and k', according to [DISJOINT-RANK-EQ].

Nonaliasing $\Gamma \vdash k \otimes k \quad \Gamma \vdash t \otimes t$

$$\frac{owner_\Gamma(k) = k'}{\Gamma \vdash k \otimes k'}\ [\text{NONALIAS-OWNER-LEFT}] \qquad \frac{owner_\Gamma(k') = k}{\Gamma \vdash k \otimes k'}\ [\text{NONALIAS-OWNER-RIGHT}] \qquad \frac{\exists i \in 1..n \quad \cdot \quad \Gamma \vdash k_i \otimes k'_i}{\Gamma \vdash c\langle k_{1..n}\rangle \otimes c\langle k'_{1..n}\rangle}\ [\text{NONALIAS-CONTEXT}]$$

$$\frac{c \neq c' \qquad \mathsf{class}\ c\ \ldots\ [\mathsf{extends}\ c''\langle\ldots\rangle]_{opt}\ \ldots \qquad \mathsf{class}\ c'\ \ldots\ [\mathsf{extends}\ c''\langle\ldots\rangle]_{opt}\ \ldots}{\Gamma \vdash c\langle\overline{k}\rangle \otimes c'\langle\overline{k'}\rangle}\ [\text{NONALIAS-CLASS}]$$

$$\frac{\Gamma \vdash e : t\ !\ \varepsilon \qquad \Gamma \vdash e' : t'\ !\ \varepsilon'}{\Gamma \vdash t \otimes t'}\ [\text{NONALIAS-TYPE}] \qquad \frac{\vdash t \leq t'' \quad \vdash t' \leq t'''}{\Gamma \vdash t'' \otimes t'''}\ [\text{NONALIAS-SUBTYPE}]$$

We provide an ad hoc collection of rules that use a variety of simple techniques to show that two expressions must not alias the same object context. More advanced types for reasoning about nonaliasing exist (e.g. linearity or uniqueness), but we omit them to keep our model simpler for both formalism and programmers. An object cannot alias the objects it owns. Types cannot be aliased if any of their context parameters cannot alias, or if they have distinct class names and a common superclass, or are both top level classes, so that one cannot be a subtype of the other. [NONALIAS-CLASS] should be understood to mean the optional superclass is the same for both classes (that is, both are present and the same, or both are absent).

We show how to type check `ParB` in the bank example with structural lock correlation and ownership in Section 3, where we already know the method `depositB` has lock effect `<[this]::this+1>`. Because `c` and `d` are owned by the same (default) owner, they have the same type `Customer<owner>`. By [LOOKUP-OWNER], we have $owner(\texttt{c}) = owner(\texttt{d})$. Then by [RANK-OWNER], we have `c` $\approx$ `d`. When `depositB` is called in `ParB`, by [LOOKUP-METHOD], the two calls have the lock effects `<[c]::c+1>` and `<[d]::d+1>`. By [NONCONF-STRUCTURAL], we have `<[c]::c+1>` # `<[d]::d+1>`. Finally by [PARALLEL], we find `par c.depositB() d.depositB()` is well-typed. Note that, our type system allows non-final expressions in types and effects via substitution (e.g. `c` and `d`), but they are only used to look up their static types at compile time, and do not depend on runtime identities.

5 Dynamic Semantics

In this section, we define the dynamic behaviour of our language, and demonstrate that a well-typed program cannot exhibit data races.

$$\begin{array}{llll} \text{Locations} & l & & \\ \text{Variables} & z ::= \ldots \mid l & \text{Objects} & o ::= \overline{f \mapsto l} \\ \text{Expressions} & e ::= \ldots \mid \mathsf{synced}\ l\ e & \text{Heaps} & H ::= \overline{l \mapsto o_t} \end{array}$$

Fig. 4. Extended syntax for dynamic semantics.

The extended syntax which includes features required for the dynamic semantics is defined in Fig. 4. The syntax of variables is extended with runtime locations l. Since variables may appear in the syntax of expressions, contexts and environments (which may contain location typing), we do not define additional typing rules for locations except [NONALIAS-LOCATION]. Synchronised state $\mathsf{synced}\ l\ e$ indicates the lock l has been acquired when evaluating e. A heap H is a mapping from locations to objects with their types; an object o maps its fields to locations. In order to formalise the key properties of the type system, we establish a connection between the static and dynamic semantics by including ownership in the dynamic semantics (preserved in the types of objects in the heap). But the ownership information does not affect how expressions are evaluated so it does not have to be available at runtime.

