Extending the Java Language for the Prevention of Data Races

by

Robert Honway Lee

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2002

Author ....
Department of Electrical Engineering and Computer Science
May 26, 2002

Certified by ....
Martin Rinard
Associate Professor
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Extending the Java Language for the Prevention of Data Races

by

Robert Honway Lee

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 2002, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

This thesis presents a language extension for Java that ensures that type-checked programs will be free of race conditions. By adding parameterization to class types, this mostly static type-system allows programmers to write generic object code independent of the concurrency protection mechanism to be used to guard it. This added flexibility makes this extension more expressive and easier to use than other similar type systems. We will present the formal type system as well as our experience with implementing a compiler for this extension.

Thesis Supervisor: Martin Rinard
Title: Associate Professor

Acknowledgments

I am grateful for the advice and guidance that I received from my advisor, Martin Rinard, over the past years. I am also grateful to the other members of my research group; especially to Chandra Boyapati for his guidance and ideas in developing this language. I would also like to thank Allison Waingold and Jon Whitney for their support, feedback and ideas on improving this project and realizing this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The multi-threaded programming model is quickly becoming more prevalent. In this model programmers can use multiple parallel flows of control which can share resources in order to optimize around performance-limiting constructs like blocking I/O calls. However, multi-threaded programming involving shared resources introduces a very severe class of bug: data-races.

A data-race can be thought of as either non-deterministic execution of program code which is intended to be deterministic, or non-atomic access to shared data intended to be atomic [8]. Race condition errors are usually the hardest to detect and eliminate, often times because they are difficult to reproduce. There are several tools that currently exist to help programmers eliminate race-conditions. These tools use techniques such as dynamic analysis [10], static analysis, and extended type systems.

Unwanted race conditions on critical sections of code can be prevented by explicit synchronization. By imposing ordering constraints on the execution of code segments, programmers can limit the possible runtime execution sequences. The abstract approach to imposing these ordering constraints is to associate a *lock* with each piece of shared data that needs to be protected. Each time the data is accessed, the accessing thread must first acquire the lock that protects the data. Correspondingly, when the thread is finished with the data, it should release the lock so that other threads may acquire it.

Synchronization (acquiring and releasing a lock) allows the programmer to control what parts of code may and may not be run synchronously. By making the acquire and release operations atomic (they occur in one logical step with no intervening steps) and having all threads acquire a lock before accessing the shared data, programmers can protect a data access against race considtions. However, synchronizing on concurrent accesses to shared data is left up to programmer discipline which is often the souce of inadvertant programming errors.

Thesis presents Parameterized Race Free Java (PRFJ), a language extension to Java, and its implementation. The extended language incorporates synchronization locks into the formal type system so that a program can be statically verified to be free of race-conditions while still reducing the amount of unnecessary overhead.

Several previously proposed type systems exist that have similar goals of producing race-free programs [3, 4, 5, 6]. However, these systems suffer from the fact that they are not flexible enough to allow programmers to specify the object protection policy at the point of object instantiation. The immediate result of this is that programmers are forced to acquire redundant locks in order to satisfy the type-checker, or are required to unnecessarily duplicate code.

Our new type system allows programmers to specify, at creation time, the protection mechanism that is to be used to guard accesses to a particular object. By making this protection policy a parameter of the object's type, programmers are able to write object code independent of the particular protection mechanism to be used, which they can postpone assigning until object instantiation.

In designing this type system, there were several key challenges:

- Making the type system sound while avoiding unnecessary locking.
- Making the type system expressive enough to accommodate common programming idioms.
- Making the language extension small enough to facilitate porting existing Java systems into PRFJ.
- Supporting the whole Java language including the runtime type-system.

This thesis will present the core language in Chapter 3 and the accommodations for the existing Java language in Chapter 4. Chapter 5 will present the formal type-system and Chapters 6 and 7 will discuss the implementation of the type-inference algorithm and the compiler.

Chapter 2

Synchronization in Java

```
public synchronized void useData()
{
    // protected code
}
```

(a)

```
private Object sharedData;

...

public void useData()
{
    synchronized(sharedData)
    {
        // protected code
    }
    //unprotected code
}
```

(b)

Figure 2-1: (a) Method-level synchronization. (b) Finer granularity explicit synchronization

The language extension that this thesis is based on is an extension of the Java programming language and makes use of Java constructs, therefore it is worthwhile to briefly summarize the synchronization constructs provided by the existing language.

Java's synchronization model is *monitor*-based [9]. In this model, each instance of an object is associated with a monitor(mutex). In order to execute protected code, the thread must first acquire the monitor associated with the object. Code can be synchronized on the method level by adding the `synchronized` modifier to the method declarator (Figure 2-1a). Threads must acquire the monitor associated with the enclosing object in order to execute synchronized methods.

It is also possible to synchronize code on a finer granularity than methods. Blocks of code can be synchronized by explicitly declaring an object to synchronize on. For example, consider the code in Figure 2-1b. The section within the `synchronized` block in method `useData()` is synchronized on the monitor for `sharedData` and not the monitor for `this`. Furthermore, any code that is in the method but outside of the synchronized block is not protected by synchronization. This might be desirable if, for example, there is a portion of the method that does not need to be synchronized because it does not modify or access any shared data. (Note that declaring a method to be synchronized is equivalent to synchronizing it's entire body on a reference to `this`).

The synchronization mechanism instructs the virtual machine (VM) to atomically perform a *lock* action on the monitor before entering a synchronized scope, and an *unlock* action upon exiting the scope. By disciplined use of the synchronization mechanism, programmers can assure that certain blocks of code may not be executed simultaneously. However, acquiring monitor locks alone does not ensure that other threads of execution will not access or modify object fields or invoke unsynchronized methods of the shared object.

Java also allows threads to communicate using condition variables. The basic notion of a condition variable is that a thread blocks until some condition becomes true. In the case of Java's built-in condition variable, the thread will wait until the notified condition is set to true by some other thread calling `notify()`. When the waiting thread calls `wait()`, it releases the monitor and the thread is put into the object's *wait-set* and no longer participates in scheduling.

At this point, because the waiting thread has released the lock on the object's monitor, other threads are free to execute synchronized code. When a thread receives the `notify` or `notifyAll` signal, the thread is removed from the wait-set and is re-enabled for scheduling. It is important to note that although a thread releases the monitor when it makes the `wait()` call, before it resumes execution after waking up, it must first re-acquire the lock.

Monitor synchronization allows programmers to restrict multiple threads from simultaneously executing code protected by the same monitor. Condition variables allow threads to block execution until some condition holds on shared data. Together, these two constructs give programmers and expressive, but undisciplined way to write multi-threaded programs.

Chapter 3

Core Language

The PRFJ type system builds on the type system presented by Boyapati and Rinard [1]. The key to the type system behind PRFJ is the idea of ownership types. Each object in the PRFJ type system has an owner associated with it. An object's owner can be itself, another object, or one of the special owners which are defined in the language.

Object owners satisfy two important properties in this type system:

- An object's owner does not change over its lifetime
- The ownership relationship forms a forest of rooted trees where the roots can have self-loops.

The reason it is necessary for an object's owner to remain the same over time is to avoid a situation where two threads are competing for access to the object and lock different objects which they believe to be the object's owner. By ensuring that an object cannot change its owner, we can statically ensure that threads will not mistakenly make an unsynchronized access to an object.

In our system, objects are protected by the root of their ownership tree. Therefore, it is necessary and sufficient for a thread to hold the lock to the root of the ownership tree to ensure exclusive access to all members of the tree.

The type of each object in this system is parameterized by the owner parameters which are assigned to the object at creation time. In this manner, it is easy for the programmer to write generic implementations of classes that are independent of the particular locking discipline that is to be used to protect it. It is only at object creation time that the programmer needs to specify the particular protection mechanism to be used.

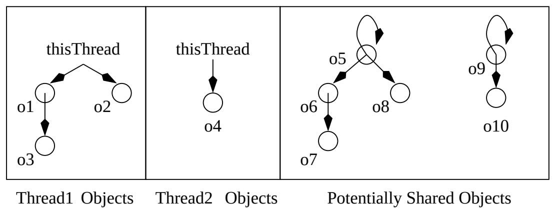

Figure 3-1: An Ownership Relation

Figure 3-1 presents an example of object ownership. In the figure, there is an arrow from object x to object y if object x owns object y. The figure shows that the `thisThread` owner of Thread 1 transitively owns objects o_1, o_2, and o_3, the `thisThread` owner of Thread 2 owns object o_4, object o_5 transitively owns objects o_5, o_6, o_7, and o_8, and object o_9 owns objects o_9 and o_{10}. In other words, objects o_1, o_2, and o_3 are thread-local to Thread 1, o_4 is local to Thread 2, and $o_5 \ldots o_1 0$ can potentially be shared across threads.

ClassDeclaration	::=	ClassModifier$_{opt}$ `class` Identifier⟨Firstowner Formal*⟩
		Super$_{opt}$ Interfaces$_{opt}$ ClassBody
Super	::=	`extends` ClassType
ClassType	::=	TypeName
Interfaces	::=	`implements` InterfaceTypeList
InterfaceTypeList	::=	InterfaceType | InterfaceTypeList , InterfaceType
InterfaceType	::=	TypeName
TypeName	::=	Identifier⟨Owner+⟩ (`[]` ⟨Owner+⟩)?
MethodDeclaratorRest	::=	MethodFormalParameters BracketsOpt (`requires` arguments)?
		(`throws` QualifiedIdentifierList)? (MethodBody | ;)
Formal	::=	Identifier | _
FirstOwner	::=	Formal | SpecialOwner
Owner	::=	Identifier | Expression | SpecialOwner
SpecialOwner	::=	`self` | `thisThread` | `readonly` | `unique`
MethodFormalParameters	::=	MethodFormalParameter (, MethodFormalParameter)*
MethodFormalParameter	::=	(`final`)? TypeName (UniquePointerInfo)? VariableDeclaratorId
UniquePointerInfo	::=	`!(e)?(w)?`

Figure 3-2: Extensions to the Java grammar

3.1 Informal Semantics

The grammar for PRFJ is given by extending the grammar for the Java programming language [9] and is shown in Figure 3-2.

The basic difference as a result of the extension is that reference types are now parameterized with owner parameters. When an object is declared and created, the fully parameterized type of the object is specified. Accordingly, that is the point at which a protection mechanism is associated with the object. A protection mechanism defines the lock which protects the object or that the object needs no lock because it is thread-local, immutable, or a unique object.

The type checker then verifies that the object is only used in accordance with the protection mechanism that is specified for the object.

3.2 Parameterized Classes

Every class in this type system is parameterized by one or more formal parameters. These parameters will be instantiated at object creation time with appropriate owner expressions that describe how the data structure is protected. The first parameter in a class type has a special meaning and is always considered the “owner” of this object. Any of the formal parameters may be used within the body of the class to parameterize field or local variable creation, as well as in method declarations.

The formal parameters in a class definition may be used to propagate the protection mechanism to within the class body. An example of such a use is in a container class. The container object would be declared with two owners, one describing the owner of the container object, and one describing the owner of the elements in the container. An example of this usage presented in the `TStack` example in Figure 3-4.

The semantics of parameterizing classes and instantiating objects are similar to much of the previous work on adding parameterized types (also known as generic types) to Java [11]. The key difference is that the parameters in generic types are themselves type values, whereas here, the parameters can actually be runtime objects.

If a formal parameter of a class is never used within the body of the class, it may be replaced with a _ or omitted and a default formal parameter will be generated. Omitting class formal owners has implications on the runtime system, which is described in Chapter 4.3.

3.3 Instantiating Objects

Objects in PRFJ are instantiated with actual parameters which describe the locking discipline to be used with that specific object. Owner parameters can be a final expression of any reference type, formal parameters, or any of the special owners: `self`, `thisThread`, `readonly`, `unique`. In order to maintain the property that the owner of an object never changes over the life of that object, it is necessary to restrict expression owners to final expressions (or `this`).

Recall that classes may declare more than one formal parameter. For example, the `TStack` class in Figure 3-4 represents a stack of `T` objects. The `TStack` class declares two formal parameters, `thisOwner` and `TOwner`. As we previously discussed, the first formal parameter, `thisOwner`, represents the owner of this object (the `TStack`). The `TOwner` parameter is later used to parameterize the instance field `head` and also the `push()` and `pop()` methods. In this case, the `TOwner` is being used to as the owner of the `TNode` objects contained within the stack.

