Ownership Types for Object Synchronisation

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. Shared-memory concurrent programming is difficult and error prone because memory accesses by concurrent threads need to be coordinated through synchronisation, which relies on programmer discipline and suffers from a lack of modularity and compile-time support. This paper exploits object structures, provided by ownership types, to enable a structured synchronisation scheme which guarantees safety and allows more concurrency within structured tasks.

1 Introduction

Shared-memory concurrent programming remains complex and error-prone, despite it having become essential in the multicore era. Most object-oriented (OO) languages adopt *unstructured parallelism*, where threads may be ubiquitously spawned with arbitrary lifetimes. It is notoriously difficult to catch concurrency errors, such as data-races, atomicity violations and deadlocks, because they are caused by unexpected thread interleaving. In practice, "most Java programs are so rife with concurrency bugs that they work only by accident" [15]. Lee argues that if we are to have any hope of simplifying parallel programming for the vast majority of programmers and applications, then parallel programming models must greatly constrain the possible interleaving of program executions [19].

Programming models with more disciplined parallelism have attracted recent interest. Based on hierarchical fork/join parallelism where tasks are managed lexically [14, 18], a number of structured parallel programming models [2, 10, 31] are emerging. They are able to forbid task interference in type systems or program analysis, guaranteeing deterministic behaviour [2, 31] or allowing the compiler to find implicit parallelism from sequential object-oriented programs [10]. While significantly simplifying concurrency, these models preclude task interference thereby having limited applicability to object-oriented programs with extensive sharing and mutation. Finally, by not allowing synchronisation, these models cannot allow task interference, which essentially reduces their application to immutable object applications; immutability makes concurrency easier, but there are undoubtedly applications that need to combine concurrency and mutability. In this paper, we use an ownership type and effect system to reason about *interference and synchronisation for parallel tasks.*

When there is potential interference between parallel tasks, we infer a *synchronisation requirement* which needs to be met by a suitable lock (or other synchronisation technique) to prevent the interference. Our type system ensures atomicity for each task, thereby guaranteeing serialisability of all tasks. When there is no interference between tasks, no synchronisation is necessary.

Meeting synchronisation requirements involves choices. Too little synchronisation may not preserve program safety, while too much synchronisation compromises concurrency and increases the chances of deadlock. Moreover, there is a tradeoff between the choices of synchronisation granularity. Choosing a coarse granularity results in less overhead (synchronisation requires runtime resources) but may reduce concurrency. For example, when we enforce synchronisation with coarse-grained locks, objects protected by the same lock can never be accessed in parallel (e.g. SafeJava, see Section 3.3). On the other hand, using a fine granularity increases the overhead of synchronisation. The fundamental problem here is that synchronisation is a whole program requirement, which is hard to localise to a single class or module. All code that accesses a piece of shared state must know and obey the same synchronisation convention regardless of who developed the code or where it is deployed. Large object-oriented programs are typically developed and deployed in a modular way by different programmers. As each programmer may follow their own conventions on synchronisation (e.g. some use finer granularity, others use coarser granularity) which are not formally or precisely specified, understanding and writing synchronised code is difficult and error prone.

We support modular reasoning about parallel tasks using ownership types and effects. Ownership types [8, 9] allow programmers to structure the object store hierarchically as an ownership tree; ownership-based effects [5, 6, 8] use ownership tree to reason about side effects in object-oriented programs and to capture potential conflict between tasks. A major contribution of the paper is the ability to automatically infer finer-grained synchronisation requirements for parallel tasks, sufficient to prevent any potential interference between them. Such implicit synchronisation eases the design and understanding of concurrent object-oriented programs by hiding lower level concurrency safety requirements. Code is less fragile.

Another key novelty comes from the combination of object ownership with structured task parallelism: when the two structures align well, we expect to see good concurrency. For example, a task synchronising on an object (coarser granularity) may be able to allow its subtasks to access the object's subobjects without explicit synchronisation, even though the sub-tasks may still be in conflict with external sub-tasks. Such structural refinement of granularity and interference-based on-demand synchronisation strategy can reduce overhead without needlessly sacrificing potential concurrency (as shown in the examples given in later sections). Moreover, with structured parallelism, it is simpler to achieve lock ordering so then there is no chance of deadlock from multiple locks.

This paper presents the basics of our model: Section 2 discusses ownership types, object aliasing and ownership-based effects; Section 3 introduces our structured parallel task model and discusses synchronisation requirements with an example; Section 4 formally presents static semantics and dynamic semantics of

the type system; Section 5 provides more discussion and reviews related work, followed by a brief summary of our contributions in Section 6. The complete formalism including rules and properties and synchronisation inference algorithms are included in an extended version of this paper [22].

2 Ownership Types and Effects in a Nutshell

Originally ownership types were proposed to provide object encapsulation [9, 8, 20, 24]. Here we use ownership types to describe effects as structured sets of objects, similar to what has been done by others [5, 8, 23, 6, 10], without enforcing encapsulation. First we provide a quick overview of ownership types, and indicate how our model of structured objects allows us to statically reason about objects being disjoint, guaranteeing non-aliasing properties for program variables. Then we discuss how program effects can be succinctly summarised using ownership structure. Ultimately our concern is to provide a practical means for reasoning about conflicting effects of concurrent behaviours.

2.1 Ownership Types and Aliasing

In an ownership type system, every object has a fixed owner, either world, for root objects, or another object; we say that an object is strictly inside its owner. The ownership relation establishes a dynamic tree structure, rooted at world, with every object created as a child of its owner. Ownership types allow us to reason statically about hierarchical object structures.

