Object Types against Races

Cormac Flanagan and Martín Abadi
[flanagan|ma]@pa.dec.com

Systems Research Center, Compaq

Abstract. This paper investigates an approach for statically preventing race conditions in an object-oriented language. The setting of this work is a variant of Gordon and Hankin's concurrent object calculus. We enrich that calculus with a form of dependent object types that enables us to verify that threads invoke and update methods only after acquiring appropriate locks. We establish that well-typed programs do not have race conditions.

1 Introduction

Concurrent object-oriented programs suffer from many of the errors common in concurrent programs of other sorts. In particular, the use of objects does not diminish the importance of careful synchronization. With objects or without them, improper synchronization may lead to race conditions (that is, two processes accessing a shared resource simultaneously) and ultimately to incorrect behavior.

A standard approach for eliminating race conditions consists in protecting each shared resource with a lock, requiring that a process acquires the corresponding lock before using the resource [5]. Object-oriented programs often rely on this approach, but with some peculiar patterns. It is common to group related resources into an object, and to attach the lock that protects the resources to this object. Processes may acquire the lock before invoking the methods of the object; alternatively, the methods may acquire this lock at the start of their execution. With constructs such as Java's `synchronized` methods [2, 10], some object-oriented languages support these synchronization patterns. However, standard object-oriented languages do not enforce proper synchronization; it remains possible, even easy, to write programs with race conditions.

This paper investigates a static-analysis approach for preventing race conditions in an object-oriented language. The approach consists in associating locks with shared object components and in verifying that appropriate locks have been acquired before each operation. In the object-oriented language that we treat, the object components are methods; they can be both invoked and updated. (Fields are a special case of methods [1].) Thus, the approach consists in associating locks with methods and in verifying that appropriate locks have been acquired before each method invocation and update. Because a method invocation may trigger other operations, several locks may be required for it; only one lock is required for a method update.

The annotations and checks necessary in our static analysis are expressed in a type system. Like standard type systems, this type system assigns a type to each of the methods of an object. In addition, it gives the set of locks that must be held before invoking the method and the lock that must be held before updating the method (or an indication that the update is forbidden). Each of these locks may be external to the object, but it may also be a special lock associated with self, that is, with this object.

Thus, we are led to a type system with dependent types, in which the type of an object refers to values, namely to locks. However, the type system is restrictive enough to preserve the important phase distinction between compile-time and run-time [7, 11]. All our checking takes place at compile-time (without excluding the possibility of further run-time checking).

The checking guarantees the absence of race conditions: if a program is well-typed, then during its execution no two threads attempt to access an object component at the same time. In addition, the checking guarantees the absence of standard run-time type errors ("message-not-understood" errors). Our approach can handle some interesting, common examples, as we demonstrate. Although it is far from complete, we believe that it represents a sensible compromise between simplicity and generality, and a worthwhile step in the ongoing investigation of the use of types for safe locking.

Background

In a recent paper [8], we developed an analogous technique for a basic calculus with reference cells but no data structures. In that paper, singleton types (types with one single element) enable the tracking of locks; existential types permit the hiding of singleton types for locks. That paper also describes a technique for avoiding deadlocks, which it should be possible to adapt to the setting of the present paper. The substantial novelty of the present paper is the treatment of objects, object types, and subtyping. Here we avoid the use of singleton types and existential types, and resort to specialized dependent types instead. These dependent types require somewhat less conceptual machinery and support more flexible subtyping relations.

In addition to our own previous work, we rely on Gordon and Hankin's concurrent object calculus **conc**$\boldsymbol{\varsigma}$ [9]. This calculus is a small but extremely expressive concurrent object-oriented language; it features a compact and elegant presentation of the concepts of expression, process, store, and configuration. We refer the reader to Gordon and Hankin's work for motivations for this calculus and additional examples and technical developments.

The calculus **conc**$\boldsymbol{\varsigma}$ extends a sequential calculus of Abadi and Cardelli [1], adopting the basic type structure and subtyping relation of that sequential calculus. Here we extend the type system with our form of dependent types. Gordon and Hankin's type system does not attempt to guarantee the absence of race conditions; our type system provides that guarantee.

For simplicity, we omit the cloning construct from **conc**$\boldsymbol{\varsigma}$. We also replace the synchronization primitives that Gordon and Hankin presented as an extension to

concς. Those primitives are two separate operations for acquiring and releasing a lock. Instead, we use the expression *lock* v *in* a, which acquires the lock denoted by v, evaluates a, then releases the lock. Like Java's `synchronized` statement, the expression *lock* v *in* a automatically guarantees the proper nesting of lock operations, helping static checking. Moreover, our calculus associates locks with objects (unlike **concς**, but like Java).

There are some other languages that we might have used as a starting point for this work instead of **concς**, in particular Di Blasio and Fisher's concurrent object calculus [6]. However, we prefer to base our work on that of Gordon and Hankin, for two main reasons. First, Di Blasio and Fisher's calculus permits object extension but not subtyping, unlike **concς** and unlike most typed object-oriented languages; we wish to treat subtyping. Furthermore, Di Blasio and Fisher's calculus combines synchronization mechanisms with the primitive operations on objects. Like Gordon and Hankin, we prefer to keep synchronization separate from object operations, although our object types do mention locks. Di Blasio and Fisher's interesting study does not address race conditions, but shows that certain synchronization guards do not have side-effects.