By allowing programmers to parameterize fields, variables, and methods with formal owner parameters, we are allowing them to propagate the ownership information into their data structures. For example, in the code from Figure 3-4:

```
TStack<thisThread, thisThread> s1 = new TStack<thisThread, thisThread>();
TStack<thisThread, self> s2 = new TStack<thisThread, self>();
final TStack<self, self> s3 = new TStack<self, self>();
TStack<s3, self> s4 = new TStack<s3, self>();
```

The declaration of `s1` says that `s1` will be a `TStack` which is owned by the `thisThread` owner; the stack constains `T` objects which are also owned by `thisThread`. The declaration of `s2` gives a `TStack` which is owned by the `thisThread` owner of `T` objects which are owned by the `self` owner. The declaration of `s3` is another permutation, this time of a `TStack` which is owned by the `self` owner where the stack contains `T` objects which are also owned by `self`. Finally, `s4` is declared to be a `TStack` which is owned by `s3` of `T` objects which are owned by `self`. In order to fully understand this example, we need to examine the properties of the special owners.

3.3.1 `thisThread` Owner

Each thread has its own `thisThread` owner. A thread also implicitly locks its `thisThread` owner and cannot access another thread's `thisThread` owner. In other terms, objects which are owned by `thisThread` can be thought of as being thread-local. As such, the thread need not acquire additional locks before accessing `thisThread`-owned objects. This is shown in the `TStack` code:

```
TStack<thisThread, thisThread> s1 = new TStack<thisThread, thisThread>();
TStack<thisThread, self> s2 = new TStack<thisThread, self>();
...
s1.push(t1);
s2.push(t2);
```

Objects owned by `thisThread` may not migrate between threads.

3.3.2 `self` Owner

The `self` owner is used to indicate that an Object owns itself. In order to safely access an object which is owned by `self`, it is necessary to lock the root owner of the object, which in this case, is the object itself. The `TStack` code shows an example of this paradigm:

```
final TStack<self, self> s3 = new TStack<self, self>();
...
synchronized(s3) {
    s3.push(t2);
}
```

Here, the call to `s3.push()` requires the root owner of `s3` (which is `s3` itself) be held. The enclosing `synchronized` block acquires the lock and makes the method call safe.

3.3.3 `unique` Owner

The `unique` owner indicates that an Object reference is unique. That is, that there is at most one reference to any unique object on the heap at a time. Knowing that a reference is unique is useful because it means that the thread holding the reference can safely access the object without synchronization because no other thread could simultaneously hold a reference to the same object.

The idea of unique pointers is useful in supporting a common programming paradigm, the producer-consumer model. In this model, one or more threads may create and initialize objects and then pass the objects to other threads to be "consumed".

In order to statically show that a pointer is unique, we make some restrictions. Non-unique objects may not be assigned to unique references. Furthermore, a unique object may only be assigned to another unique reference through a special form:

```
x = y--;    //x = y; y = null;
m(y--);     //m(y); y = null;
```

The special form for dereferencing a unique reference evaluates to the reference and then immediately assigns it to point to `null`. As a result, the dereferenced pointer can safely be assigned to another unique reference.

Special care must also be taken to ensure that unique pointers do not escape as a result of method calls. A pointer escapes the method if, at the end of the method, there is a new reference to the object. To allow programmers to pass unique pointers as arguments to method calls, we require that the method being called declare that it will not cause the argument to escape by assigning a new reference to point to it. It can do this by annotating the argument declaration with `!e` to signify that the variable has a non-escaping type.

If a variable has a non-escaping type, then it means that the reference stored in the variable will not escape to any object field or to another thread. A variable with a non-escaping type can be assigned only to other variables with non-escaping types. Similarly, it can passed as a method argument only if the type of the argument is specified to be non-escaping in the method declaration.

The code below is an example of where a unique `Message` object is passed as an argument to a `display` method that declares that the `Message` argument will not escape.

```
class Message<thisOwner> {...};

class Util<thisOwner, MsgOwner> {
   void display(Message<MsgOwner>!e m) requires(m) {...}
}

...
Util<self, unique> u = new Util<self, unique>();
Message<unique>    m = new Message<unique>();
u.display(m);
```

In addition to specifying that unique objects may only be passed to methods which declare that references to the argument will not escape, there is an additional restriction on class formal parameters. Not all classes can be instantiated with the `unique` owner. For example, in the code for `TStack` in Figure 3-4, the `TOwner` parameter must not be instantiated with `unique` because the code in the `TNode` class will escape the reference.

We can control this by imposing a constraint on the `TStack` class that `TOwner` must not be unique.

```
class TStack<thisOwner, TOwner> where (TOwner!=unique)
{ ... }
```

3.3.4 `readonly` Owner

The `readonly` owner indicates that an Object is read-only. Read-only objects can only be read from, and not written to. Because of this, it is safe for multiple threads to access them without synchronization. One way to create a readonly object is to first create the object with the unique owner. After it is initialized and written to, the `m--` form can be used to assign it to a readonly reference. For example, the code below shows how you would create a readonly reference to a Message object.

```
Util<self, unique> u = new Util<self, unique>();
Message<unique>    m = new Message<unique>();
// write to m
Message<readonly> rm = m--;
```

By using the `m--` form, the assignment to `rm` guarantees that `rm` is still the only reference to the Message object at this point in the code. Because the declared type of `rm` is read-only, the static type system guarantees that the only additional references to the `rm` object created will also be owned by the `readonly` owner.

Immutable classes may also declare themselves to be readonly by specifying the `readonly` owner as their first formal parameter. For example, the specification file (see Chapter 4.4 for the `java.lang.String` class specifies that it is a readonly class. This means that all instances of String are readonly and can safely be referenced without locking.

3.4 Parameterized Methods

In some cases, it is desirable to define a method which defines argument types and a return type which are polymorphic over some set of owners. We introduce the ability to define method-level formal owner parameters by simply parameterizing the types with fresh iden-

```
public static TStack<thisThread, any> makeStack(TNode<any> tNode) requires() {
    TStack<thisThread, any> ts = new TStack<thisThread, any>();
    ts.push(tNode);
    return ts;
}
```

Figure 3-3: An example of a parameterized method

tifiers. This is useful for writing more generic code and allows the programmer to further abstract away the protection mechanism from the actual code.

For example, suppose we want to define a factory method for `TStack` objects which takes an initial `TNode` object as an argument. Figure 3-3 shows an example implementation.

Here, the `any` parameter defines a method-level formal owner which can be instantiated with any actual owner. This allows for a more generic implementation of this factory method rather than multiple implementations for each possible `TNode` owner.

Allowing the use of parameterized methods does, however, cause some difficulty. For example, what should happen if a null literal is passed in as an argument to the `makeStack()` method in Figure 3-3? The type of a null-literal is a special case in that it defines no formal owner parameters. The question then becomes, what is the `any` formal method-level parameter bound to and what type does the `ts` object (and ultimately the method return value) have? Chapters 6 and 4.3 will describe how problems such as these are resolved.

```
class TStack<thisOwner, TOwner> {
    TNode<this, TOwner> head = null;

    public void push(T<TOwner> value) requires (this) {

        TNode<this, TOwner> newNode = new TNode<this, TOwner>();
        newNode.init(value, head);

        head = newNode;
    }

    public T<TOwner> pop() requires (this) {

        if(head == null)
        {
            return null;
        }

        T<TOwner> value = head.value();
        head = head.next();
        return value;
    }
}

class TNode<thisOwner, TOwner> {

    T<TOwner> value;
    TNode<thisOwner, TOwner> next;

    public void init(T<TOwner> v, TNode<thisOwner, TOwner> n) requires(this) {
        this.value = v;
        this.next = n;
    }

    public T<TOwner> value() requires(this) {
        return value;
    }

    public TNode<thisOwner, TOwner> next() requires(this) {
        return next;
    }
}

class T<thisOwner> {
    int x = 0;
}

class TStackDriver<owner> {

    public static void main(String<readonly>[]<readonly> args) requires(this) {

        T<thisThread> t1 = new T<thisThread>();
        T<self> t2 = new T<self>();

        TStack<thisThread, thisThread> s1 = new TStack<thisThread, thisThread>();
        TStack<thisThread, self> s2 = new TStack<thisThread, self>();
        final TStack<self, self> s3 = new TStack<self, self>();
        TStack<s3, self> s4 = new TStack<s3, self>();
        s1.push(t1);
        s2.push(t2);

        synchronized(s3) {
            s3.push(t2);
        }
    }
}
```

Figure 3-4: Example Code for `TStack`, a Stack of `T` objects

Chapter 4

Extending the Type System for Java

This chapter discusses some of the details of the Java programming language which the initial type-system [1] was expanded to include.

4.1 Arrays

In order to support arrays, we introduce a special form for declaring array types. In Java, arrays are first-class objects in their own right [9]. For example, arrays have a `length` property as well as all of the methods that they inherit from `java.lang.Object`.

Array types declare some base element-type and also define the dimension of nesting. In our type system, array objects essentially have 2 sets of owners parameters, one for the owners of the base type, and one for the actual array object. The example below shows a declaration of an array of `TStack<thisThread, self>` objects where the array is self-owned.

```
TStack<thisThread, self>[]<self> t = new TStack<thisThread, self>[10];
```

It is necessary to make the distinction between parameterizing the array object and parameterizing the array elements. For example, if you have an one-dimensional array of reference types, it is possible to race on the array itself (to have two threads competing to read and update a given element) and also to race on the contents of an array element.

4.2 Static Fields

Supporting static fields in Java requires some special handling. Because it is possible to globally reference static class fields, they must be treated as global variables. Thus, they can be accessed from any program point (or more importantly, from any running thread). We handle this by requiring that all accesses of static fields of a class `A` hold a lock on the `A.class` object. By using the singleton class object [9] to control access to static fields, we can protect these global accesses. For example, the following is a valid access of the static field `counter` in class `StaticCounter`:

```
synchronized(StaticCounter.class) {
  StaticCounter.counter++;
}
```

Although there are other ways of referencing the class Object for the `StaticCounter` class, this is the only way that we support as it is the only semantic approach that is resolved statically.

While this treatment of static fields may lead to extra synchronization, it is necessary in order to maintain the soundness of the type-system. In general, heavy use of static fields is not good as they are essentially global variables and thus limit the programmer's ability to reason locally. Chapter 8 on results discusses an alternative paradigm to lessen the synchronization burden involved with static fields while maintaining a good programming style.

4.3 Runtime Type System

Up until now, the type system described has been entirely static. However, Java's type system is not entirely static (namely runtime downcasting and the `instanceof` operator). Unfortunately, the type-system so far is not adequate to handle these operations. For example, consider the following code:

```
TStack<thisThread, self> ts = new TStack<thisThread, self>();
Object<thisThread> o = ts;
...
TStack<thisThread, thisThread> tt = (TStack<thisThread, thisThread>)o;  // bad!!!
```

In the existing type system, we can verify that the declaration of `ts` and `o` are legal. Furthermore, we can verify the assignment of `ts` to `o` and the implicit upcast that is involved with that. However, on the downcast of `o` to type `TStack<thisThread, thisThread>`, we can only verify that the first `thisThread` parameter is legal. This is because an object's owner can never change, so any downcast must preserve the same first owner. However, the second parameter (and any subsequent parameters) are not statically verifiable.

At runtime, the JVM will be able to check that the `o` object is of type `TStack` (unparameterized). However, the JVM has no way of telling dynamically if the object was created as a `TStack<thisThread, self>` or `TStack<thisThread, thisThread>`.

In order to support the runtime features of Java's type within the JVM specification, we extended the compiler to propagate some owner information in the body of the objects at runtime. The transformation we chose is similar to the type-passing approach to generics as presented by Viroli et al [12]. Without modifying the semantics of the JVM or Java bytecodes, this is a very lightweight approach while still maintaining a sound type-system.

```
public class $OD {
    private Object owner;
    public static $OD THISTHREAD = new $OD(``thisThread'');
    public static $OD READONLY = new $OD(``readonly'');
    public static $OD UNIQUE = new $OD(``unique'');
    public static $OD SELF = new $OD(``self'');

    public $OD(Object o) {
        owner = o;
    }
    public boolean equals(Object o) {
        return (this == o);
    }
}
```

Figure 4-1: Code for the `$OD` class which is used to wrap runtime owners.

The basic approach is to determine which classes need to be instrumented to carry ownership information around in runtime. Any class which specifies formal owner parameters must carry the corresponding owner expressions in the body of each instance. The reason for this is that if any of those formal parameters are used to instantiate other objects, the runtime information must be propagated to those subsequently created objects. If a particular class definition makes no use of a formal parameter, it may specify _ as the parameter name. This indicates that there is no need to propagate that particular parameter at runtime.