Owners are often called *(ownership) contexts* in ownership type systems. Classes are parameterised by formal context parameters, possibly with constraints such as inside $\preceq$, strictly inside $\prec$ or disjoint $\#$. The first formal context parameter of a class determines the owner of this object within the class. Types are formed by binding actual contexts to their class parameters; the first context argument of a type determines the actual owner of objects of the type. Within a class definition, contexts can use formal class parameters, the root context world, the current context this and final expressions (see Section 4). Consider the following Java-style example:

```
class Customer {
  final Account<this> sav = new Account<this> ();
  final Account<this> chq = new Account<this> ();
  void transfer(int amt) <this> { sav.withdraw(amt); chq.deposit(amt); }
}
```

The context argument of the Account type for the read-only `sav` and `chq` fields specifies the owner for the objects referenced by those fields. In this case, it is the current object **this** of class `Customer`, meaning that a customer owns savings and cheque accounts. We omit formal parameters of a class when none are used in its definition, as for Customer. Ownership types are often used for object encapsulation by constraining references. For instance, accounts cannot be directly accessed unless through their owning customer. This enables object

access control and localised reasoning on objects. However, in this paper, we do not enforce encapsulation or constrain references because our type system relies only on (ownership) effects: ownership determines object structure, and structured objects allow us to succinctly summarise effects. Encapsulation can still be enforced independently if desired.

Even without enforcing encapsulation, ownership types allow us to derive more distinctions for reasoning about object aliasing. We rely on this in two related ways: we can determine when effects are disjoint, and when synchronisation requirements are non-overlapping. For example, objects with different (i.e. non-aliased) owners must be non-aliased. Also, objects cannot alias their owners: $x \prec y$ implies $x \otimes y$ (we write this *must-not-alias* relation with $\otimes$). Beside ownership, there are a variety of type-based techniques for determining when reference variables must not alias the same object, such as uniqueness [17], linearity [32], regions [30], freshness and more. They can be used in our type system to enhance alias reasoning, but we do not consider all of them for the simplicity of the formalism. In the above code, we can determine that `sav` and `chq` refer to distinct objects, because they are initialised to fresh objects in their declaration, and are not permitted to be re-assigned. Alternatively, if we want `sav` and `chq` to be mutable fields, we can simply let them have different owners in order to distinguish the objects they refer to.

2.2 Ownership Effects

Effect systems [25] offer abstract views of program behaviours, identifying, for example, subsets of memory affected by read and write operations. The key idea for ownership effect systems is to specify memory effects on whole subtrees of objects. An effect on an object subsumes effects on objects that it owns. This allows effects to be abstracted to higher level contexts where the details of individual objects may be unknown.

For the Customer class above, the method `transfer` has effect **this**, which means the body of the method may access (read or write, directly or indirectly via calls) the fields of the current object and any object inside (transitively owned by) the current object. The body of the `transfer` method calls `sav.withdraw` to access the `balance` field in the savings account, defined by:

```
class Account {
  int balance = 0;
  void deposit(int amount) <this> { balance += amount; }
  void withdraw(int amount) <this> { balance -= amount; }
}
```

The effect of the call `sav.withdraw` is {`sav`}, found by substituting the target `sav` for the declared method effect {**this**} in class `Account`. Similarly the effect of `chq.deposit` is {`chq`}. The effect of the `transfer` is sound because the combined effect of its body, {`sav, chq`}, is subsumed by (that is, is a smaller effect than) the effect of the calling context (the `Customer` object): both the `sav` and `chq` objects are inside the `Customer` object in the ownership tree.

Such static knowledge of effects is needed in the type system of Section 4 to determine the effect of tasks and then infer synchronisation to protect them if they may interfere with each other. An effect, denoted as ε, is simply a set of objects indicating a read or write dependency on that object; semantically the effect includes any sub-objects of the specified set. *Disjointness* of effects $\varepsilon_1 \# \varepsilon_2$ says that the two effects do not overlap: this is interpreted semantically, so disjointness implies that there are no common sub-objects for the two given sets. If two program behaviours have disjoint effects then they are non-interfering. Because ownership is tree structured, disjointness of owners is inherited by their sub-objects; thus we can separate whole collections of objects just by separating their owners. In summary, ownership type systems model hierarchical object structures; each context in an effect specifies a tree of objects rooted at the context, thus allowing large effects to be summarised in a single context.

3 Structured Parallelism and Synchronisation

Fork-join frameworks [14, 18] typically assume that subtasks do not interfere with one another; recent type systems enforce such non-interference [2, 10]. We present a similar structured model for parallel tasks, but allow tasks to interfere.

3.1 Structured Parallelism with Effects

Since all possible sibling tasks can be identified lexically, the synchronisation requirement for a task need only prevent potential interference with its siblings, and not with any other tasks in the program, which are dealt with by the parent. Such a model relies on knowledge of the task structure and an effect system which can track all the effects of subtasks. Since the effects of subtasks are subsumed by the effect of their parents, subtasks may inherit their parents' protection. For instance, subtasks of a parent task are correctly synchronised with the subtasks of the parent's siblings through the synchronisation (if any) of the parent tasks.