Other pieces of related work are discussed in our recent paper. These rely on a variety of techniques, including program-verification methods and data-flow analyses, for example. One of the most relevant is the work of Kobayashi and Sumii [13, 16], which develops a type-based techniques for avoiding deadlocks (not necessarily race conditions) in the context of a process calculus. Another one is Warlock [15], a system for partial detection of race conditions and deadlocks in ANSI C programs. We are not aware of any work that specifically addresses race conditions in object-oriented programs. In another direction, there have been intriguing explorations of the combination of dependent types with objects and subtyping, with an emphasis on logical frameworks rather than programming languages [3, 12].

Outline

The next section presents the syntax and informal semantics of the concurrent object-oriented language that we treat; an appendix contains a formal semantics. Section 3 develops our type system for this calculus. Section 4 shows some example applications of our type system. Section 5 considers formal properties. Finally, section 6 concludes. Proofs are omitted.

2 A Concurrent Object Calculus

This section describes our variant of Gordon and Hankin's concurrent object calculus. It is largely a review.

2.1 Syntax

We define the sets of *results*, *denotations*, *lock states*, and *terms* by the grammar:

Syntax

$u, v ::=$	results
$\quad x$	$\quad$ variable
$\quad p$	$\quad$ name
$d ::=$	denotations
$\quad [\ell_i = \varsigma(x_i)b_i \ {}^{i \in 1..n}]^m$	$\quad$ object
$m ::=$	lock states
$\quad \bullet$	$\quad$ locked
$\quad \circ$	$\quad$ unlocked
$a, b, c ::=$	terms
$\quad u$	$\quad$ result
$\quad (\nu p)a$	$\quad$ restriction
$\quad p \mapsto d$	$\quad$ denomination
$\quad u.\ell$	$\quad$ method invocation
$\quad u.\ell \Leftarrow \varsigma(x)b$	$\quad$ method update
$\quad let\ x{=}a\ in\ b$	$\quad$ let
$\quad a \upharpoonright b$	$\quad$ parallel composition
$\quad lock\ u\ in\ a$	$\quad$ lock acquisition
$\quad locked\ p\ in\ a$	$\quad$ lock acquired

Results include both variables and names. Variables represent intermediate values, and are bound by methods $\varsigma(x)b$ and by let expressions $let\ x{=}a\ in\ b$; both of these constructs bind the variable x with scope b. Names represent the addresses of stored objects. They are introduced by a restriction $(\nu p)a$, which binds the name p with scope a. We let $fn(a)$ and $fv(a)$ denote the sets of free names and free variables in a, respectively. We write $a\{\!\{u \leftarrow b\}\!\}$ to denote the capture-free substitution of b for all free occurrences of u in a. We write $a = b$ to mean that a and b are equal up to the renaming of bound variables and bound names, and the reordering of object methods.

2.2 Informal Semantics

A denotation $[\ell_i = \varsigma(x_i)b_i \ {}^{i \in 1..n}]^m$ describes an object containing a collection of methods. Each method has a label ℓ_i and consists of a self parameter x_i and a body b_i. In addition, the object has an associated lock whose state is described by m. If $m = \bullet$, the lock is held by some term in the program; if $m = \circ$, the lock is unlocked. (As a straightforward extension, each object could have several associated locks.)

A denotation may appear in a denomination $p \mapsto d$, which maps the name p to the denotation d. Intuitively, this term represents the portion of the store containing the object d, and p represents the address of that object. The term $(\nu p)a$ introduces a fresh name p and then evaluates a. This operation corresponds to allocating a fresh address p at which objects can be stored. Thus the language separates name introduction $(\nu p)a$ from name definition $p \mapsto d$; the type system forbids programs with multiple definitions of the same name.

A method invocation $u.\ell$ invokes the method ℓ of the object u. A method update $u.\ell \Leftarrow \varsigma(x)b$ replaces the method ℓ of u with $\varsigma(x)b$. The term *let* $x{=}a$ *in* b first evaluates a to yield a result, binds x to this result, and then evaluates b. A parallel composition $a \upharpoonright b$ evaluates both a and b in parallel. The result of this parallel composition is the result of b; the subterm a is evaluated only for effect.

The lock operation *lock* u *in* a functions in a similar manner to Java's `synchronized` statement: the lock on the object u is acquired; then the subterm a is evaluated; and finally the lock is released. The implementation of this construct relies on an auxiliary construct *locked* p *in* a, which indicates that the lock p has been acquired and that the term a is being evaluated. Locks are not reentrant, so expressions like *lock* u *in lock* u *in* a will deadlock; similarly, calls to the recursive method ℓ of $[\ell = \varsigma(x) \mathit{lock}\ x\ \mathit{in}\ x.\ell]^\circ$ will deadlock (cf. Java).

The appendix contains a detailed formal semantics of the language. It is a chemical semantics in the style of Berry and Boudol [4]. It consists of a group of structural congruence rules, which permit the rearrangement of terms and imply for example that $\upharpoonright$ is associative, and a group of reduction rules, which model proper computation steps.

A typical structural congruence rule is:

$$a \upharpoonright \mathcal{E}[\, b\,] \equiv \mathcal{E}[\, a \upharpoonright b\,] \qquad\qquad (\text{if } \mathit{fn}(a) \cap \mathit{bn}(\mathcal{E}) = \varnothing)$$

where $\mathcal{E}$ denotes an evaluation context:

$$\mathcal{E} ::= [\,\cdot\,] \mid \mathit{let}\ x{=}\mathcal{E}\ \mathit{in}\ b \mid \mathcal{E} \upharpoonright b \mid a \upharpoonright \mathcal{E} \mid (\nu p)\mathcal{E} \mid \mathit{locked}\ p\ \mathit{in}\ \mathcal{E}$$

and the *binding names* $\mathit{bn}(\mathcal{E})$ of an evaluation context are the names p bound by a restriction $(\nu p)\mathcal{E}'$ that encloses the hole $[\,\cdot\,]$.