One other place where it is necessary to instrument for runtime parameters is in methods which declare method-level owners. Recall that methods may declare new "formal owner parameters" in order to allow programmers to declare more polymorphic methods. So for example, the `makeStack()` method in `TStack` declares a method-level parameter, `any`, and uses it to parameterize a new object.

```
class TStack {
TNode head = null;

  private $OD[] $_ods;
  public TStack($OD[] ods) {
    $_ods = ods;
  }

  public void push(T value) {

    TNode newNode = new TNode(new $OD[]{new $OD(this), $_ods[1]});
                 //newNode: TNode<this, TOwner>
    newNode.init(value, head);

    head = newNode;
  }

  public T<TOwner> pop() {

    if(head == null) {
      return null;
    }

    T value = head.value();    //value : T<TOwner>
    head = head.next();
    return value;
  }

  public static TStack makeStack($OD[] ods, TNode tNode) requires() {
    TStack ts = new TStack(new $OD[]{ods[0]});
    ts.push(tNode);
    return ts;
  }
}
```

Figure 4-2: Example translation of `TStack` class

The owner descriptors which are passed into the parameterized objects are stored in an instance field named `$_ods`. Within the class body, references to formal parameters will resolve to the corresponding index into the `$_ods` array.

When instantiating an object of a class that requires ownership passing, the owner parameters of the parameterized class type encoded by owner descriptor (`$OD`) objects. In the parameterized class type, the owner parameters are either final expressions, special owners, or formal parameters (as shown in the extended grammar in Figure 3-2). If the owner parameter is a final expression, then a new owner descriptor object (`$OD`) is created. If the parameter is a special owner, one of the singleton `$OD` objects is used. Lastly if the owner parameter is a formal parameter, then the owner descriptor is a reference into the `$OD` array that contains the `$OD` which was used to parameterize the current object.

To translate type casts and `instanceof` checks, the owner parameters in the object are compared to the statically bound owners. So for example, to do a downcast to a parameterized type, not only must the runtime Java type of the object be cast-able to the Java type in question, but the owner descriptors in the object's `$_ods` field must also match the statically bound owner descriptors. Similarly, when evaluating an `instanceof` expression, not only does the Java type have to be compared, but the owner descriptors in the target object need to be compared to the statically bound owner descriptors. Figure 4-3 shows a more detailed translation of some `TStack` client code.

While the transformations shown in Figures 4-2 and 4-3 seem unwieldy, they are all performed at the bytecode-level, so there are optimizations that our implementation makes that cannot be shown. For example, in translating the downcast of `obj` to `TStack<thisThread,`

```
01  TStack<thisThread, self> t0 = new TStack<thisThread, self>();
02  TStack<thisThread, thisThread> t1 = new TStack<thisThread, thisThread>();
03  TStack<self, thisThread> t2 = new TStack<self, thisThread>();
04  Object<thisThread> obj0
05  obj0 = t2
06  obj0 = t0
07  t0 = (TStack<thisThread, self>)obj
08  t0 = (TStack<thisThread, thisThread>)obj
09  boolean b = (obj instanceof TStack<thisThread, self>)
```

(a)

```
01  TStack t0 = new TStack(new $OD[]{$OD.THISTHREAD, $OD.SELF});
02  TStack t1 = new TStack(new $OD[]{$OD.THISTHREAD, $OD.THISTHREAD});
03  TStack t2 = new TStack(new $OD[]{$OD.SELF, $OD.THISTHREAD}});
04  Object obj0
05  obj0 = t2
06  obj0 = t0
07  try {
      if((obj instanceof TStack) &&
         (obj.$_ods[0].equals($OD.THISTHREAD) &&
         (obj.$_ods[1].equals($OD.SELF))
      { t0 = (TStack)obj; }
      else
      { throw new Exception();}
    } catch (Exception e) { throw new ClassCastException();}
08  try {
      if((obj instanceof TStack) &&
         (obj.$_ods[0].equals($OD.THISTHREAD) &&
         (obj.$_ods[1].equals($OD.THISTHREAD))
      { t0 = (TStack)obj; }
      else
      { throw new Exception();}
    } catch (Exception e) { throw new ClassCastException();}
09  boolean b;
    try {
      b = ((obj instanceof TStack) &&
           (obj.$_ods[0].equals($OD.THISTHREAD)) &&
           (obj.$_ods[1].equals($OD.SELF))) &&
    } catch (Exception e) { b = false;}
```

(b)

Figure 4-3: Example (a) `TStack` client code and (b) translation

`self>` in line 7 of Figure 4-3, the scope of the exception handler only encompasses the computation of the predicate expression of the `if` statement. Furthermore, if the `else` clause of the `if` statement is invoked, only the `ClassCastException` is thrown. The transformations shown were chosen because they are the most simply stated while being semantically equivalent to the implementation.

A more formal description of the transformations needed to build the runtime type system is given in Figure 4-5.

4.4 Specification classes

In order to allow the safe use of standard (or pre-compiled) Java classes, we introduce parameterized specification files. The specification files simply re-define signatures of class, method and field declarations using parameterized owner-types. By type-checking client code against these specifications but compiling against the pre-compiled binaries, we can, assuming the specifications are correct, still maintain a sound type system.

Furthermore, because the specification files are simply an interface specification, they can be extracted from PRFJ source code and then used in a later partial compilation where client code can be type-checked against the extracted specification files. This allows us to maintain the ability to do separate compilation.

An example specification for `java.util.Vector` is shown in figure 4-4. The method signatures given here have the same syntax and meaning as method signatures found in normal class definitions, except in this case, they are only used to type-check against. The specification files need to be generated by hand and there is currently no way to check the

```
public specification class Vector<vOwner, eOwner>
{
    public boolean add(Object<eOwner> o) requires(this) {}
    public void addElement(Object<eOwner> o) requires (this) {}
    public int capacity() requires (this) {}
    public void clear() requires (this) {}
    public Object<eOwner> elementAt(int i) requires(this) {}
    public java.util.Enumeration<thisThread, eOwner> elements() requires() {}
    public boolean equals(Object<any> o) requires(this, o) {}
    public boolean remove(Object<eOwner> obj) requires(this) {}
    ...
}
```

Figure 4-4: Example specification file for `java.util.Vector`

compiled binaries against the written specification.

4.5 Exceptions

Exceptions in Java are just like any other Object, except that they can be used with the `throw` operator. Accordingly, we type-check exceptions as we would any other object in the system except that we allow the `throw` and `try-catch` operators to operate on `Exception` objects that are owned by any owner. The reason is that both operators happen synchronously and neither operator can modify the `Exception` object. Therefore, throwing and catching exceptions will not lead to data races.

By default, in this system, exceptions are owned by `unique`. This is because in most programming idioms, exceptions are instantiated, initialized, thrown, caught and discarded. Exceptions are also sometimes explicity caught and rethrown, but still with a unique reference. There is generally not a need to create multiple references to exception objects. This heuristic simplifies the programmer's task of annotating class types.

4.6 Constructors

Constructors in Java require special handling. Because constructors instantiate new objects, the thread calling the constructor implicitly holds the lock on the new object being created. A call to a constructor can be viewed as two operations, allocating the object and initializing the object. In allocating the object, the calling thread has unique access. In order to ensure that the calling thread has unique access in the initialization portion of the constructor call, the body of the constructor must not escape the `this` pointer.

4.7 Java Threads

The original type-system for race-free concurrent Java presented a `fork()` construct for spawning and running new threads. In Java, new threads are created by calling the `start()` method on an object that subclasses `java.lang.Thread`. When a new thread is created and run, the type-checker needs to ensure that objects protected by the `thisThread` owner in the creating thread's context are not referenced by the new thread.

The way this is handled is by creating a new context in which to evaluate the body of the new thread. In that context, a fresh `otherThread` owner is substituted for the `thisThread` owner of objects which originated from other threads. Thus, any types protected by `thisThread` in the new thread's context will refer to the new thread, and there is no way for code in the new thread's context to reference the thread-local owner of the creating thread.

4.8 Interfaces

Interfaces in Java simply define a specification of method signatures and field declarations for which any implementing subtype must provide implementations [9]. Interface definitions, like class definitions, may be parameterized with formal parameters which can be used in the method signatures. On instantiation of an implementing subclass, the actual parameters in the class type are substituted into the method signatures to type check method calls on that object. For example, suppose you have an interface `A` and implementing subtype `B`:

```
interface A<owner1> {
  void foo(Object<owner1> obj);
}

class B<bOwner> implements A<bOwner> {
  void foo(Object<bOwner> obj) {
    ...
  }
}
```

Consider the following snippet:

```
A<thisThread> b = new B();
b.foo(new Object<thisThread>());
```

To type-check the method call `b.foo()`, the parameter of the receiver object (`thisThread`) is substituted for `bOwner` which is substituted for `owner1` in the interface definition. Other than an extra level of substitution, there is no other special handling required for Java interfaces.

$$\mathcal{T}\llbracket \mathit{ClassDecl} \rrbracket = \mathcal{T}\llbracket \texttt{class}\ \mathit{Identifier}\langle f_1 \ldots f_n \rangle \{\mathit{fields\ methods\ constructors}\} \rrbracket =$$
$$\texttt{class}\ \mathcal{T}\llbracket \mathit{Identifier} \rrbracket\{\ \texttt{public}\ \$\texttt{OD}[]\ \texttt{\_ods};\ \mathcal{T}\llbracket \mathit{fields} \rrbracket\ \mathcal{T}\llbracket \mathit{methods} \rrbracket\ \mathcal{T}\llbracket \mathit{constructors} \rrbracket\}$$
$$\text{where } \mathit{DeclaresParameters}(\mathit{ClassDecl})$$

$$\mathcal{T}\llbracket \mathit{ClassDecl} \rrbracket = \mathcal{T}\llbracket \texttt{class}\ \mathit{Identifier}\langle f_1 \ldots f_n \rangle \{\mathit{fields\ methods\ constructors}\} \rrbracket =$$
$$\texttt{class}\ \mathcal{T}\llbracket \mathit{Identifier} \rrbracket\{\ \mathcal{T}\llbracket \mathit{fields} \rrbracket\ \mathcal{T}\llbracket \mathit{methods} \rrbracket\ \mathcal{T}\llbracket \mathit{constructors} \rrbracket\}$$
$$\text{where } \neg \mathit{DeclaresParameters}(\mathit{ClassDecl})$$

$$\mathcal{T}\llbracket \mathit{Identifier} \rrbracket = \mathit{Identifier}$$

$$\mathcal{T}\llbracket \mathit{Field} \rrbracket = \mathcal{T}\llbracket \mathit{TypeName} \rrbracket\ \mathcal{T}\llbracket \mathit{Identifier} \rrbracket\ \mathcal{T}\llbracket \mathit{VariableInitializer} \rrbracket$$

$$\mathcal{T}\llbracket \mathit{TypeName} \rrbracket = \mathcal{T}\llbracket \mathit{Identifier} \rrbracket$$

$$\mathcal{T}\llbracket \mathit{Constructor} \rrbracket = \mathcal{T}\llbracket \mathit{Identifier} \rrbracket\ (\$\texttt{OD}[]\texttt{ods},\ \mathcal{T}\llbracket \mathit{FormalParameters} \rrbracket)\ \{\ \texttt{\_ods} = \texttt{ods};\ \mathcal{T}\llbracket \mathit{MethodBody} \rrbracket\ \}$$
$$\text{where } \mathit{DeclaresParameters}(\mathit{ClassOwner}(\mathit{Constructor}))$$

$$\mathcal{T}\llbracket \mathit{Constructor} \rrbracket = \mathcal{T}\llbracket \mathit{Identifier} \rrbracket\ (\mathcal{T}\llbracket \mathit{FormalParameters} \rrbracket)\ \{\ \mathcal{T}\llbracket \mathit{MethodBody} \rrbracket\ \}$$
$$\text{where } \neg \mathit{DeclaresParameters}(\mathit{ClassOwner}(\mathit{Constructor}))$$

$$\mathcal{T}\llbracket \mathit{Method} \rrbracket = \mathcal{T}\llbracket \mathit{TypeName} \rrbracket\ \mathcal{T}\llbracket \mathit{Identifier} \rrbracket\ (\$\texttt{OD}[]\texttt{ods},\ \mathcal{T}\llbracket \mathit{FormalParameters} \rrbracket)\ \{\ \mathcal{T}\llbracket \mathit{MethodBody} \rrbracket\}$$
$$\text{where } \mathit{DeclaresMethLevParams}(\mathit{Method})$$

$$\mathcal{T}\llbracket \mathit{Method} \rrbracket = \mathcal{T}\llbracket \mathit{TypeName} \rrbracket\ \mathcal{T}\llbracket \mathit{Identifier} \rrbracket\ (\mathcal{T}\llbracket \mathit{FormalParameters} \rrbracket)\ \{\ \mathcal{T}\llbracket \mathit{MethodBody} \rrbracket\}$$
$$\text{where } \neg \mathit{DeclaresMethLevParams}(\mathit{Method})$$

$$\mathcal{T}\llbracket \texttt{new}\ \mathit{ClassName}\langle o_1 \ldots o_n \rangle(\mathit{Arguments}); \rrbracket = \texttt{new}\ \mathit{ClassName}(\mathcal{T}\llbracket o_1 \ldots o_n \rrbracket,\ \mathcal{T}\llbracket \mathit{Arguments} \rrbracket);$$
$$\text{where } \mathit{DeclaresParameters}(\mathit{Declaration}(\mathit{ClassName}))$$

$$\mathcal{T}\llbracket \mathit{MethodCall} \rrbracket = \mathcal{T}\llbracket \mathit{Expression}(\mathit{Arguments}) \rrbracket = \mathcal{T}\llbracket \mathit{Expression} \rrbracket(\mathcal{T}\llbracket \mathit{ListOfMethLevParams}(m) \rrbracket,$$
$$\mathcal{T}\llbracket \mathit{Arguments} \rrbracket)\ \text{where } \mathit{Expression} \text{ binds to } m \in \mathit{Method}\ \&\&\ \mathit{DeclaresMethLevelParams}(m)$$

$$\mathcal{T}\llbracket \mathit{Owner}_1 \ldots \mathit{Owner}_n \rrbracket = \texttt{new}\ \$\texttt{OD}[n]\{\mathcal{T}\llbracket \mathit{Owner}_1 \rrbracket, \ldots, \mathcal{T}\llbracket \mathit{Owner}_n \rrbracket\}$$

$$\mathcal{T}\llbracket \mathit{Owner} \rrbracket = \texttt{ods}[\mathit{MethLevParamIndex}(\mathit{MethodOwner}(\mathit{Owner}), \mathit{Owner})]$$
$$\text{where } \mathit{DeclaresMethLevParam}(\mathit{MethodOwner}(\mathit{Owner}), \mathit{Owner})$$

$$\mathcal{T}\llbracket \mathit{Owner} \rrbracket = \texttt{ods}[\mathit{ParameterIndex}(\mathit{ClassOwner}(\mathit{MethodOwner}(\mathit{Owner})), \mathit{Owner})]$$
$$\text{where } \mathit{DeclaresParameter}(\mathit{ClassOwner}(\mathit{MethodOwner}(\mathit{Owner})), \mathit{Owner})$$