We adopt a parallel-let form in which subtasks are forked, and may return a result which can only be used in the continuation after all subtasks complete:

$$\begin{array}{l}\textbf{par}\ \{\\ \quad x_1 = \textbf{sync}\ (\pi_1)\ e_1,\\ \quad \cdots\ ,\\ \quad x_n = \textbf{sync}\ (\pi_n)\ e_n\\ \}\ e\end{array}$$

Here, e is the continuation for the tasks e_i, which assign their return values to the local variables x_i which may be used in e. Each task is guarded by an inferred synchronisation requirement **sync** (π_i) which depends on the overlap between the effect of e_i and the effects of all its siblings, as discussed below. We present a particular example of an effect system in Section 4. For now, we simply assume that an effect ε of an expression e is a set denoting all memory locations or objects which may be read or written in the computation of e; overlaps of such effects are used to define tasks' synchronisation requirements π which correspond to the sets of objects that the tasks must synchronise on. The notation $\varepsilon_1 \# \varepsilon_2$

means that the two effects are guaranteed to be disjoint (and the computations of e_1 and e_2 are non-interfering).

The intended operational behaviour for synchronisation is simple. Sibling tasks synchronising on the same object are mutually excluded within the same **par**. A dynamic semantics for our type system is presented in Section 4, which ensures a task will only proceed if its sync requirement π does not overlap with those of the currently active sibling tasks.

Each synchronisation requirement for a **par** can be implemented as a lock-set. These lock-sets can be minimised to anti-chains in the ownership order. To ensure deadlock freedom it is only necessary to ensure that individual locks for the tasks of a par are acquired in a consistent order; there is no risk of deadlock from locking associated with different pars. It is possible to refine coarse-grain locks, pushing sync requirements onto subtasks, adapting techniques discussed elsewhere [29, 16]. Such fine-grain approaches should increase concurrency at the cost of greater locking overhead. They will still preserve data-race freedom, but may allow interleaving of subtask behaviours, so program serialisability may not be preserved. The coarse grain approach adopted in this paper preserves atomicity, hence serialisability, of all tasks. Instead of using lock-sets, we could use other means, such as software transactions, to implement the synchronisation requirements, which then become static advice about which objects need to be checked for conflict and need for task roll-back. Our focus here is to combine object ownership types with effects of tasks to infer synchronisation requirements.

3.2 Ownership-Based Synchronisation Requirements

For two parallel tasks with effects ε_1 and ε_2, we consider two different kinds of constraints on their sync requirements π_1 and π_2. Each of the constraints are sufficient to prevent concurrent activation of potentially interfering tasks thus ensuring safety; we denote this safety condition as $(\pi_1)\ \varepsilon_1 \perp (\pi_2)\ \varepsilon_2$.

The first constraint says that a mutex, where the sync requirements overlap, is always good enough, irrespective of the effects:

$$\pi_1 \cap \pi_2 \neq \emptyset \tag{1}$$

In practice we often want to maximise concurrency, so we would only choose this form if we know that the effects definitely overlap. Then we can simply choose π_1 and π_2 to be an arbitrary singleton mutex. Because the sync requirements overlap, our dynamic model ensures that the two tasks can only be run in mutual exclusion—thus ensuring serialisability.

At runtime, the effects and sync requirements correspond to sets of objects. However, static effect systems only capture approximations of the runtime effects, and have to deal with possible aliasing amongst program variables. In the `transfer` method of the `Customer` example of Section 2, we might allow `sav.withdraw` and `chq.deposit` to run in parallel. If we could not guarantee that `sav` and `chq` were never aliased, then we would need to protect them with `sync(sav)` and `sync(chq)`. The second form of constraint copes with possible (rather than definite) overlap of effects, but attempts to allow maximal concurrency when the possible overlap is not actually manifested at runtime (e.g. if `sav` and `chq` referred to distinct accounts).

```
class Customer {
  final Account<this> sav = new Account;
  final Account<this> chq = new Account;
  void transfer(int amt) <this> {
    par {
      sync()  sav.withdraw(amt),
      sync()  chq.deposit(amt)
    };
  }
  int creditCheck() <this> {
    return par {
      s = sync()  sav.balance(),
      c = sync()  chq.balance()
    } (s + c);
}}

class Bank {
  ...
  void touch(Customer<this> cus1, Customer<this> cus2) {
    par {
      sync(cus1)  cus1.transfer(10),
      sync(cus2)  cus2.creditCheck()
    };
} }
```

Fig. 1. Customer with *Inferred Synchronisation*

$$\pi_1 \otimes \pi_2 \implies \varepsilon_1 \# \varepsilon_2 \tag{2}$$

This is a variation of the *conditional must-not-aliasing* safety requirement introduced in [26]. It allows us to write sync requirements, without knowing whether they will actually inhibit concurrent activation or not. The safety condition (2) says that if the sync requirements are not aliases (hence will not block concurrent activation) then the task side effects must not be aliased (that is, non-interfering). The converse statement may be easier to understand: whenever the tasks may interfere, then the sync requirements must also. This means that we can choose the sync requirements for potentially overlapping subtasks by restricting the sync requirement to just that part of the effect which may overlap with the other task. We give an example of this below.

3.3 An Example

We adopt the bank account example of [5] to discuss our inferred synchronisation (details of the algorithm are omitted in the short paper). In the listing of Figure 1 we provide a two-level ownership structure: a bank owns a list of customers while a customer may own a list of accounts, as illustrated for our model in Figure 2.

