Ownership Types Restrict Aliasing

MEng. Computing Final Year Project Report

Alexander Buckley
arb@doc.ic.ac.uk

Supervisor: Sophia Drossopoulou

Department of Computing
Imperial College of Science, Technology and Medicine
180 Queens Gate, London SW7 2BZ, UK

June 2000
Acknowledgements

My very greatest thanks must go to Dr. Sophia Drossopoulou for the many hours of guidance, discussion and teaching that I have benefitted from during this project, and for patiently listening to so many of my “intuitively obvious” expositions of some ownership feature or another. Also, I thank Professor Chris Hankin for commentary along the way and acting as second marker, and David Clarke of Macquarie University, Sydney, for his willingness to develop our early understanding with increasingly lengthy emails.
Abstract

Object-oriented programming languages allow references to create multiple aliases to the same object. This permits object sharing, and gives programmers the ability to model real-world phenomena through networks of interacting objects. However, the possibility that an object’s internal state may be modified through an alias to it adds complexity and makes reasoning and optimisation of the code more difficult.

Ownership types embed the notion of object ownership into the type system. Every object has an owner that strongly encapsulates the objects that it owns. Aliasing may only occur where the programmer uses ownership types that permit it.

This thesis presents an implementation of ownership types for Java, and extends it with new notions of ownership. It also presents a scheme for dynamic validation of a program’s ownership model.
Table of Contents

1. INTRODUCTION .......................................................................................................................... 5
   1.1 ALIASING .......................................................................................................................... 5
   1.2 ALIAS CONTROL IN JAVA ............................................................................................ 5
   1.3 OWNERSHIP TYPES ...................................................................................................... 7
   1.4 MY CONTRIBUTION ......................................................................................................... 11
   1.5 REPORT STRUCTURE ................................................................................................... 12

2. BACKGROUND ........................................................................................................................... 13
   2.1 THE GENEVA CONVENTION ........................................................................................ 13
   2.2 ALIAS DETECTION .......................................................................................................... 15
   2.3 ALIAS ADVERTISEMENT AND PREVENTION .......................................................... 16
   2.4 THE FLEXIBLE ALIAS PROTECTION MODEL .......................................................... 17
   2.5 CURRENT RESEARCH DIRECTIONS FOR OWNERSHIP ......................................... 21

3. THE LANGUAGE JOT – JAVA WITH OWNERSHIP TYPES ................................................ 23
   3.1 LANGUAGE GRAMMAR ................................................................................................. 23
   3.2 APPLYING OWNERSHIP TYPES TO JAVA ................................................................. 24
   3.3 TYPE RULES .................................................................................................................. 28

4. IMPLEMENTATION .................................................................................................................. 30
   4.1 TOOLS & TECHNIQUES ................................................................................................. 30
   4.2 GRAMMAR EXTENSION ............................................................................................... 37
   4.3 STRUCTURE OF THE PRE-PROCESSOR ..................................................................... 44
   4.4 PRE-PROCESSOR STAGE #1 - INSPECTING THE AST ................................................. 45
   4.5 PRE-PROCESSOR STAGE #2 – TYPE-CHECKING ....................................................... 52
   4.6 TEST PLANS ................................................................................................................. 65

5. EXTENDING OWNERSHIP TYPES IN JOT+* ........................................................................ 66
   5.1 ANONYMOUS OWNERSHIP CONTEXTS ..................................................................... 66
   5.2 POLYMORPHIC OWNERSHIP CONTEXTS ................................................................. 79
   5.3 FORMALISING JOT+* .................................................................................................... 88
   5.4 IMPLEMENTING JOT+* ................................................................................................. 91

6. OWNERSHIP AT RUNTIME ..................................................................................................... 98
   6.1 DYNAMIC OWNERSHIP SUPPORT ............................................................................. 98
   6.2 DESIGN ........................................................................................................................... 99
   6.3 IMPLEMENTATION ........................................................................................................ 102
   6.4 FURTHER WORK ........................................................................................................... 109

7. DISCUSSION ............................................................................................................................ 112
   7.1 USING GENERIC JAVA TYPE PARAMETERS TO SYNTHESE CONTEXTS .................... 112
   7.2 A MONOTONE FRAMEWORK FOR OWNERSHIP .................................................... 113

8. EVALUATION .......................................................................................................................... 119
   8.1 CONCLUSIONS .............................................................................................................. 119
   8.2 ACHIEVEMENTS .......................................................................................................... 120
   8.3 FURTHER WORK .......................................................................................................... 122

APPENDIX A – TEST CASES ........................................................................................................ 124

BIBLIOGRAPHY ........................................................................................................................... 174
1. Introduction

1.1 Aliasing

One of the most common activities in OO programming is implementing an abstract data type by aggregating objects into a logical container object. An OO language provides information hiding [IH] by allowing objects to encapsulate certain state and behaviour and present a public interface to it; programmers may then code to the interface of an object rather than its internal implementation [DP].

