Universes for Race Safety

D. Cunningham [a,1] S. Drossopoulou[a,2] S. Eisenbach[a,3]

[a] *Department of Computing*
Imperial College London
U.K.

Abstract

Race conditions occur when two incorrectly synchronised threads simultaneously access the same object. Static type systems have been suggested to prevent them. Typically, they use annotations to determine the relationship between an object and its "guard" (another object), and to guarantee that the guard has been locked before the object is accessed. The object-guard relationship thus forms a tree similar to an ownership type hierarchy.
Universe types are a simple form of ownership types. We explore the use of universe types for static identification of race conditions. We use a small, Java-like language with universe types and concurrency primitives. We give a type system that enforces synchronisation for all object accesses, and prove that race conditions cannot occur during execution of a type correct program.
We support references to objects whose ownership domain is unknown. Unlike previous work, we do so without compromising the synchronisation strategy used where the ownership domain of such objects is fully known. We develop a novel technique for dealing with non-final (i.e. mutable) paths to objects of unknown ownership domain using effects.

Keywords: ownership, synchronisation, concurrency, race condition, effects, atomicity

A race condition is an error that can occur in concurrent programs when two threads are not properly synchronised, and thus can simultaneously access the same object. This can then lead to corruption of data structures, and eventual software failure. To date, many well-known pieces of software have fallen foul of race conditions, often long after their initial development, sometimes leading to denial-of-service attacks or other security problems.

Programmers typically attempt to avoid race conditions through disciplined programming; by ensuring that the right lock is taken during all shared object accesses. They must choose which locks guard their objects and respect this relationship everywhere in their code [21].

Previous work [11,3] uses ownership type annotations [5,28] (or "guard" annotations that resemble ownership) to restrict variable bindings to objects in specific regions of the heap. Every region has an associated lock, so the type systems know which locks protect a block of code without precisely knowing which objects will be

[1] Email: dc04@doc.ic.ac.uk
[2] Email: scd@doc.ic.ac.uk
[3] Email: sue@doc.ic.ac.uk

accessed. This requires all types to be explicitly annotated by ownership parameters, allowing the expression of complex ownership structures, albeit at the expense of heavier annotation.

Universes [26,8] are a simple, yet powerful form of ownership types used in the JML tools [22]. Universes let programmers succinctly specify the topological relationship between objects using just a few keywords. As such, the owner of an object is implicitly understood by the type system, and implicitly stored by the run-time environment. Thus, programmers need not explicitly declare them.

We developed a type system for race safety using universes to partition the heap. As in [3,4,11,12,13,14] we treat objects in the same *ownership domain* (i.e. all objects sharing the same owner) as guarded by the same lock. At run-time we associate this lock with the objects' owner. All objects have an implicit reference to their owner.

Universes also allow references to objects whose owner is unknown through the annotation `any` [9] (in earlier work [26] called `readonly`, which is distinct from `final` because it describes the referenced object). This is not supported by [11,3].

The presence of `any` was a challenge for us, but turned out to increase the expressiveness of our language. The `any` annotation allows the expression of data structures that contain objects from various ownership domains. Use of such data structures does not require us to compromise the design of other data structures in our program. This improves upon [3,11], where in particular one sometimes has to alter the design of unrelated data structures so that they take the lock once for each access in an iteration. Iterating over the unrelated structures would be atomic in our system but could not be atomic in [3,11].

When the type does not indicate the owner of an object, we use paths as an alternative mechanism to guarantee correct synchronisation. We use an effect system where these paths are not final as would be required in [3,11].

Previous work [3,11] required entire ownership domains to be locked even if only a single object is accessed. It is straightforward to extend our system with single object locks.

The rest of this paper is organised as follows: In section 1 we explain universes and their suggested use for concurrency in an example program. In section 2 we give a formal model of universes. In section 3 we give (and prove) a simple type system that guarantees race safety. We discuss implementation in section 4 and related work in section 5. We discuss future directions in section 6 and conclude with section 7.

1 An example of Universes and Race Safety

The run-time state of an object-oriented program consists of a graph of objects linked by field references. In an ownership system, each object is owned by another object. The ownership relation describes a tree structure whose root is `null`. This tree structure represents the encapsulation inherent in the design of a program [5,28].

In a universe type system, types consist of a class and a keyword that indicates the topological relationship. We call the keyword an *ownership type qualifier*; it is one of `rep`, `peer`, and `any`.

```
1  class Stud { int mark ; boolean roomClean  }

2  class Dept {  // Closed list
3      rep DeptStudNode first;
4      void releaseMarks () { ... }
5  }

6  class DeptStudNode {
7      peer Stud s;
8      peer DeptStudNode next;
9  }

10 class Hall {  // Open list
11     rep HallStudNode first;
12     void cleanRooms () { ... }
13 }

14 class HallStudNode {
15     any Stud s;
16     peer HallStudNode next;
17 }
```

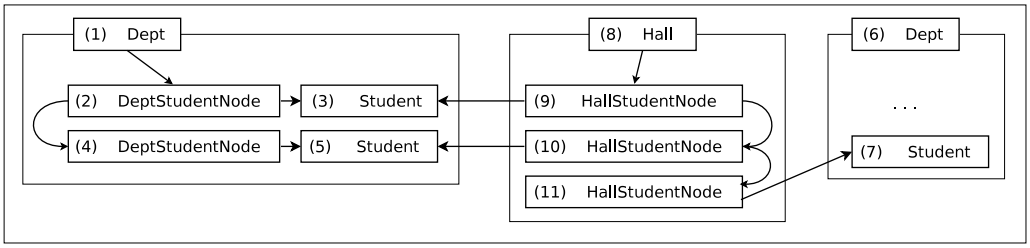


Fig. 1. Example program showing heap hierarchy structure

Types are relative to some *observer* object. When the observer has the same owner as another object, the second object is a `peer` of that observer. When the observer is the owner of another object, the second object is a `rep` of that observer. Any object can be `any`.

Consider, for example, the heap diagram from Fig. 1. Each object has an address, *e.g.,* (1), and a class name, *e.g.,* `Dept`. Owned objects are drawn in the domains of their owners; the tree is represented by the nesting of the boxes, (1) owns $(2-5)$, and (8) owns $(9-11)$. From observer (1), (3) has type `rep Stud` [4], but from observer (2), the (3) has type `peer Stud`. Thus, the type of (3) is relative to the observer. Also, from observer (8) the object (3) has type `any Stud`, and the object (1) has type `peer Dept` [5].

In Fig. 1 we show source code where types contain universe annotations, and which could give rise to the heap in the diagram. For example, class `Dept` has field `first` of type `rep DeptStudNode`, which, in the diagram corresponds to the reference from (1) to (2). On the other hand, `HallStudNode` has field `s` of type `any Stud`, which, in the diagram corresponds to the reference from (9) to (3).

Thus, through `any`, students (owned by their respective departments), can be accessed also from their halls of residence. We call the list inside `Dept` a *closed* list as the students are enclosed in the ownership domain of the list. In contrast the list inside `Hall` is *open* because its students can be in any department.

We now discuss the use of the tree-hierarchy imposed by the universe types to avoid races: We require that the run-time system records the owner of an object (which does not change). We associate a lock with each object, with objects guarded by their owner's lock rather than their own. Any accesses to a field of an object, for example $e'.f$ or $e'.f = ...$, must be within a `sync` e block where e resolves to an object that is part of the same ownership domain as e'. This is in contrast to Java's `synchronized` which locks e and not the whole of its ownership domain.

[4] It trivially also has type `any Stud`.

[5] The reference from (8) to (3) is illegal in systems enforcing owners-as-dominators [5,28] but is legal in universe types, which, instead, enforce owners-as-modifiers.

```
18  void releaseMarks () {
19    sync (this) {
20      rep DeptStudNode i = this.first;
21      sync (i) {
22        while (i!=null) {
23          i.s.mark = ...;
24          i = i.next;
25  } } } }
```

```
26  void cleanRooms () {
27    sync (this) {
28      rep HallStudNode i = this.first;
29      sync (i) {
30        while (i!=null) {
31          sync (i.s)
32            { i.s.roomClean = true; }
33          i = i.next;
34  } } } }
```

Fig. 2. Method bodies for Fig. 1

One can also think of `sync` e as locking the object owning e. We propose that `synchronized` be no-longer used, in favour of `sync`. Nested ownership domains are disjoint; code that accesses both must take both locks. The ownership domain is statically verifiable when the ownership type qualifier of e' is not `any`.

Consider the code for `releaseMarks` from Fig. 2. Adhering to the rule set out above, the field access to `this.first` (line 20) is enclosed within `sync (this)` (line 19). More interesting is the body of the `while` loop, where a statically unknown number of field accesses through `i.next` (line 24) is correctly synchronised by acquiring a *single* lock, `sync (i)`, before the loop (line 21). Even though `i` will point to different objects at each iteration, the synchronisation is correct, because the field `next` is `peer` and thus all these objects will have the same owner. The same is true when we access the student (line 23).

A challenge we needed to tackle is, how to avoid races when the ownership domain of the accessed object is unknown, *i.e.,* when it has type `any C` for some class `C`. In such a case, any accesses of the form $p.f$ or $p.f = ...$, where p is a path [6] must be within a `sync` p block provided that the block does not assign to any of the fields appearing in p.

The difference between the body of `cleanRooms` in Fig. 2 and `releaseMarks` is that in the former, `HallStudNode` has an `any` pointer to `Stud`. Thus, the student is not necessarily a peer of the node `i`; therefore, when we access `i.s` (line 32) the `sync (i)` (line 29) is no longer sufficient. We must lock the owner of the student `i.s` and this is possible through the "fresh" `sync (i.s)` (line 31) even though `i.s` is `any`. We must be sure however that the body of the `sync (i.s)` block does not write to the field `s`, otherwise the type system would reject our program.

Note that in `releaseMarks`, students will receive their marks *atomically* (there is never a state visible where a subset of students have their marks) but this is not the case for the cleaning of rooms. A student may notice their room has been cleaned whereas another student's room has not. In general, we must lock individual elements when iterating through an open list. This is not necessary for a closed list.

In [11], an open list of students can be written if we design the student so that it has a final field that stores the owner. In other words, we create a class that can be referenced by a variable whose type does not specify an owner such as `s` of `HallStudNode`. However, this change is global to the program so every other reference (*e.g.,* the field `s` of `DeptStudNode`) must use the same type that does not specify an owner. This means that we cannot make a closed list of students, because the owner of the student is no longer indicated by its type. The only solution is to use open lists everywhere, which have the undesirable property that we cannot lock

[6] A path is a sequence of field accesses starting from a parameter or `this`.

```
// We have to design Stud like this:

class Stud {
   final Object owner;
   /* fields guardedby this.owner */
}

// Therefore, the HallStudNode looks like:

class HallStudNode<x> {
   Stud s guardedby x;
   HallStudNode<x> next guardedby x;
}

// Hall locks its students like so:

class Hall<x> {  // Open list
   HallStudNode<this> first guardedby x;
   void cleanRooms () {
      sync (x) {  //protects fields of this
         HallStudNode<this> i=this.first;
         sync (this) {  //protets nodes
            while (i) {
               final Stud s' = i.s;
               sync (s'.owner) {
                  s'.roomClean = true;
               }
               i = i.next;
} } } } }


// But a Stud is a Stud, so we must design
// DeptStudNode in the same manner:

class DeptStudNode<x> {
   Stud s guardedby x;
   DeptStudNode<x> next guardedby x;
}


// And thus we have to lock each student

class Dept<x> {  // must be open too!
   DeptStudNode<this> first guardedby x;
   void releaseMarks () {
      sync (x) {
         DeptStudNode<this> i=this.first;
         sync (this) {
            while (i) {
               final Stud s' = i.s;
               sync (s'.owner) {
                  s'.mark = ...;
               }
               i = i.next;
} } } } }
```

```
// We have to design student like this:

class Stud<x> {
   /* fields here */
}


// Therefore, HallStudNode looks like:

class HallStudNode<x> {
   Stud<self> s;
   HallStudNode<x> next;
}

// Hall locks its students like so:

class Hall<x> {  // Open list
   HallStudNode<this> first;
   void cleanRooms () {
      sync (x) {
         HallStudNode<this> i=this.first;
         sync (this) {
            while (i) {
               final Stud<self> s'=i.s;
               sync (s') {
                  s'.roomClean = true;
               }
               i = i.next;
} } } } }


// Since the students we reference have type
// Stud<self>, this field must be the same:

class DeptStudNode<x> {
   Stud<self> s;
   DeptStudNode<x> next;
}

// But we do not own the student, so we must
// lock each student individually.

class Dept<x> {  // must be open too!
   DeptStudNode<this> first guardedby x;
   void releaseMarks () {
      sync (x) {
         DeptStudNode<this> i=this.first;
         sync (this) {
            while (i) {
               final Stud<self> s'=i.s;
               sync (s') {
                  s'.mark = ...;
               }
               i = i.next;
} } } } }
```

Fig. 3. Example code in the systems of [11] (left) and [4] (right)

all the elements of the list at once, we have to acquire the same lock once for each student. Another implication is that iterating through the list cannot be atomic (as in `releaseMarks`).