Instead of explicit synchronisation, we infer sync requirements to guarantee the task safety; these are shown in grey italic. In class `Customer`, subtasks are spawned (in the `par` block) to access the accounts in the `transfer` and

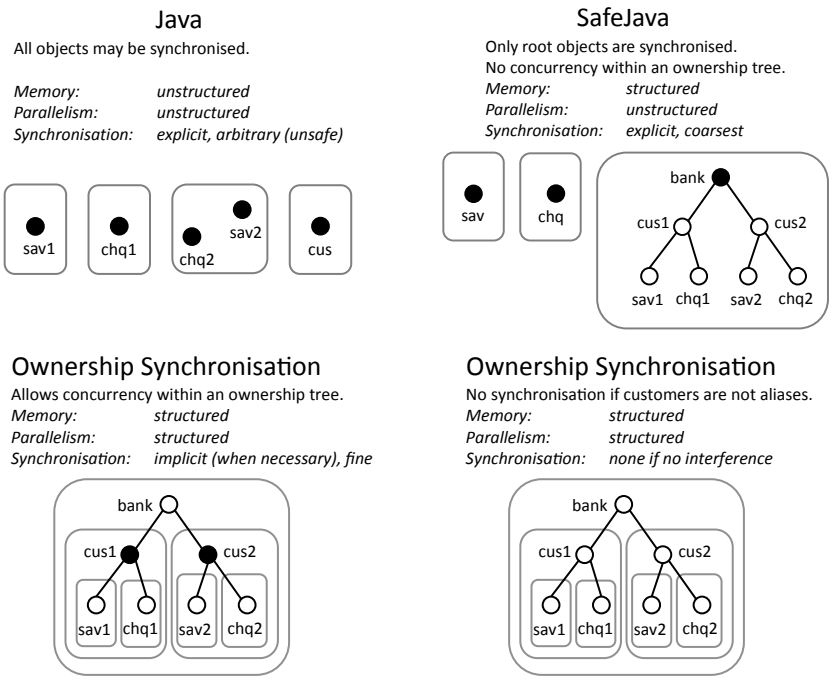

Fig. 2. Comparison of models: circles are objects; filled circles are synchronised; objects are in ownership trees; boxes are tasks; tasks contain objects they use and subtasks.

`creditCheck` methods. In both of these methods, all the subtasks have inferred `sync()`—the empty parenthesis means no synchronisation is required. *Why is this safe?* Let us look at the `transfer` method. The effect, as declared, of the `transfer` is the current object `this`. Such an effect is used by the type system as a safety contract between the tasks created inside the method body (subtasks) and the task that calls the method (parent task). When the method is called, the parent task guarantees that the call is safe—no other tasks may interfere with it. Within the method body of `transfer`, the calls on `sav.withdraw(amt)` and `chq.deposit(amt)` cannot interfere: the call on `sav.withdraw(amt)` will only access objects inside `sav` while `chq.deposit(amount)` will only access objects inside `chq`—being initialised with new account objects, the final fields `sav` and `chq` must refer to distinct objects. Since `sav` and `chq` can own no objects in common, there is no chance of interference. Therefore, no synchronisation is required for these tasks.

Consider the `touch` method in the `Bank`. The call `cus1.transfer(10)` has side-effect {`cus1`} and `cus2.creditCheck()` has {`cus2`}. But we cannot guarantee that `cus1` and `cus2` are not aliases, so those two calls may interfere. Synchronisation is required. By synchronising each call on different variables, `cus1` and `cus2`, we only block concurrent execution when they are actual run-time aliases. Our type system allows this conditional blocking by safety condition (2). To perform a safe inter-customer transfer (not in the example), a bank-level task

Programs	P	::=	$\overline{L}\ e$
Classes	L	::=	$\mathsf{class}\ c\langle\overline{p}\rangle\ [\lhd\ t]_{opt}\ \mathsf{where}\ \overline{p\ R\ p}\ \{\overline{[\mathsf{final}]_{opt}\ t\ f};\ \overline{M}\}$
Constraints	R	::=	$\prec\ \mid\ \preceq\ \mid\ \#$
Methods	M	::=	$t\ m(\overline{t\ x})\ \varepsilon\ \{e\}$
Effects	ε, π	::=	$\emptyset\ \mid\ \{k\}\ \mid\ \varepsilon \cup \varepsilon$
Contexts	k	::=	$\mathsf{world}\ \mid\ p\ \mid\ e$
Types	t	::=	$c\langle\overline{k}\rangle$
Variables	z	::=	$x\ \mid\ \mathsf{this}$
Expressions	e	::=	$z\ \mid\ \mathsf{new}\ t\ \mid\ e.f\ \mid\ e.f = e\ \mid\ e.m(\overline{e})\ \mid\ \overline{a}; e$
Tasks	a	::=	$x = s\ \pi\ e$
Sync states	s	::=	sync
Environments	Γ	::=	$\emptyset\ \mid\ \Gamma, p\ \mid\ \Gamma, p\ R\ p\ \mid\ \Gamma, z : t\ \mid\ \Gamma, \pi \otimes \pi$
Identifiers	c, p, f, m, x		

Table 1. Syntax and Type Environment

would process transactions on two customers; our inference algorithm would add both customers into the sync requirement for the task.

We illustrate the difference between Java, SafeJava and our model in Figure 2, whose caption explains the notation. In Java, all shared objects may be accessed by any thread. Java allows arbitrary synchronisation which does not provide safety. The simplest mechanism for thread-safety is to employ the per-object monitor. SafeJava [5] extends previous type systems for data race freedom [12], with ownership types where objects are encapsulated within threads or on the heap. In SafeJava, at most one thread can be active within a tree at any one time; this may be problematic for some applications.