Languages with pointers typically allow multiple pointers to refer to the same memory location, or in an OO language, to the same object. An object is said to be aliased when multiple references exist to it; those references are said to be aliases. Generally, understanding what a program does in the presence of aliases becomes more complex because run-time information about the topology of the system is required in order to understand the effects of state changes. (Incidentally, this is as true for simple C programs making assignments of the form `int q; int *p = &q;` as it is for large, complex OO systems.) The inability to localise references to an object leads to a loss of modularity when reasoning about programs.

We wish to be able to reason more strongly about the encapsulation provided by an aggregate container in the presence of aliasing. This means developing a mechanism for truly encapsulating the internal state of objects against explicit attempts (possibly mistaken, possibly malicious) to access such state.

It is important to note that aliasing is not always undesirable. Often, multiple objects holding references to the same data object is a necessary and natural part of an object-oriented design; it is how data sharing is represented [JAC]. For example, a Person object may hold a reference to another Person object under the instance variable “friend”. Even as the details of that Person change, e.g. their name changes on marriage, it is fundamentally still the same person in real life and the same object in the system. Indeed, a number of papers [GC][FAP] remark that the property of object identity makes aliasing a powerful technique in the right circumstances; all parties are aware of the correct information “automatically”. However, because aliasing is so powerful, it is necessary to restrict it.

1.2 Alias control in Java

Java allows a programmer to encapsulate internal state and behaviour of an object through language features which specify the visibility of member variables and methods (the `private`, `protected` and `public` keywords). In fact, the semantics of these keywords do not provide the level of encapsulation that is often supposed, and it surprises most Java programmers to learn that marking a variable or method as `private` hides it only from other classes; instances of the same class can access each other’s private or protected members and methods with impunity. An example is shown in figure 1.

This example is shown merely to make the reader aware that the semantics of `private` is not always what the reader would naturally expect. (It may well be true that objects of the same class accessing each other’s private members and methods is not a common OO idiom). But there is a more serious semantic difference between a natural notion of a “private” variable, (that it is fully and permanently encapsulated within an object), and Java’s declared semantics. Even when the internal state of an object is hidden from other objects, and the
Ownership Types Restrict Aliasing

A technique from figure 1 is not practised, it is still not possible to guarantee that other references are not held to the private variables. Figure 2 shows what leads to this situation.

There is nothing to stop a method in class A from returning a reference to its private instance variable. The caller of such a method could itself reference the object, store the received reference, pass it on to its callers or callees, or indeed set it to null.

The reason why \texttt{A.checkIt()} is returning a reference is likely to be due to programmer confusion over the behaviour of their program. Sun themselves fell foul of this in the JDK 1.1.1, where a method in a system-level object returned a reference to a private array of highly sensitive data [CT]. Once exposed to the caller, the reference could be used to update elements of the array which, due to its role in Java’s security architecture, could lead to malicious code being allowed to perform any activity it desired on the system. A Sun programmer was mistaken in returning a direct reference to the array; a copy should have been returned, or some other semantics defined for the message sent to the system-level object.

It is clear that Java does not enforce true information hiding nor rigorous encapsulation, as mechanisms exist to subvert even the strongest intentions of the programmer implementing an abstraction via aggregation of objects. Where the programmer is less sure of his program’s semantics, internal state may be exposed even more easily. The net result of this is that aliasing may occur more frequently and surreptitiously than many programmers would expect.

\begin{verbatim}
MEng. Computing, Imperial College
\end{verbatim}


1.3 Ownership types

Work by Clarke, Noble, Vitek and Potter has produced a series of papers that form the basis for this project’s study. The notion of “flexible alias protection” was developed in [FAP], and key concepts from it formalised into Java’s type system in [OT]. Subsequent papers have given examples of the capabilities of the approach [MS], and developed its theoretical grounding [WA].