The type system of [4] is even more restrictive, as a closed list implementation can only contain self-owned objects. This means objects contained in an open list also can only exist in the root ownership domain. The code for both solutions is given in Fig. 3.

2 Formal Preliminaries

Universes are introduced in [26], and given a type theoretic presentation in [7]. We use ideas from [26] but with some differences: We decided that owners-as-modifiers, while useful for verification, are not needed for type soundness and race safety. Our type system allows field assignments through `any` objects as long as the heap

$$\mathcal{M} : (Id^c \times Id^m) \to TypeSig \qquad \mathcal{MBody} : (Id^c \times Id^m) \to SrcExpr$$
$$\mathcal{F} : (Id^c \times Id^f) \to Type \qquad c \in Id^c \qquad f \in Id^f$$

$$\begin{array}{rrl} e \in SrcExpr & ::= & \texttt{this} \mid \texttt{x} \mid \texttt{null} \mid \texttt{new}\ t \mid e.f \mid e.f = e \mid e.m(e) \\ & \mid & (t)\ e \mid \texttt{spawn}\ e \mid \texttt{sync}\ e\ e \\ t \in Type & ::= & u\ c \\ u \in Universe & ::= & \texttt{rep} \mid \texttt{peer} \mid \texttt{any} \mid \texttt{self} \\ TypeSig & ::= & t\ m(t) \\ \Gamma \in Environment & = & Type \times Type \qquad \Gamma(\texttt{this}) = \Gamma{\downarrow_1}, \Gamma(\texttt{x}) = \Gamma{\downarrow_2} \end{array}$$

We define $(\leq_c)$, the reflexive transitive closure of the inheritance in the program.

We define $(\leq_u)$: $\texttt{self} \leq_u \texttt{peer} \leq_u \texttt{any} \quad \texttt{rep} \leq_u \texttt{any}$

Thus we define the subtype relation $(\leq)$: $u\ c \leq u'\ c' \iff u \leq_u u' \wedge c \leq_c c'$

Fig. 4. Source program definition

remains well-formed. Therefore our type system is more permissive and could be restricted to also require owners-as-modifiers.

For sequences, we use the notation $\overline{e}$ in the style of [19], and sometimes as $e_{1..n}$, both of which are distinct from the undecorated e. The ith element of $e_{1..n}$ is e_i and of $\overline{e}$ is $\overline{e}{\downarrow_i}$. We use similar notation when we access the ith element of a tuple: $(a, b, c){\downarrow_2} = b$. We use an underscore _ to represent a variable whose value can be arbitrary. To denote that a particular construct *e.g.,* `new` c occurs within an expression e, we sometimes write $\texttt{new}\ c \in e$. $\mathbb{P}$ is the powerset.

2.1 Syntax and Semantics

Programs are defined in Fig. 4, and consist of the three functions $\mathcal{M}$, $\mathcal{MBody}$, and $\mathcal{F}$, which define the method signatures, method bodies, and field types of each class in the program, together with $(\leq_c)$ which gives the inheritance relationship between classes. Note that all types t are annotated with an ownership type qualifier u which can be one of the three keywords `rep`, `any`, `peer`, or `self` which is the type of `this` and thus a specialisation of `peer`. It prevents the type system losing type information during local member access. We use `spawn` e to start a new thread to execute e, and `sync` $e\ e'$ to acquire the lock that guards the object e while we execute the expression e'. We give the run-time syntax in Fig. 5 and use a small step semantics.

The program state consists of the heap h and a sequence of expressions $\overline{e}$. It is reduced with respect to a base stack frame σ according to the rules in Fig. 6. The (INTERLEAVE) rule uses the single-threaded semantics. The base stack frame contains the value of `this` and `x`. Single-threaded execution steps are decorated with actions, ranged over by β. If a step accesses an address a, then its action is a, otherwise its action is τ. At the multi-threaded level, each step may introduce at most one more thread, stopped threads are never eliminated from the system, and we wrap actions

$$
\begin{array}{rcll}
s \in \mathit{State} & : & \mathit{RunExpr} \times \mathit{Heap} & \\
h \in \mathit{Heap} & : & \mathit{Addr} \to \mathit{Object} & \\
\mathit{Object} & : & (\mathit{Val} \times \mathit{Id}^c \times (\mathit{Id}^f \to \mathit{Val})) & \texttt{//} \text{ owner, class, fields} \\
a \in \mathit{Addr} & : & \mathbb{N} & \\
v, w \in \mathit{Val} & ::= & a \mid \texttt{null} & \\
\sigma \in \mathit{Stack} & ::= & (a, v) & \sigma(\texttt{this}) = \sigma{\downarrow_1},\ \sigma(\texttt{x}) = \sigma{\downarrow_2} \\
\beta \in \mathit{Actions} & ::= & a \mid \tau & \\
e \in \mathit{RunExpr} & ::= & v \mid \texttt{this} \mid \texttt{x} \mid \texttt{new}\ t \mid e.f \mid e.f = e \mid e.m(e) \mid (t)\ e \mid \texttt{spawn}\ e & \\
 & \mid & \texttt{sync}_e\ e\ e \mid \texttt{synced}_e\ w\ e \mid \texttt{frame}\ \sigma\ e & \\
E[\cdot] & ::= & E[\cdot].f \mid E[\cdot].f = e \mid v.f = E[\cdot] \mid E[\cdot].m(e) \mid v.m(E[\cdot]) & \\
 & \mid & (t)\ E[\cdot] \mid \texttt{sync}_e\ E[\cdot]\ e \mid \texttt{synced}_e\ w\ E[\cdot] &
\end{array}
$$

Fig. 5. Run-time state and syntax

with the index of the thread that caused them.

Method calls are modelled by substituting the call construct with the method body in question and the stack which records the receiver and parameters. The `frame` σ e construct marks the boundaries between the different calling contexts in the run-time expression, and holds the new stack σ which is used to execute the method body e. One can see from the `sync` syntax of the run-time language, and the (Call) semantics rule, that `sync` expressions in the method body are translated slightly by S during method call. This does not affect the behaviour of the program, it was needed so that we could prove soundness. [7] The subtree e' is duplicated and recursively translated, to be held in the subscript of the new `sync` construct.

The semantics rules (Cast) and (New) use the universe type system to constrain their behaviour (*e.g.,* in (New) we set the owner field this way), this makes the proofs simple. The syntax allows for the expression `new self` c but this will always result in a stuck execution [8]. We also allow the expression `new any` c, where the owner of the new object is chosen arbitrarily.

For interleaved execution we use the context $C[\cdot]$, which extends the evaluation context syntax $E[\cdot]$ to add the stack frame construct:

$$C[\cdot] ::= \ldots \mid \texttt{frame}\ \sigma\ C[\cdot]$$

In rule (Lock), note that it is the owner of the object $w = h(a){\downarrow_1}$ that is actually locked. This is because we are locking the whole ownership domain, not just the

[7] We demonstrate the life-cycle of a `sync` construct with an example: The source expression `sync` $e\ e''$ is translated into the run-time expression $\texttt{sync}_e\ e'\ e''$. Initially $e = e'$ but as the expression reduces, the e' will reduce until it reaches an object a, and then the lock that guards that object w is taken. The subscript persists on this $\texttt{synced}_e\ w\ e''$ expression until e'' terminates and the lock released. The subscript expression e will always remain as a record of the initial locking expression, even after it has reduced to an object and the main body is executing.

[8] It would reduce to an unused address that is the same as σ(`this`), an existing address. In practice we could disallow the use of `self` and `any` when constructing new objects, in either the syntax or the static type system.

$$\frac{}{\sigma \vdash \texttt{this}, h \overset{\tau}{\rightsquigarrow} \sigma(\texttt{this}), h}\,(\textsc{This})$$

$$\frac{}{\sigma \vdash \texttt{x}, h \overset{\tau}{\rightsquigarrow} \sigma(\texttt{x}), h}\,(\textsc{Var})$$

$$\begin{array}{rcl} S & : & SrcExpr \to RunExpr \\ S(\texttt{sync}\ e_1\ e_2) & = & \texttt{sync}_{S(e_1)}\ S(e_1)\ S(e_2) \\ S(Con(e_1 \ldots e_n)) & = & Con(S(e_1) \ldots S(e_n)) \\ \multicolumn{3}{c}{\text{(for all other constructs } Con \in SrcExpr)} \end{array}$$

$$\frac{e = S(\mathit{MBody}(h(a)\downarrow_2, m))}{\sigma \vdash a.m(v), h \overset{\tau}{\rightsquigarrow} \texttt{frame}\ (a, v)\ e, h}\,(\textsc{Call})$$

$$\frac{h, \sigma \vdash v : t}{\sigma \vdash (t)\ v, h \overset{\tau}{\rightsquigarrow} v, h}\,(\textsc{Cast})$$

$$\frac{h' = h[a\downarrow_3(f) \mapsto v]}{\sigma \vdash a.f = v, h \overset{a}{\rightsquigarrow} v, h'}\,(\textsc{Assign})$$

$$\frac{\sigma' \vdash e, h \overset{\beta}{\rightsquigarrow} e', h'}{\sigma \vdash \texttt{frame}\ \sigma'\ e, h \overset{\beta}{\rightsquigarrow} \texttt{frame}\ \sigma'\ e', h'}\,(\textsc{Frame1})$$

$$\frac{\sigma \vdash e_i, h \overset{\beta}{\rightsquigarrow} e_i', h'}{\sigma \vdash e_{1..n}, h \overset{(i,\beta)}{\rightsquigarrow} e_{1..i-1}\ e_i'\ e_{i+1..n}, h'}\,(\textsc{Interleave})$$

$$\frac{\begin{array}{l} h(a) \text{ undefined} \\ h' = h[a \mapsto (\_, c, \lambda f.\texttt{null})] \\ h', \sigma \vdash a : u\ \_ \end{array}}{\sigma \vdash \texttt{new}\ u\ c, h \overset{\tau}{\rightsquigarrow} a, h'}\,(\textsc{New})$$

$$\frac{\begin{array}{l} e_i = C[\texttt{spawn}\ e'] \\ e_{n+1} = \texttt{frame}\ \mathit{Active}(\sigma, C[\cdot])\ e' \end{array}}{\sigma \vdash e_{1..n}, h \overset{(i,\tau)}{\rightsquigarrow} e_{1..i-1}\ C[\texttt{null}]\ e_{i+1..n+1}, h}\,(\textsc{Spawn})$$

$$\frac{}{\sigma \vdash a.f, h \overset{a}{\rightsquigarrow} h(a)\downarrow_3(f), h}\,(\textsc{Field})$$

$$\frac{\begin{array}{l} e_i = C[\texttt{sync}_{e'}\ a\ e] \qquad w = h(a)\downarrow_1 \\ \forall j \in \{1..n\}\ .\ \mathit{Locked}(e_j, w) \Longrightarrow i = j \\ e'' = C[\texttt{synced}_{e'}\ w\ e] \end{array}}{\sigma \vdash e_{1..n}, h \overset{(i,\tau)}{\rightsquigarrow} e_{1..i-1}\ e''\ e_{i+1..n}, h}\,(\textsc{Lock})$$

$$\frac{\sigma \vdash e, h \overset{\beta}{\rightsquigarrow} e', h'}{\sigma \vdash E[e], h \overset{\beta}{\rightsquigarrow} E[e'], h'}\,(\textsc{Ctx})$$

$$\frac{e_i = C[\texttt{synced}_{e'}\ w\ v]}{\sigma \vdash e_{1..n}, h \overset{(i,\tau)}{\rightsquigarrow} e_{1..i-1}\ C[v]\ e_{i+1..n}, h}\,(\textsc{UnLock})$$

$$\frac{}{\sigma \vdash \texttt{frame}\ \sigma'\ v, h \overset{\tau}{\rightsquigarrow} v, h}\,(\textsc{Frame2})$$

Fig. 6. Small step operational semantics

object specified by the `sync` block:

$$\texttt{sync}_{e'}\ a\ e \rightsquigarrow \texttt{synced}_{e'}\ w\ e$$

The predicate $\mathit{Locked}(e, w)$ determines whether the thread e has the lock on object w: It holds whenever the construct $\texttt{synced\_}\ w\ \_$ is a subexpression of e.