For our purposes, the two most interesting reduction rules are:

$$\begin{array}{rcll}
(p \mapsto [\ldots]^\circ) \upharpoonright \mathit{lock}\ p\ \mathit{in}\ a & \rightarrow & (p \mapsto [\ldots]^\bullet) \upharpoonright \mathit{locked}\ p\ \mathit{in}\ a & \text{(Red Lock)} \\
(p \mapsto [\ldots]^\bullet) \upharpoonright \mathit{locked}\ p\ \mathit{in}\ u & \rightarrow & (p \mapsto [\ldots]^\circ) \upharpoonright u & \text{(Red Locked)}
\end{array}$$

where $[\ldots]$ represents an object (excluding its lock state). The rule (Red Lock) evaluates a lock operation by acquiring the lock associated with p, and yielding the term *locked* p *in* a. Subsequent reduction steps may then evaluate a. Once a is reduced to some result u, the rule (Red Locked) releases the lock on p and returns u as the result of the locked expression.

2.3 An Example

For clarity, we present example programs in an extended language with integer constants and operations, and we abbreviate *let* $x{=}a$ *in* b to $a; b$ when $x \notin \mathit{fv}(b)$.

A counter that has a read method and an increment method, and initially contains the integer n, can be defined as:

$$\begin{array}{rl}
\mathit{count}_n \triangleq [\!\!\! & \mathit{val} = \varsigma(x)n, \\
& \mathit{read} = \varsigma(x)x.\mathit{val}, \\
& \mathit{inc} = \varsigma(x)\mathit{let}\ t{=}x.\mathit{val} + 1\ \mathit{in}\ x.\mathit{val} \Leftarrow \varsigma(x)t]
\end{array}$$

The following program allocates a counter (initially containing 0), increments the counter, and then reads the value of the counter.

$$(\nu p)(p \mapsto count_0 \upharpoonright (p.inc; p.read))$$

As expected, this program reduces to $(\nu p)(p \mapsto count_1 \upharpoonright 1)$, since the counter works correctly in a sequential setting. In the presence of concurrency, however, the counter may exhibit unexpected behavior. To illustrate this danger, we consider the following program, which creates a counter and then increments it twice, in parallel.

$$(\nu p)(p \mapsto count_0 \upharpoonright p.inc \upharpoonright p.inc)$$

This program is non-deterministic. It may reduce to $(\nu p)(p \mapsto count_2 \upharpoonright p \upharpoonright p)$, as expected. Alternatively, if the evaluations of the two calls to *inc* are interleaved, the program may also reduce to $(\nu p)(p \mapsto count_1 \upharpoonright p \upharpoonright p)$, which is presumably not what the programmer intended. Thus the program has a race condition: two threads may attempt to update the method *val* simultaneously, with incorrect results.

We can fix this error by adding appropriate synchronization to the counter:

$$\begin{aligned} sync\_count_n \triangleq [&val = \varsigma(x)n, \\ &read = \varsigma(x) lock\ x\ in\ x.val, \\ &inc = \varsigma(x) lock\ x\ in\ let\ t{=}x.val + 1\ in\ x.val \Leftarrow \varsigma(x)t] \end{aligned}$$

In this synchronized counter, the method *val* is protected by the lock of the counter. This lock should be held whenever the method *val* is invoked or updated, and thus the methods *read* and *inc* both acquire that lock. The modified counter implementation is race-free and will behave correctly even if used by multiple threads, provided those threads access *val* only through *read* and *inc*, or acquire the lock before accessing *val* directly. We revisit this example in later sections.

3 The Type System

Race conditions, such as that in $count_n$, are a common bug in concurrent object-oriented programs, just as they are in concurrent programs of other kinds. In practice, race conditions are often avoided by the same strategy that we employed in $sync\_count_n$; each mutable component is protected by a lock, and is only accessed when that lock is held. In this section, we describe a type system that supports this programming discipline.

3.1 The Type Language

The set of types in our system is described by the grammar:

Types

$A, B ::= [\ell_i : \varsigma(x_i)A_i \cdot r_i \cdot s_i \ ^{i \in 1..n}] \mid \mathit{Proc} \mid \mathit{Exp}$	types
$r ::= \{u_1, \ldots, u_n\}$	permissions
$s ::= u \mid +$	protection annotations

An object type $[\ell_i : \varsigma(x_i)A_i \cdot r_i \cdot s_i \ ^{i \in 1..n}]$ describes an object containing n methods labeled $l_1, \ldots, l_n$. Each method l_i has result type A_i, *permission* r_i, and *protection annotation* s_i. The permission r_i is a set of results describing the locks that must be held before invoking l_i. Because the method invocation may trigger other operations, we allow r_i to contain more than one lock. The protection annotation s_i is a result describing the lock that must be held before updating l_i; we refer to s_i as the lock that protects l_i (although additional locks may be required for invoking l_i). In the case where l_i is never updated, s_i may alternatively be the symbol '+'. Since methods are commonly protected by the *self lock* (that is, the lock of the object itself), the description of each method also binds the self variable x_i; this variable may occur free in A_i, r_i, and s_i.

An example type is $[l : \varsigma(x)A \cdot \varnothing \cdot +]$, which describes an object containing a single method l with result type A. The permission $\varnothing$ indicates that no locks need to be acquired before invoking this method; the protection annotation '+' indicates that the method cannot be updated. The type $[l : \varsigma(x)A \cdot \{x\} \cdot x]$ is similar, except that it describes an object whose method l can be updated. The self lock of the object must be acquired before invoking or updating that method. The type $[l : \varsigma(x)A \cdot \{x\} \cdot x, l' : \varsigma(x)A \cdot \{x, p\} \cdot p]$ mentions an additional method l' protected by an external lock p. Since the lock that protects l (the self lock x) must be acquired before invoking l', the code for l' may update or invoke l without any further locking.