Ownership types form a static type system that indicates object ownership via type parameters [OT]. The notion of ownership is that when an object owns another object, it encapsulates it so that no updates occur to the owned object without the authority of the owning object. The nested hierarchy of objects that occurs naturally in system design can be more formally reflected in the object structures created by a programmer, and in this sense ownership is just as obvious a concept as sharing is – both are meant to reflect programmers' intuition about the natural shape of their data structures. [SAB] noted that when the controlled sharing of information via encapsulation makes the idea of “ownership” of objects difficult to resolve; ownership types embed per-object ownership directly into the type system.

Each object created in the system has a context associated with it. This context represents the “interior” of the object, and is known as its representation - rep. When a variable is declared at an ownership type, then its owner is specified as well as its class. A class definition declaring fields at ownership types may use “rep” as the fields’ owner: when an object of this class is instantiated, those fields declared as rep are considered to be owned by it. Each new object of this class that is instantiated has its own representation context, so the fields in each new object are considered as owned by that new object. The objects which may access fields within an object’s representation context are the object itself and those other fields also within that context (i.e. also declared rep within the object’s class definition).

The representation context of an object is absolutely private to it; other objects, even of the same class, cannot access it. Therefore, rep gives “per-object” protection and allows an object to truly encapsulate its internal state. The ownership type system considers that fields declared with rep in their type are invisible to objects that are not the owner of the field (i.e. it is their representation context which the field is declared within). Expressions are thus untypeable if an object access occurs which would subvert another object’s representation context, i.e. access fields owned by that representation context without the owning object being in the access path. The owner of an object in fact dominates the objects that it owns, in that access paths from higher in the graph to the owned objects must pass through the owning object.

A program begins execution in a root context. The owner for objects in this context is effectively the system, and such objects are globally available. The name of this context is norep – everything in it is part of nobody’s representation.

Example

Figure 3 shows an object graph [IOO] where objects are represented by boxes and inter-object references denoted by continuous lines. The arrows indicate, for example, that the “car” object has a reference to the “driver” object. A simple ownership structure [OT] based on representation containment is imposed on an object.

The notion is that we wish an object implementing an abstraction of a car to be owned by nobody, so that it may be referenced by anybody. Within the car, there is a reference to an object representing an engine, and that object is owned by the car. Hence, a direct reference
from an object in the root context (owned by no-one, or more correctly, noRep) to the engine object is not allowed; the car must feature in all access paths from higher objects than it in the object graph to the engine. The car also holds a reference to a driver object that is owned by noRep, so can be referenced from anywhere (i.e. aliased freely). This is implemented by the code in figure 4.

```java
class Engine {
  void start() { … }
  void stop() { … }
}
class Driver { … }
class Car {
  rep   Engine engine;
  noRep Driver driver;
  Car() {
    engine = new rep Engine();
    driver = null;
  }
  rep Engine getEngine() {
    return engine;
  }
  void setEngine(rep Engine e) {
    engine = e;
  }
  void go() {
    if (driver != null)
      engine.start();
  }
}
```

```java
class CarFactory {
  void deliverCar() {
    noRep Driver bob =
      new noRep Driver();
    noRep Car car = new noRep Car();
    car.driver = bob;
    car.go();
    car.engine.stop();
    // Fails because car.engine is
    // within the rep context of car,
    // and a reference to it may not
    // be made outside that context.
    // Make a new engine object in the
    // rep context of the CarFactory.
    rep Engine e = new rep Engine();
    car.setEngine(e);
    // Fails because the formal
    // parameter of setEngine()
    // requires it to be within the
    // rep of the Car object.
    car.setEngine(e);
  }
}
```

Figure 4

An ownership type is a type annotated with context declarations. Two variables instantiated as ownership types, but annotated with different context declarations, are considered different types.

In figure 5, in method Aggregate.doWork(), the representation context of the current instance of the Aggregate class is passed as a context parameter to the new LinkedList object. According to the declaration of the LinkedList class, it is bound to the formal context parameter m. Within LinkedList, m may be used to in field declarations as either the owner of those fields or as an actual context context parameter for instances of other classes.
A graphical representation of the ownership structure built by this code is shown in figure 6. An instance of a LinkedList has only a single representation object, called top. It refers to the first Link in the list, and because it is within the representation context of its particular instance of LinkedList, may not be referenced outside it.

When new data is to be added to the list, LinkedList.push() instantiates a new Link object and passes the actual context parameter \( m \) to it. This, recall, is bound to the representation context of the object instantiating the LinkedList. The formal context parameter \( n \) in the class signature `class Link<n> { .. }` becomes bound to this context, and \( n \) is used as the owner of the field `data` in a Link. As such, a Link holds a reference to a data object of some class \( X \), whose owner is the representation context of the Aggregate object.