The function $\mathit{Active}(\sigma, e)$ provides the σ' from the innermost $\texttt{frame}\ \sigma'\ \_$ within the thread e according to the context rules for $C[\cdot]$. If there is no such $\texttt{frame}\ \sigma'\ \_$, it returns σ.

$$\frac{\Gamma \vdash e : t' \quad t' < t}{\Gamma \vdash e : t}\ (\textsc{Sub}) \qquad \frac{}{\Gamma \vdash \texttt{null} : t}\ (\textsc{Null}) \qquad \frac{}{\Gamma \vdash \texttt{x} : \Gamma(\texttt{x})}\ (\textsc{Var}) \qquad \frac{\Gamma \vdash e : t'}{\Gamma \vdash (t)\ e : t}\ (\textsc{Cast})$$

$$\frac{\Gamma \vdash e : u\ c \quad \mathcal{F}(c, f) = t}{\Gamma \vdash e.f : u \triangleright t}\ (\textsc{Field}) \qquad \frac{u \neq \texttt{self}}{\Gamma \vdash \texttt{new}\ u\ c : u\ c}\ (\textsc{New}) \qquad \frac{}{\Gamma \vdash \texttt{this} : \Gamma(\texttt{this})}\ (\textsc{This})$$

$$\frac{\Gamma \vdash e : t \quad \Gamma \vdash e' : t'}{\Gamma \vdash \texttt{sync}\ e\ e' : t'}\ (\textsc{Sync}) \qquad \frac{\Gamma \vdash e : t'}{\Gamma \vdash \texttt{spawn}\ e : t}\ (\textsc{Spawn}) \qquad \frac{\Gamma \vdash e : u\ c \quad \Gamma \vdash e' : t \quad \mathcal{F}(c, f) = u \triangleright t}{\Gamma \vdash e.f = e' : t}\ (\textsc{Assign})$$

$$\frac{\Gamma \vdash e : u\ c \quad \Gamma \vdash e' : t \quad \mathcal{M}(c, m) = t_r\ m(u \triangleright t)}{\Gamma \vdash e.m(e') : u \triangleright t_r}\ (\textsc{Call}) \qquad \frac{}{a, w \vdash a, w : \texttt{self}}\ (\textsc{Self})$$

$$\frac{}{\_, w \vdash \_, w : \texttt{peer}}\ (\textsc{Peer}) \qquad \frac{}{a, \_ \vdash \_, a : \texttt{rep}}\ (\textsc{Rep}) \qquad \frac{}{\_, \_ \vdash \_, \_ : \texttt{any}}\ (\textsc{Any})$$

$$\frac{\begin{array}{l}\forall c' \geq c\ .\ \mathcal{F}(c', f) = t \implies \mathcal{F}(c, f) = t \\ \forall c' \geq c\ .\ \mathcal{M}(c', m) = t'_r\ m(t'_x) \implies \mathcal{M}(c, m) = t_r\ m(t_x) \\ \qquad\qquad (\text{where } t_r \leq t'_r, t_x \geq t'_x) \\ \forall \mathcal{M}(c, m) = t_r\ m(t_x)\ .\ (\texttt{self}\ c, t_x) \vdash \mathit{MBody}(c, m) : t_r\end{array}}{\vdash c}\ (\textsc{WFClass})$$

A program is well-formed iff $\forall c. \vdash c$

Fig. 7. Static universe type system

2.2 Universe Type System

The universe type system is given in Fig. 7. The judgement $\Gamma \vdash e : t$ gives the type t of an expression e with respect to an environment Γ. As there is only one method parameter, this environment is simply a pair of the types of `this` and `x`.

Universe annotations have meaning only with respect to an observer as discussed in section 1. The type annotations in field and method signatures are meant with respect to the object that contains them. When we are typing method bodies, the annotations within are considered with respect to the object `this`, whatever value `this` might have at run-time. Thus the type returned by the type system is also

$$\frac{h,\sigma \vdash \sigma(\texttt{this}) : t}{h,\sigma \vdash \texttt{this} : t}(\textsc{This}) \qquad \frac{h,\sigma \vdash \sigma(\texttt{x}) : t}{h,\sigma \vdash \texttt{x} : t}(\textsc{Var}) \qquad \frac{}{h,\sigma \vdash \texttt{new}\ t : t}(\textsc{New})$$

$$\frac{}{h,\sigma \vdash \texttt{null} : t}(\textsc{Null}) \qquad \frac{h,\sigma \vdash e : t'}{h,\sigma \vdash (t)\ e : t}(\textsc{Cast}) \qquad \frac{h,\sigma \vdash e : t}{h,\sigma \vdash \texttt{synced}_{e''}\ a\ e : t}(\textsc{Synced})$$

$$\frac{\begin{array}{l}h,\sigma \vdash e : u\ c \\ \mathcal{M}(c,m) = t_r\ m(u \rhd t) \\ h,\sigma \vdash e' : t\end{array}}{h,\sigma \vdash e.m(e') : u \rhd t_r}(\textsc{Call}) \qquad \frac{\begin{array}{l}h,\sigma' \vdash e : t \\ h,\sigma \vdash \sigma'(\texttt{this}) : u\ \_\end{array}}{h,\sigma \vdash \texttt{frame}\ \sigma'\ e : u \rhd t}(\textsc{Frame})$$

$$\frac{\begin{array}{l}h,\sigma \vdash e : t' \\ t' < t\end{array}}{h,\sigma \vdash e : t}(\textsc{Sub}) \qquad \frac{\begin{array}{l}h,\sigma \vdash e : u\ c \\ \mathcal{F}(c,f) = t\end{array}}{h,\sigma \vdash e.f : u \rhd t}(\textsc{Field}) \qquad \frac{\begin{array}{l}h,\sigma \vdash e : u\ c \\ h,\sigma \vdash e' : t \\ \mathcal{F}(c,f) = u \rhd t\end{array}}{h,\sigma \vdash e.f = e' : t}(\textsc{Assign})$$

$$\frac{\begin{array}{l}h,\sigma \vdash e : t \\ h,\sigma \vdash e' : t'\end{array}}{h,\sigma \vdash \texttt{sync}_{e''}\ e'\ e : t}(\textsc{Sync}) \qquad \frac{\begin{array}{l}h(a)\downarrow_2 = c \\ \sigma(\texttt{this}), h(\sigma(\texttt{this}))\downarrow_1 \vdash a, h(a)\downarrow_1 : u\end{array}}{h,\sigma \vdash a : u\ c}(\textsc{Addr})$$

$$\frac{\forall i \in \{1..n\}\ .\ h,\sigma \vdash e_i : t_i}{h,\sigma \vdash e_{1..n} : t_{1..n}}(\textsc{Threads}) \qquad \frac{h,\sigma \vdash e : t'}{h,\sigma \vdash \texttt{spawn}\ e : t}(\textsc{Spawn})$$

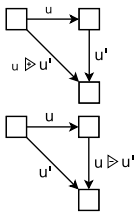
$$\frac{\begin{array}{l}h(a) = (\_, c, flds) \\ \forall \mathcal{F}(c,f) = t\ \ .\ \ h,(a,\_) \vdash flds(f) : t\end{array}}{h \vdash a}(\textsc{WFAddr})$$

Heap well-formedness:
$\vdash h \iff \forall a \in dom(h).h \vdash a$

Fig. 8. Run-time universe type system

$u \rhd u'$		u' self	u' peer	u' rep	u' any
u	self	self	peer	rep	any
	peer	peer	peer	any	any
	rep	rep	rep	any	any
	any	any	any	any	any

$u \rhd u'$		u' self	u' peer	u' rep	u' any
u	self	self	peer	rep	any
	peer	peer	peer	any	any
	rep	any	any	peer	any
	any	any	any	any	any



We extend to types by defining: $u \rhd (u'\ c) = (u \rhd u')\ c$ and $u \rhd (u'\ c) = (u \rhd u')\ c$

Fig. 9. Universe composition and decomposition

meant in respect to `this`. The ownership type qualifier `self` (a specialisation of `peer`) is used for the parameter `this`, e.g.

$$(\texttt{self Dept}, \_) \vdash \texttt{this.first.s} : \texttt{rep Student}$$
$$(\texttt{self Hall}, \_) \vdash \texttt{this.first.s} : \texttt{any Student}$$

The ownership type qualifier `self` has a special purpose – when calling local methods and accessing local fields, we want the type of such accesses to be exactly the annotation u given in the class. Note that $\texttt{self} \rhd u = u$. If we were to use `peer` instead of `self` as the type of `this`, then we would lose information in the case where $u = \texttt{rep}$ and the type system would be unnecessarily restrictive.

There is one aspect of this type system that deserves detailed discussion because we use it later. The purpose of $u \rhd u'$ is to determine the type that best describes an object that is "twice removed" from the observer by references of type u and u'. In other words, we have two subsequent references and two respective types, and we want to know the type of the most distant object from the observer. $(\rhd)$ is defined in Fig. 9. The object (1) in Fig. 1 observes the object (8) to be `peer`. (8) considers (9) to be `rep`, so (1) considers (9) to be $\texttt{peer} \rhd \texttt{rep} = \texttt{any}$.

We use $(\rhd)$ to 'translate' a type u' from one observer to another, where the old observer is u with respect to the new observer. This is useful for class member lookups where the type of the member is from the perspective of the object that contains it, but we want a type from the caller's perspective.

The complement of $(\rhd)$ is $(\triangleright)$, which we use to translate a type to another observer. The object (1) observes the objects $(2, 3)$ to be `rep`, however object (2) considers (3) to be $\texttt{rep} \triangleright \texttt{rep} = \texttt{peer}$.

Finally, we require classes to be well-formed. The types of method bodies must agree with their signatures. Note that when typing a method body, we use `self` in the type of `this`. This is consistent with our notion of observer for method bodies as described above. We also require consistency between field and method signatures in subclasses.

To prove soundness of this system, we need a type system for run-time expressions. This type system is capable of typing addresses using the owner stored in the heap, and typing variables `x` and `this` using the stack σ. The judgement is $h, \sigma \vdash e : t$, it is given in Fig. 8. In Appendix A we give several lemmas, including a substitution lemma mapping the static to the run-time type system. We finally give the soundness theorem for single threads and the multithreaded system, the proofs of which are in [7].