DeclaresParameters : *ClassDecl* → *Boolean*; Returns `true` iff *ClassDecl* declares class-level formal parameters
DeclaresParameter : *ClassDecl* → *Owner* → *Boolean*; Returns `true` iff *ClassDecl* defines *Owner* as a class-level formal parameter
ParameterIndex : *ClassDecl* → *Owner* → *Integer*; Returns the integer index of *Owner* in *ClassDecl*'s parameter list
ListOfMethLevParams : *Method* → (`listof` *Owner*); Returns the list of method-level formal parameters that are declared
DeclaresMethLevParams : *Method* → *Boolean*; Returns `true` iff *Method* declares method-level formal parameters
DeclaresMethLevParam : *Method* → *Owner* → *Boolean*; Returns `true` iff *Method* defines *Owner* as a method-level parameter.
MethLevParamIndex : *Method* → *Owner* → *Integer*; Returns the integer index of *Owner* in *Method*'s parameter list
ClassOwner : *MemberDecl* → *ClassDecl*; Returns the *ClassDecl* which defines a given *MemberDecl*
MethodOwner : *Expression* → *Method*; Returns the *Method* which declares the *Expression*.
Declaration : *ClassName* → *ClassDecl*; Returns the *ClassDecl* corresponding to a given *ClassName*

Figure 4-5: Translation Function to standard Java

```
$\mathcal{T}[\![$freshvar =$(TypeName)Expression]\!] =$ try { if(($\mathcal{T}[\![Expression]\!]$ instanceof $\mathcal{T}[\![TypeName]\!]$) &&
                                                   ($\mathcal{T}[\![Expression]\!]$._ods[0].equals($\mathcal{T}[\![o_1]\!]$)) &&
                                                   ($\mathcal{T}[\![Expression]\!]$._ods[1].equals($\mathcal{T}[\![o_2]\!]$)) &&
                                                   ...
                                                   ($\mathcal{T}[\![Expression]\!]$._ods[n − 1].equals($\mathcal{T}[\![o_n]\!]$)))
                                                   { freshVar = ($\mathcal{T}[\![TypeName]\!]$)Expression; }
                                               else { throw new Exception(); }
                                           catch(Exception ex) { throw new ClassCastException(); }

$\mathcal{T}[\![$freshBool =($Expression$ instanceof $TypeName$)$]\!] =$
                                       try {
                                               freshBool = (($\mathcal{T}[\![Expression]\!]$ instanceof $\mathcal{T}[\![TypeName]\!]$) &&
                                               ($\mathcal{T}[\![Expression]\!]$._ods[0].equals($\mathcal{T}[\![o_1]\!]$)) &&
                                               ($\mathcal{T}[\![Expression]\!]$._ods[1].equals($\mathcal{T}[\![o_2]\!]$)) &&
                                               ...
                                               ($\mathcal{T}[\![Expression]\!]$._ods[n − 1].equals($\mathcal{T}[\![o_n]\!]$))) }
                                       catch(Exception ex) { freshBool = false; }


$\mathcal{T}[\![$self$]\!] =$ $OD.SELF
$\mathcal{T}[\![$readonly$]\!] =$ $OD.READONLY
$\mathcal{T}[\![$thisThread$]\!] =$ $OD.THISTHREAD
$\mathcal{T}[\![$unique$]\!] =$ $OD.UNIQUE
```

Figure 4-6: Translation Function continued

Chapter 5

The Formal Type System

This chapter presents the type system described in Section 3.1 which was largely built on the type system presented by Boyapati and Rinard [1]. The grammar for the type system was shown in the beginning of Section 3.1. While the full type system appears in Appendix A, this chapter will highlight and discuss some of the more interesting and pertinent rules.

We first define a number of predicates used in the type system informally. These predicates are based on similar predicates from [17] and [6]. We refer the reader to those papers for their precise formulation.

For a program P,

- *ClassOnce(P)* - No class is declared twice in P
- *WFClasses(P)* - There are no cycles in the class hierarchy
- *FieldsOnce(P)* - No class contains two fields with the same name, either declared or inherited
- *MethodsOncePerClass(P)* - No method name appears more than once per class
- *OverridesOK(P)* - Overriding methods have the same return type and parameter types as the methods being overridden. The requires clause of the overriding method must be the same or a subset of the requires clause of the methods being overridden
- *RO(e)* - The root owner of the final expression e

A typing environment E is a mapping of variables and fields to well-formed types and formal owners and can be described as:

$E ::= \emptyset \mid E, [\text{final}]_{opt}\ t\ x \mid\ E, \text{owner}_{formal}\ f$

A lock set ls is the set of all locks which are statically determined to be held at a program point. It can be described as:

$ls ::= \text{thisThread} \mid ls, e_{final} \mid ls, \text{RO}(e_{final})$

The type system is defined using the following judgments. The typing rules for these judgments can be found with the full set of typing rules in appendix A.

Judgment	Meaning
$P \vdash defn$	$defn$ is a well-formed class definition
$P; E \vdash wf$	E is a well-formed typing environment
$P; E \vdash meth$	$meth$ is a well-formed method
$P; E \vdash field$	$field$ is a well-formed field
$P; E \vdash t$	t is a well-formed type
$P; E \vdash t_1 \sqsubseteq t_2$	t_1 is a subtype of t_2
$P; E \vdash field \in cn\langle f_{1..n}\rangle$	class cn with formal parameters $f_{1..n}$ declares/inherits $field$
$P; E \vdash meth \in cn\langle f_{1..n}\rangle$	class cn with formal parameters $f_{1..n}$ declares/inherits $meth$
$P; E \vdash_{\text{final}} e : t$	e is a final expression with type t
$P; E \vdash_{\text{owner}} o$	o can be an owner
$P; E \vdash \text{R}ootOwner(e) = r$	r is the root owner of the final expression e
$P; E \vdash e : t$	expression e has type t, provided we have all the necessary locks
$P; E; ls \vdash e : t$	expression e has type t
$P; E \vdash e : t_1 \vert t_2$	expression e has type either t_1 or t_2, provided we have all the necessary locks
$P; E; ls \vdash e : t_1 \vert t_2$	expression e has type either t_1 or t_2

5.1 Type checking references

[EXP REF]

$$\frac{\begin{array}{c} P;\ E;\ ls \vdash e : cn\langle o_{1..n}\rangle \\ P;\ E \vdash ([\text{final}]_{opt}\ t\ fd) \in cn\langle f_{1..n}\rangle \\ P;\ E \vdash \text{RootOwner}(e) = r \quad\quad r \in ls \end{array}}{P;\ E;\ ls \vdash e.fd : t[e/\text{this}][o_1/f_1]..[o_n/f_n]}$$

To type-check an object-field reference expression $e.fd$, the type-rule ensures that e is a well-formed expression of some class type $cn\langle o_{1..n}\rangle$. The rule also requires that the class $cn\langle o_{1..n}\rangle$ declares a non-static field fd of some type t.

In order for the field access to be thread-safe, the rule must ensure that the thread performing the field access holds the lock on the root owner of the expression e. Otherwise, some intervening thread might change the value of $e.fd$ prior to the access.

5.2 Type checking static references

[EXP STATIC REF]

$$\frac{\begin{array}{c} P;\ E \vdash cn\langle f_{1..n}\rangle \\ P;\ E \vdash (\text{static}\ t\ fd) \in cn\langle f'_{1..m}\rangle \\ P;\ E \vdash \text{cn.class} \in ls \end{array}}{P;\ E;\ ls \vdash cn.fd : t}$$

[FINAL CLASS OBJECT]

$$\frac{P;\ E \vdash cn\langle f_{1..n}\rangle}{P;\ E \vdash_{final} \text{cn.class} : \text{java.lang.Class}}$$

The type-rule for static field references is similar to the one that proves non-static field references. First, the rule must prove that there is some well-formed class $cn\langle f_{1..n}\rangle$ and that cn declares a static field $t\ fd$. In Chapter 4.2, we discussed how this type system protects static field accesses with the corresponding `java.lang.Class` object. The type-rule summarizes this by requiring that `cn.class` be held in the lockset ls.

5.3 Type checking method calls

[EXP INVOKE]

$$\frac{\begin{array}{c} P;\ E;\ ls \vdash e : cn\langle o_{1..n}\rangle \\ P;\ E \vdash (t\ mn(t_j\ y_j\ ^{j\in 1..k})\ \mathsf{requires}(e'_{1..m})...)\in cn\langle f_{1..n}\rangle \\ P;\ E;\ ls \vdash e_j : t_j[e/\mathrm{this}][o_1/f_1]..[o_n/f_n] = cn'\langle o_{j_{1..n}}\rangle \\ o_{j_1} \neq \mathsf{unique,\ readonly} \\ P;\ E \vdash \mathrm{RootOwner}(e'_i[e/\mathrm{this}][o_1/f_1]..[o_n/f_n]) = r'_i \\ r'_i \in ls \end{array}}{P;\ E;\ ls \vdash e.mn(e_{1..k}) \colon t[e/\mathrm{this}][o_1/f_1]..[o_n/f_n]}$$

[EXP STATIC INVOKE]

$$\frac{\begin{array}{c} P;\ E \vdash_{\mathrm{static}} (t\ mn(t_j\ y_j\ ^{j\in 1..k})\ \mathsf{requires}(e'_{1..m})...)\in cn\langle f_{1..n}\rangle \\ P;\ E;\ ls \vdash e_j : t_j = cn'\langle o_{j_{1..n}}\rangle \\ o_{j_1} \neq \mathsf{unique,\ readonly} \\ P;\ E \vdash \mathrm{RootOwner}(e_i) = r_i \\ r_i \in ls \end{array}}{P;\ E;\ ls \vdash cn.mn(e_{1..k}) \colon t}$$

The type rules to type-check method invocations are shown above. In order to check a non-static method invocation, we have to check that the receiver expression is of the correct type. Next, we must check that the calling thread holds all of the root owner locks that are listed in the requires clause of the method declaration. We must also check to see that the number of arguments and the argument types match the declaration. The argument and return types that are declared by the method must be renamed when checking the callsite. The owner parameters of the receiver are substituted for the formal class parameters and the receiver expression is substituted for this.

To type check static declarations, the rule is similar except that there is no receiver object. Accordingly, the rule checks that the class declares a static method and that the number of arguments and the types of the arguments and return value match the calling context. The rule also requires the calling thread to hold the locks on the root owners of the expressions in the requires clause of the method.

5.4 Type checking synchronized blocks

[EXP SYNC]

$$\frac{P;\ E \vdash_{final} e_1 : t_1 \qquad P;\ E;\ ls,\ e_1 \vdash e_2 : t_2}{P;\ E;\ ls \vdash \textsf{synchronized}\ e_1\ \{\ e_2\ \} : t_2}$$

The synchronized keyword is used to acquire a single lock for the scope of the associated block. Here, the type rule says that the object being synchronized on must be a final field or variable of a reference type. The synchronized block is then well-typed if, by adding the synchronized final expression e_1 to the lockset ls, we can type-check the body of the synchronized block.