By way of contrast, Figure 2 highlights the ability for tasks in our model to concurrently access objects inside an ownership tree (i.e. protected by the same coarse-grained lock). Unlike in SafeJava, accounts can be both encapsulated and accessed concurrently. If potential interference is inferred (because the two customer variables may be aliases for the same object), then synchronisation may be needed, but not necessarily at the top of the hierarchy. The tasks may run in parallel if the two customers are different objects. If we can determine that the two customers are indeed distinct, then no synchronisation is required at all, as in the last diagram. The diagrams make it clear that our model provides a general structure of active objects and synchronisation patterns. This generality is achieved without explicit synchronisation.

4 A Type System for Ownership Synchronisation

Our type system focuses on static checking of ownership synchronisation requirements, ensuring that runtime computations cannot interfere with one another. In this section, we summarise key type rules and a small step dynamic semantics. The syntax of a core language with ownership types and effects is given in Table 1. We allow final expressions to be used as owner contexts (including final fields and read-only variables) since they cannot change [5]. Type environments Γ record context parameters, the assumed ordering between them, and the types of variables.

[VARIABLE] $$\frac{z : t \in \Gamma}{\Gamma \vdash z : t \,!\, \emptyset}$$

[NEW] $$\frac{\Gamma \vdash t}{\Gamma \vdash \mathsf{new}\ t : t \,!\, \emptyset}$$

[SELECT] $$\frac{\Gamma \vdash e : t \,!\, \varepsilon \qquad (t'\ f) \in \mathit{fields}(t, e)}{\Gamma \vdash e.f : t' \,!\, \varepsilon \cup \{e\}}$$

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

[UPDATE] $$\frac{\Gamma \vdash e : t \,!\, \varepsilon \qquad (t'\ f) \in \mathit{fields}(t, e) \qquad \Gamma \vdash e' : t' \,!\, \varepsilon'}{\Gamma \vdash e.f = e' : t' \,!\, \varepsilon \cup \varepsilon' \cup \{e\}}$$

[CALL] $$\frac{\mathit{method}(m, t, e, \overline{e}) = t'\ \overline{t}\ \varepsilon'\ \ldots \qquad \Gamma \vdash e : t \,!\, \varepsilon \qquad \Gamma \vdash \overline{e : t \,!\, \varepsilon}}{\Gamma \vdash e.m(\overline{e}) : t' \,!\, \bigcup \overline{\varepsilon} \cup \varepsilon \cup \varepsilon'}$$

[PARALLEL] $$\frac{\Gamma \vdash \overline{e : t \,!\, \varepsilon} \qquad \Gamma, \overline{x : t} \vdash e : t \,!\, \varepsilon \qquad \forall k \in \bigcup \overline{\pi}\ \cdot\ \Gamma \vdash k \qquad \forall i, j \in 1..|\overline{e}|\ \cdot\ i \neq j \implies \Gamma \vdash (\pi_i)\ \varepsilon_i \perp (\pi_j)\ \varepsilon_j}{\Gamma \vdash \overline{x = \mathsf{sync}\ \pi\ e}; e : t \,!\, \bigcup \overline{\varepsilon} \cup \varepsilon}$$

[SUBSUMPTION] $$\frac{\Gamma \vdash e : t' \,!\, \varepsilon' \qquad \vdash t' \leq t \qquad \Gamma \vdash \varepsilon' \sqsubseteq \varepsilon}{\Gamma \vdash e : t \,!\, \varepsilon}$$

Table 2. Expression Typing Rules

The parallel-let expression is the only construct for expressing parallelism in this simple language. In the examples, we have used essentially the same syntax, with the explicit keyword **par**, which we now omit in the abstract syntax. Like conventional let expressions, our parallel-let introduces local variables to be used in the expression after the semicolon. The semicolon serves as a join boundary; the expression after the semicolon will only be evaluated after all the concurrent expressions join. The sync requirements π may be declared by the programmer or inferred by the compiler. The type system considers π given and ensures they are sufficient for guaranteeing safety.

Table 2 defines expression types and effects, which are largely standard except `[PARALLEL]`. The *fields()* and *method()* functions look up types for fields and methods from the class definition. `[FINAL]` states that accessing final fields has no side effect, because they are read-only. Sync requirements are checked in the last premise of `[PARALLEL]`. The elements in a sync requirement must be valid contexts, such as final expressions.

In Table 3 the antecedent for `[S-MUTEX]` states that the sync requirements overlap, and thus will always ensure the tasks execute in mutual exclusion, which is safe. `[S-COND]` corresponds to safety condition (2) of Section 3. Details of these rules are left to [22].

[S-MUTEX] $$\frac{k \in \pi \qquad k \in \pi'}{\Gamma \vdash (\pi)\ \varepsilon \perp (\pi')\ \varepsilon'}$$

[S-COND] $$\frac{\Gamma, \pi \otimes \pi' \vdash \varepsilon \# \varepsilon'}{\Gamma \vdash (\pi)\ \varepsilon \perp (\pi')\ \varepsilon'}$$

Table 3. Synchronisation Rules

Locations	l		
Variables	z	::=	$... \mid l$
Sync states	s	::=	$... \mid \mathsf{synced} \mid \mathsf{rel}$
Objects	o	::=	$\overline{f \mapsto l}$
Heaps	H	::=	$\overline{l \mapsto o_t}$
Eval states	S	::=	$H; e$
Eval contexts	E	::=	$[\,] \mid E.f \mid E.f = e \mid l.f = E \mid E.m(\overline{e}) \mid l.m(E) \mid \overline{l}, E, \overline{e}$

Table 4. Extended Syntax for Dynamic Semantics

Table 4 defines the extended syntax and features used in the dynamic semantics. In order to formalise the 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; ownership is only used for static type checking so it need not be available at runtime. We extend the syntax of variables with runtime locations l. Sync state synced indicates locks have been acquired and rel indicates locks have been released. A heap is a mapping from locations to objects with their types; an object maps its fields to locations. The initial state is $(\emptyset; e)$ where e is the body of the main method. We use standard evaluation contexts to reduce the number of evaluation rules.