3 Race Safety

3.1 Static Types for Race Safety

In Fig. 10 we give a type system that requires correct synchronisation and thus guarantees race safety. The judgement $\mathbb{L}, \Gamma \vdash e : F$ denotes that the expression e is race free if all locks l in the *synchronisation set* $\mathbb{L}$ have been acquired for the duration of its execution. A lock l is either an ownership type qualifier $u \neq \texttt{any}$ or a path. The set F is the *effect* of e, *i.e.,* the set of fields that e may write to as it executes.

$$\frac{}{\emptyset, \Gamma \vdash \texttt{x} : \emptyset}\,(\textsc{Var}) \qquad \frac{}{\emptyset, \Gamma \vdash \texttt{this} : \emptyset}\,(\textsc{This}) \qquad \frac{}{\emptyset, \Gamma \vdash \texttt{new}\ t : \emptyset}\,(\textsc{New})$$

$$\frac{}{\emptyset, \Gamma \vdash \texttt{null} : \emptyset}\,(\textsc{Null}) \qquad \frac{\mathbb{L}, \Gamma \vdash e : F}{\mathbb{L}, \Gamma \vdash (t)\ e : F}\,(\textsc{Cast}) \qquad \frac{\emptyset, \Gamma \vdash e : \_}{\emptyset, \Gamma \vdash \texttt{spawn}\ e : \emptyset}\,(\textsc{Spawn})$$

$$\frac{\begin{array}{l}\mathbb{L}', \Gamma \vdash e : F' \\ \mathbb{L}' \subseteq \mathbb{L} \qquad F' \subseteq F \\ \forall p \in \mathbb{L}.\mathbb{L}, \Gamma \vdash p : \_ \\ \mathbb{L} \# F\end{array}}{\mathbb{L}, \Gamma \vdash e : F}\,(\textsc{Sub}) \qquad \frac{\begin{array}{l}\mathbb{L}, \Gamma \vdash e : F \\ \Gamma \vdash_{gb} e : l \\ \mathbb{L} \cup \{l\}, \Gamma \vdash e' : F\end{array}}{\mathbb{L}, \Gamma \vdash \texttt{sync}\ e\ e' : F}\,(\textsc{Sync}) \qquad \frac{\begin{array}{l}\mathbb{L}, \Gamma \vdash e : F \\ \Gamma \vdash_{gb} e : l \\ l \in \mathbb{L}\end{array}}{\mathbb{L}, \Gamma \vdash e.f : F}\,(\textsc{Field})$$

$$\frac{\begin{array}{ll}\mathbb{L}, \Gamma \vdash e : F & \Gamma \vdash_{gb} e : l \\ \mathbb{L}, \Gamma \vdash e' : F & l \in \mathbb{L} \quad f \in F\end{array}}{\mathbb{L}, \Gamma \vdash e.f = e' : F}\,(\textsc{Assign}) \qquad \frac{\begin{array}{ll}\mathbb{L}, \Gamma \vdash e : F & \Gamma \vdash e : u\ c \\ \mathbb{L}, \Gamma \vdash e' : F & \mathcal{Eff}(c, m)\downarrow_2 \subseteq F \\ \mathbb{L}' \in \mathcal{Eff}(c, m)\downarrow_1 & (u, e, e') \rhd \mathbb{L}' \subseteq \mathbb{L}\end{array}}{\mathbb{L}, \Gamma \vdash e.m(e') : F}\,(\textsc{Call})$$

$$\frac{\begin{array}{l}\forall c' \geq c\ .\ F' = \mathcal{Eff}(c', m)\downarrow_2 \implies \mathcal{Eff}(c, m)\downarrow_2 \subseteq F', \\ \qquad \mathbb{L}' \in \mathcal{Eff}(c', m)\downarrow_1 \implies \exists \mathbb{L} \in \mathcal{Eff}(c, m)\downarrow_1\ .\ \mathbb{L} \subseteq \mathbb{L}' \\ \forall \mathcal{M}(c, m) = t_r\ m(t_x), \mathbb{L} \in \mathcal{Eff}(c, m)\downarrow_1\ . \\ \qquad \mathbb{L}, (\texttt{self}\ c, t_x) \vdash \mathcal{MBody}(c, m) : \mathcal{Eff}(c, m)\downarrow_2\end{array}}{\vdash c}\,(\textsc{WFClass})$$

where (#) and ($\rhd$) are defined below:

$\mathbb{L} \# F \iff \forall f \in F, p \in \mathbb{L}\ .\ f \notin p$

$$\frac{\begin{array}{l}\Gamma \vdash e : u\ \_ \\ u \neq \texttt{any}\end{array}}{\Gamma \vdash_{gb} e : u}\,(\textsc{Univ}) \qquad \frac{}{\Gamma \vdash_{gb} p : p}\,(\textsc{Path})$$

$$\begin{array}{lll}(u, \_, \_) \rhd u' = u \rhd u' & \text{if } u \rhd u' \neq \texttt{any} & (\text{undefined if } u \rhd u' = \texttt{any}) \\ (\_, p, \_) \rhd p' = p'[p/\texttt{this}] & \text{if } p' = \texttt{this}\ldots & \\ (\_, \_, p) \rhd p' = p'[p/\texttt{x}] & \text{if } p' = \texttt{x}\ldots & \\ (u, e, e') \rhd \mathbb{L} = \{\ (u, e, e') \rhd l \mid l \in \mathbb{L}\ \} & & \\ \qquad (\text{undefined if } (u, e, e') \rhd l \text{ is undefined for any } l \in \mathbb{L}) & &\end{array}$$

The function $\mathcal{Eff}$ returns pairs of sets of synchronisation sets and sets of fields, and paths are defined below (static paths do not contain addresses a):

$\mathcal{Eff} : (Id^c \times Id^m) \to (\mathbb{P}(\mathbb{P}(\mathbb{L})) \times \mathbb{P}(Id^f))$ $\qquad p ::= \texttt{this} \mid \texttt{x} \mid a \mid p.f$

Fig. 10. Static race safety type system

Well-typed expressions do not overwrite fields appearing in their synchronisation set, and their locks are prefix complete (e.g. $x.f \in \mathbb{L} \Rightarrow x \in \mathbb{L}$):

Lemma 3.1 *The effects of well-typed expressions do not undermine their locks.*

$$\mathbb{L}, \Gamma \vdash e : F \Longrightarrow \mathbb{L} \# F, \quad \forall p \in \mathbb{L}.\mathbb{L}, \Gamma \vdash p : \_$$

Proof: induction on the derivation of $\mathbb{L}, \Gamma \vdash e : F$.

We say that an expression is *internally synchronised* if it can be typed with an empty synchronisation set, otherwise it is *externally synchronised*.

We now discuss the type system in greater detail. Suppose we have $\mathbb{L}, \Gamma \vdash e : \_$ The synchronisation set and effect of variables and constants are empty, *c.f.,* (NULL), (VAR), (THIS). This also holds for (NEW), as object creation does not interact with other threads. A cast does not require more locks or produce more effects than its sub-term, *c.f.,* (CAST). Spawning requires the new thread to be internally synchronised, and therefore requires its sub-term to have an empty synchronisation set. Since the sub-term is executed in a new thread, its effect is of no interest to the current thread, therefore the whole expression has empty effect, *c.f.,* (SPAWN).

The (SUB) rule is a form of subsumption as it increases the effect and synchronisation set, provided that none of the fields in the new effects F appear in any of the paths of the new synchronisation set $\mathbb{L}$, thus preserving lemma 3.1.

The (FIELD) and (ASSIGN) rules are similar. They calculate the lock l that guards the object access in question, using the *guarded by* judgement $\Gamma \vdash_{gb} e : l$. This lock must be acquired before the execution of the access in order to guarantee race safety, therefore l is included in the synchronisation set $\mathbb{L}$. The judgement finds the owner via the ownership type qualifier u provided that $u \neq$ `any` (UNIV), or uses the path p when e is such a path (PATH). If both rules are applicable, then the locks obtained will be syntactically different (e.g. `self` and `this`), but will indicate the same owner. [9]

The (SYNC) rule calculates the synchronisation set for the expression `sync` $e\ e'$ by removing the lock that guards the object accessed by e from the synchronisation set of e'.

Because methods in our system are not necessarily internally synchronised, we extend their signatures through $\mathcal{Eff}$, whose shape is defined in Fig. 10, which returns a set of synchronisation sets, and a set of fields to which the method body may assign. The (CALL) rule thus requires that these locks and assignments are included in the resulting synchronisation set and effects F. [10] [11] Because the synchronisation sets expressed in $\mathcal{Eff}$ are given from the perspective of the target of the method call, they need to be translated into the perspective of the receiver before being used. This is done through the operaor $\triangleright$, defined in Fig. 10.

Well-formed classes, *c.f.,* (WFCLASS), requires, in addition to the requirements imposed for universe type soundness, that: Firstly, if a method m existed in a

[9] Obviously, if neither rule is applicable the expression is type incorrect.

[10] The reason we use a *set* of synchronisation sets, rather than one single synchronisation set, is, that there may be more than one way to correctly synchronise a method call. E.g. a method with body `this.f.s = x` might have $\mathcal{Eff}$ as follows: $(\{\{\texttt{this}, \texttt{this.f}\}, \{\texttt{this}, \texttt{rep}\}, \{\texttt{self}, \texttt{this.f}\}, \{\texttt{self}, \texttt{rep}\}\}, \{\texttt{s}\})$.

[11] The synchronisation sets will grow with the number of accesses in a method body. Therefore, in practice we would need a better syntax that scales more favourably, e.g. `this|self`, `this.first|rep`. This is outside the scope of the current paper.

superclass, then the superclass's synchronisation sets and effects should be larger than those in the subclass. Secondly, each of synchronisation sets $\mathbb{L}$ in $\mathcal{Eff}(c, m)\downarrow_1$ should be sufficient for correct synchronisation of the body of m.

3.1.1 Examples

We now discuss the application of our type rules with some examples. We first consider the body of `releaseMarks` in Fig. 2. We have an environment Γ_1 where $\Gamma_1(\texttt{this}) = \texttt{self Dept}$. Because our tiny language does not include local variables, we will consider `i` as a field in class `Dept` of type `rep DeptStudentNode`, and mentally map each appearance of `i` in Fig. 2, onto `this.i`. Thus,

$$\begin{array}{l}\emptyset, \Gamma_1 \vdash \texttt{sync (this)}\ \{\ \texttt{this.i = this.first;} \\ \qquad\qquad\qquad\qquad\quad \texttt{sync (this.}i\texttt{)}\ \{\texttt{this.i.s.mark} = \ldots\texttt{;} \\ \qquad\qquad\qquad\qquad\qquad\qquad \texttt{this.i = this.i.next}\}\ \} : \{\texttt{i, mark}\}\end{array}$$

On the other hand, in Fig. 2, line 31, we obtain the following with an environment Γ_2 where $\Gamma_2(\texttt{this}) = \texttt{self Hall}$:

$$\begin{array}{l}\{\texttt{this, this.i}\}, \Gamma_2 \vdash \texttt{sync (this.i.s)}\ \{\ \texttt{this.i.s.roomClean = ...;} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{this.i = ...}\ \} : \{\texttt{roomClean, i}\}\end{array}$$

Because `this.i.s` has type `any Student`, the type system can only use the guard rule (Path). The type system accepts the above synchronised block because we are not assigning to the fields `i` or `s` within the block. However, the expression `sync(this.i) {... this.i = this.i.next }` would be type incorrect in Γ_2, and thus we are forced to lock at every loop iteration.

The method `badCode()` in Fig. 11 accesses and synchronises `this.getFirst()` (line (9)). This is not a path, but has type `rep DeptStudentNode` so the type system can use (Univ) rule to accept the code. The `sync (this.first2)` block (line (12)) fails type checking because the path being locked is `any` and also comprises a field `first2` which is assigned during the synchronised block. The final access (line (17)) is not a path and has type `any`, so no amount of synchronisation will persuade the type system to accept it.