The reference that is being synchronized on must be final because if it changes, then at some points in the body of the synchronized block the root owner locks may not be held anymore. There are additional rules that establish that the this reference and class-expression such as cn.class are final expressions and so they may be synchronized on.

5.5 Type checking fork

[EXP FORK]

$$\frac{P;\ E;\ ls \vdash e : cn\langle o_{1..n}\rangle \qquad P;\ E;\ ls \vdash cn\langle o_{1..n}\rangle \sqsubseteq \text{java.lang.Thread} \qquad P;\ E[\text{otherThread}/\text{thisThread}];\ \text{thisThread} \vdash (\text{void}\ run()\ \textsf{requires}(\text{this})\ \ldots\ ) \in cn\langle f_{1..n}\rangle}{P;\ E;\ ls \vdash \text{e.start():void}}$$

The type rule for starting a new thread is shown above. In Java, a new thread starts executing when the start() method is called on a subclass of java.lang.Thread. The rule checks that the receiver of the invocation is a well-defined subclass of Thread. However, the environment may contain some types which have the thisThread owner. Because the thisThread owner in those types refers to the initial thread, those owners must be renamed in order to type-check the body of the new thread. This is shown by renaming thisThread to otherThread in the environment E and using that to type-check the body of the new thread.

5.6 Type checking unique assignments

[EXP UNIQUE ASSIGN]

$$\frac{\begin{array}{c} P;\ E;\ ls \vdash e : cn\langle o_{1..n}\rangle \\ P;\ E \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \\ P;\ E \vdash \text{RootOwner}(e) = r \quad r \in ls \\ P;\ E;\ ls \vdash e' : t[e/\text{this}][o_1/f_1]..[o_n/f_n] \\ P;\ E;\ ls \vdash \text{t} = \text{cn'}\langle o'_{1..n}\rangle \quad o'_1 = \mathsf{unique} \end{array}}{P;\ E;\ ls \vdash e.fd = e'\texttt{--} : t[e/\text{this}][o_1/f_1]..[o_n/f_n]}$$

[ROOTOWNER UNIQUE/READONLY]

$$\frac{P;\ E \vdash e : \mathsf{Object}\langle unique\rangle \mid \mathsf{Object}\langle readonly\rangle}{P;\ E \vdash \text{RootOwner}(e) = \emptyset}$$

Objects that are owned by the unique owner may not be assigned to fields and variables directly. The type-rule shown above shows how a unique object can be safely stored. This rule is similar to the normal, non-unique, assignment expression rule which can be found in Appendix A. First, the left-hand-side and right-hand-side types are computed with the proper name substitutions (e for this and the actual owner parameters for the formal class parameters). As with the rule for a field reference, we must check to see that the root owner of the receiver expression is in the lockset ls.

The rule also checks to see that the object being assigned is owned by unique. The rule then proves that the expression e'-- can be assigned to the field $e.fd$. Remember that this is equivalent to assigning the field reference and atomically setting the old reference, e--, to null.

5.7 Type checking unique and readonly in method calls

[EXP INVOKE UNIQUE/READONLY]

$$\frac{\begin{array}{c} P;\ E;\ ls \vdash e : cn\langle o_{1..n}\rangle \\ P;\ E \vdash (t\ mn(t_j\ y_j\ ^{j\in 1..k})\ \textsf{requires}(e'_{1..m})...)\in cn\langle f_{1..n}\rangle \\ P;\ E;\ ls \vdash e_j : t_j[e/\text{this}][o_1/f_1]..[o_n/f_n] = cn'\langle o_{j_{1..n}}\rangle \\ (o_{j_1} = \textsf{unique})\ \&\&\ (e_j \neq e'_j\texttt{--}) \Rightarrow y_j!e \\ o_{j_1} = \textsf{readonly} \Rightarrow y_j!w \\ P;\ E \vdash \text{RootOwner}(e'_i[e/\text{this}][o_1/f_1]..[o_n/f_n]) = r'_i \\ r'_i \in ls \end{array}}{P;\ E;\ ls \vdash e.mn(e_{1..k}):\ t[e/\text{this}][o_1/f_1]..[o_n/f_n]}$$

[EXP STATIC INVOKE UNIQUE/READONLY]

$$\frac{\begin{array}{c} P;\ E \vdash_{\text{static}} (t\ mn(t_j\ y_j\ ^{j\in 1..k})\ \textsf{requires}(e'_{1..m})...)\in cn\langle f_{1..n}\rangle \\ P;\ E;\ ls \vdash e_j : t_j = cn'\langle o_{j_{1..n}}\rangle \\ (o_{j_1} = \textsf{unique})\ \&\&\ (e_j \neq e'_j\texttt{--}) \Rightarrow y_j!e \\ o_{j_1} = \textsf{readonly} \Rightarrow y_j!w \\ P;\ E \vdash \text{RootOwner}(e_i) = r_i \\ r_i \in ls \end{array}}{P;\ E;\ ls \vdash cn.mn(e_{1..k}):\ t}$$

As mentioned in Chapter 3.3.3, arguments to method calls which are owner by the unique owner must be treated with care. This is to prevent the accidental escape of a reference to a unique object. The way that we ensure this is by checking that arguments in method invocations that are owned by the unique owner are passed to the method safely. Safely passing the argument means that either the argument is "dereferenced" with the e-- form, or that the corresponding method parameter is annotated with the `!e` modifier. Recall that the `!e` modifier requires that the method be checked to ensure that it will not escape that parameter.

Similarly, readonly method arguments are checked to see that the corresponding method parameters are annotated with the `!w` modifier signifying that the argument will not be written to.

5.8 Type checking constructor calls

[EXP NEW]

$$\frac{\begin{array}{c} P;\,E \vdash cn\langle f_{1..n}\rangle \\ P;\,E \vdash (cn(t_j\ y_j\ {}^{j\in 1..k})\ \mathsf{requires}\ (e'_{1..p})) \in cn\langle f_{1..n}\rangle \\ P;\,E;\,ls \vdash e_j : t_j[o_1/f_1]..[o_n/f_n] = cn'\langle o_{j_{1..n}}\rangle \\ o_{j_1} \neq \mathsf{this} \\ (o_{j_1} = \mathsf{unique})\ \&\&\ (e_j \neq e'_j\texttt{--}) \Rightarrow y_j!e \\ o_{j_1} = \mathsf{readonly} \Rightarrow y_j!w \\ P;\,E \vdash \mathrm{RootOwner}(e'_i[o_1/f_1]..[o_n/f_n]) = r'_i \quad r'_i \in ls \end{array}}{P;\,E;\,ls \vdash \mathsf{new}\ cn\langle o_{1..n}\rangle(e_{1..m}) : cn\langle o_{1..n}\rangle}$$

Type checking a constructor call is similar to checking a method invocation. First, the rule checks to see that there is a well-typed constructor declared by the class in question. Then the arguments are type-checked replacing the actual owner parameters supplied to the constructor call. In this case, none of the argument types can declare that they are owned by this as the this reference will not be valid until after the constructor executes. The rule also checks to see that the proper locks are held on the expressions in the requires clause.

Chapter 6

Type Inference

To convert a pure Java program into race-free Java, the programmer must annotate the types of fields and variables in the program with the correct owner parameters. In a large system, this requires changing many files and understanding the implicit protection mechanisms that are used in the original source.

In order to ease the burden of adding the extra type-annotations on the programmer, the compiler supports local-variable type-inference. Programmers need only fully parameterize the types that appear in method or constructor signatures as well as field declarations. From that, it is possible to infer the parameterized types of local variables.

6.1 Inference

The inference algorithm first starts by assigning default owners to all unparameterized types. For example, when the algorithm runs on the code in Figure 6-1, it generates the following assignments:

```
00  class A<owner1, owner2>
01  {
02      Object<owner2> foo(Object<owner2> obj)
03      {
04          Object o = obj;
05          Vector v = new Vector();
06          v.addElement(o);
07          return v.remove(0);
08      }
09
10      void bar()
11      {
12          A a = new A();
13          o = a.foo();
14      }
15  }
```

```
00  class A<owner1, owner2>
01  {
02      Object<owner2> foo(Object<owner2> obj)
03      {
04          Object<owner2> o = obj;
05          Vector<?o2, owner2> v =
                new Vector<?o2, owner2>();
06          v.addElement(o);
07          return v.remove(0);
08      }
09
10      void bar()
11      {
12          A<?o4, self> a = new A<?o4, self>();
13          o = a.foo();
14      }
15  }
```

Figure 6-1: Incompletely typed code before and after inference

```
04          Object<?o1> o = obj;
05          Vector<?o2, ?o3> v = new Vector<?o2, ?o3>();
11          A<?o4, ?o5> a = new A<?o4, ?o5>();
```

Each type is augmented with the number of default owners as its type declaration declares. For example, in line 11, the class `A` defines two owner parameters, so the type of the local variable `a` is assigned two fresh default owners, `?o4` and `?o5`. The default owners represent type-variables in the parameterized class types.

The next step is to collect all of the constraints between the default owner assignments. For example, the assignment of local variable `o` to method parameter `obj` in line 4 adds the constraint that `?o1` = `owner2`. The full set of constraints generated are:

```
04          ?o1 = owner2;
06          ?o3 = ?o1;
07          ?o3 = owner2;
13          ?o5 = self;
```

Once the constraints are collected, they are unified using the standard Union-find algorithm in psuedo-linear time [14]. If there are conflicts in unifying constraints, then this results in a type-error. This is because there is no possible assignment of owner parameters that would make the conflicting expressions type check together.

While multiple conflicting constraints on the same default owner are not allowed, it is permissible (and expected) that multiple consistent constraints may be found for a given default owner. For example, in the above code, `?o3` is equal to `?o1` and to `owner2`. This is allowed, because the constraint from line 4 is consistent with these constraints.

When there are not enough constraints to fully determine default owners, these default owners are replaced with the `thisThread` special owner. This is because if there are no constraints relating the default owner in question, then it is safe to assign it to be the thread-local owner.

Chapter 7

Compiler Structure

The race-free Java compiler that we have implemented is structured as shown in Figure 7-1. This chapter will give a high-level overview of each stage of the compilation process and how it was implemented. The compiler is built using the Kopi Java compiler framework [13].

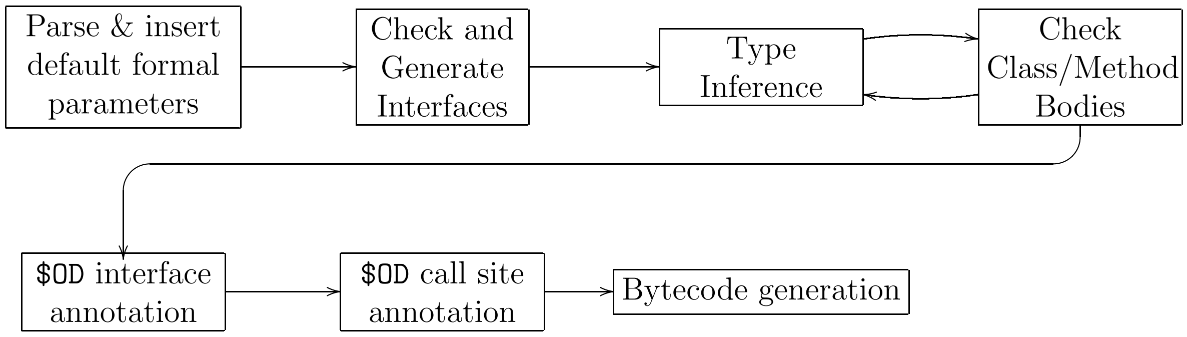
Parse & insert default formal parameters
Check and Generate Interfaces
Type Inference
Check Class/Method Bodies
$OD interface annotation
$OD call site annotation
Bytecode generation

Figure 7-1: High-level compiler dataflow

7.1 Parser

The Kopi parser was modified to handle the new extended grammar which was presented in Figure 3-2. This stage only enforces that source files conform the grammar and performs no checks for semantic correctness. The parser stage is also responsible for inserting default formal parameters for class declarations and method signatures that are missing parameters.

7.2 Check and Generate Interfaces

This stage of the compiler generates interfaces for class and interface definitions and for field and method signatures. At this stage, interfaces of different modules are checked against one another to make sure that they are semantically correct. For example, at this stage, class definitions are verified to check that they correctly declare superclasses and interfaces with the formal parameters. Method and field declarations are also checked against a variety of semantic rules. In addition to performing standard Java checks such as checking overriding and overloading, methods and fields are also checked to see that they make correct use of the new parameterized types.