Table 5 presents a small step operational semantics. The transition rules are mostly standard [12, 11, 2], where each single step is atomic and task interleaving is modelled as random choices (in `[R-INT]`). The label on the transition ε is the effect of the transition, which may either be empty ($\emptyset$) or a singleton location ($\{l\}$). For simplicity, we adopt the object creation semantics from [6] where all fields are initialised with new objects, hence `[R-NEW]` has no side effect.

The sync requirement of a task must be met in order for the task to be executed. The *synced* function tracks the total requirements for all the active (synced) tasks in its argument; it is used to ensure non-overlapping of sync requirements from different active tasks. This semantics is similar to [11], which abstracts away from a specific implementation as seen in [12]. The premise of `[R-ACQ]` blocks a task unless its sync requirement has no overlap with *synced*, the other active tasks' synchronisation, thus preventing any possible interference. (This premise just uses set intersection; there is no effect subsumption or context ordering as we do not depend on ownership information at runtime.) Once a task is active, it remains non-interfering with any other active task. This is the essence of the safety guarantee for serialisability. Note that `[R-ACQ]` is an atomic step no matter how many objects need to be synchronised in the sync requirement, hence is not prone to deadlock (see discussion in Section 3). `[R-REL]` removes a task from the active set, and releases its sync; its π no longer contributes to the active requirements *synced*. `[R-JOI]` ensures the order of sequential execution. The reduction rules for other expressions are mostly standard.

In database systems, serial means that transactions do not overlap in time and cannot interfere with each other, i.e., as if all transactions in the system had executed serially, one after another. Serialisability is usually proved by using an *acyclic precedence graph* [27]. A precedence graph for a schedule contains a node for each transaction and an edge from transaction T_i to T_j if an action in

$$\texttt{[R-SEL]}\quad \frac{H(l) = o_t \qquad (t'\ f) \in \mathit{fields}(t,l)}{H; l.f \xrightarrow{\{l\}} H; H(l)(f)}$$

$$\texttt{[R-FIN]}\quad \frac{H(l) = o_t \qquad (\mathsf{final}\ t'\ f) \in \mathit{fields}(t,l)}{H; l.f \xrightarrow{\emptyset} H; H(l)(f)}$$

$$\texttt{[R-CAL]}\quad \frac{H(l) = o_t \qquad \mathit{method}(m,t,l,\overline{l}) = \ldots\ e}{H; l.m(\overline{l}) \xrightarrow{\emptyset} H; e}$$

$$\texttt{[R-NEW]}\quad \frac{l \notin \mathit{dom}(H) \qquad H_1 = H, l \mapsto \emptyset_t \qquad (\overline{\ldots\ t\ f}) = \mathit{fields}(t,l) \qquad \forall i \in 1..|\overline{f}| \quad \cdot \quad H_i; \mathsf{new}\ t_i \xrightarrow{\emptyset} H_{i+1}; l_i}{H; \mathsf{new}\ t \xrightarrow{\emptyset} H_{|\overline{f}|+1}[l \mapsto (\overline{f \mapsto l})_t]; l}$$

$$\texttt{[R-ASS]}\quad H; l.f = l' \xrightarrow{\{l\}} H[l \mapsto H(l)[f \mapsto l']]; l'$$

$$\texttt{[R-SYN]}\quad \frac{H; \overline{e} \xrightarrow{\emptyset} H'; \overline{e'}}{H; \overline{a}, x = \mathsf{sync}\ \{\overline{e}\}\ e, \overline{a'}; e' \xrightarrow{\emptyset} H; \overline{a}, x = \mathsf{sync}\ \{\overline{e'}\}\ e, \overline{a'}; e'}$$

$$\texttt{[R-ACQ]}\quad \frac{\{\overline{l}\} \cap \mathit{synced}(\overline{a}, \overline{a'}) = \emptyset}{H; \overline{a}, x = \mathsf{sync}\ \{\overline{l}\}\ e, \overline{a'}; e' \xrightarrow{\emptyset} H; \overline{a}, x = \mathsf{synced}\ \{\overline{l}\}\ e, \overline{a'}; e'}$$

$$\texttt{[R-REL]}\quad H; \overline{a}, x = \mathsf{synced}\ \pi\ l, \overline{a'}; e \xrightarrow{\emptyset} H; \overline{a}, x = \mathsf{rel}\ \pi\ l, \overline{a'}; e$$

$$\texttt{[R-INT]}\quad \frac{H; e \xrightarrow{\varepsilon} H'; e'}{H; \overline{a}, x = \mathsf{synced}\ \pi\ e, \overline{a'}; e'' \xrightarrow{\varepsilon} H'; \overline{a}, x = \mathsf{synced}\ \pi\ e', \overline{a'}; e''}$$

$$\texttt{[R-JOI]}\quad H; \overline{x = \mathsf{rel}\ \pi\ l}; e \xrightarrow{\emptyset} H; [\overline{l/x}]e \qquad\qquad \texttt{[R-CTX]}\quad \frac{H; e \xrightarrow{\varepsilon} H'; e'}{H; E[e] \xrightarrow{\varepsilon} H'; E[e']}$$