Additional auxiliary definitions for dynamic semantics and properties are given below. [STATE] defines the consistency between type environments and runtime heaps; location typing provided in the type environment must match locations and their types in the heap which are well-formed by [OBJECT].

$$\text{[NONALIAS-LOCATION]}\quad \frac{l \neq l'}{\Gamma \vdash l \otimes l'} \qquad \text{[SYNCHRONISED]}\quad \frac{\Gamma \vdash e : t\ !\ \varphi}{\Gamma \vdash \mathsf{synced}\ l\ e : t\ !\ l ::: \varphi}$$

$$\text{[STATE]}\quad \frac{\Gamma \vdash e : t\ !\ \varphi \qquad H = \overline{l \mapsto o_t} \qquad \Gamma \vdash \overline{l \mapsto o_t}}{\Gamma \vdash (H; L; e) : t\ !\ \varphi}$$

$$\text{[OBJECT]}\quad \frac{\Gamma \vdash l : t\ !\ \emptyset \qquad \Gamma \vdash t \qquad \Gamma \vdash \overline{l : t\ !\ \emptyset} \qquad fields(t, l) = \overline{\ldots\ f}}{\Gamma \vdash l \mapsto (\overline{f \mapsto l})_t}$$

We present a small step operational semantics in Fig. 5 where each reduction step is considered atomic. Evaluation states contain a heap, a lock store and an expression to be evaluated. The lock store records all locks that are currently held. Like in the formalism of lock types [9], locks are not reentrant; that is, an expression cannot reacquire a lock that it already holds. The evaluation of a program starts in an initial state, $\emptyset; \emptyset; e$, where e is the body of the main method with empty heap and lock store. Evaluation then takes place according to the rules which specify the behaviour of the various constructs in the language.

We use the conventional form of evaluation contexts to reduce the number of evaluation rules:

$$\begin{array}{rl} E ::= & [\,] \mid E.f \mid E.f = e \mid l.f = E \mid E.m(e) \\ & \mid l.m(E) \mid \mathsf{sync}\ E\ e \mid \mathsf{par}\ E\ e \mid \mathsf{par}\ e\ E \end{array}$$

which defines the order of evaluation of subexpressions in compound terms, except for parallel expressions which evaluate nondeterministically via their choice of subexpression—this is a standard way to model concurrency [9, 8, 1].

$$\frac{H(l) = o_t \qquad (t'\ f) \in \mathit{fields}(t,l)}{H; L; l.f \xrightarrow{\mathsf{rd}\ l \to f} H; L; H(l)(f)}\ [\text{EVAL-SELECTION}] \qquad \frac{H(l) = o_t \qquad (\mathsf{final}\ t'\ f) \in \mathit{fields}(t,l)}{H; L; l.f \longrightarrow H; L; H(l)(f)}\ [\text{EVAL-FINAL}]$$

$$\frac{H' = H[l \mapsto H(l)[f \mapsto l']]}{H; L; l.f = l' \xrightarrow{l \to f} H'; L; l'}\ [\text{EVAL-UPDATE}] \qquad \frac{H(l) = o_t \qquad \mathit{method}(m,t,l,l') = \ldots\ e}{H; L; l.m(l') \longrightarrow H; L; e}\ [\text{EVAL-CALL}]$$

$$\frac{\begin{array}{c} l \notin \mathit{dom}(H) \qquad H_1 = H, l \mapsto \emptyset_t \qquad \overline{[\mathsf{final}]_{opt}\ t\ f} = \mathit{fields}(t,l) \\ \forall i \in 1..|\overline{f}| \quad \cdot \quad H_i; L; \mathsf{new}\ t_i \longrightarrow H_{i+1}; L; l_i \qquad H' = H_{|\overline{f}|+1} \end{array}}{H; L; \mathsf{new}\ t \longrightarrow H'[l \mapsto (\overline{f \mapsto l})_t]; L; l}\ [\text{EVAL-NEW}]$$

$$\frac{l \notin L}{H; L; \mathsf{sync}\ l\ e \longrightarrow H; L \cup l; \mathsf{synced}\ l\ e}\ [\text{EVAL-ACQUISITION}] \qquad \frac{H; L; e \xrightarrow{\varphi} H; L'; e'}{H; L; \mathsf{synced}\ l\ e \xrightarrow{l::\varphi} H; L'; \mathsf{synced}\ l\ e'}\ [\text{EVAL-SYNCED}]$$

$$\frac{L' = L \setminus l}{H; L; \mathsf{synced}\ l\ l' \longrightarrow H; L'; l'}\ [\text{EVAL-RELEASE}] \qquad \frac{}{H; L; \mathsf{par}\ l\ l' \longrightarrow H; L; l'}\ [\text{EVAL-JOIN}] \qquad \frac{H; L; e \xrightarrow{\varphi} H'; L'; e'}{H; L; E[e] \xrightarrow{\varphi} H'; L'; E[e']}\ [\text{EVAL-CONTEXT}]$$

Fig. 5. Small step operational semantics: $H; L; e \xrightarrow{\varphi} H; L; e$.