The interfaces exported by this stage of compilation contain a variety of "substitution tables" which define what formal parameters were declared and how they should be substituted for in the code within the corresponding body. For example, consider the class definition below.

```
class A<owner1, owner2> extends B<owner1, owner2, owner2>
                        implements C<owner1, owner2>, D<owner1, self>
{...}
```

```
class B<bo1, bo2, bo3> {...}
interface C<co1, co2> {...}
interface D<do1, do2> {...}
```

The class declaration interface exported for class `A` would define an exported interface which would contain a tables to map superclass/interface contexts to substitutions: in the context of superclass `B`, the actual parameter value of `owner1` should be substituted for the formal parameter `bo1` and `owner2` for `bo2` and `bo3`. Similarly, in the contexts of the the interfaces `C` and `D`, the exported interface defines substitutions for the appropriate parameter values.

For method definitions, this stage of the compiler also determines which, if any, of the formal parameters declared in the class-types of its arguments are method-level parameters. This can only be done after the interface for the enclosing class definition is solidified. The set of method-level formal parameters is reflected in the method's exported interface because there is further type-checking that must be conducted at each of the method's call-sites.

7.3 Type Inference and Body Checking

After generating interfaces, the next stage in compilation is to infer the types of unannotated local variables and to type-check all of the class and method bodies. The type-inference process is by nature somewhat intermingled with the type-checking process. The basic process is that when an unannotated type is encountered, it is assigned some fresh parameters. The corresponding expression can then be used in the remainder of the type-checking process.

The problem is that given an expression, in order to retrieve the type of that expression and discover that it is unannotated, you must first attempt to type-check it. The first pass of the type-checker will yield the unannotated type but not perform any real semantic checking. A second, later pass, will enforce the semantic rules once the proper type of the expression has been inferred. For example, consider the following code using the `TStack` class given in figure 3-4.

```
TNode<thisThread> myNode = new TNode<thisThread>();
TStack<thisThread> localTStack = new TStack<thisThread>();
TStack myStack = new TStack();

myStack.push(myNode);
...
localTStack = myStack;
```

The expression to be type-checked is the method call to `push()` on the `myStack` object. In the first pass, `myStack` is not completely typed. Therefore, the only useful information that is yielded in the first pass is that the `myStack` object has a type `TStack<?x1, thisThread>`. It is not until later, when the assignment expression `localTStack = myStack;` is type-checked that the local-inference constraints for `myStack` are solved and it is given its full type of `TStack<thisThread, thisThread>`. Now, in the second pass, the compiler can check all of the applicable semantic rules, such as verifying that the necessary locks are held, which would not have been possible in the first pass.

7.4 Owner Descriptor Interface Transformation

During this stage, the compiler modifies all of the class and method interfaces that must be updated to support owner-passing as described in Section 4.3 and in Figure 4-5. For classes that declare formal parameters which must be passed in at runtime, an additional instance field `$_ods` of type `$OD[]` is inserted into the class definition. Furthermore, all constructors must be modified to add an additional `$OD[]` parameter and code to store the array parameter into the corresponding field.

Similarly methods which define method-level formal parameters must have an additional `$OD[]` parameter. Additional code to store this parameter is not necessary for method-level parameters because the owner descriptors only need to stay in scope for the extent of the method body.

7.5 Owner Descriptor Body Transformation

After the class and method interfaces have been modified to incorporate owner descriptors, there are several transformations that must be made to the body code. First of all, method and constructor call sites that are bound to methods or constructors that have had their signatures altered must also be modified to pass the new `$OD[]` parameter. The parameter to be passed is synthesized at call-site based on the source parameterizations. For example, suppose that we have the following constructor call to class `A` which requires owner passing:

```
new A<owner1, owner2, self>();
```

Statically, it is known where parameters `owner1` and `owner2` are declared. Suppose that `owner1` is the first class-level formal parameter and `owner2` is the second method-level parameter. The owner descriptor array to be passed into the `A` constructor call then consists of three expressions, one for each of the parameters. The first expression would be `this.$_ods[0]` to reference the first class-level formal parameter. The next expression would be `ods[1]` where `ods` corresponds to the method-level parameter in the call-site's scope. The last expression for the `self` parameter is a static field reference into the `$OD` class.

The intermediate representation (IR) is modified to insert a new expression which corresponds to the new array. The contents of the array are statically bound to either references into the current object's `$_ods` field (if it has one), the current method's `$OD[]` parameter (if it has one), or one of the singleton `$OD` objects for special owners. The compile-time type of the expression being transformed determines which `$OD` expressions comprise the owner descriptor array expression.

Dynamic downcast and `instanceof` expressions also undergo a transformation at this stage. The expressions are modified to not only check that the Java-type specified in the expression is a supertype of the runtime Java-type of the object, but also that the runtime owner descriptors match the owner parameters in the static parameterized type.

Chapter 8

Experience

We used the compiler for the extended language to type check existing Java benchmarks which we translated into PRFJ. We also ported several single and multi-threaded Java benchmarks into our race-free extension.

In implementing these various benchmarks, we found that the PRFJ language extension is expressive enough to accommodate the commonly used protection mechanisms. Various language features such as type-inference, parameterized methods, and specification files significantly ease the burden on the programmer by reducing the number type annotations, allowing polymorphic methods, and allowing separate compilation. Table 8.1 shows the programming overhead in converting several of the benchmarks.

The `tsp` and `elevator` benchmarks were taken from benchmarks described in [18]. The `mst`, `bh`, `bisort`, `power`, `em3d`, and `voroni` programs were taken from the Java version of the Olden benchmarks.

We found that many of the necessary modifications could be categorized into several types

Benchmark	Description	LOC	Lines Changed
tsp	"Traveling salesman problem" solver	826	32
elevator	Elevator simulator	569	35
chat	Chat client/server program	542	53
mst	"Minimum Spanning Tree" solver	401	19
bh	Barnes-Hut benchmark	1301	103
bisort	Bitonic Sort benchmark	381	21
power	Parallel power pricing prog.	775	36
em3d	Emulation of 3d electromagnetic waves	454	27
voroni	Computes Voroni diagram for set of points	1003	73

Table 8.1: Programming Overhead

of changes. Because our inference algorithm only guarantees discovery of local variables, field declarations of reference (and array) types must still be fully parameterized. Also, reference and array types that appear in method signatures need to be fully specified.

Another major source of changes to the original Java source code is to handle static fields. Recall that because static variables can be referenced globally in the Java namespace, they must be locked on every access. In many cases, the static fields in question are merely there to act as a global constant-holder. In such cases, read only type-safe enumerations [16] could be used along with the `final` keyword to eliminate much of the unnecessary locking and code changes that results from using static members. In fact, in the `bh` benchmark, 36 of lines of code that needed to be changed were to synchronize on static fields.

Our experience shows that there is still an overhead in instrumenting existing Java code. The type system was designed to make the programmer's synchronization discipline explicit and enforcable. This makes it easy for programmers to write new race-free code because the only overhead that they incur is formalizing the locking discipline. On the other hand in order to, translate Java code into PRFJ, the programmer must first infer what locking discipline was used in the original code, and then formalize it. It is not always straightforward to discover the locking conventions because the issues are so subtle (this is the same reason that data-races are a problem to begin with.)

One example of the subtlety in determining these conventions is read only objects. In some cases, such as in the `elevator` benchmark, access to certain data structures are unsynchronized because the original programmer wrote the data-structure code to be immutable. In order to correctly translate this code into PRFJ, we must first examine the code for the data-structure to determine that it is in fact immutable before we can translate it.

We measured a negligible runtime performance slowdown with the benchmarks that we ran. Most of the code did not make much use of polymorphism and so the extent of the additions to the code to propagate runtime types was simply a matter of passing an extra parameter in places and an extra constructor call (to create the owner descriptors) in others.

The similar type-passing work in supporting generic types for Java [12] showed a minimal load-time performance hit by statically assigning the type descriptors to constructor call-sites and creating all possible type descriptors used in a class. Because the descriptors in the generics work represent types themselves (as opposed to owner expressions), they are able to reuse many of the descriptors. For example, each instantiation of Vector<String> would use the same String type descriptor which would be created once in the static class initializer. However, in our system, this load-time optimization would not work as well because the values being passed at runtime represent Objects, not types. Without being able to achieve the same reuse, this approach would suffer a large startup overhead from creating unnecessary descriptors.

Although the runtime cost of type-passing is low, it is possible to further improve upon this. Up until this point, programs in our system benefit from separate compilation. That is, it is possible to compile modules of a program separately in the presence of the appropriate interface declarations. With a whole program analysis, we could easily determine whether or not a particular type will be the subject of a runtime type operation, and consequently, whether type-passing is necessary.

Chapter 9

Related Work

There has been much research work in detecting and/or preventing data races in programs.

Several approaches such as the Extended Static Checker for Java (ESC/Java) and Warlock use programmer annotations to statically detect potential race conditions. While these tools are useful, they are not sound approaches. Particularly, they cannot certify that a program is free of race conditions and may report races that are not truly data races.

There have also been several dynamic approaches to data-race detection. The Eraser [10] system is the most notable of these approaches. Eraser used binary rewriting to examine code on the fly and to detect breakdowns in protection mechanisms. Running a multi-threaded program with Eraser entails a runtime overhead. Furthermore, Eraser is known to report both false positive and false negative results.

A newer dynamic approach to detecting data-races from IBM Research [15] boasts huge runtime savings as a result of aggressive dynamic and static optimizations. More importantly, their system guarantees that every possible race condition that is encountered is reported.

This is an important result, because this means that the exact timing behavior does not need to occur in order for it to be detected.

There has been some recent work in static type systems for multi-threaded OO programs. The most similar work is Race Free Java. In Race Free Java [6], types are annotated with parameters which specify locks that are used to guard data members. Like our system, these parameterizations can be used to specify that a single lock should guard an entire data structure. The Race Free Java system also supports thread-local classes, but does not support unique or read-only objects. With Race Free Java, programmers are able to specify a finer granularity of locking policy than in our system.

However, Race Free Java does not allow the programmer to write code independently of the protection mechanism that is to be used to guard it. In our system, the programmer can write generic code for a data structure once, and specify at *object-creation* time what protection mechanism should be used on a per-object basis. So, for example, different *Queue* objects that are thread-local, self-synchronized, part of a larger data-structure, contain thread-local elements, contain self-synchronized elements, etc. can all be instantiated from the same implementation.

Another static approach to preventing data races is the Guava [3] type system. In the Guava approach, the type-system consists of three orthogonal hierarchies, *Monitors*, *Objects*, and *Values.* Monitors are instances that are shared between threads and must be synchronized on similar to the root objects of ownership trees in our system. Objects in Guava are similar to thread-local objects or field objects owned by the containing object in our system. Values are similar to unique objects in our system. The main difference between our system and Guava is that like Race Free Java, Guava does not allow the programmer to write generic object code and defer specifying the protection mechanism until object-creation.

9.1 Extensions

We recently extended our race-free Java system to also statically prevent deadlocks. This section will give a brief overview of the deadlock preventing extension. The full detail of this extension is beyond the scope of this thesis but can be found in [2].

A deadlock is a condition in a multi-threaded programs when two or more threads in the program "halt" because they are each waiting to acquire locks which the other threads hold. The canonical example of this is the dining philosopher's problem. In this problem, a group of philosophers are seated at a circular table with one chopstick between them all. The premise is that a philosopher can pick up one chopstick at a time but requires both chopsticks in order to eat a bite. After taking a bite, he then returns both chopsticks to where he found them.

The classic problem is that if each philosopher reaches to his right and picks up a chopstick, then reaches to his left, the entire meal will halt, because each philosopher will find themselves with one chopstick trying to find another. The analogy is directly applicable to concurrent programs in which multiple locks are acquired to process some transaction and then released.

The solution to the dining problem is to number the chopsticks and instruct each dinner attendee to pick up the chopsticks in descending order. Similarly, the deadlock extension requires that there be an ordering among locks and that the order in which locks are acquired is enforced.

In the extension, programmers specify a partial ordering relationship between locks. Once there is an ordering relationship between various locks, the type-checker statically verifies that these locks are only acquired in decreasing order of strength. More specifically, in order to acquire a lock, the new lock must be strictly weaker than all of the locks currently held. The reason that the type-system only requires a partial order and not a total order is that only locks that will simultaneously be held need to have an ordering because they will not cause a deadlock.

In order to type-check method calls, each method must declare in its signature a list of locks that the method could potentially acquire. From the call-site, the type-checker verifies that the weakest locks in the lockset are stronger than the least upper bound of the list of locks that is declared to be locked by the method. What this ensures is that no matter what lock is acquired in the new method, it will be weaker than any lock that is held at the call-site.

Chapter 10

Conclusion

Reliable thread-safe concurrent programming has always been difficult to achieve. Multi-threaded programs are difficult enough for humans to reason about, much less program consistently. The type-system behind Parameterized Race-Free Java offers a sound and expressive way to develop safe multi-threaded programs. By formalizing the implicit protection mechanisms that programmers have in mind while writing concurrent code, we are able to use the type-checker to catch these programmer errors before they become bugs, as well as allow better documentation of locking disciplines.