Finally, we give examples of method calls. Assume a method `clean()` [12] in class `Student` such that:

$$\mathcal{Eff}(\texttt{Student}, \texttt{clean}) = (\{\{\texttt{self}\}\}, \{\texttt{cleanRoom}\})$$

Then, in class `Dept`, it holds that $\Gamma_1 \vdash \texttt{this.first.s} : \texttt{rep Student}$. Thus, by application of (Call), we obtain:

$$\{\texttt{rep}\}, \Gamma_1 \vdash \texttt{this.first.s.clean()} : \{\texttt{cleanRoom}\}$$

In class `Hall`, $\Gamma_2 \vdash \texttt{this.first.s} : \texttt{any Student}$. Here, the call `this.first.s.clean()` causes a type error. On the other hand, with a method `makeBed()`, where:

$$\mathcal{Eff}(\texttt{Student}, \texttt{makeBed}) = (\{\{\texttt{self}\}, \{\texttt{this}\}\}, \{\texttt{cleanRoom}\})$$

[12] For simplicity, we ignore method parameters.

```
1  class BrokenDept extends Dept {
2      any Student first2;
3      any DeptStudentNode getFirst2() {
4          sync (this) { return this.first2; }
5      }
6      rep DeptStudentNode getFirst() {
7          sync (this) { return this.first; }
8      }
9      void badCode() {
10         sync (this) {
11             sync (this.getFirst()) {
12                 this.getFirst().s = NULL;
13             }
14             sync (this.first2) { // FAIL
15                 this.first2 = this.first2.next;
16                 this.first2.s = NULL;
17             }
18             sync (???) {
19                 this.getFirst2().s = NULL; //FAIL
20 }   }   }   }
```

Fig. 11. Example

We would obtain

$$\mathbb{L}, \Gamma_2 \vdash \texttt{this.first.s.makeBed()} : \{\texttt{cleanRoom}\}$$
$$(\text{where } \mathbb{L} = \{\texttt{this}, \texttt{this.first}, \texttt{this.first.s}\})$$

3.2 Run-time Type System

As is standard, we give a run-time type system in order to prove soundness and race safety (presented in Fig. 12). We type run-time expressions e according to a heap h and stack σ. The judgement has the shape $\mathbb{L}, h, \sigma \vdash e : F$. The meanings of $\mathbb{L}$ and F are unchanged. The function $h(\sigma, p)$ executes p in the given heap and stack to retrieve a value in a finite number of steps bounded by the size of p. If this process attempts to dereference `null`, we define it to return `null`. In Fig. 10 we used (PATH) and (Univ) to derive locks from expressions. We needed to extend this functionality to derive locks from partially executed expressions so we added the rule (VAL) and replaced (PATH) by the rules (VAR) and (FIELD). (UNIV) was changed to use the run-time universe type system, which understands partially executed expresions. (CALL) needed us to extend ($\triangleright$) to translate locks in the context of partially executed targets and arguments.

The type system is lifted to the sequence of threads that ultimately comprises our model of the run-time state by (THREADS). We require all the threads to be internally synchronised and also that no two threads have the same lock. The shape of the judgement is $h, \sigma \vdash \overline{e}$.

We use the predicate $Virgin(e)$ to note that e has not yet been executed *i.e.,* contains no addresses, `synced` or `frame` constructs. $Reachable(e)$ (Fig.13)

$$\frac{h(\sigma,p)=v}{h,\sigma\vdash_{gb} v:p}\text{(VAL)} \qquad \frac{p\in\{\texttt{x},\texttt{this}\}}{h,\sigma\vdash_{gb} p:p}\text{(VAR)} \qquad \frac{h,\sigma\vdash_{gb} p:p'}{h,\sigma\vdash_{gb} p.f:p'.f}\text{(FIELD)}$$

$$\frac{\begin{array}{ll}\mathbb{L},h,\sigma\vdash e:F & \mathbb{L}'\in \mathit{Eff}(c,m)\downarrow_1\\ \mathbb{L},h,\sigma\vdash e':F & \mathit{Eff}(c,m)\downarrow_2\subseteq F\\ h,\sigma\vdash e:u\ c & (h,\sigma,u,e,e')\Subset\mathbb{L}'\subseteq\mathbb{L}\end{array}}{\mathbb{L},h,\sigma\vdash e.m(e'):F}\text{(CALL)} \qquad \frac{\begin{array}{ll}\mathbb{L}',h,\sigma'\vdash e:F & \\ \sigma'=(a,v) & h,\sigma\vdash a:u\ \_\\ \mathbb{L}=(h,\sigma,u,a,v)\Subset\mathbb{L}' & \end{array}}{\mathbb{L},h,\sigma\vdash \texttt{frame}\ \sigma'\ e:F}\text{(FRAME)}$$

$$\frac{\begin{array}{ll}h,\sigma\vdash_{gb} e':l & \mathbb{L},h,\sigma\vdash e':F\\ h,\sigma\vdash_{gb} e'':l & \mathbb{L},h,\sigma\vdash e'':F\\ \mathbb{L}\cup\{\ l\ \},h,\sigma\vdash e:F & \end{array}}{\mathbb{L},h,\sigma\vdash \texttt{sync}_{e''}\ e'\ e:F}\text{(SYNC)} \qquad \frac{\begin{array}{ll}h,\sigma\vdash_{gb} a:l & h(a)\downarrow_1=w\\ h,\sigma\vdash_{gb} e':l & \mathbb{L},h,\sigma\vdash e':F\\ \mathbb{L}\cup\{\ l\ \},h,\sigma\vdash e:F & \end{array}}{\mathbb{L},h,\sigma\vdash \texttt{synced}_{e'}\ w\ e:F}\text{(SYNCED)}$$

$$\frac{}{\emptyset,h,\sigma\vdash a:\emptyset}\text{(ADDR)}$$

(VAR) (THIS) (NULL) (SUB) (CAST) (NEW) (FIELD) (ASSIGN) (SPAWN) (UNIV) are the same as in Fig. 10 but with Γ replaced by h,σ

$$\frac{\begin{array}{l}\forall i\in\{1..n\}\ .\ \emptyset,h,\sigma\vdash e_i:\_\\ \forall i,j\in\{1..n\},w\ .\ \mathit{Locked}(e_i,w)\wedge\mathit{Locked}(e_j,w)\Longrightarrow i=j\end{array}}{h,\sigma\vdash e_{1..n}}\text{(THREADS)}$$

$$\begin{aligned}(\_,\_,u,\_,\_)\Subset u' &= u\Subset u' && \text{if } u\Subset u'\neq\texttt{any}(\text{undefined if } u\Subset u'=\texttt{any})\\ (h,\sigma,\_,e,\_)\Subset p' &= p'[p/\texttt{this}] && \text{where } h,\sigma\vdash_{gb} e:p,\quad p'=\texttt{this}\ldots\\ (h,\sigma,\_,\_,e)\Subset p' &= p'[p/\texttt{x}] && \text{where } h,\sigma\vdash_{gb} e:p,\quad p'=\texttt{x}\ldots\\ (h,\sigma,u,e,e')\Subset\mathbb{L} &= \{\ (h,\sigma,u,e,e')\Subset l\mid l\in\mathbb{L}\ \}\\ & && (\text{undefined if } (h,\sigma,u,e,e')\Subset l \text{ is undefined for any } l\in\mathbb{L})\end{aligned}$$

Fig. 12. Run-time race safety type system

denotes that the subterms of e have been executed in the right order, *e.g.*, $\mathit{Reachable}(a.f=y.f)$ but $\neg\mathit{Reachable}(y.f=a.f)$. We extend this to sequences of expressions $\mathit{Reachable}(\overline{e})$ if all the expressions are reachable.

Because of the instrumentation of `sync` with a subscript that records the initial lock expression, we need to use the same substitution as used in the semantics rule (CALL) when defining the following substitution lemma:

$$
\begin{array}{ll}
\mathit{Reachable}(e) & \Longleftarrow e \in \{\texttt{x}, \texttt{this}, v, \texttt{new}\ t\} \\
\mathit{Reachable}(e.f) & \Longleftarrow \mathit{Reachable}(e) \\
\mathit{Reachable}((t)e) & \Longleftarrow \mathit{Reachable}(e) \\
\mathit{Reachable}(e_1.f = e_2) & \Longleftarrow \mathit{Reachable}(e_1) \wedge \mathit{Virgin}(e_2) \\
\mathit{Reachable}(v.f = e) & \Longleftarrow \mathit{Reachable}(e) \\
\mathit{Reachable}(e_1.m(e_2)) & \Longleftarrow \mathit{Reachable}(e_1) \wedge \mathit{Virgin}(e_2) \\
\mathit{Reachable}(v.m(e)) & \Longleftarrow \mathit{Reachable}(e) \\
\mathit{Reachable}(\texttt{spawn}\ e) & \Longleftarrow \mathit{Virgin}(e) \\
\mathit{Reachable}(\texttt{sync}_{e_1}\ e_2\ e_3) & \Longleftarrow \mathit{Virgin}(e_1) \wedge \mathit{Reachable}(e_2) \wedge \mathit{Virgin}(e_3) \\
\mathit{Reachable}(\texttt{synced}_{e_1}\ w\ e_2) & \Longleftarrow \mathit{Virgin}(e_1) \wedge \mathit{Reachable}(e_2) \\
\mathit{Reachable}(\texttt{frame}\ \sigma\ e) & \Longleftarrow \mathit{Reachable}(e)
\end{array}
$$

Fig. 13. Definition of *Reachable*

Lemma 3.2 *Static race safety implies run-time race safety*

$$
\left.
\begin{array}{l}
\mathbb{L}, \Gamma \vdash e : F \\
h, \sigma \vdash \texttt{x} : \Gamma(\texttt{x}) \\
h, \sigma \vdash \texttt{this} : \Gamma(\texttt{this})
\end{array}
\right\} \Longrightarrow
\begin{array}{l}
\mathbb{L}, h, \sigma \vdash S(e) : F \\
\mathit{Virgin}(S(e))
\end{array}
$$

Proof: Induction over derivation of $\mathbb{L}, \Gamma \vdash e : F$

Using the above lemma in the case of method calls, it is possible to prove the soundness of the race safety type system. Firstly we state soundness for single-threaded execution. Note that we require the heap to be well-formed. This is necessary so that field accesses yield objects of the correct owner. We give some lemmas that lead to this result in Appendix B.

Lemma 3.3 *The type of a thread is preserved over the execution of that thread.*

$$
\left.
\begin{array}{l}
\mathit{Reachable}(e) \\
\vdash h \\
h, \sigma \vdash e : t \\
\mathbb{L}, h, \sigma \vdash e : F \\
\sigma \vdash e, h \rightsquigarrow e', h'
\end{array}
\right\} \Longrightarrow
\begin{array}{l}
\mathbb{L}, h', \sigma \vdash e' : F \\
\mathit{Reachable}(e')
\end{array}
$$

Proof: Induction over derivation of $\mathbb{L}, h, \sigma \vdash e : F$

The soundness of the complete type system can now be stated:

Theorem 3.4 *Well-typedness of the system is preserved over execution.*

$$\left.\begin{array}{l} \vdash h \\ h, \sigma \vdash \overline{e} : \overline{t} \\ h, \sigma \vdash \overline{e} \\ \sigma \vdash \overline{e}, h \rightsquigarrow \overline{e}', h' \\ \mathit{Reachable}(\overline{e}) \end{array}\right\} \Longrightarrow \begin{array}{l} h', \sigma \vdash \overline{e}' \\ \mathit{Reachable}(\overline{e}') \end{array}$$

Proof: Case analysis of $\sigma \vdash \overline{e}, h \rightsquigarrow \overline{e}', h'$

Now we work towards a theorem of race safety, i.e. that well-typed programs can exhibit no race conditions. Firstly the following lemma states that objects are only accessed if the appropriate lock has been acquired by the thread in question. Note the use of the action a to denote an access of address a by the execution step.

Lemma 3.5 *Objects are only accessed while their owners are locked.*

$$\left.\begin{array}{l} \mathbb{L}, h, \sigma \vdash e : \_ \\ \sigma \vdash e, h \overset{a}{\rightsquigarrow} \_, \_ \end{array}\right\} \Longrightarrow \begin{array}{l} (\exists l \in \mathbb{L}. h, \sigma \vdash_{gb} a : l) \vee \\ \mathit{Locked}(e, h(a)\downarrow_1) \end{array}$$

Proof: Induction over structure of $\mathbb{L}, h, \sigma \vdash e : \_$

The multi-threaded case follows. We are dealing with entire threads here (as opposed to sub-terms), so we do not need the set $\mathbb{L}$ of locks taken in the current context.

Theorem 3.6 *Objects are only accessed while their owners are locked by the corresponding thread.*

$$\left.\begin{array}{l} h, \sigma \vdash e_{1..n} \\ \sigma \vdash e_{1..n}, h \overset{(i,a)}{\rightsquigarrow} \_, \_ \end{array}\right\} \Longrightarrow \mathit{Locked}(e_i, h(a)\downarrow_1)$$

Proof: Case analysis of $\sigma \vdash e_{1..n}, h \overset{(i,a)}{\rightsquigarrow} \_, \_$

For race conditions, we use the definition from [10], where the state of a multi-threaded system exhibits an *instantaneous race condition* if the semantics allows two possible execution steps of different threads, that both access the same object. The required non-determinism is provided by the (INTERLEAVE) rule.