Table 5. Small Step Operational Semantics

T_i occurs before an action in T_j and they conflict. Serialisable means that the precedence graph for any possible schedule is acyclic. In our system, all tasks run in a completely isolated fashion, even though they may execute concurrently with other (non-interfering) tasks. To prove serialisability, we represent a schedule with an arbitrary number of transitions. For example, a schedule $S \xRightarrow{G} S_3$ is a sequence of transitions $S \xrightarrow[x_1]{\varepsilon_1} S_1 \xrightarrow[x_2]{\varepsilon_2} S_2 \xrightarrow[x_3]{\varepsilon_3} S_3$, where actions are denoted by their task names and effects. We only track actions in active (synced) tasks because [R-INT] is the only case where tasks may interfere. We write $G \vdash x_1\ \varepsilon_1 \rhd x_3\ \varepsilon_3$ to denote that, in the schedule G, an action in task x_1 with effect ε_1 occurs before another action in task x_3 with effect ε_3.

Theorem 1 (Serialisability) states that tasks associated with conflicting actions must always appear in the same order in a schedule, so the precedence graph is indeed acyclic as required. Because tasks are lexically scoped and hierarchically defined, all parallel tasks are serialisable. We can therefore rely on conventional techniques for reasoning about state-based updates in a sequential style. Serialisability subsumes data race freedom such as that provided in [5, 11].

Theorem 1 (Serialisability)
Given $H; (\overline{x = s\ \pi\ e}; e)$ *is a well-formed eval state and* $H; (\overline{x = s\ \pi\ e}; e) \stackrel{G}{\Longrightarrow} H'; (\overline{x = s'\ \pi\ e'}; e)$*, if* $G \vdash x_i\ \varepsilon_i \rhd x_j\ \varepsilon_j$ *and* $\varepsilon_i \cap \varepsilon_j \neq \emptyset$*, then* $G \not\vdash x_j\ \varepsilon'_j \rhd x_i\ \varepsilon'_i$.

5 Discussion and Related Work

This research attempts to combine concurrency and OO using ownership types and effects to reason about structured object synchronisation. Programmers can take the advantage of domain knowledge of object structures, specified via ownership types and effects, to design concurrency and synchronisation. In general, finer-grained object structures not only allow finer-grained reasoning on task effects and their overlapping, but also provide the opportunity for finding more allowable concurrency in structured parallelism, supported by our structured synchronisation scheme. With the ability to locally reason about tasks, programmers may control and possibly reduce interference between tasks to allow more tasks to run in parallel; they may also split tasks to achieve a finer granularity of execution and synchronisation.

We adopt a whole-task synchronisation approach for ensuring parallel tasks are executed in isolation, like previous type systems for atomicity [13]. By making all tasks serialisable and synchronisation implicit, we simplify concurrent programming and allow programmers to focus on their program logic as they would for sequential programs within any task. However, structured parallelism is not as flexible as a free-range threading model. Fortunately, it is possible to ensure threads do not interfere with serialisable tasks [22], so that programmers can choose to use threads or tasks as appropriate.

Our type and effect system supports modular, compile-time type checking, and relies only on effects for preventing task interference, but otherwise placing no restriction on references as seen in previous ownership types for encapsulation. For example, an owned object can be typed and referenced from outside its owner by naming the owner argument of the type with a read-only variable. But this may be somewhat limited due to the naming restriction of ownership types [5]. To completely remove naming restriction and provide for liberal reference to owned objects in ownership type systems, we have proposed ownership abstraction [21] and variances [20]. For simplicity we do not adopt these earlier schemes in this paper. In addition, like [5], our type system does not distinguish read and write effects as done in [8, 6]. This is somewhat over-conservative in that it does not allow simultaneous reads of an object. We believe that adding a treatment of read-write effects to our type system should not be difficult, at the cost of extra effect annotations.

Reasoning about concurrent programs is difficult, in part because behaviour of code is rather informally and imprecisely specified. Effect annotations enhance program reasoning [6, 4]. Our type system uses explicit and checkable type and effect annotations as a means of recording programmer intent. On the other hand, annotations add programming overhead. With sensible defaults we can reduce the annotation load so that not all type occurrences need explicit annotations. Different ownership inference techniques have been studied elsewhere, e.g. [28, 5].

Much work has addressed the challenges of shared-memory concurrent programming. Here we restrict attention to directly related work. In JOE [8], Clarke and Drossopoulou introduced an ownership-based effect system. MOJO [6] developed these ideas further, focusing on the use of multiple ownership to increase the precision of effects. Ownership domains [1] provided a richer structure than ownership tree, which may be useful for more precise effect reasoning. In this paper, we only consider ownership trees for objects, but it should be feasible to extend the ideas to other object structures.

Previous attempts to utilise ownership types for object synchronisation are rather limited and structureless. In SafeJava [5], objects are encapsulated within threads or on the heap. Concurrent access to shared objects on the heap from different threads is guaranteed to be safe by synchronising on the root objects—an object can only be accessed when holding the lock on its root owner. Moreover, no external reference to non-root objects is allowed. Thus, within a root object, all behaviour is single-threaded; this is the coarsest-grained ownership synchronisation. In [11], data-races on objects are controlled by synchronisations on their direct owners; different levels of object access do not rely on higher-level synchronisations, and must be separately synchronised. It is finer-grained than SafeJava. Unlike these previous techniques which rely on a fixed granularity throughout system, we provide a more unified and structured ownership synchronisation discipline. Moreover, [5, 11] and other type-based race detection techniques [12] only check for the sufficiency of synchronisation, called the *locking discipline*, requiring all accesses to shared data to be protected by a common lock regardless of thread interference. This can lead to excessive locking even when synchronisation is unnecessary, for example, there is no need to protect a location shared by different tasks but will never be accessed concurrently. In our model, we apply synchronisation only when there is potential interference, hence achieve a more precise interference-based discipline.