As a slightly more complicated example, a suitable type for the synchronized counter $\mathit{sync\_count}_n$ described earlier is:

$$[\mathit{val} : \varsigma(x)\mathit{Int} \cdot \{x\} \cdot x, \quad \mathit{read} : \varsigma(x)\mathit{Int} \cdot \varnothing \cdot +, \quad \mathit{inc} : \varsigma(x)[\,] \cdot \varnothing \cdot +]$$

This type states that the method *val* is protected by the self lock, which must be acquired before invoking or updating that method. The methods *read* and *inc* are read-only; they cannot be updated. Furthermore, since these methods perform the necessary synchronization internally, no locks need to be held when invoking these methods.

In addition to object types, the type language also includes the types *Exp* and *Proc*. The type *Exp* describes all results that may be returned by expressions; the type *Proc* is a supertype of *Exp* that also covers terms that never return results, such as a denomination $p \mapsto d$.

3.2 Clean and Defined Names

In addition to checking that the appropriate locks are held whenever a method is invoked or updated, the type system also verifies that each lock is held by at most one thread at any time. That is, for each name p, there is at most one term of the form *locked* p *in* $\ldots$ in the program.

Verifying this mutual exclusion property is a little tricky, since any term that contains the denomination $p \mapsto [\ldots]^\circ$ can potentially acquire the lock on p via the reduction rule (Red Lock). Therefore, we introduce the notion of *clean names*, and we say that p is a clean name of a term if the term includes either *locked* p *in* $\ldots$ or $p \mapsto [\ldots]^\circ$ in an evaluation context. (The restriction to evaluation contexts excludes some nonsensical programs.) The set of clean names of a term is preserved during evaluation, even though the set of locks held by the term may vary. The type system checks that for any parallel composition $a \upharpoonright b$, the clean names of the subterms a and b are distinct. This check ensures that a lock cannot be simultaneously held by two terms executing in parallel.

The type system also verifies that every name that is introduced is associated with a unique denotation. We say that a name is *defined* by a term if it is associated with a denotation in an evaluation context.

Clean and defined names

$p \in \mathit{clean}(a)$ if $a = \mathcal{E}[\, p \mapsto [\ldots]^\circ \,]$ or $a = \mathcal{E}[\, \mathit{locked}\ p\ \mathit{in}\ b \,]$ and $p \notin \mathit{bn}(\mathcal{E})$

$p \in \mathit{defined}(a)$ if $a = \mathcal{E}[\, p \mapsto d \,]$ and $p \notin \mathit{bn}(\mathcal{E})$

3.3 Type Rules

We define the type system using the following six judgments and associated rules. In these judgments, an *environment* E is a sequence of bindings of results to types, of the form $\varnothing, u_1 : A_1, \ldots, u_n : A_n$.

Judgments

$E \vdash \diamond$	E is a well-formed environment
$E \vdash A$	given E, type A is well-formed
$E \vdash r$	given E, permission r is well-formed
$E \vdash A <: B$	given E, A is a subtype of B
$E \vdash r <: r'$	given E, r is a subpermission of r'
$E; r \vdash a : A$	given E and r, term a has type A

Type rules

(Env $\varnothing$)
$$\frac{}{\varnothing \vdash \diamond}$$

(Env u)
$$\frac{E \vdash A \quad u \notin \mathit{dom}(E)}{E, u : A \vdash \diamond}$$

(Perm)
$$\frac{E \vdash \diamond \quad r \subseteq \mathit{dom}(E)}{E \vdash r}$$

(Type Proc)
$$\frac{E \vdash \diamond}{E \vdash \mathit{Proc}}$$

(Type Exp)
$$\frac{E \vdash \diamond}{E \vdash \mathit{Exp}}$$

(Type Object) (ℓ_i distinct)
$$\frac{E \vdash \diamond \quad E, x_i : [\,] \vdash B_i {<:} \mathit{Exp} \quad E, x_i : [\,] \vdash r_i \quad s_i \in r_i \cup \{+\} \quad \forall i \in 1..n}{E \vdash [\ell_i : \varsigma(x_i) B_i {\cdot} r_i {\cdot} s_i \ ^{i \in 1..n}]}$$

(Val Object) (where $A = [\ell_i : \varsigma(x_i) B_i {\cdot} r_i {\cdot} s_i \ ^{i \in 1..n}]$)
$$\frac{\begin{gathered} E = E_1, p : A, E_2 \quad E \vdash \diamond \quad E; r_i\{\!\{x_i \leftarrow p\}\!\} \vdash b_i\{\!\{x_i \leftarrow p\}\!\} : B_i\{\!\{x_i \leftarrow p\}\!\} \\ \mathit{defined}(b_i) = \mathit{clean}(b_i) = \varnothing \quad \forall i \in 1..n \end{gathered}}{E; \varnothing \vdash p \mapsto [\ell_i = \varsigma(x_i) b_i \ ^{i \in 1..n}]^m : \mathit{Proc}}$$

(Val u)
$$\frac{E, u : A, E' \vdash \diamond}{E, u : A, E'; \varnothing \vdash u : A}$$