We prove that no well-typed state can ever have an instantaneous race condition, and by theorem 3.4, all intermediate states of execution will be free from instantaneous race conditions. We show, for an arbitrary well-typed run-time state, that if two possible execution steps can access the same object, then those steps must be steps of the same thread:

Theorem 3.7 *Race safety*

$$\left.\begin{array}{l} h, \sigma \vdash \overline{e} \\ \sigma \vdash \overline{e}, h \overset{(i,a)}{\rightsquigarrow} \_, \_ \\ \sigma \vdash \overline{e}, h \overset{(j,a)}{\rightsquigarrow} \_, \_ \end{array}\right\} \Longrightarrow i = j$$

Proof: Application of Theorem 3.6 and (THREADS).

4 Implementation Issues

We hope to implement our type system, and have considered ways to achieve good performance within our semantics.

The constraints on extending classes may seem severe, particularly the requirement that an overriding method cannot have more effects F than the original method. However, if we forsake separate compilation, we can infer the effect F of each method, and thus we do not need to restrict inheritance. We believe the constraints on the synchronisation set required for a call to be race safe are not so severe because most functions will be internally synchronised, whereas one cannot hide field assignments F within a function. In general separate compilation does not mix well with concurrency since concurrency is concerned with the effect of the rest of the program. Separate compliation requires specification at module boundaries, and a behaviour specification is quite hard to write and maintain. In our work such a specification is represented by the sets $\mathbb{L}$ and F.

In practice, we could use more accurate techniques of alias detection, *e.g.,* using a points-to analysis, to refine the type system's judgement about whether a field assignment affects a path p in $\mathbb{L}$. This would allow us to reject fewer correct programs. We could easily distinguish between identically named fields in different classes by prepending the class name to all fields.

It may be useful to use the method of [4] for preventing deadlock. We can require the programmer to write additional type annotations to firstly divide the heap into a statically bounded set of regions [24] and secondly specify a partial order over these regions. Types are therefore augmented with a region identifier, which specifies the region where the owner of the object lies (locks are associated with the owner). The type system would check that the specified order has no cycles, and that assignments do not let variables of a certain region reference objects of other regions.

To actually prevent deadlocks, the type system has to ensure that locks are taken (i.e. sync blocks are nested) in the order specified. Method signatures can be annotated with the lowest lock that they take, and thus each call can be checked to make sure its context has not already locked any higher than the method will lock.

Since we require an implicit owner field in all objects, for casts and synchronisation of `any` expressions, there may unnecessary memory overhead [13]. If this becomes an issue, we propose introducing new types to lay alongside `rep`, `peer`,

[13] In some cases, previous work required the programmer to use an explicit owner field, see Fig. 3

and `self` which would have identical semantics except that they may not be cast to `any`. Objects of these types would not require an owner field since the owner would always be statically known. We anticipate that programmers would use these types only when memory use was a problem. Static analysis could also be used to infer which objects are never cast to `any`, and optimise away the owner field.

Other type systems [2,12,14,3,4] have extra features such as thread local storage and final variables. We did not formalise them, as although they are useful in practice, they are well-understood and can be easily added to an implementation.

4.1 Distinguishing Reads and Writes

We consider two simultaneous accesses of the same object by different threads to be a race condition, but this need only be so if one of the accesses is a write. Distinguishing between reads and writes would allow more liberal synchronisation. We now show how this can be done with an extension of our formalism.

We need to distinguish between read and write accesses, i.e. distinguish between field lookups and field assignments. We also need read/write locks, i.e. we need a pair of constructs like $\texttt{sync}^R(...)$ and $\texttt{sync}^W(...)$ that attempt to acquire the read/write lock on an object, respectively. The semantics would allow multiple threads to have the read lock at any one time, so long as no other thread was writing. This would be reflected in the rule (Threads) which would ensure that if a thread is writing then no other thread can write. These locks have already been implemented in Java, and are easy to add to the type system.

4.2 Single Object Locking

While we need to lock the whole ownership domain when iterating through nodes as in Fig. 2 (line 21), we do not need to lock all the peers of an object if we do not use it for iteration. E.g. we do not need to lock the peers of `this` when we access `this.first`. We'd like to give the programmer the choice of when to lock the whole domain (as is currently available with `sync`) and when to lock only a single object. Then, it would be possible for two threads to execute in parallel the `releaseMarks` method of the two departments in the diagram of Fig. 1. We may extend the type system and semantics as shown in Fig. 14.

Programmers use `syncobj` when they only want to lock the object in question e.g. Fig. 2 (line 19), and the old `sync` construct if they want to lock that object and all of its peers. We do not let a thread lock a whole domain if any of the objects in that domain have been locked. Likewise, we do not let a thread lock a single object if another thread has locked that object's domain. We use the old predicate $\mathit{Locked}(e, w)$ but we also add $\mathit{LockedObj}(e, a)$ which holds when e contains `syncedobj_` a `_`.

We augment the lock syntax so the whole-domain locks are now denoted with $(*, u)$ or $(*, p)$, whereas the single object locks are denoted with $(1, p)$. We do not need $(1, u)$, as this would describe a whole domain. We updated the (Univ) and (Path) rules to wrap their result in $(*, \ldots)$ and give a new guard rule that returns $(1, p)$ if the expression is a path. The type rule for threads guarantees that no two threads have conflicting locks.

$$\frac{\mathbb{L}, \Gamma \vdash e : F \quad \Gamma \vdash_{gb} e : l = (*, \_) \quad \mathbb{L} \cup \{l\}, \Gamma \vdash e' : F}{\mathbb{L}, \Gamma \vdash \texttt{sync}\ e\ e' : F}\ (\textsc{Sync})$$

$$\frac{\mathbb{L}, \Gamma \vdash e : F \quad \Gamma \vdash_{gb} e : l = (1, \_) \quad \mathbb{L} \cup \{l\}, \Gamma \vdash e' : F}{\mathbb{L}, \Gamma \vdash \texttt{syncobj}\ e\ e' : F}\ (\textsc{SyncObj})$$

$$\frac{\begin{array}{l} e_i = C[\texttt{sync}_{e'}\ a\ e] \qquad w = h(a)\downarrow_1 \\ \forall j \in \{1..n\}\ .\ \mathit{Locked}(e_j, w) \implies i = j \\ \qquad \forall b\ .\ (h(b)\downarrow_1 = w \wedge \mathit{LockedObj}(e_j, b)) \implies i = j \end{array}}{\sigma \vdash e_{1..n}, h \overset{(i,\tau)}{\rightsquigarrow} e_{1..i-1}\ C[\texttt{synced}_{e'}\ w\ e]\ e_{i+1..n}, h}\ (\textsc{LockDomain})$$

$$\frac{\begin{array}{l} e_i = C[\texttt{syncobj}_{e'}\ a\ e] \\ \forall j \in \{1..n\}\ .\ \mathit{Locked}(e_j, h(a)\downarrow_1) \implies i = j \\ \qquad \mathit{LockedObj}(e_j, a) \implies i = j \end{array}}{\sigma \vdash e_{1..n}, h \overset{(i,\tau)}{\rightsquigarrow} e_{1..i-1}\ C[\texttt{syncedobj}_{e'}\ a\ e]\ e_{i+1..n}, h}\ (\textsc{LockObj})$$

Source syntax:	Runtime syntax:
$e ::= ... \mid \texttt{syncobj}\ e\ e$	$e ::= ... \mid \texttt{syncobj}_e\ e\ e \mid \texttt{syncedobj}_e\ a\ e$

$$\frac{\begin{array}{l} \forall i \in \{1..n\}\ .\ \emptyset, h, \sigma \vdash e_i : \_ \\ \forall i, j \in \{1..n\}, h(a)\downarrow_1 = w\ . \\ \qquad \mathit{Locked}(e_i, w) \wedge \mathit{Locked}(e_j, w) \implies i = j \\ \qquad \mathit{Locked}(e_i, w) \wedge \mathit{LockedObj}(e_j, a) \implies i = j \\ \qquad \mathit{LockedObj}(e_i, a) \wedge \mathit{LockedObj}(e_j, a) \implies i = j \end{array}}{h, \sigma \vdash e_{1..n}}\ (\textsc{Threads})$$

Fig. 14. Single object locking

An efficient implementation might use a counter in the owner of an object to record how many of its child objects have been individually locked, rather than iterating through them all. Lock-free programming techniques can be used to ensure this has negligible performance cost compared to a standard mutex implementation. There has been a lot of research [1] into increasing the performance of lock operations in the most common cases. We have implemented a proof-of-concept prototype of a single object lock. It lacks features such as re-entrancy and readers/writers but early tests show that it performs only a few percent worse than a `java.util.concurrent.locks.ReentrantLock`. The performance can be further improved by refining our naive implementation to use ideas from previous work.

We believe single object locking (in the context of static race safety checkers)

is a novel idea. None of the prior work supports it. It is a small extension of our formalism, and as such we did not incorporate it into our proofs; however, in practice it should allow more parallelism without much linear cost.

4.3 Atomicity

As described by Flanagan [15], there are program misbehaviours due to thread interactions that are not classed as race conditions by the standard definition [14]. Ideally, we would like to identify *atomicity violations* [13,14,15] in programs, which would include bugs such as stale value errors that we currently cannot detect. In fact, a program that does not type in our system, can be "corrected" by wrapping each access in a tiny synchronised block, thus converting all its race conditions to stale value errors.

Atomicity checking basically requires that non-redundant `sync` blocks are always nested (never composed) in an atomic block. As noted elsewhere [27], checking atomicity relies on checking race safety, so it is natural to provide this atomicity checking as an extension to our basic system.

We require the programmer to specify that certain blocks of code are intended to be atomic with the syntax `atomic` e. The type system will ensure that the execution of these blocks in the context of arbitrary race-free threads will be equivalent to a serialised execution with no interleaving of other threads. Our extension is a strengthening of our existing type system.

We propose the additional judgement $\mathbb{A}, \mathbb{B}, \Gamma \vdash e : F$ which indicates what locks need to be taken to guarantee an expression is atomic. $\mathbb{A}$ and $\mathbb{B}$ are sets of locks just like $\mathbb{L}$, and are supplied for each method by the programmer (we leave inference as further work). F is the set of accessed fields we used previously. Object accesses generate locks in $\mathbb{A}$ and $\mathbb{B}$ like they have previously in $\mathbb{L}$. Locks are not eliminated from $\mathbb{B}$ by (Sync). $\mathbb{A}$ is the set of locks that need to be taken before e will be atomic. $\mathbb{B}$ is the set of locks required for an expression to be a *both mover* [15]. If e is atomic and e' is a both mover (or vice versa), then $e; e'$ is atomic, however $e; e'$ is only a both mover if both e and e' are both movers. The value chosen for $\mathbb{A}$ is illustrated by the rule for a sequential composition operator $(;)$, if one was added to the model:

$$\frac{\mathbb{A}, \mathbb{B}, \Gamma \vdash e : F \quad \mathbb{A}', \mathbb{B}', \Gamma \vdash e' : F \qquad \mathbb{A}'' \in \{\mathbb{A} \cup \mathbb{B}', \mathbb{A}' \cup \mathbb{B}\}}{\mathbb{A}'', \mathbb{B} \cup \mathbb{B}', \Gamma \vdash e; e' : F} \text{(Seq)}$$

For $e; e'$ to be atomic, we require enough locks so that e is atomic and e' is a both mover, or vice versa. For $e; e'$ to be a both mover we need to take all the locks generated by their accesses, regardless of the synchronisation present in e and e'. The idiom $\mathbb{A}'' \in \{\mathbb{A} \cup \mathbb{B}', \mathbb{A}' \cup \mathbb{B}\}$ is used where one sub-term is executed after another, and is used in (Assign), (Sync), and also in (Call) which needs to consider the body of the method and thus involves 3 sets. The full system is presented in Fig. 15. We include $\mathbb{A}$ and $\mathbb{B}$ in method type signatures, which we model with $\mathcal{E}\mathit{ff}(c, m)\downarrow_3$.