By implementing the type-system and compiler, we were able to demonstrate that converting programs from Java into the extended language is a tractable and manageable task. Furthermore, with the assistance of type-inference, the overhead in writing programs in this language is further lessened. This language extension gives programmers a powerful tool to prevent race-conditions and is expressive enough to allow the programmer to write generic code independently of locking discipline while still producing sound, data-race free code.

Appendix A

Formal Type Rules

$\boxed{\vdash P : t}$

[PROG]

$$\frac{\begin{array}{c} \mathit{ClassOnce}(P)\ \ \mathit{WFClasses}(P)\ \ \mathit{FieldsOnce}(P) \\ \mathit{MethodsOncePerClass}(P)\ \ \mathit{OverridesOK}(P) \\ P = \mathit{defn}_{1..n}\ \ \mathit{local}_{1..l}\ \ e \\ P \vdash \mathit{defn}_i \quad P;\ \mathit{local}_{1..l};\ \text{thisThread} \vdash e : t \end{array}}{\vdash P : t}$$

$\boxed{P \vdash \mathit{defn}}$

[CLASS]

$$\frac{\begin{array}{c} \text{if } (f_1 \neq \text{self} \mid \text{thisThread}) \text{ then } g_1 = \text{owner}_{formal}\ f_1 \\ \forall_{i=2..n}\ g_i = \text{owner}_{formal}\ f_i \quad E = g_{1..n},\ \text{final}\ cn\langle f_{1..n}\rangle\ \mathsf{this} \\ P;E \vdash c \quad P;E \vdash \mathit{field}_i \quad P;E \vdash \mathit{meth}_i \end{array}}{P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle\ \mathsf{extends}\ c\ \{\ \mathit{field}_{1..j}\ \ \mathit{meth}_{1..k}\ \}}$$

$\boxed{P; E \vdash_{owner} o}$

[OWNER THISTHREAD]

$$\frac{P;\ E \vdash \mathit{wf}}{P;\ E \vdash_{owner} \text{thisThread}}$$

[OWNER OTHERTHREAD]

$$\frac{P;\ E \vdash \mathit{wf}}{P;\ E \vdash_{owner} \text{otherThread}}$$

[OWNER SELF]

$$\frac{P;\ E \vdash \mathit{wf}}{P;\ E \vdash_{owner} \text{self}}$$

[OWNER UNIQUE]

$$\frac{P;\ E \vdash \mathit{wf}}{P;\ E \vdash_{owner} \text{unique}}$$

[OWNER READONLY]

$$\frac{P;\ E \vdash \mathit{wf}}{P;\ E \vdash_{owner} \text{readonly}}$$

[OWNER FINAL]

$$\frac{P;\ E \vdash_{final} e : t}{P;\ E \vdash_{owner} e}$$

[OWNER FORMAL]

$$\frac{\begin{array}{c} P;\ E \vdash \mathit{wf} \\ E = E_1,\ \text{owner}_{formal}\ f,\ E_2 \end{array}}{P;E \vdash_{owner} f}$$

$\boxed{P; E \vdash_{final} e}$

[FINAL VAR]

$$\frac{\begin{array}{c} P;\ E \vdash \mathit{wf} \\ E = E_1,\ \text{final}\ t\ x,\ E_2 \end{array}}{P;\ E \vdash_{final} x : t}$$

[FINAL REF]

$$\frac{\begin{array}{c} P;\ E \vdash (\text{final}\ t\ \mathit{fd}) \in cn\langle f_{1..n}\rangle \\ P;\ E \vdash_{final} e : cn\langle o_{1..n}\rangle \end{array}}{P;\ E \vdash_{final} e.\mathit{fd} : t[o_1/f_1]..[o_n/f_n]}$$

[FINAL THIS]

$$\frac{P;\ E \vdash \text{this} : \text{cn}\langle o_{1..n}\rangle}{P;\ E \vdash_{final} \text{this} : \text{cn}\langle o_{1..n}\rangle}$$

[FINAL CLASS OBJECT]

$$\frac{P;\ E \vdash cn\langle f_{1..n}\rangle}{P;\ E \vdash_{final} \text{cn.class} : \text{java.lang.Class}}$$

$\boxed{P; E \vdash_{static} e}$

[STATIC REF]

$$\frac{\begin{array}{c} P;\ E \vdash cn_0\langle o1..n\rangle \quad P;\ E \vdash cn_1\langle f1..n\rangle \\ P;\ E \vdash (\text{static}\ cn_0\langle o1..n\rangle\ \mathit{fd}) \in cn_1\langle f1..m\rangle \\ (o_{1..n} \cap f_{1..m}) - (\mathsf{self} + \mathsf{thisThread} + \mathsf{readonly} + \mathsf{unique}) = \emptyset \end{array}}{P;\ E \vdash_{static} cn_1.fd : cn_0\langle o1..n\rangle}$$

$\boxed{P; E \vdash wf}$

[ENV $\emptyset$]

$$\frac{}{P;\ \emptyset \vdash wf}$$

[ENV OWNER]

$$\frac{P;\ E \vdash wf \quad f \notin Dom(E)}{P;\ E,\ \text{owner}_{formal}\ f \vdash wf}$$

[ENV X]

$$\frac{P;\ E \vdash t \quad x \notin Dom(E)}{P;\ E,\ [\text{final}]_{opt}\ t\ x \vdash wf}$$

$\boxed{P; E \vdash t}$

[TYPE INT]

$$\frac{P;\ E \vdash wf}{P;\ E \vdash \text{int}}$$

[TYPE OBJECT]

$$\frac{P;E \vdash_{owner}\ o}{P;\ E \vdash \mathsf{Object}\langle o\rangle}$$

[TYPE SELF-SYNCHRONIZED CLASS]

$$\frac{\begin{array}{c}P \vdash \mathsf{class}\ cn\langle \text{self}\ f_{2..n}\rangle\ \ldots \\ P;E \vdash_{owner}\ o_{2..n}\end{array}}{P;E \vdash\ cn\langle \text{self}\ o_{2..n}\rangle}$$

[TYPE READONLY CLASS]

$$\frac{\begin{array}{c}P \vdash \mathsf{class}\ cn\langle \text{readonly}\ f_{2..n}\rangle\ \ldots \\ P;E \vdash_{owner}\ o_{2..n}\end{array}}{P;E \vdash\ cn\langle \text{readonly}\ o_{2..n}\rangle}$$

[TYPE THREAD-LOCAL CLASS]

$$\frac{\begin{array}{c}P \vdash \mathsf{class}\ cn\langle \text{thisThread}\ f_{2..n}\rangle\ \ldots \\ P;E \vdash_{owner}\ o_{2..n}\end{array}}{P;E \vdash\ cn\langle \text{thisThread}\ o_{2..n}\rangle}$$

[TYPE C]

$$\frac{\begin{array}{c}f_1 \neq \text{self} \mid \text{thisThread} \\ P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle\ \ldots \\ P;E \vdash_{owner}\ o_{1..n}\end{array}}{P;E \vdash\ cn\langle o_{1..n}\rangle}$$

$\boxed{P; E \vdash t_1 \sqsubseteq t_2}$

[SUBTYPE REFL]

$$\frac{P;\ E \vdash t}{P;\ E \vdash t \sqsubseteq t}$$

[SUBTYPE TRANS]

$$\frac{P;\ E \vdash t_1 \sqsubseteq t_2 \quad P;\ E \vdash t_2 \sqsubseteq t_3}{P;\ E \vdash t_1 \sqsubseteq t_3}$$

[SUBTYPE CLASS]

$$\frac{\begin{array}{c}P;E \vdash cn_1\langle o_{1..n}\rangle \\ P \vdash \mathsf{class}\ cn_1\langle f_{1..n}\rangle\ \mathsf{extends}\ cn_2\langle f_1\ o*\rangle\ \ldots\end{array}}{P;\ E \vdash cn_1\langle o_{1..n}\rangle \sqsubseteq cn_2\langle f_1\ o*\rangle\ [o_1/f_1]..[o_n/f_n]}$$

$\boxed{P; E \vdash field}$

[FIELD INIT]

$$\frac{P;\ E;\ \text{thisThread} \vdash\ e\ :\ t}{P;\ E \vdash\ [\text{final}]_{opt}\ t\ fd\ =\ e}$$

$\boxed{P; E \vdash field \in c}$

[FIELD DECLARED]

$$\frac{P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle\ldots\ \{\ \ldots\ field\ \ldots\ \}}{P;\ E \vdash\ field \in\ cn\langle f_{1..n}\rangle}$$

[FIELD INHERITED]

$$\frac{\begin{array}{c}P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle\ldots\ \{\ \ldots\ field\ \ldots\ \} \\ P \vdash \mathsf{class}\ cn'\langle g_{1..m}\rangle\ \mathsf{extends}\ cn\langle o_{1..n}\rangle\ldots\end{array}}{P;\ E \vdash\ field[o_1/f_1]..[o_n/f_n]\ \in\ cn'\langle g_{1..m}\rangle}$$

$\boxed{P; E \vdash method}$

[METHOD]

$$\frac{\begin{array}{c}g_i\ =\ \text{final}\ arg_i \quad P;\ E,\ g_{1..n} \vdash_{final}\ e_i\ :\ t_i \\ P;\ E,\ g_{1..n} \vdash \text{RootOwner}(e_i)\ =\ r_i \\ P;\ E,\ g_{1..n},\ local_{1..l};\ \text{thisThread},\ r_{1..m} \vdash\ e\ :\ t\end{array}}{P;\ E \vdash\ t\ mn(arg_{1..n})\ \mathsf{requires}\ (e_{1..m})\{local_{1..l}\ \ e\}}$$

[STATIC METHOD]

$$\frac{\begin{array}{c}g_i\ =\ \text{final}\ arg_i \quad P;\ E,\ g_{1..n} \vdash_{final}\ e_i\ :\ t_i \\ P;\ E,\ g_{1..n} \vdash \text{RootOwner}(e_i)\ =\ r_i \\ P;\ E,\ g_{1..n},\ local_{1..l};\ \text{thisThread},\ r_{1..m} \vdash\ e\ :\ t \\ \forall e_i : e_i \neq \text{this}\end{array}}{P;\ E \vdash_{\text{static}}\ t\ mn(arg_{1..n})\ \mathsf{requires}\ (e_{1..m})\{local_{1..l}\ \ e\}}$$

$\boxed{P; E \vdash constructor}$

[CONSTRUCTOR]

$$\frac{\begin{array}{c}g_i\ =\ \text{final}\ arg_i \quad P;\ E,\ g_{1..n} \vdash_{final}\ e_i\ :\ t_i \\ P;\ E,\ g_{1..n} \vdash \text{RootOwner}(e_i)\ =\ r_i \\ P;\ E,\ g_{1..n},\ local_{1..l};\ \text{thisThread},\ r_{1..m},\ RO(\mathsf{this}) \vdash\ e\ :\ t\end{array}}{P;\ E \vdash\ cn(arg_{1..n})\ \mathsf{requires}\ (e_{1..m})\{local_{1..l}\ \ e\}}$$

$\boxed{P; E \vdash meth \in c}$

[METHOD DECLARED]

$$\frac{P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle\ldots\ \{\ \ldots\ meth\ \ldots\ \}}{P;\ E \vdash\ meth\ \in\ cn\langle f_{1..n}\rangle}$$

[METHOD INHERITED]

$$\frac{\begin{array}{c}P \vdash \mathsf{class}\ cn\langle f_{1..n}\rangle\ldots\ \{\ \ldots\ meth\ \ldots\ \} \\ P \vdash \mathsf{class}\ cn'\langle g_{1..m}\rangle\ \mathsf{extends}\ cn\langle o_{1..n}\rangle\ldots\end{array}}{P;\ E \vdash\ meth[o_1/f_1]..[o_n/f_n]\ \in\ cn'\langle g_{1..m}\rangle}$$

$\boxed{P; E \vdash \text{RootOwner}(e)\ =\ r}$

[ROOTOWNER THISTHREAD]

$$\frac{P;\ E \vdash\ e\ :\ cn\langle thisThread\ o*\rangle\ \mid\ \mathsf{Object}\langle thisThread\rangle}{P;\ E \vdash \text{RootOwner}(e)\ =\ \mathsf{thisThread}}$$

[ROOTOWNER OTHERTHREAD]

$$\frac{P;\ E \vdash\ e\ :\ cn\langle otherThread\ o*\rangle\ \mid\ \mathsf{Object}\langle otherThread\rangle}{P;\ E \vdash \text{RootOwner}(e)\ =\ \mathsf{otherThread}}$$

[ROOTOWNER SELF]

$$\frac{P;\ E \vdash e : cn\langle self\ o*\rangle \mid \mathsf{Object}\langle self\rangle}{P;\ E \vdash \mathrm{RootOwner}(e) = e}$$

[ROOTOWNER UNIQUE/READONLY]

$$\frac{P;\ E \vdash e : \mathsf{Object}\langle unique\rangle \mid \mathsf{Object}\langle readonly\rangle}{P;\ E \vdash \mathrm{RootOwner}(e) = \emptyset}$$