Java 7 provides library support for structured parallelism based on Lea's fork/join framework [18], which in turn, is based on Cilk [14]. Although able to access shared memory, parallel tasks are typically assumed to work with distinct subsets of memory, and consequently, should not interfere with one another. Deterministic Parallel Java (DPJ) [2] proposed a region-based effect system to enforce noninterference among tasks in the fork/join model, thus guaranteeing deterministic semantics. Synchronisation is never needed because tasks may never conflict with one other. Such determinism is useful for certain programs (e.g. scientific applications) where interference is limited, but too strong in realistic object-oriented programs with extensive sharing and mutation. The latest work on DPJ [3] recognises the necessity and challenge for nondeterminism, by allowing interference in some branches of the task tree. Programmers are required to explicitly declare atomic blocks and place every access to shared memory inside atomic blocks, so that data race freedom can be guaranteed by software transactional memory (STM). Beside the limitations of STM at this moment [7], it does not support threads because thread interference cannot be captured by their type and effect systems. Rather than regions, [10] uses an ownership-based effect system to reason about data dependencies and allow the compiler to parallelise code wherever task interference is not possible. These models are

special cases of ours (except some special features, such as index-parameterised array types [2]), where the inferred sync requirements are empty. By scheduling parallel tasks whose sync requirements overlap to execute in sequential order, our behavioural model is reduced to theirs. With synchronisation, our model allows more concurrency than deterministic programming which requires all potentially interfering tasks to be executed in a specified order, though synchronisation may have runtime overhead.

6 Conclusion

In this paper we have proposed the use of ownership types to enforce a structured synchronisation discipline for safe execution of structured tasks. We use ownership effects to reason about interference between parallel tasks, and establish synchronisation requirements that guarantee serialisability. Synchronisation requirements can be inferred to simplify concurrent programming, thus lowering the bar for programmers—they are not responsible for choosing appropriate synchronisation. The strong serialisability property simplifies reasoning about concurrent programs by allowing every piece of code to be reasoned about separately without fear of interference. Although this paper has focused on structured parallelism, it can be extended to coexist with conventional unstructured threads. With this work, we hope to stimulate further exploration of how object and effect structure can facilitate safe and efficient concurrent programming.

References

1. J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *ECOOP*, 2004.
2. 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.
3. R. L. Bocchino Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe nondeterminism in a deterministic-by-default parallel language. In *POPL*, 2011.
4. C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *POPL*, 2003.
5. C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *OOPSLA*, 2001.
6. N. Cameron, S. Drossopoulou, J. Noble, and M. Smith. Multiple Ownership. In *OOPSLA*, 2007.
7. C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *ACM Queue*, 2008.
8. D. Clarke and S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *OOPSLA*, 2002.
9. D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, 1998.
10. A. Craik and W. Kelly. Using ownership to reason about inherent parallelism in object-oriented programs. In *CC*, 2010.

11. D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universe Types for Race Safety. In *VAMP*, 2007.
12. C. Flanagan and M. Abadi. Types for safe locking. In *ESOP*, 1999.
13. C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for Java. *TOPLAS*, 30(4):1–53, 2008.
14. M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, 1998.
15. B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.
16. G. Golan-Gueta, N. G. Bronson, A. Aiken, G. Ramalingam, M. Sagiv, and E. Yahav. Automatic fine-grain locking using shape properties. In *OOPSLA*, 2011.
17. J. Hogg. Islands: aliasing protection in object-oriented languages. In *OOPSLA*, 1991.
18. D. Lea. A Java fork/join framework. In *Java Grande*, 2000.
19. E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, 2006.
20. Y. Lu and J. Potter. On ownership and accessibility. In *ECOOP*, 2006.
21. Y. Lu and J. Potter. Protecting representation with effect encapsulation. In *POPL*, 2006.
22. Y. Lu, J. Potter, and J. Xue. *Ownership Types for Object Synchronisation*. Extended version, available at `http://www.cse.unsw.edu.au/~ylu/0813.pdf`.
23. Y. Lu, J. Potter, and J. Xue. Validity invariants and effects. In *ECOOP*, 2007.
24. Y. Lu, J. Potter, and J. Xue. Ownership downgrading for ownership types. In *APLAS*, 2009.
25. J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL*, 1988.
26. M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *POPL*, 2007.
27. C. Papadimitriou. *The theory of database concurrency control*. Computer Science Press, Inc., New York, NY, USA, 1986.
28. A. Potanin, J. Noble, D. Clarke, and R. Biddle. Generic ownership for generic Java. In *OOPSLA*, 2006.
29. J. Potter and A. Shanneb. Incremental lock selection for composite objects. *Journal of Object Technology*, 6(9):477–494, 2007.
30. M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *POPL*, 1994.
31. M. T. Vechev, E. Yahav, R. Raman, and V. Sarkar. Automatic verification of determinism for structured parallel programs. In *SAS*, 2010.
32. P. Wadler. Is there a use for linear logic? In *PEPM*, 1991.