[14] Programmers often use "race condition" to describe these errors as well.

$$\frac{\_, \_, \Gamma \vdash e : \_}{\emptyset, \{\bot\}, \Gamma \vdash \texttt{spawn}\ e : \emptyset}\ (\textsc{Spawn})$$

$$\frac{e \in \{\texttt{null}, \texttt{x}, \texttt{this}, \texttt{new}\ t\}}{\emptyset, \emptyset, \Gamma \vdash e : \emptyset}\ (\textsc{Triv})$$

$$\frac{\begin{array}{l}\mathbb{A}, \mathbb{B}, \Gamma \vdash e : F \qquad \Gamma \vdash_{gb} e : l \\ \mathbb{A}' \cup \{l\}, \mathbb{B}', \Gamma \vdash e' : F \quad l \in \mathbb{B}' \\ \mathbb{A}'' \in \{\mathbb{A} \cup \mathbb{B}', \mathbb{A}' \cup \mathbb{B}\}\end{array}}{\mathbb{A}'', \mathbb{B} \cup \mathbb{B}', \Gamma \vdash \texttt{sync}\ e\ e' : F}\ (\textsc{Sync})$$

$$\frac{\begin{array}{l}\mathbb{A}', \mathbb{B}', \Gamma \vdash e : F' \\ \mathbb{A}' \subseteq \mathbb{A}, \qquad \mathbb{B}' \subseteq \mathbb{B}, \\ F' \subseteq F, \quad \mathbb{A}, \mathbb{B} \# F\end{array}}{\mathbb{A}, \mathbb{B}, \Gamma \vdash e : F}\ (\textsc{Sub})$$

$$\frac{\begin{array}{l}\mathbb{A}, \mathbb{B}, \Gamma \vdash e : F \qquad \Gamma \vdash_{gb} e : l \\ \mathbb{A}', \mathbb{B}', \Gamma \vdash e' : F \quad l \in \mathbb{A}'', \mathbb{B}'' \quad f \in F \\ \mathbb{B}'' = \mathbb{B} \cup \mathbb{B}' \quad \mathbb{A}'' \in \{\mathbb{A} \cup \mathbb{B}', \mathbb{A}' \cup \mathbb{B}\}\end{array}}{\mathbb{A}'', \mathbb{B}'', \Gamma \vdash e.f = e' : F}\ (\textsc{Assign})$$

$$\frac{\mathbb{A}, \mathbb{B}, \Gamma \vdash e : F}{\mathbb{A}, \mathbb{B}, \Gamma \vdash (t)\ e : F}\ (\textsc{Cast})$$

$$\frac{\begin{array}{l}\mathbb{A}, \mathbb{B}, \Gamma \vdash e : F \\ \Gamma \vdash_{gb} e : l \\ l \in \mathbb{A}, \mathbb{B}\end{array}}{\mathbb{A}, \mathbb{B}, \Gamma \vdash e.f : F}\ (\textsc{Field})$$

$$\frac{\begin{array}{l}\mathbb{A}_1, \mathbb{B}_1, \Gamma \vdash e_1 : F \qquad \mathbb{A}_2, \mathbb{B}_2, \Gamma \vdash e_2 : F \\ \Gamma \vdash e_1 : u\ c \qquad (\mathbb{A}_3, \mathbb{B}_3) \in \mathcal{Eff}(c, m)\downarrow_3 \\ \mathbb{B}_4 = (u, e_1, e_2) \triangleright \mathbb{B}_3 \quad \mathcal{Eff}(c, m)\downarrow_2 \subseteq F \\ \mathbb{A}_4 = (u, e_1, e_2) \triangleright \mathbb{A}_3 \qquad \mathbb{B}_4 \subseteq \mathbb{B}_1 \cup \mathbb{B}_2 \\ \mathbb{A}_5 \in \{\mathbb{A}_1 \cup \mathbb{B}_2 \cup \mathbb{B}_4, \mathbb{B}_1 \cup \mathbb{A}_2 \cup \mathbb{B}_4, \\ \qquad\qquad \mathbb{B}_1 \cup \mathbb{B}_2 \cup \mathbb{A}_4\}\end{array}}{\mathbb{A}_5, \mathbb{B}_1 \cup \mathbb{B}_2, \Gamma \vdash e_1.m(e_2) : F}\ (\textsc{Call})$$

Fig. 15. Static atomicity type system

If a new thread is spawned part-way through an atomic block e, it could take a lock not yet taken by e, and see state that should not have been visible until after e had completed. Therefore, the execution was not atomic according to the theory of reduction [23]. We prevent this by requiring the lock $\bot$ for `spawn` to be a both-mover. Such a lock is impossible to acquire, and thus `spawn` is never a both mover. This means there can be only one `spawn` in an atomic block, in the centre of the nesting of `sync` blocks. If we were to distinguish between the locks required for an expression to be a left mover and right mover, as in [15] (we currently treat such expressions as atomic), we could let spawn be a left mover. This would allow the spawning of any number of threads after the acquisition of locks in an atomic section.

5 Related Work

Previous race safety work can be divided into two categories. There are those that use ownership or guard annotations to specify the locking discipline as a relationship between objects, and there are those that use a finite set of programmer-supplied or inferred region names, and specify the locking discipline as a relationship between objects and regions.

The latter work [2,18,30] results in fewer annotations since regions are easier to infer than ownership types. However, they have the disadvantage that the set of locks is finite, and thus the program does not scale as well to many threads. This was noted in [25,18]. The more recent papers [18,30] have been able to infer the synchronisation completely. For us this would mean being able to infer the set $\mathbb{L}$, which is desirable, and we will attempt it as further work.

Of the former variety, the first work to exclude race conditions from object-oriented programs using a static type system was [10], using the concurrent object calculus. This idea was subsequently refined to more concrete models of object-oriented languages [11], which included parameterised classes and even some degree of inference [12]. These papers were variations on the same approach: The programmer supplied guard annotations in their classes; the guard annotations were a form of ownership types; this is clear through their use of final expressions and parameters, although individual fields were owned instead of entire objects. Similar results were also obtained using ownership types directly [3,4]. Our work complements these approaches by discussing a different kind of ownership type system (universes) and its application to race safety. Although it is interesting to see how static race safety can be achieved using universes, our major contributions are increased expressiveness and greater concurrency.

There is a substantial difference between our work and that just discussed; we allow paths of non-final field types (in fact our formalism does not have the `final` type qualifier) whereas [11,3] require paths to be constructed from only final field dereferences. The price we pay for forsaking this restriction is that we must find another way to ensure that the meaning of paths is not affected by the side-effects of the lock expression. For this we use a system of effects, and for this to work we require the effect of overriding methods to be restricted to that of the method they override. A combination of our approach and that of [11,3]would achieve the best of both worlds.

Effects are also used to prove preservation of properties of ownership type system in [29]. A concept similar to universes was studied in conjunction with synchronisation in [20]. This was mainly for the purpose of verifying object invariants rather than absence of race conditions. Objects can "change hands" over time, therefore their owners are not constant at run-time. Also, there is no concept of `peer`.

Locking an object in our system does not lock the whole tree as in [20,3,4]. We lock only the immediate ownership domain and further lock acquisitions are required if deeper objects are accessed. With more locks, we reduce contention and let more threads execute in parallel than [3,4].

6 Further Work

Universes may seem under-powered because they cannot express the relationship between objects when they are neither `rep` nor `peer`. However we expect that using generics [9], we will have a lot more power, e.g. when we parameterise a list to hold elements of a particular domain. We expect the extension to include generics to be straightforward.

We would also like to use *path dependent types*, which are currently being studied in the context of universes. These would allow us to use types like `x.`f`.rep`, which characterise the objects owned by `x.`f, greatly increasing our expressiveness.

We soon hope to implement our semantics and type system and try it out by adapting existing concurrent programs.

6.1 Inferring Locks

Recent work [18,30] has investigated the complete inference of synchronisation, *i.e.,* inferring not only the fields F but the set $\mathbb{L}$ as well. This would lead to systems where the programmer does not write any locking code (only using `atomic`); such systems would have all the benefits of software transactional memory [17]. We doubt we will be able to infer ownership annotations, but perhaps we can drop back to a finite set of inferred regions if ownership annotations are not available. Thus, the programmer can choose to add ownership annotations if more parallelism is required.

6.2 STMBench7

We have considered the applicability of our system (compared to the related work), on a benchmark suite [16]. This is a complex datastructure, a tree with a graph at each leaf, with operations that scan, search, and access single elements. It is demanding to synchronise efficiently, correctly, and without introducing deadlocks. The language the authors used did not check their synchronisation. They presented two locking disciplines, the simpler of which uses a single lock for the whole datastructure. In our system this is achieved by making everything `peer`, although with single object locking, many operations that access single elements can run in parallel. Their more complex locking discipline relies on the number of levels in the tree being statically bounded (they used 6 levels). It associates a lock for each level, and a lock for all the leaf graphs.

We found that their example was hard to type-check in our system or the previous work. Firstly, one cannot parameterise the `Node` class to specify the next level:

```
class Node <ourLock,below> {
        Node <below,???> left, right;
}
```

If we do not use parameters, but use a final field to hold the lock (in our system this means the fields `left` and `right` are `any`), then one must lock each node individually. It would not be possible to lock the whole tree before a scan. Locking each node during the scan would cause deadlock unless all the scans were top down or bottom up. One can work around this by defining a class for each level; however, this does not permit iteration through the layers.

The problem is that using field annotations to specify the locking discipline does not work well when the latter is disjoint from the structure of the data. The natural locking discipline is to exploit the hierarchy of the tree, using `rep` for each `left` and `right` field in the node, but we cannot scan the tree from the bottom-up in this

case, as it might deadlock with concurrent top-down scans. Even considering each leaf graph in its own ownership domain is problematic when iterating over them. We leave solving these problems as further work.

7 Conclusion

We wanted to use universes for race safety because we believe that they are simpler and thus a more programmer-friendly type system. We have given a language, semantics, and race safety type system. We have proved that our system prevents races; full proofs will soon be available from [6]. We took the approach of defining a minimal system and then giving a number of straightforward extensions that add features to the type system and allow more parallelism in programs. These extensions can a) distinguish between read/writes, b) prevent deadlocks, c) verify atomicity, and d) allow locks to be taken at the granularity of single objects.

Ownership types are important for good race safety type systems because they allow the type system to understand the extent of heap accesses in while loops and recursive functions. This is also the area where our work differs most. Our system has the qualifier `any`, which can be used to implement open data-structures without constraining the ownership hierarchy and thus the synchronisation of the rest of the program.

We found that our type system required fewer annotations than previous work [11,3] and that the cases of accessing objects from different domains, it could understand more programs. We also used a system of effects, that we do not believe has been tried before, which lets the programmer use non-final paths.

Another advantage of using universes is that they have already been implemented in JML[22]; we hope to extend JML with race safety features based on our type system.

In the future we would like to extend and refine our system to include generics, and to study atomicity in greater detail.

Acknowledgement

We would like to thank P. Müller and W. Dietl for comments and suggestions that were invaluable in the writing of this paper. We would also like to thank C. Flanagan and C. Boyapati for their elucidating correspondence, and the SLURP group, particularly T. Allwood for comments and suggestions.

References

[1] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano. Retrospective: Thin locks. In Kathryn S. McKinley, editor, *Twenty Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979–1999): A Selection*, April 2004.

[2] David F. Bacon, Robert E. Strom, and Ashis Tarafdar. Guava: A dialect of Java without data races. In *Proceedings of the ACM Conference on Object-Oriented Systems, Languages, and Applications*, pages 382–400, Minneapolis, Minnesota, October 2000.

[3] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the 16th Annual ACM Symposium on Object-Oriented Programming Systems, Languages, and Applications*, Tampa Bay, Florida, October 2001.

[4] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 211–230, New York, NY, USA, 2002. ACM Press.

[5] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33:10 of *ACM SIGPLAN Notices*, pages 48–64, New York, October 18–22 1998. ACM Press.