(Val Select)
$$\frac{E; \varnothing \vdash u : [\ell_i : \varsigma(x_i) B_i {\cdot} r_i {\cdot} s_i \ ^{i \in 1..n}] \quad j \in 1..n}{E; r_j\{\!\{x_j \leftarrow u\}\!\} \vdash u.\ell_j : B_j\{\!\{x_j \leftarrow u\}\!\}}$$

(Val Update) (where $A = [\ell_i : \varsigma(x_i) B_i {\cdot} r_i {\cdot} s_i \ ^{i \in 1..n}]$)
$$\frac{\begin{gathered} E; \varnothing \vdash u : A \quad E; r_j\{\!\{x_j \leftarrow u\}\!\} \vdash b\{\!\{x_j \leftarrow u\}\!\} : B_j\{\!\{x_j \leftarrow u\}\!\} \quad s_j \neq + \quad j \in 1..n \\ \mathit{defined}(b) = \mathit{clean}(b) = \varnothing \end{gathered}}{E; \{s_j\{\!\{x_j \leftarrow u\}\!\}\} \vdash u.\ell_j \Leftarrow \varsigma(x_j) b : A}$$

(Val Let)
$$\frac{\begin{gathered} E; r \vdash a : A \quad E, x : A; r \vdash b : B \quad E \vdash A {<:} \mathit{Exp} \quad E \vdash B {<:} \mathit{Exp} \\ \mathit{defined}(b) = \mathit{clean}(b) = \varnothing \end{gathered}}{E; r \vdash \mathit{let}\ x{=}a\ \mathit{in}\ b : B}$$

(Val Res)
$$\frac{\begin{gathered} E, p : A; r \vdash a : B \quad E \vdash r \quad E \vdash B \\ p \in \mathit{defined}(a) \quad p \in \mathit{clean}(a) \end{gathered}}{E; r \vdash (\nu p) a : B}$$

(Val Par)
$$\frac{\begin{gathered} E; \varnothing \vdash a : \mathit{Proc} \quad E; r \vdash b : B \\ \mathit{defined}(a) \cap \mathit{defined}(b) = \varnothing \quad \mathit{clean}(a) \cap \mathit{clean}(b) = \varnothing \end{gathered}}{E; r \vdash a \upharpoonright b : B}$$

(Val Lock)
$$\frac{\begin{gathered} E; \varnothing \vdash u : [\,] \quad E; r \cup \{u\} \vdash a : A \\ \mathit{defined}(a) = \mathit{clean}(a) = \varnothing \end{gathered}}{E; r \vdash \mathit{lock}\ u\ \mathit{in}\ a : A}$$

(Val Locked)
$$\frac{\begin{gathered} E; \varnothing \vdash p : [\,] \quad E; r \cup \{p\} \vdash a : A \\ p \notin \mathit{clean}(a) \end{gathered}}{E; r \vdash \mathit{locked}\ p\ \mathit{in}\ a : A}$$

(Val Subsumption)
$$\frac{E; r \vdash a : A \quad E \vdash A {<:} B \quad E \vdash r {<:} r'}{E; r' \vdash a : B}$$

(Subperm)
$$\frac{E \vdash r \quad E \vdash r' \quad r \subseteq r'}{E \vdash r {<:} r'}$$

(Sub Refl)
$$\frac{E \vdash A}{E \vdash A {<:} A}$$

(Sub Trans)
$$\frac{E \vdash A {<:} B \quad E \vdash B {<:} C}{E \vdash A {<:} C}$$

(Sub Exp)
$$\frac{E \vdash A \quad A \neq \mathit{Proc}}{E \vdash A {<:} \mathit{Exp}}$$

(Sub Proc)
$$\frac{E \vdash A}{E \vdash A {<:} \mathit{Proc}}$$

(Sub Object)

$$\frac{E \vdash [\ell_i : \varsigma(x_i)B_i{\cdot}r_i{\cdot}s_i \;{}^{i\in 1..n+m}]}{E \vdash [\ell_i : \varsigma(x_i)B_i{\cdot}r_i{\cdot}s_i \;{}^{i\in 1..n+m}]{<:}[\ell_i : \varsigma(x_i)B_i{\cdot}r_i{\cdot}s_i \;{}^{i\in 1..n}]}$$

Many of the rules of the type system are based on corresponding rules in Gordon and Hankin's system, which is in turn based on Abadi and Cardelli's calculi. The novel aspects of our system mainly pertain to locking; they include the treatment of permissions and dependent types.

The core of the system is the set of rules for the judgment $E; r \vdash a : A$ (read "a is a well-typed expression of type A in typing environment E with permission r"). Our intent is that, if this judgment holds, then a is race-free and yields results of type A, provided the free variables of a are given bindings consistent with the typing environment E, and the current thread holds at least the locks in r.

The type system thus tracks the set of locks that are assumed to be held at each program point. The rule (Val Object) checks that each method body is race-free under the assumption that the locks described by the method's permission are held. The rule (Val Select) ensures that these locks are held whenever the method is invoked. The rule (Val Update) ensures that the lock protecting a method is held whenever that method is updated. The rule (Val Lock) for *lock* u *in* a typechecks a under the assumption that the lock u is held. The rule (Val Subsumption) allows for subsumption on both types and permissions: if $E \vdash r{<:}r'$, then any term that is race-free with permission r is also race-free with the superset r' of r.

The type system provides dependent types, that is, a type may contain a result that refers to an object. In some cases, an object can be the referent of several results, for example, its self variable and some external name for the object. The type rules contain a number of substitutions that support changing the result used to refer to a particular object. For example, the rule (Val Select) for a method invocation $u.\ell_j$ replaces occurrences of the self variable x_j in the type B_j with the result u, since x_j and u refer to the same object, and x_j is going out of scope. A similar substitution is performed on the permission r_j. The rules (Val Object) and (Val Update) rely on analogous substitutions.