[ROOTOWNER FINAL TRANSITIVE]

$$\frac{\begin{array}{c} P;\ E \vdash e : cn\langle o_{1..n}\rangle \mid \mathsf{Object}\langle o_1\rangle \\ P;\ E \vdash_{final} o_1 : c_1 \\ P;\ E \vdash \mathrm{RootOwner}(o_1) = r \end{array}}{P;\ E \vdash \mathrm{RootOwner}(e) = r}$$

[ROOTOWNER FORMAL]

$$\frac{\begin{array}{c} P;\ E \vdash e : cn\langle o_{1..n}\rangle \mid \mathsf{Object}\langle o_1\rangle \\ E = E_1,\ owner_{formal}\ o_1,\ E_2 \end{array}}{P;\ E \vdash \mathrm{RootOwner}(e) = \mathrm{RO}(e)}$$

$\boxed{P; E \vdash e : t}$

[EXP TYPE]

$$\frac{\exists_{ls}\ P;\ E;\ ls \vdash e : t}{P;\ E \vdash e : t}$$

$\boxed{P; E; ls \vdash e : t}$

[EXP SUB]

$$\frac{P;\ E;\ ls \vdash e : t' \quad P;\ E;\ ls \vdash t' \sqsubseteq t}{P;\ E;\ ls \vdash e : t}$$

[EXP NEW]

$$\frac{\begin{array}{c} P;\ E \vdash cn\langle f_{1..n}\rangle \\ P;\ E \vdash (cn(t_j\ y_j\ ^{j\in 1..k})\ \mathsf{requires}\ (e'_{1..p})) \in cn\langle f_{1..n}\rangle \\ P;\ E;\ ls \vdash e_j : t_j[o_1/f_1]..[o_n/f_n] = cn'\langle o_{j_{1..n}}\rangle \\ o_{j_1} \neq \mathsf{this} \\ (o_{j_1} = \mathsf{unique})\ \&\&\ (e_j \neq e'_j\texttt{--}) \Rightarrow y_j!e \\ o_{j_1} = \mathsf{readonly} \Rightarrow y_j!w \\ P;\ E \vdash \mathrm{RootOwner}(e'_i[o_1/f_1]..[o_n/f_n]) = r'_i \quad r'_i \in ls \end{array}}{P;\ E;\ ls \vdash \mathsf{new}\ cn\langle o_{1..n}\rangle(e_{1..m}) : cn\langle o_{1..n}\rangle}$$

[EXP SEQ]

$$\frac{P;\ E;\ ls \vdash e_1 : t_1 \quad P;\ E;\ ls \vdash e_2 : t_2}{P;\ E;\ ls \vdash e_1; e_2 : t_2}$$

[EXP VAR]

$$\frac{P;\ E \vdash wf \quad E = E_1,\ [\mathrm{final}]_{opt}\ t\ x,\ E_2}{P;\ E;\ ls \vdash x : t}$$

[EXP VAR INIT]

$$\frac{P;\ E;\ ls \vdash e : t}{P;\ E;\ ls \vdash [\mathrm{final}]_{opt}\ t\ x = e}$$

[EXP VAR ASSIGN]

$$\frac{\begin{array}{c} E = E_1, t\ x,\ E_2 \quad P;\ E;\ ls \vdash e : t \\ P;\ E;\ ls \vdash \mathrm{t} = \mathrm{cn}\langle o_{1..n}\rangle \\ o_1 \neq \mathsf{unique,\ readonly} \end{array}}{P;\ E;\ ls \vdash x = e : t}$$

[EXP UNIQUE VAR ASSIGN]

$$\frac{\begin{array}{c} E = E_1, t\ x,\ E_2 \quad P;\ E;\ ls \vdash e : t \\ P;\ E;\ ls \vdash \mathrm{t} = \mathrm{cn}\langle o_{1..n}\rangle \qquad o_1 = \mathsf{unique} \end{array}}{P;\ E;\ ls \vdash x = e\texttt{--}\!: t}$$

[EXP SYNC]

$$\frac{\begin{array}{c} P;\ E \vdash_{final} e_1 : t_1 \\ P;\ E;\ ls,\ e_1 \vdash e_2 : t_2 \end{array}}{P;\ E;\ ls \vdash \mathsf{synchronized}\ e_1\ \{\ e_2\ \} : t_2}$$

[EXP FORK]

$$\frac{\begin{array}{c} P;\ E;\ ls \vdash e : cn\langle o_{1..n}\rangle \\ P;\ E;\ ls \vdash cn\langle o_{1..n}\rangle \sqsubseteq \mathrm{java.lang.Thread} \\ P;\ g[\mathrm{otherThread/thisThread}];\ \mathrm{thisThread} \vdash (\mathrm{void}\ run()\ \mathsf{requires}(\mathrm{this})\ \ldots\ ) \in cn\langle f_{1..n}\rangle \end{array}}{P;\ E;\ ls \vdash \mathrm{e.start():void}}$$

[EXP INVOKE]

$$\frac{\begin{array}{c} P;\ E;\ ls \vdash e : cn\langle o_{1..n}\rangle \\ P;\ E \vdash (t\ mn(t_j\ y_j\ ^{j\in 1..k})\ \mathsf{requires}(e'_{1..m})...) \in cn\langle f_{1..n}\rangle \\ P;\ E;\ ls \vdash e_j : t_j[e/\mathrm{this}][o_1/f_1]..[o_n/f_n] = cn'\langle o_{j_{1..n}}\rangle \\ o_{j_1} \neq \mathsf{unique,\ readonly} \\ P;\ E \vdash \mathrm{RootOwner}(e'_i[e/\mathrm{this}][o_1/f_1]..[o_n/f_n]) = r'_i \\ r'_i \in ls \end{array}}{P;\ E;\ ls \vdash e.mn(e_{1..k}) \colon t[e/\mathrm{this}][o_1/f_1]..[o_n/f_n]}$$

[EXP STATIC INVOKE]

$$\frac{\begin{array}{c} P;\ E \vdash_{\mathrm{static}} (t\ mn(t_j\ y_j\ ^{j\in 1..k})\ \mathsf{requires}(e'_{1..m})...) \in cn\langle f_{1..n}\rangle \\ P;\ E;\ ls \vdash e_j : t_j = cn'\langle o_{j_{1..n}}\rangle \\ o_{j_1} \neq \mathsf{unique,\ readonly} \\ P;\ E \vdash \mathrm{RootOwner}(e_i) = r_i \\ r_i \in ls \end{array}}{P;\ E;\ ls \vdash cn.mn(e_{1..k}) \colon t}$$

[EXP INVOKE UNIQUE/READONLY]

$$\frac{\begin{array}{c} P;\ E;\ ls \vdash e : cn\langle o_{1..n}\rangle \\ P;\ E \vdash (t\ mn(t_j\ y_j\ ^{j\in 1..k})\ \mathsf{requires}(e'_{1..m})...) \in cn\langle f_{1..n}\rangle \\ P;\ E;\ ls \vdash e_j : t_j[e/\mathrm{this}][o_1/f_1]..[o_n/f_n] = cn'\langle o_{j_{1..n}}\rangle \\ (o_{j_1} = \mathsf{unique})\ \&\&\ (e_j \neq e'_j\texttt{--}) \Rightarrow y_j!e \\ o_{j_1} = \mathsf{readonly} \Rightarrow y_j!w \\ P;\ E \vdash \mathrm{RootOwner}(e'_i[e/\mathrm{this}][o_1/f_1]..[o_n/f_n]) = r'_i \\ r'_i \in ls \end{array}}{P;\ E;\ ls \vdash e.mn(e_{1..k}) \colon t[e/\mathrm{this}][o_1/f_1]..[o_n/f_n]}$$

[EXP STATIC INVOKE UNIQUE/READONLY]

$$\frac{\begin{array}{c} P;\ E \vdash_{\mathrm{static}} (t\ mn(t_j\ y_j\ ^{j\in 1..k})\ \mathsf{requires}(e'_{1..m})...) \in cn\langle f_{1..n}\rangle \\ P;\ E;\ ls \vdash e_j : t_j = cn'\langle o_{j_{1..n}}\rangle \\ (o_{j_1} = \mathsf{unique})\ \&\&\ (e_j \neq e'_j\texttt{--}) \Rightarrow y_j!e \\ o_{j_1} = \mathsf{readonly} \Rightarrow y_j!w \\ P;\ E \vdash \mathrm{RootOwner}(e_i) = r_i \\ r_i \in ls \end{array}}{P;\ E;\ ls \vdash cn.mn(e_{1..k}) \colon t}$$

[EXP REF]

$$\frac{P;\, E;\, ls \vdash e : cn\langle o_{1..n}\rangle \quad P;\, E \vdash ([\text{final}]_{opt}\ t\ fd) \in cn\langle f_{1..n}\rangle \quad P;\, E \vdash \text{RootOwner}(e) = r \quad r \in ls}{P;\, E;\, ls \vdash e.fd : t[e/\text{this}][o_1/f_1]..[o_n/f_n]}$$

[EXP STATIC REF]

$$\frac{P;\, E \vdash cn\langle f_{1..n}\rangle \quad P;\, E \vdash (\text{static}\ t\ fd) \in cn\langle f'_{1..m}\rangle \quad P;\, E \vdash \text{cn.class} \in ls}{P;\, E;\, ls \vdash cn.fd : t}$$

[EXP ASSIGN]

$$\frac{P;\, E;\, ls \vdash e : cn\langle o_{1..n}\rangle \quad P;\, E \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \quad P;\, E \vdash \text{RootOwner}(e) = r \quad r \in ls \quad P;\, E;\, ls \vdash e' : t[e/\text{this}][o_1/f_1]..[o_n/f_n] \quad P;\, E;\, ls \vdash \text{t} = \text{cn'}\langle o'_{1..n}\rangle \quad o'_1 \neq \textsf{unique, readonly}}{P;\, E;\, ls \vdash e.fd = e' : t[e/\text{this}][o_1/f_1]..[o_n/f_n]}$$

[EXP UNIQUE ASSIGN]

$$\frac{P;\, E;\, ls \vdash e : cn\langle o_{1..n}\rangle \quad P;\, E \vdash (t\ fd) \in cn\langle f_{1..n}\rangle \quad P;\, E \vdash \text{RootOwner}(e) = r \quad r \in ls \quad P;\, E;\, ls \vdash e' : t[e/\text{this}][o_1/f_1]..[o_n/f_n] \quad P;\, E;\, ls \vdash \text{t} = \text{cn'}\langle o'_{1..n}\rangle \quad o'_1 = \textsf{unique}}{P;\, E;\, ls \vdash e.fd = e'\text{--} : t[e/\text{this}][o_1/f_1]..[o_n/f_n]}$$

Bibliography

[1] C. Boyapati, M. Rinard *A Parameterized Type System for Race-Free Java Programs* Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2001

[2] C. Boyapati, R. Lee, M. Rinard *A Type System for Preventing Data Races and Deadlocks in Java Programs* Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2002

[3] D.F. Bacon, R.E. Strom, and A. Tarafdar. *Guava: A dialect of Java without data races.* Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2000

[4] C. Flanagan and M. Abadi. *Types for safe locking.* In European Symposium on Programming (ESOP), March 1999

[5] C. Flanagan and M. Abadi. *Object types against races.* In Conference on Concurrent Theory (CONCUR), August 1999

[6] C. Flanagan and S.N. Freund. *Type-based race-detection for Java.* In Programming Language Design and Implementation (PLDI), June 2000

[7] J. Aldrich, C. Chambers, E.G. Sirer, S. Eggers *Static Analyses for Eliminating Unnecessary Synchronization from Java Programs*

[8] R. Netzer, B. Miller *What are Race Conditions? Some issues and Formalizations* ACM Letters on Programming Languages and Systems, Vol. 1, No. 1, March 1992

[9] J. Gosling, B. Joy, G. Steele *The Java Language Specification* Addison-Wesley, 1996

[10] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson *Eraser: A dynamic data race detector for multi-threaded programs* Symposium on Operating Systems Principles (SOSP), October 1997

[11] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler *GJ: Extending the Java programming language with type parameters*, 1998

[12] M. Viroli, A. Natali. *Parametric Polymorphism in Java: an approach to translation based on reflective features.* In Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) 2000.

[13] *The Kopi Project* Available online at http://dms.at/kopi/

[14] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1991.

[15] JD. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, M. Sridharan. *Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs* Programming Language Design and Implementation (PLDI) 2002

[16] J. Bloch. *Effective Java Programming Language Guide*. Addison-Wesley, 2001

[17] M. Flatt, S. Krishnamurthi, M. Felleisen. *Classes and mixins.* In Principles of Programming Languages (POPL), January 1998

[18] C.v. Praun, T. Gross. *O*bject Race Detection. In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2001.