The effect of the `owner` annotation for the `next` field in Link, is that when a Link is instantiated, the `next` object becomes owned by the object instantiating the Link. Because all Links are explicitly created within the representation of a LinkedList object (in `LinkedList.push()`), that particular instance of a LinkedList owns all the Link objects. Accesses to each Link’s `next` object must be made through sending a message to the LinkedList instance, which may write expressions that utilise the `next` references of its Links (e.g. `top.next.next.next`). Outside the LinkedList instance’s representation context, a Link’s `next` reference is invisible.

```java
class Aggregate {
    void doWork() {
        LinkedList<rep> myList =
            new LinkedList<rep>();
        rep Data d = new rep Data();
        myList.push(d);
    }
}

class LinkedList<m> {
    rep Link<m> top = null;
    void push(m X data) {
        rep Link<m> newTop =
            new rep Link<m>(data);
        newTop.next = top;
        top = newTop;
    }
    m X pop() {
        rep Link<m> oldTop = top;
        rep Link<m> top =
            oldTop.next;
        return top.data;
    }
    boolean isEmpty() {
        return top == null;
    }
}
```

Note that if figure 6, if the Aggregate class performed `LinkedList<rep> myList2 = new LinkedList<rep>()` then that second LinkedList object would have its own representation context which owned its Links; each LinkedList’s structure of Links is fully protected against aliases from outside that instance of a LinkedList. The data objects of `myList2` would be owned by the representation context of the Aggregate, just like the data objects of `myList`, and we could call `myList2.push(d)` to store the Aggregate’s `rep Data` object in two containers simultaneously.

Formally, an ownership type consists of a class name, a context representing the owner, and bindings for the context parameters. It has the form \( c < M | M^* > \), where \( c \) is the underlying class, \( M \) is the owner and \( M^* \) represents one or more formal context parameters for the class’ definition. The Java notation is more readable: `owner Link<n> next` is formally `Link<owner|n> next`. 
The key facility that ownership types provide is restricting access to fields which are declared with the rep annotation. In a given class, the only expression statically guaranteed to denote the object which owns any rep objects is this. Other expressions representing access paths to objects may denote different objects, for whom the rep in the type of their fields and methods is different. This is formalised as the Static Visibility constraint, and occurs frequently in the semantics of the restricted Java language used by [OT].

![Object Graph](image)

**Figure 6**

On object graphs such as figure 6, three properties hold if a program is well-typed:

- **Role separation** - Two different ownership types appearing in the same context are not compatible, regardless of the ensuing binding.

  For example, variables declared as Link<n> x and Link<n> y are incompatible even if m and n are context parameters bound to the same context. This guarantees that a class will behave the same way regardless of the bindings to its formal context parameters (albeit making static type-checking conservative). Specifically, we have a static guarantee that values of two ownership types will not be mixed within a class. This is the analogue of the “no role confusion” invariant in the full flexible alias protection model.

- **Restricted visibility** - Objects assigned to fields, passed as arguments to method calls, returned from field access or method call must be visible in both the context of the caller and the callee.

  Essentially, this stops the creation of unwanted dynamic aliases, which typically occur through those mechanisms listed above. [OT] states explicitly: “One could imagine that such references were valid, even though assignment was not. They aren’t.” In our example, car.engine is forbidden because os static visibility, and hence car.engine.stop() too.

  In particular, one cannot access an object with rep in its type from outside its owner. This invariant is maintained by requiring the semantics of well-typed programs to obey the static visibility constraint mentioned previously.
- **Representation containment** - All paths from the root of the system must pass through an object’s owner.

This invariant prevents “representation exposure”; since the only path from the root of the system to an object’s representation is through its owner, the owner is conceptually “in control” of its representation. The rest of the system must use its interface (i.e. methods) to affect changes in fields in its representation.

### 1.4 My contribution

This project is about the implementation and extension of a type-checking tool for programs written in the language JOT – Java with Ownership Types.

An implementation of the ownership type system in [OT] was built by Clarke, Noble and Potter that used the Pizza toolkit [PZ] that extends Java with parametric polymorphism. It extended Pizza’s parser and type-checking phases (not the compiler itself) but as it developed, the authors felt the need to formalise the alias protection model. The capabilities of the eventual implementation are exemplified by [MS]; the authors themselves have not developed it further, and advised not trying to do so.