In order to accommodate self-dependent types, where the description of an object's method may refer to the object itself, the rule (Type Object) checks that the result type and permission of each method is well-formed in an extended environment that contains a binding for the self variable. Because types may refer to results, the rules (Val Let) and (Val Res) ensure that a type that is lifted outside a result binding is still well-formed. The rule (Val Res) also has a similar requirement on permissions.

The type rules include conditions on the clean and defined names of subterms. The rule (Val Par) for $a \upharpoonright b$ requires that the defined names of the subterms a and b be disjoint. Furthermore, the clean names of the subterms must also be disjoint. The latter condition implies that the two subterms cannot simultaneously hold the same lock. The rule (Val Res) for $(\nu p)a$ requires that the name p being introduced be defined in a, and that the lock associated with p is either unlocked or is held by a. The rule (Val Locked) disallows nested acquisitions of the same lock. In addition, in order to ensure that the clean and defined names of a term are invariant under evaluation, the type rules require that terms not in evaluation contexts do not have any clean or defined names.

The rule (Sub Object) defines the usual subtyping relation on object types (appropriately adapted to our type syntax). Since the protection annotation $+$ can be considered a variance annotation [1], we could extend the type system with a more powerful subtyping rule. This rule would allow the result types and permissions of immutable components to behave covariantly. We conjecture that the extended system would still be race-free.

4 Examples

In this section we show a few applications of our type system in examples. For convenience, we use the abbreviation $b.\ell \triangleq \mathit{let}\ x{=}b\ \mathit{in}\ x.\ell$ when b is not a result.

4.1 Counters

The unsynchronized counter implementation count_n described earlier can be assigned the type:

$$[\mathit{val} : \varsigma(x)\mathit{Int}\cdot\{x\}\cdot x,\quad \mathit{read} : \varsigma(x)\mathit{Int}\cdot\{x\}\cdot+,\quad \mathit{inc} : \varsigma(x)[\,]\cdot\{x\}\cdot+]$$

This type states that the method *val* is protected by the self lock of the object, and this self lock must be acquired before invoking the methods *read* and *inc*.

The method *val* may be considered private to the implementation of the counter, and can be dropped via subtyping, yielding:

$$[\mathit{read} : \varsigma(x)\mathit{Int}\cdot\{x\}\cdot+,\quad \mathit{inc} : \varsigma(x)[\,]\cdot\{x\}\cdot+]$$

This type describes the public interface to the counter; it states that the self lock of the counter must be acquired before invoking the counter's methods. This interface expresses a synchronization protocol that is sufficient to ensure that the counter operates correctly. The type system requires that this protocol be obeyed by each client of the counter. Programs that do not obey this synchronization protocol, such as $(\nu p)(p \mapsto \mathit{count}_0 \upharpoonright p.\mathit{inc} \upharpoonright p.\mathit{inc})$, are forbidden.

4.2 Input Streams

In some cases, we may wish to provide similar synchronized and unsynchronized interfaces to the same object. For example, an input stream may provide both a synchronized method *read* for reading characters from the stream, and a faster but unsynchronized method read'. (The Modula-3 I/O package provides both of these methods [14].)

An outline implementation of such an input stream might be:

$$\begin{array}{lll} instream \triangleq [\mathit{buffer} = \varsigma(x)\ldots, & & \text{internal data structure} \\ \quad read' = \varsigma(x)\cdots \mathit{buffer}\cdots, & & \text{fast, unsynchronized read} \\ \quad read = \varsigma(x) lock\ x\ in\ x.read'] & & \text{slower, synchronized read} \end{array}$$

The method *buffer* contains some internal data structures of the input stream and is protected by the self lock. The method $read'$ assumes that the self lock is held, and returns the next input character after some manipulation of *buffer*. The method *read* does not assume that the self lock is held; it first acquires that lock and then dispatches to $read'$.

A suitable type for this input stream is:

$$[\mathit{buffer} : \varsigma(x)Buffer\cdot\{x\}\cdot x, \quad read' : \varsigma(x)Char\cdot\{x\}\cdot+, \quad read : \varsigma(x)Char\cdot\varnothing\cdot+]$$

Subtyping then allows us to view an input stream as having either the synchronized interface $[read : \varsigma(x)Char\cdot\varnothing\cdot+]$ or the faster but unsynchronized interface $[read' : \varsigma(x)Char\cdot\{x\}\cdot+]$.

4.3 Lines and Points

The examples above describe objects whose components are protected by the self lock of the object. In addition, object components can also be protected by a lock external to the object. To illustrate this possibility, we consider the following example consisting of point and line objects.

$$\begin{aligned} point \triangleq [&x = \varsigma(s)0, \\ &y = \varsigma(s)0, \\ &bmp = \varsigma(s) let\ t{=}s.x + 1\ in\ s.x \Leftarrow \varsigma(s)t] \end{aligned}$$

$$\begin{aligned} line \triangleq [&start = \varsigma(s)pt_1, \\ &end = \varsigma(s)pt_2, \\ &bmp = \varsigma(s) lock\ s\ in\ (s.start.bmp;\ s.end.bmp)] \end{aligned}$$

A point contains a method *bmp* that increments the x-coordinate of the point. (An analogous method for y is omitted for brevity.) Each line object includes two methods for its end points, *start* and *end*, and a method *bmp* that increments the x-coordinate of both end points of the line. This method first acquires the self lock of the line, then calls the method *bmp* of both end points. These points do not perform any synchronization internally; their mutable methods x and y are protected by the lock of the enclosing line object. Appropriate types for lines and points are:

$$\begin{aligned} Point_z &\triangleq [x : \varsigma(s)Int\cdot\{z\}\cdot z,\ y : \varsigma(s)Int\cdot\{z\}\cdot z,\ bmp : \varsigma(s)[\ ]\cdot\{z\}\cdot+] \\ Line &\triangleq [start : \varsigma(s)Point_s\cdot\varnothing\cdot+,\ end : \varsigma(s)Point_s\cdot\varnothing\cdot+,\ bmp : \varsigma(s)[\ ]\cdot\varnothing\cdot+] \end{aligned}$$

where the type $Point_z$ describes a point whose mutable methods are protected by the lock z. The type *Line* states that the methods *start* and *end* yield points whose mutable components are protected by the lock of the enclosing line object.