The label φ on the transition is the effect that takes place during the transition (when $\emptyset$, it may be omitted). For simplicity, in [EVAL-NEW], we adopt the object creation semantics from [4] where all fields are initialised with new objects. This may not be the case in practice, but it does not affect our results because it has no side-effect. In [EVAL-ACQUISITION], the premise blocks unless the lock to be acquired is not held (i.e. not in the lock store); after the lock is acquired (recorded in $\mathsf{synced}\ l$) the expression e becomes active and may progress by [EVAL-SYNCED]. Note [EVAL-SYNCED] yields an effect guarded by the acquired lock l; this corresponds to [SYNCHRONISATION] in the static semantics. [EVAL-RELEASE] removes a lock from the held lockset. [EVAL-JOIN] ensures the order of sequential execution. The reduction rules for other expressions are standard.

Finally, we formalise some of the key properties of the type system and sketch their proofs. Theorem 1 asserts preservation of types and effects over the reduction of well-typed expressions.

Theorem 1 (Preservation)
If $\Gamma \vdash (H; L; e) : t\ !\ \varphi$ *and* $H; L; e \xrightarrow{\varphi'} H'; L'; e'$, *then* $\Gamma \vdash (H'; L'; e') : t\ !\ \varphi$ *and* $\Gamma \vdash \varphi' \sqsubseteq \varphi$.

Proof. The proof proceeds by structural induction on the derivation of term evaluation with a set of substitution lemmas, which is largely standard as seen in [5, 6, 14].

Theorem 2 states that parallel expressions have no conflict effects during their execution. Since parallel expressions are lexically structured, this theorem applies to any possible interleaves.

Theorem 2 (Nonconflict)
If $\Gamma \vdash (H; L; \mathsf{par}\ e_1\ e_2) : t\ !\ \varphi$, $H; L; e_1 \xrightarrow{\varphi_1} H_1; L_1; e_1'$ *and* $H; L; e_2 \xrightarrow{\varphi_2} H_2; L_2; e_2'$, *then* $\Gamma \vdash \varphi_1 \# \varphi_2$.

Proof. By [STATE], we have $\Gamma \vdash \mathsf{par}\ e_1\ e_2 : t\ !\ \varphi$. By [PARALLEL], we have $\Gamma \vdash e_1 : t_1\ !\ \varphi_1'$, $\Gamma \vdash e_2 : t_2\ !\ \varphi_2'$ and $\Gamma \vdash \varphi_1' \# \varphi_2'$. From [STATE], we know $\Gamma \vdash (H; L; e_1) : t_1\ !\ \varphi_1'$ and $\Gamma \vdash (H; L; e_2) : t_2\ !\ \varphi_2'$. By Theorem 1, we can have $\Gamma \vdash \varphi_1 \sqsubseteq \varphi_1'$ and $\Gamma \vdash \varphi_2 \sqsubseteq \varphi_2'$. Finally, by Lemma 1, we have the result.

Lemma 1 (Subeffects preserve nonconflict)
If $\Gamma \vdash \varphi_1 \sqsubseteq \varphi_1'$, $\Gamma \vdash \varphi_2 \sqsubseteq \varphi_2'$ *and* $\Gamma \vdash \varphi_1' \# \varphi_2'$, *then* $\Gamma \vdash \varphi_1 \# \varphi_2$.

Proof. By induction on the derivations of $\Gamma \vdash \varphi \# \varphi$. Case [NONCONF-STRUCTURAL] uses Lemmas 2 and 3. Other cases are straightforward.

Lemma 2 (Lock abstraction preserves subeffects and rank) *If* $\Gamma \vdash \omega \ll \omega'$, *then* $\Gamma \vdash \omega \sqsubseteq \omega'$ *and* $\Gamma \vdash \omega \approx \omega'$.

Proof. By [LOCK-ABSTRACT], $\omega = k{+}m$ and $\omega' = k'{+}n$ hold, where $owner_\Gamma^{n-m}(k) = k'$ and $0 \le m \le n$. Choosing $i = n$ in [RANK-OWNER] shows $\Gamma \vdash \omega \approx \omega'$. Finally, choosing $i = n - m$ in [SUBREG-FOREST] shows $\Gamma \vdash \omega \sqsubseteq \omega'$.