[6] Dave Cunningham. Universes for Race Safety - proofs, 2007. available from http://www.doc.ic.ac.uk/~dc04/race_safety_proofs/.

[7] Dave Cunningham, Adrian Francalanza, Sophia Drossopou-lou, Werner Dietl, and Peter Mueller. UJ: Type Soundness for Universe Types, 2006. preliminary version available at http://www.doc.ic.ac.uk/~scd/uj.pdf.

[8] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[9] Werner Dietl, Sophia Drossopoulou, and Peter Mueller. GUJ: Generic Universe Types for Java-like languages. In *Proceedings of ECOOP'07*, 2007.

[10] Cormac Flanagan and Martin Abadi. Object types against races. In *International Conference on Concurrency Theory*, pages 288–303, 1999.

[11] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. *ACM SIGPLAN Notices*, 35(5):219–232, 2000.

[12] Cormac Flanagan and Stephen N. Freund. Type inference against races. In *SAS*, pages 116–132, 2004.

[13] Cormac Flanagan, Stephen N. Freund, and Marina Lifshin. Type inference for atomicity. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 47–58, New York, NY, USA, 2005. ACM Press.

[14] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 338–349, New York, NY, USA, 2003. ACM Press.

[15] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 1–12, New York, NY, USA, 2003. ACM Press.

[16] Rachid Guerraoui, Michał Kapałka, and Jan Vitek. STMBench7: A benchmark for software transactional memory. Technical report, EPFL, 2006.

[17] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM Press.

[18] Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Lock inference for atomic sections. In *First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06)*., June 2006. (to appear).

[19] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[20] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 137–147, Washington, DC, USA, 2005. IEEE Computer Society.

[21] Doug Lea. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[22] G. Leavens, E. Poll, C. Clifton, Y. Cheon, and C. Ruby. Jml reference manual, 2002.

[23] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[24] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, Calif.*, pages 47–57, January 1988.

[25] Bill McCloskey, Feng Zhou, David Gay, and Eric Brewer. Autolocker: synchronization inference for atomic sections. *SIGPLAN Not.*, 41(1):346–358, 2006.

[26] P. Müller. *Modular Specification and Verification of Object-Oriented programs*, volume 2262. Springer-Verlag, 2002.

[27] Mayur Naik and Alex Aiken. Conditional must not aliasing for static race detection. *SIGPLAN Not.*, 42(1):327–338, 2007.

[28] James Noble, John Potter, David Holmes, and Jan Vitek. Flexible alias protection. In *Proceedings of ECOOP'98*, Brussels, Belgium, July 20 - 24, 1998.

[29] Matthew Smith and Sophia Drossopoulou. Cheaper Reasoning with Ownership Types. In *ECOOP International Workshop on Aliasing, Confinement and Ownership (IWACO 2003)*, 2003.

[30] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. *SIGPLAN Not.*, 41(1):334–345, 2006.

A Universe Lemmas

Lemma A.1 *The* $(\rhd)$ *operator composes types:*

$$\left.\begin{array}{l} a, w \vdash a', w' : u \\ a', w' \vdash a'', w'' : u' \end{array}\right\} \Longrightarrow a, w \vdash a'', w'' : u \rhd u'$$

Proof: Case analysis of u and u'.

Lemma A.2 *The* $(\rhd)$ *operator decomposes types:*

$$\left.\begin{array}{l} a, w \vdash a', w' : u \\ a, w \vdash a'', w'' : u' \end{array}\right\} \Longrightarrow a', w' \vdash a'', w'' : u \rhd u'$$

Proof: Case analysis.

Firstly we guarantee that at all times after an object is constructed, both its class and its owner remain constant. As a corollary, execution will not affect the universe type judgement of another expression. We present a "substitution" lemma (although there is no substitution here since we are using a stack to hold the arguments. We require h and σ to be consistent with Γ, but the expression e is the same on both sides. Finally we state soundness. In the soundness theorem, m is either n or $n+1$. New threads can initially have any type, but must maintain this type as they execute.

Lemma A.3 *Ownership and class membership are constant:*

$$\left.\begin{array}{l} h(a) = (v, c, \_) \\ \_ \vdash \_, h \rightsquigarrow \_, h' \end{array}\right\} \Longrightarrow h'(a) = (v, c, \_)$$

Proof: Induction over structure of reduction.

Lemma A.4 *The run-time types of expressions are preserved over the execution of other expressions.*

$$\left.\begin{array}{l} h, \sigma \vdash e : t \\ \_ \vdash \_, h \rightsquigarrow \_, h' \end{array}\right\} \Longrightarrow h', \sigma \vdash e : t$$

Proof: Induction over the structure of $h, \sigma \vdash e : t$.

Lemma A.5 *Static type safety implies run-time type safety with respect to a suitable stack.*

$$\left.\begin{array}{l}\Gamma \vdash e : t \\ h, \sigma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \\ h, \sigma \vdash \mathtt{this} : \Gamma(\mathtt{this})\end{array}\right\} \Longrightarrow h, \sigma \vdash S(e) : t$$

Proof: Induction over the structure of $\Gamma \vdash e : t$.

Theorem A.6 *Run-time types and heap well-formedness are preserved over execution.*

$$\left.\begin{array}{l}\vdash h \\ h, \sigma \vdash e : t \\ \sigma \vdash e, h \rightsquigarrow e', h'\end{array}\right\} \Longrightarrow \begin{array}{l}\vdash h' \\ h', \sigma \vdash e' : t\end{array}$$

Proof: Induction over the structure of $h, \sigma \vdash e : t$.

Theorem A.7 *The well-typedness of all threads in a system is preserved over a step of multithreaded execution.*

$$\left.\begin{array}{l}\vdash h \\ h, \sigma \vdash e_{1..n} : t_{1..n} \\ \sigma \vdash e_{1..n}, h \rightsquigarrow e'_{1..m}, h'\end{array}\right\} \Longrightarrow \begin{array}{l}\vdash h' \\ h', \sigma \vdash e'_{1..m} : t_{1..m}\end{array}$$

Proof: Case analysis of $\sigma \vdash e_{1..n}, h \rightsquigarrow e'_{1..m}, h'$.

B Race Safety Lemmas

Lemma B.1 *Path resolution is preserved over execution.*

$$\left.\begin{array}{l}h(\sigma, p) = v \\ \sigma \vdash p, h \rightsquigarrow e, h'\end{array}\right\} \Longrightarrow \begin{array}{l}e = p', h = h' \\ h'(\sigma, p') = v \\ \forall f \notin p \;.\; f \notin p'\end{array}$$

Proof: induction over $\sigma \vdash p, h \rightsquigarrow e, h'$.

A given expression should be guarded by the same lock no matter what state of execution the expression has reached. We require heap well-formedness because we need the universe type judgements within the guard logic to be preserved.

Lemma B.2 *Guards are preserved over execution.*

$$\left.\begin{array}{l} h, \sigma \vdash_{gb} e : l \\ \vdash h \\ \sigma \vdash e, h \rightsquigarrow e', h' \end{array}\right\} \Longrightarrow h', \sigma \vdash_{gb} e' : l$$

Proof: Case analysis of $h, \sigma \vdash_{gb} e : l$.

If the heap has changed enough since the lock p was taken, so that p resolves to a different object, then p no longer guards the objects it used to. This lemma ensures heap changes are not sufficient, by ensuring the effect of the executing expression does not conflict with p.

Lemma B.3 *Path resolution is preserved over execution of other expressions.*

$$\left.\begin{array}{l} h(\sigma, p) = v \\ \sigma' \vdash e, h \rightsquigarrow e', h' \\ \_, h, \sigma' \vdash e : F \\ \{p\} \# F \end{array}\right\} \Longrightarrow h'(\sigma, p) = v$$

Proof: Induction over steps of resolution.

The next lemma helps to prove that the guard of an expression should be unaffected by the execution of other expressions. However, we must ensure the reducing expression will not interfere with any paths that the guard might be using.

Lemma B.4 *Guards are preserved over the execution of other expressions*

$$\left.\begin{array}{l} h, \sigma \vdash_{gb} e : l \\ \sigma' \vdash e', h \rightsquigarrow \_, h' \\ \_, h, \sigma \vdash e' : F' \\ \{l\} \# F' \end{array}\right\} \Longrightarrow h', \sigma \vdash_{gb} e : l$$

Proof: Induction over $h, \sigma \vdash_{gb} e : l$.

A similar result relies on the fact that if $Virgin(e)$ then only (FIELD) and (VAR) are used in the derivation of $h, \sigma \vdash_{gb} e : p$ then and neither rule uses h or σ.

Lemma B.5 *Virgin guards are preserved over the execution of other expressions.*

$$\left.\begin{array}{l} h, \sigma \vdash_{gb} e : l \\ Virgin(e) \\ \_ \vdash \_, h \rightsquigarrow \_, h' \end{array}\right\} \Longrightarrow h', \sigma \vdash_{gb} e : l$$

Proof: Induction over $h, \sigma \vdash_{gb} e : l$.

The following lemma is used to show that the part of an expression that has not executed yet is not affected by the part of that expression that is executing.

Lemma B.6 *Types of virgin expressions are preserved over the execution of other expressions.*

$$\left.\begin{array}{l} \mathbb{L}, h, \sigma \vdash e : F \\ Virgin(e) \\ \_ \vdash \_, h \rightsquigarrow \_, h' \end{array}\right\} \Longrightarrow \mathbb{L}, h', \sigma \vdash e : F$$

Proof: Induction over $\mathbb{L}, h, \sigma \vdash e : F$.

This lemma requires that the locks taken by one thread are disjoint to the locks guarding a path. This means that the modifications made by the executing thread cannot affect the resolution of the path.

Lemma B.7 *Path resolution is preserved over the execution of other expressions when locks do not collide.*

$$\left.\begin{array}{l} h(\sigma, p) = v \\ \sigma' \vdash e', h \rightsquigarrow \_, h' \\ \emptyset, h, \sigma' \vdash e' : \_ \\ \mathbb{L}, h, \sigma \vdash p : \_ \\ \{h(a)\downarrow_1 | h, \sigma \vdash_{gb} a : l, l \in \mathbb{L}\} \cap \{w | Locked(e', w)\} = \emptyset \end{array}\right\} \Longrightarrow h'(\sigma, p) = v$$

Proof: Induction over $\mathbb{L}, h, \sigma \vdash e : F$.

The following lemma requires that the executing thread's locks are disjoint to the locks required to guard a lock, and can therefore show that the derivation of the lock is unaffected by the execution.

Lemma B.8 *Guards are preserved over the execution of other expressions when locks do not collide.*

$$\left.\begin{array}{l} h, \sigma \vdash_{gb} e : l \\ l \in Path \Longrightarrow \mathbb{L}, h, \sigma \vdash l : \_ \\ \sigma' \vdash e', h \rightsquigarrow \_, h' \\ \emptyset, h, \sigma' \vdash e' : \_ \\ \{h(a)\downarrow_1 | h, \sigma \vdash_{gb} a : l, l \in \mathbb{L}\} \cap \{w | Locked(e', w)\} = \emptyset \end{array}\right\} \Longrightarrow h', \sigma \vdash_{gb} e : l$$

Proof: Induction over $\mathbb{L}, h, \sigma \vdash e : F$.

Finally, this lemma shows that the execution of one thread will not interfere with the typing of another thread.

Lemma B.9 *Types are preserved over the execution of other expressions when locks do not collide.*

$$\left.\begin{array}{l} \mathbb{L}, h, \sigma \vdash e : F \\ \sigma' \vdash e', h \rightsquigarrow \_, h' \\ \emptyset, h, \sigma' \vdash e' : \_ \\ \mathit{Reachable}(e) \\ \forall w. \neg(\mathit{Locked}(e, w) \wedge \mathit{Locked}(e', w)) \\ \{h(a)\downarrow_1 | h, \sigma \vdash_{gb} a : l, l \in \mathbb{L}\} \cap \{w | \mathit{Locked}(e', w)\} = \emptyset \end{array}\right\} \Longrightarrow \mathbb{L}, h', \sigma \vdash e : F$$

Proof: Induction over $\mathbb{L}, h, \sigma \vdash e : F$.