Interestingly, the type *Line* also permits the following line object:

$$\begin{aligned} line' \triangleq [&start = \varsigma(s)pt_1, \\ &end = \varsigma(s)pt_2, \\ &bmp = \varsigma(s)(lock\ s\ in\ s.start.bmp);\ (lock\ s\ in\ s.end.bmp)] \end{aligned}$$

When a thread runs the method *bmp* of this object, it acquires and releases the self lock twice; it does not hold the lock continuously throughout the execution of the method. Therefore, another thread may observe *line*′ in an intermediate state where *start* has been updated but *end* has not. Such an intermediate state might violate higher-level invariants, so the interleaving of the two threads might be regarded as a higher-level race condition. Our type system does not address such errors directly.

4.4 Encoding Functions as Race-Free Objects

We encode function abstraction and application in our calculus as follows, much as in other object calculi [1, 9]:

Encoding functions

$$\begin{aligned} \lambda(x)b &\triangleq [new = \varsigma(s)[arg = \varsigma(s)s.arg,\ val = \varsigma(s)let\ x{=}s.arg\ in\ b]] && \text{for } s \notin fv(b) \\ b(a) &\triangleq let\ y{=}b.new\ in\ lock\ y\ in\ (y.arg \Leftarrow \varsigma(z)a).val && \text{for } y, z \notin fv(a) \end{aligned}$$

In the absence of cloning, we need to use a method *new* to create a fresh object with the usual methods *arg* and *val*. The self lock is acquired before accessing *arg* and *val*. Locking is needed for both methods because *arg* is mutable and *val* invokes *arg*.

This translation provides an encoding for the simply-typed call-by-value λ-calculus; a function of type $A \rightarrow B$ is mapped to an object of type:

$$[new : \varsigma(s)[arg : \varsigma(s)A \cdot \{s\} \cdot s,\ val : \varsigma(s)B \cdot \{s\} \cdot +] \cdot \varnothing \cdot +]$$

This translation cannot encode dependent function types, in which the result type depends on the argument value. Encoding dependent function types in our calculus seems to require an extension, for example allowing the use of terms (and not just results) as locks.

4.5 Other Encodings (Sketches)

We can translate programs of the imperative object calculus **imp**$\boldsymbol{\varsigma}$ [1] into our calculus in a straightforward manner, much as Gordon and Hankin. (Since our calculus does not include cloning, this translation only works for clone-free programs.) Each translated program includes a single global lock, which protects all object components in the program. Since the program is single-threaded, this lock needs to be acquired only once, at the start of the program's execution;

it is then held throughout the execution, allowing unrestricted invocations and updates of object components.

Gordon and Hankin describe an encoding of the π-calculus into their concurrent object calculus. Their encoding is based on an implementation of channels. A similar approach works in our setting, but not as neatly. Because the locks of our calculus are not semaphores, our implementation of channels uses busy-waiting; for example, reading from a channel may involve looping until a value is written to that channel by some other thread.

5 Well-Typed Programs Don't Have Races

The fundamental property of the type system is that well-typed programs do not have race conditions. We formalize the notion of a race condition as follows. A term b *reads* $p.\ell$ if there exists some $\mathcal{E}$ such that $b = \mathcal{E}[\ p.\ell\ ]$ and $p \notin bn(\mathcal{E})$. Similarly, a term b *writes* $p.\ell$ if there exists some $\mathcal{E}$, x, and c such that $b = \mathcal{E}[\ p.\ell \Leftarrow \varsigma(x)c\ ]$ and $p \notin bn(\mathcal{E})$. A term *accesses* $p.\ell$ if it either reads or writes $p.\ell$. A term b has an *immediate race condition* if there exists some c_1, c_2, and $p.\ell$ such that $b \equiv \mathcal{E}[\ c_1 \upharpoonright c_2\ ]$, c_1 and c_2 both access $p.\ell$, and at least one of those accesses is a write. Finally, a term b has a *race condition* if its evaluation may yield a term with an immediate race condition, that is, if there exists a term c such that $b \rightarrow^* c$ and c has an immediate race condition.

The type system ensures that, in a well-typed program, a thread that accesses a method holds the appropriate locks. The following lemma is crucial in establishing that property.

Lemma 1. *If $E; r \vdash b : B$ and b accesses $p.\ell$ then $E; \varnothing \vdash p : [\ell : \varsigma(x)A \cdot r' \cdot s]$ and $s\{\!\{x \leftarrow p\}\!\} \in clean(b) \cup r \cup \{+\}$. Furthermore, if the access is a write, then $s \neq +$.*

Since each lock can be held by at most one term at any time, a well-typed program does not have an immediate race condition.

Lemma 2. *If $E; \varnothing \vdash b : B$ then b does not have an immediate race condition.*

Furthermore, typing is invariant under reduction.