Lemma 3 (Lock abstraction preserves structural correlation)
If $\Gamma \vdash \omega \ll \omega'$ *and* $\Gamma \vdash \varepsilon \sqsubseteq \omega$, *then* $\Gamma \vdash \varepsilon \sqsubseteq \omega'$.

Proof. By Lemma 2 and [SUBEFF-TRANS].

Lemma 4 (Sub-side-effects preserve disjointness)
If $\Gamma \vdash \varepsilon_1 \sqsubseteq \varepsilon_1'$, $\Gamma \vdash \varepsilon_2 \sqsubseteq \varepsilon_2'$ *and* $\Gamma \vdash \varepsilon_1' \# \varepsilon_2'$, *then* $\Gamma \vdash \varepsilon_1 \# \varepsilon_2$.

Proof. Easy induction on the derivations of $\Gamma \vdash \varepsilon \# \varepsilon$.

Data race freedom means that parallel expressions cannot cause conflicting side-effects without synchronisation. We formalise race freedom to show that arbitrary interleaves of evaluations do not cause conflicting side-effects [8]. To facilitate the proofs, we introduce a lockset lookup function $locks(e)$ which simply extracts all the locks currently held by e (i.e. all $\mathsf{synced}\ l$ in e).

Theorem 3 (Race freedom)
Given $H; L; \mathsf{par}\ e_1\ e_2$ *is reachable from the initial state and* $\Gamma \vdash (H; L; \mathsf{par}\ e_1\ e_2) : t\ !\ \varphi$, *if* $H; L; e_1 \xrightarrow{L_1::\varepsilon_1} H_1; L_1; e_1'$ *and* $H; L; e_2 \xrightarrow{L_2::\varepsilon_2} H_2; L_2; e_2'$, *then* $\Gamma \vdash \varepsilon_1 \# \varepsilon_2$.

Proof. By Lemma 5, we have $locks(e_1) \cap locks(e_2) = \emptyset$. By Lemma 6, we have $L_1 \subseteq locks(e_1)$ and $L_2 \subseteq locks(e_2)$. By set disjointness, we have $L_1 \cap L_2 = \emptyset$. By Theorem 2, we have $\Gamma \vdash L_1{::}\varepsilon_1 \# L_2{::}\varepsilon_2$. By [NONCONF-SIDEEFF], we have the result.

Lemma 5 (Mutual exclusion)
If $H; L;$ par $e_1\ e_2$ *is reachable from the initial state, then* $locks(e_1) \cap locks(e_2) = \emptyset$.

Proof. Easy induction on the derivation of term evaluation. The only interesting case is [EVAL-ACQUISITION], which ensures at any time a lock can only be acquired at most once.

Lemma 6 (Lockset)
If $H; L_0; e \xrightarrow{L::\varepsilon} H'; L_0'; e'$, *then* $L \subseteq locks(e)$.

Proof. Let $L = l::L'$, we need to show $(l \cup L') \subseteq locks(e)$. We have $l \subseteq locks(e)$ by [EVAL-SYNCED] and $L' \subseteq locks(e)$ by induction.

6 Related Work

In earlier type systems for safe locking [9, 10], shared fields are *guarded by* a fixed lock. This follows the recommended programming practice, where program annotations are added to data declarations, to document which locks are intended to guard which data. Any code accessing a field must hold the specified lock, regardless of its context. Requiring a fixed lock to protect certain data is inflexible as it does not allow clients to devise their own synchronisation policies. For modularity, lock type systems ask the programmer to annotate a method declaration with a *requires* clause to specify the locks that must be held at each call-site of the method. The specified locks are then used to check to see if the method body is race-free or not. This requires effects to set up a precondition for the method, thereby placing a restriction on where this method can be used. In contrast, lock effects describe the computational behaviour of a method body.

Instead of associating locks with fields, [11] associates locks with hierarchical regions, protecting state across many objects. A lock object can be the object itself or one of its final fields; it may also be renamed to hide its representation from clients. Method effects can be specified as annotations in terms of *requires* clauses or side effects. Like lock types, it does not offer lock abstraction and is not context sensitive in checking unsynchronised access to state.

SafeJava [3, 2] relies on the object encapsulation enforced by an owners-as-dominators type system to provide a clear point for access control for safe locking. SafeJava is an extension to the lock types model, where the guarded-by requirement is specified implicitly for every field: the required lock is always the root of the ownership tree that the field belongs to. Only access to the root objects' fields and methods needs to be directly synchronised. Using universes types, [8] follows a fine-grained ownership-based locking convention, where an object's corresponding lock is its direct owner rather than any transitive root owner. But different levels of object access do not rely on higher-level locks, and must be separately synchronised. In general, these approaches rely on encapsulation, while our effect-based approach simply describes the effects of code and places no restriction on references. Our own work in [18] explores how to use

ownership types to infer synchronisation requirement for structured parallelism. It does not consider explicit locks and does not use lock effects.