The tool developed here is a pre-processor for the Java compiler, *javac*, that accepts programs written in Java and annotated with ownership features (denoted by the file extension “.jot”). It checks the basic ownership type rules from [OT] and for programs that are type-correct, rewrites them without ownership features into a corresponding file with a “.java” extension for ordinary compilation.

The ownership type system is then extended to feature novel concepts of ownership arising from study of how ownership restricts can be used in programs. The JOT language is extended to JOT+*, with suitable constructs and enhanced type rules, and the pre-processor made capable of checking JOT+* programs. A test suite is given that covers features from JOT and JOT+*.

![Diagram](attachment:file.png)
To contrast with the static analysis driven by the ownership type system, a scheme for validating object ownership at run-time is developed and implemented. [DAP] proposes techniques for tracking the owner of an object at run-time, but it is for an unusual language; I support programs written in JOT, including any standard Java program.

1.5 Report structure

The remainder of the report is structured as follows. Chapter 2 summarises well-known techniques for the management of aliasing, and describes some of the earlier published work that led to the development of ownership types. Chapter 3 gives formal and descriptive specifications of the type system to be implemented, and chapter 4 describes tools, design and implementation techniques. Chapter 5 extends the basic ownership type system with anonymous and polymorphic ownership, and chapter 6 describes the scheme of run-time support for ownership. Chapter 7 discusses a variety of theoretical concepts. Chapter 8 concludes the project by summarising what has been achieved and suggesting further research.
2. Background

This section describes work in the field of aliasing in object-oriented programs, and outlines recent proposals for developing and implementing ownership.

2.1 The Geneva Convention

A seminal document in the OO research community was “The Geneva Convention On The Treatment of Object Aliasing” [GC]. Written in 1991, it provides an overview of the aliasing problem, why it is particularly important to the OO paradigm, and suggests a taxonomy of approaches for handling it.

The paper starts by making clear the dual activities of formal verification and practical programming, and how aliasing causes difficulties in both. Invariants (in the form of pre- and post-conditions for a block of code) may be violated in the presence of aliasing, and proving that aliasing cannot occur is asserted to be not always straightforward. An example for a Pascal-like language is given:

- if \(x\) and \(y\) are aliases
- then (* pre: \(x = \text{true} \)* \(y := \text{false} (* \text{post: } x = \text{true} *) \)) will not hold.

Providing motivating examples of aliasing is often more difficult than it appears: it must be possible to contrive the aliasing in the first place, and then find an example where that aliasing is undesirable. Even where object constructions lead to aliasing, it may not induce a problem that needs to be solved. However, a fine example is given for a matrix multiplication routine \(\text{mult}(\text{left}, \text{right}, \text{result})\), which stores the product of its first two parameters into the third. This works perfectly well until a programmer writes \(\text{mult}(a, b, a)\) which does not look so unreasonable. But if the implementer of the routine did not consider that the arguments are aliased – which requires no greater language construct than pointers – then the matrix represented by the first actual parameter will be overwritten by the result as the computation proceeds, leading to a wholly incorrect data structure pointed to by \(a\) at the end.

The paper then defines its terms, with a Smalltalk flavour: all variables are really pointers to objects, and even “primitive” values like integers and booleans are represented with objects. Objects have methods that use instance variables, local variables and parameters, and all parameter-passing is by reference. During the execution of a method, there exists the concept of a “context”, which is the set of “object address values” associated with variables. In other words, a context maps references to their concrete objects.

The paper defines an “access path” as a sequence of variable names that has the overall effect of addressing a particular object; this will be familiar as the \(x.y.z\) notation for “navigating” through nested object structures. Formally, the evaluation of the first variable yields not only a new object, but a context mapping references to instance variables. Within that context, the second variable in the access path is evaluated, etc. Within any method, objects may be accessed through paths rooted at a number of points, including \(\text{this}\) (for an access to an instance variable), a result returned by another method, a method’s formal parameter, a static variable accessible from the method scope or a local variable bound to any of these. An object is aliased with respect to some context if there are two or more routes to it.

A valuable contribution made by the paper is the treatment of aliasing in conceptual terms. Aliasing occurs when one object is accessed through more than one of its possible “roles” in
a program, and aliasing is a problem whenever these roles conflict. A role is analogous to an access path, so that different roles are indicated by different access paths. In the matrix multiplication example, the role of a source matrix will be indicated by the name of the first formal parameter in mult(); the role of a destination matrix will be indicated by the third. For one object to play both source and destination roles simultaneously is clearly dangerous. Yet by passing it to mult() twice, so binding it to the access paths of both the first and third formal parameters, this is precisely what we request.