Lemma 3. *If $E; r \vdash b : B$ and $b \rightarrow c$ then $E; r \vdash c : B$.*

Using the previous lemmas, we can prove that well-typed programs do not have race conditions.

Theorem 1. *If $E; \varnothing \vdash b : B$ then b does not have a race condition.*

6 Conclusion

As this paper shows, a simple type system can help detect and avoid some synchronization errors in concurrent object-oriented programs. Our type system builds on the underlying object constructs: it extends standard object types with locking information. Through operational arguments, we establish that well-typed programs do not have race conditions.

A static-analysis technique such as ours is necessarily incomplete. In practice, it probably should be complemented with mechanisms for escaping its requirements, that is, with means for asserting that program fragments do not have race conditions even when these fragments do not typecheck. Complementarily, we are currently investigating type systems more sophisticated and liberal than the one presented in this paper. In the context of Java, we are also considering implementations of our methods.

Acknowledgments

We thank Andrew Gordon and Paul Hankin for permission to adapt their macros and inference rules.

Appendix: Formal Semantics

The formal semantics of our calculus closely follows that given by Gordon and Hankin. It consists of a group of structural congruence rules ($\equiv$), which permit the rearrangement of terms, and a group of reduction rules ($\rightarrow$), which model proper computation steps. In addition to the rules listed here, we also use a set of rules that imply that $\equiv$ is a congruence relation.

Structural congruence rules

(Struct Res $\mathcal{E}$)

$$\frac{p \notin fn(\mathcal{E}) \cup bn(\mathcal{E})}{(\nu p)\mathcal{E}[\ a\ ] \equiv \mathcal{E}[\ (\nu p)a\ ]}$$

(Struct Par $\mathcal{E}$)

$$\frac{fn(a) \cap bn(\mathcal{E}) = \varnothing}{a \Rsh \mathcal{E}[\ b\ ] \equiv \mathcal{E}[\ a \Rsh b\ ]}$$

Reduction rules

(Red Let)

$$\frac{}{let\ x{=}p\ in\ b\ \rightarrow\ b\{\!\{x \leftarrow p\}\!\}}$$

(Red Select)

$$\frac{d = [\ell_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]^m \quad j \in 1..n}{(p \mapsto d) \Rsh p.\ell_j\ \rightarrow\ (p \mapsto d) \Rsh b_j\{\!\{x_j \leftarrow p\}\!\}}$$

(Red Update)

$$\frac{d = [\ell_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]^m \quad j \in 1..n \quad d' = [\ell_j = \varsigma(x)b, \ell_i = \varsigma(x_i)b_i\ ^{i\in (1..n)-\{j\}}]^m}{(p \mapsto d) \Rsh (p.\ell_j \Leftarrow \varsigma(x)b)\ \rightarrow\ (p \mapsto d') \Rsh p}$$

(Red Lock) (where $[\ldots] = [\ell_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]$)

$$\frac{}{(p \mapsto [\ldots]^\circ) \Rsh lock\ p\ in\ a\ \rightarrow\ (p \mapsto [\ldots]^\bullet) \Rsh locked\ p\ in\ a}$$

(Red Locked) (where $[\ldots] = [\ell_i = \varsigma(x_i)b_i\ ^{i\in 1..n}]$)

$$\frac{}{(p \mapsto [\ldots]^\bullet) \Rsh locked\ p\ in\ u\ \rightarrow\ (p \mapsto [\ldots]^\circ) \Rsh u}$$

(Red $\mathcal{E}$)
$$\frac{a \ \rightarrow \ a'}{\mathcal{E}[\ a\ ] \ \rightarrow \ \mathcal{E}[\ a'\ ]}$$

(Red Struct)
$$\frac{a \equiv a' \quad a' \ \rightarrow \ b' \quad b' \equiv b}{a \ \rightarrow \ b}$$

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, 1996.
3. David Aspinall and Adriana Compagnoni. Subtyping dependent types. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 86–97, July 1996.
4. Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.
5. Andrew D. Birrell. An introduction to programming with threads. Research Report 35, Digital Equipment Corporation Systems Research Center, 1989.
6. Paolo Di Blasio and Kathleen Fisher. A calculus for concurrent objects. In *CONCUR'96: Concurrency Theory*, volume 1119 of *Lecture Notes in Computer Science*, pages 655–670. Springer Verlag, 1996.
7. Luca Cardelli. Phase distinctions in type theory. Unpublished, 1988.
8. Cormac Flanagan and Martín Abadi. Types for safe locking. In *Proceedings of the 8th European Symposium on Programming, ESOP '99*, pages 91–108. Springer-Verlag, March 1999.
9. Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In Uwe Nestmann and Benjamin C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages*, volume 16.3 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998. An extended version appears as Technical Report No. 457 of the University of Cambridge Computer Laboratory, February 1999.
10. James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley, 1996.
11. Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 341–354, January 1990.
12. Martin Hofmann, Wolfgang Naraschewski, Martin Steffen, and Terry Stroup. Inheritance of proofs. *Theory and Practice of Object Systems*, 4(1):51–69, 1998.
13. Naoki Kobayashi. A partially deadlock-free typed process calculus. In *Proceedings of the 12th Annual IEEE Symposium on Logic in Computer Science*, pages 128–139, 1997.
14. Greg Nelson, editor. *Systems Programming in Modula-3*. Prentice Hall, 1991.
15. Nicholas Sterling. Warlock: A static data race analysis tool. In *USENIX Winter Technical Conference*, pages 97–106, 1993.
16. Eijiro Sumii and Naoki Kobayashi. A generalized deadlock-free process calculus. In Uwe Nestmann and Benjamin C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages*, volume 16.3 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.