Locksmith [24] is a static race checker for C programs. It performs a whole-program analysis, based on an effect system, to accumulate lock correlation constraints, which are then resolved to ensure that all memory accesses are consistently protected by a common lock. Locksmith's lock correlations are intended to be inferred by a static analysis tool—they are not geared towards modular specification of locking requirements, but rely on a global program analysis.

Chord [22] is a static race checker for Java programs, based on a context-sensitive may alias analysis. Unlike Locksmith, Chord omits linearity checking, and unsoundly assumes must-aliasing of locks at the same allocation site in order to reduce the number of false positives produced. The later work [21] adds a *disjoint reachability analysis* to distinguish correlated lock/object pairs allocated at the same allocation site but in different iterations of a loop. Intuitively, if a lock reaches (via one or more field dereferences) an object and they were allocated in the same iteration, then locks allocated in different iterations must reach different objects at the same allocation site. This introduces a form of *conditional must-not aliasing* relation: two objects are not aliased under the assumption that their respective protecting locks are not aliased. The structural lock correlation we propose in this paper achieves a form of conditional must-not aliasing relation by exploiting ownership structure, and establishes such a relation in terms of modular specifications enabled by ownership-based lock effects.

7 Conclusion

Concurrent programs are difficult to reason about. Usually their behaviours are only informally and imprecisely specified. Our effect-based model supports modular checking of lock usage. Programmers must express their high-level design intentions via effect contracts on methods, thus helping to avoid intention-related bugs—the most dominant bug category in real-world applications [13]. We rely on lock correlations where structural locks protect data they own. Structural lock correlation is preserved through abstraction, as checked by the ownership-based type and effect system. Our approach is flexible, allowing locking requirements to be satisfied differently in different contexts. Lock effects also serve as an enforceable contact at interface boundaries, contributing to safety and composability of program components.

This paper does not consider unstructured threads. Our paper [19] explored reasoning about threads in order to track what side-effects may occur in parallel. We capture the effects of active threads in a form that mimics the tree structure of thread creation; it then compares these effects with those of any subsequent expressions to detect potential interference in a flow-sensitive manner. Such a technique can be adapted to the ownership-based effects described in this paper to detect data races between threads, but we leave that for future work.

Acknowledgements. This research is supported by Australian Research Grants, DP0987236 and DP130101970.

References

1. R. L. Bocchino Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for Deterministic Parallel Java. In *OOPSLA*, 2009.
2. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *OOPSLA*, 2002.
3. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *OOPSLA*, 2001.
4. N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple Ownership. In *OOPSLA*, 2007.
5. D. Clarke. *Object Ownership and Containment*. PhD thesis, The University of New South Wales, Sydney, Australia, 2001.
6. D. Clarke and S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *OOPSLA*, 2002.
7. D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, 1998.
8. D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universe Types for Race Safety. In *VAMP*, 2007.
9. C. Flanagan and M. Abadi. Types for safe locking. In *ESOP*, 1999.
10. C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI*, 2000.
11. A. Greenhouse and W. L. Scherlis. Assuring and evolving concurrent programs: annotations and policy. In *ICSE*, 2002.
12. D. Lea. A Java fork/join framework. In *Java Grande*, 2000.
13. S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes - a comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, 2008.
14. Y. Lu. *Object Validity, Effects and Accessibility with Ownership*. PhD thesis, The University of New South Wales, Sydney, Australia, 2008.
15. Y. Lu and J. Potter. On ownership and accessibility. In *ECOOP*, 2006.
16. Y. Lu and J. Potter. Protecting representation with effect encapsulation. In *POPL*, 2006.
17. Y. Lu, J. Potter, and J. Xue. Validity invariants and effects. In *ECOOP*, 2007.
18. Y. Lu, J. Potter, and J. Xue. Ownership types for object synchronisation. In *APLAS*, 2012.
19. Y. Lu, J. Potter, C. Zhang, and J. Xue. A type and effect system for determinism in multithreaded programs. In *ESOP*, 2012.
20. P. Müller and A. Poetzsch-Heffter. Universes: A type system for controlling representation exposure. *PLFP*, 1999.
21. M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *POPL*, 2007.
22. M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI*, 2006.
23. J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *ECOOP*, 1998.
24. P. Pratikakis, J. S. Foster, and M. Hicks. Locksmith: context-sensitive correlation analysis for race detection. In *PLDI*, 2006.