It has been pointed out elsewhere [IBM] since that the correct pattern for OO methods providing a matrix multiplication service, is for them to return a reference to a new matrix object (instantiated internally, computed accordingly and then returned). This technique would offer a method with the signature Matrix mult(Matrix a, Matrix b), but this still fails to solve the problem of the algorithm itself being sensitive to the remaining arguments being aliased — again, the problem is that the programmer’s notion of the semantics of the operation are not precisely identified by the signature, leading to the caller having an arbitrary and possibly different notion of the its semantics, and that this semantic mismatch may lead to the integrity of the operation being compromised. Taking the Java equivalent of an example from the Geneva Convention, a method f(A a, B b) may indeed find a and b to be aliased if A is a subclass of B, or B a subclass of A. The different classes and names indicates different roles for the parameters, and there is no guarantee that sharing of the role is appropriate at some later stage.

Because of subsumption (“an instance of a subclass is always acceptable in place of an instance of the superclass”), we can write:

```java
class Teacher extends Person { … }
class School {
  void setTutorFor(Person a, Teacher b) { // Person a is taught by b … }
}

Teacher x = new Teacher();
mySchool.setTutorFor(x, x); // A teacher may later find themselves attending their own lesson.
```

This is probably an oversight in the class definition; there should also be a class Student that extends Person, and it is on a Student that setTutorFor operates. Nevertheless, this form of aliasing clearly has the potential to be widespread in OO programs.

[GC] then explains why OO languages suffer from aliasing to a greater extent than do traditional procedural languages. In procedural languages, aliasing is dynamic in that it only lasts as long as the scope of a particular block of code, as aliases disappear when that finishes. (It is assumed that global variables are well-known and aliasing them is unnecessary.) But the ability of an object to encapsulate variables which retain their values between method invocations means that when a method ends, only the dynamic aliases resulting from its parameters and temporary variables go away. New aliases caused by instance variables remain, and this static aliasing lays the path for the problems.

Execution of a method may change the behaviour of a seemingly uninvolved object without even accessing it. This is because methods may use inter-object references to invoke further methods on different objects, in a manner unknown (due to encapsulation) of the original caller. The state of an object is thus the state of the transitive closure of objects referenced by it, and two objects are said to be effectively aliased if their transitive closures have a non-empty intersection, i.e. somewhere further down the object graph (in terms of reachable references), aliasing occurs.

The example is given of a Bank object with two Portfolio objects. Each Portfolio object accepts credit and debit messages (method invocations), and forwards them on to its private
Ownership Types Restrict Aliasing

A Portfolio also has a `transferTo(int amount, Portfolio p)` method for debiting its own account by the amount specified, and invoking `p.credit(amount)`.

Where `p1` and `p2` are Portfolio objects, it is obvious that invoking `p1.transferTo(100, p2)` will not decrease the amount of money in `p1` when it is aliased with `p2`. This may break some invariant but the code in the Bank object making the invocation on `p1` can perform the `p1==p2` comparison for object identity easily enough. But even if `p1` and `p2` are different objects, their Account objects may be aliased, in which case `p1` will still not lose any money. But the Bank cannot check for the aliasing of Account objects, as they really are encapsulated (assuming no methods return them) by `p1` and `p2`. In fact, in Java, the `p1` object can peek `p2`'s private `Account` field, as they are both `instanceof Portfolio` — but this would rarely be the result of a recognised OOA technique.

[GC] concerns itself with the treatment of aliasing for its remainder. These are broadly categorised as:

- **Alias detection** - Determining, at compile-time or run-time, what the potential or actual situation is regarding aliases.
- **Alias advertisement** - Annotations that support detection by declaring aliasing properties of methods.
- **Alias prevention** - Constructs that disallow aliasing in a statically checkable fashion.
- **Alias control** - Where aliasing cannot be determined until runtime, methods that prove the lack, or restriction, of aliasing.

The paper ends with a statement that was to determine much of the direction for research in the coming years: “Component reuse requires adequate description of component behaviour, and this can be only given if components are sufficiently encapsulated for their behaviours to be predictable.”

## 2.2 Alias detection

Well-developed data-flow techniques are available for statically detecting potential or actual aliases via inspection of source code. Knowing where aliases occur is important semantic information that can greatly affect the quality of optimised code, such as the ability of a compiler to perform code motion [IAPP].

Most work has been done to trace aliasing in C and FORTRAN programs. The overall problem is broken down into four classes:

- Intraprocedural May Alias
- Intraprocedural Must Alias
- Interprocedural May Alias
- Interprocedural Must Alias

The complexity of these problems depends on the programming language features assumed present. For example, with single-level pointers and no parameter passing by reference, they are all have polynomial-time solutions [IAPP]; however, even the addition of multi-level pointers makes them all NP-hard. As a middle ground, solving these problems in a language with no pointers but with parameter passing by reference can be done in polynomial time for
Ownership Types Restrict Aliasing

the interprocedural cases – but this is no use for an OO language, because it relies on
pointers or references to objects for the invocation of methods!

Aliasing is usually between code in different scopes (traditionally created by nested
procedure calls; therefore in OOP, persistent instance variables anywhere in a network of
object references potentially store aliases), interprocedural analysis is required and is always
NP-hard for constructs likely to be found in any real OO program worth checking for
aliases.

A related technique [PDPA] partitions the statements of a program into independent sets in
terms of their effects on pointer aliasing. Each set can then be analysed for aliasing
independently and with different analysis algorithms depending on the pointer types features
(single-level, multiple-level, etc). This work is based on C.

Run-time checks are of course possible but frustrated by many factors. Even in a modern
OO language like Java, encapsulation often forbids a single programmer performing relevant
checks (as on Account in the Bank example), and comparing two pointers for identity in
C++ will fail if they point to the same object, but via different classes from its multiple
inheritance hierarchy. In any case, all of this assumes that the class design makes clear to the
programmer that the possibility of aliasing must be considered, and it is usual for this
clarification to be absent.

2.3 Alias advertisement and prevention

The popular programmer’s tool Lint [LINT] has long recognised that aliasing can have
detrimental effects, even in procedural languages where it is phrased as “unexpected sharing
of storage”:

- The /*@only@*/ annotation indicates that a reference is the only pointer to the object
  that it points to (and in C, this brings memory management challenges.) An only
  variable can be passed as an actual parameter corresponding to a function’s formal
  parameter declared with the only annotation, in which case the original reference is a
dead pointer and the storage it points to may not be used.

- The /*@abstract@*/ annotation on structures indicates that they are an ADT and their
  internal representation is not to be exposed unless the client code is in a corresponding
  .c file to the definition’s .h file. The /*@observer@*/ annotation restricts a function to
  read-only access to the ADT structure.

In OOP, the notion of “full alias encapsulation” has existed since 1991. This is the term
given to proposals by Hogg [ISL] and Almeida [BALL] which seek to build a conceptual
“aliasing boundary” around an object’s shadow (the set of objects reachable in an object
from references starting “this.”). Objects outside the shadow are statically prevented from
having references to objects inside the shadow; similarly, objects inside the shadow are
prevented from having references to objects outside it. Within the shadow, aliasing is
unrestricted between objects, and this is also true outside the shadow; it is only across the
aliasing boundary that aliasing is prevented. Both schemes, however, allow dynamic aliases
to cross from an external object into the aliasing boundary for an object. Dynamic aliases are
formal method parameters and local variables that exist on the stack and go out of scope
when their block exits. Restrictions are enforced to ensure that this access does indeed
disappear at the end of a block, and that a dynamic alias is not stored permanently in an
instance variable (which would make it a static alias, permanently available to break the
aliasing boundary). It is not universally considered that this dynamic aliasing is safe [OA].
To move data into the shadow, both systems ensure the uniqueness of a variable passed as a parameter to the interface of the encapsulating object. Islands use destructive reads that nullify the variable as the side-effect of it being read, so that only one reference to it remains in the system. This can be implemented efficiently by delaying physical destruction of the variable in the calling context for as long as possible, but its semantics are unfamiliar to most programmers. Boyland [AK] proposes an alternative approach to unique variables that does not require destructive reads, as an invariant is imposed on the program that existing aliases are dead (out of scope, null or used no further) when the unique variable is read. A unique variable modifier is introduced and the invariant checked statically.

Balloons [BALL] use an alternative approach that copies an actual parameter object as it is passed to the interface of the encapsulated object. Copying actual parameters is familiar as pass-by-value in procedural languages, but is not entirely in keeping with the OO paradigm because frequent copying of objects devalues the notion of object identity. It is also computationally expensive.

[JAC] focuses on access control capabilities for encapsulating object access, albeit with only two levels of control: full access and read-only access. It statically restricts the effects of aliasing by adding a readonly modifier to both instance variables and method signatures. A nontransferable modifier prevents formal method parameters from propagating aliases on as actual parameters in further method invocations, although they can be stored in nontransferable instance variables.

Confined types [CT] are a mechanism similar to full alias encapsulation, in that they restrict external references to objects considered inside a certain protection domain. A confined type is a class or interface definition annotated with the confined keyword, and instances of confined types are called confined objects. The protection domain is chosen to be a Java package, such that instances of confined classes can only be accessed from within a single package. The motivation is that references to confined objects (those instantiated at a confined type) cannot be transferred out of their confining package. Static checking of this invariant is performed using Bokowski's CoffeeStrainer [CS] tool, although link-time support is needed to prevent unchecked programs declaring themselves to be in a package and overriding confined classes within it. Confined types may be declared on a per-class basis, so it is claimed that they are more modular than Balloons, which require whole-program analysis.

### 2.4 The Flexible Alias Protection model

**Rationale**

The aim of Flexible Alias Protection is to support benign uses of aliasing while providing significant protection from aliasing problems. The main conceptual problem with aliasing is that it permits changes to the internal state of an object without that object even being aware of the aliasing occurring, let alone the state change. But the aliasing is technically causing no harm until any of the aliases depend on the state of aliased object; then, it may have changed unexpectedly, via another alias. Therefore, we do not try to restrict aliasing by constraining the references between objects. Instead, we allow aliasing to occur as desired, as long as changes made within aliased objects are invisible to all other aliases. Therefore, OO programs which employ aliasing must do so in ways that avoid critical dependencies on the mutable state of aliased objects.

For example, if the rogue applet from section 1.2 never updated the (mutable) array to which it has a reference, then its reference to the array is of no concern. The other alias – that reference from the system itself – is not being updated in a way that it would certainly not
Ownership Types Restrict Aliasing

Support via its own interface. It should be noted that aliasing is not a problem *as long as the program is designed with aliasing in mind*; then state changes seen in common are expected. For example, the JVM itself will doubtless have a variety of internal aliases for its security arrays, but because of the common encapsulation boundary around all the JVM’s components – they are all working towards implementing the same abstraction of a VM, after all - this sharing is entirely in order. It is only when an “external” object, such as an untrusted applet, attempts to associate with the array that an aggregation’s encapsulation boundary broken. Flexible Alias Protection gives roles to objects so that “external” objects would be those with different roles to objects in the JVM.

In an untyped language, personal discipline simulates the effect of a type system on object access and assignment, and eventually formal discipline evolves into most programming languages. The success and acceptance of a formal type system depends on the extent to which it supports or constrains typical programming styles. Therefore, the notional aim of Flexible Alias Protection is to use techniques similar to type checking to provide guarantees about the occurrence of aliasing in programs, without compromising the use of popular OO idioms.

**Key points**

To protect aggregate objects from undesirable aliasing of its internal state, the members of that state are divided into two categories:

- An aggregate’s *representation* objects are the member variables that should not be accessible from outside the aggregate. An aggregate may freely operate on its representation objects, but its methods must never return them.

- An aggregate’s *argument* objects (or just “arguments”) are the member variables that can be publicly accessed from other objects. Arguments are available and modifiable from outside the aggregate, so the aggregate can conceptually only depend on the immutable part of its arguments’ states. That is, when arguments have public methods, only those that do not update the argument’s internal state can be called by the aggregate. Public methods that read the argument’s internal state (specifically, its own arguments and not its representation) and return it are acceptable.

An individual object can be an argument of more than one container; because none of its containers can cause it to change its value (that would require sending it a message that updated its internal state), there is no problem with all of them aliasing it.

The category into which an individual member variable of a class falls is given by annotating its declaration with an *aliasing mode*. Representation objects are annotated `rep` and argument objects are annotated `arg`. Static type-checking ensures that no variable annotated `rep` is returned from a method that was called from outside the object (i.e. its invocation was not of the form `this.<method name>(...)`). Type-checking also defends against assigning representation objects to arguments, as this would lead to the possibility of them being aliased. It also prevents an argument being assigned to a representation object, as pre-existing aliases would lead to a similar situation.

The aliasing mode of an object’s argument may be supplemented with a *role*. Roles may be considered as partitioning all the arguments of an object into subcategories. Arguments with different roles must be kept separate, so cannot be assigned to one another. A method may only return an argument as a particular role if the argument was passed into the object as the same role.

This example demonstrates these modes.
class Hashtable<Hashable, Item> {
    private Array<HashtableEntry<Hashable, Item>> contents;
    private int size;
    public void put(Hashable key, Item val) { ... }
    public Item get(Hashable key) { ... }
}

The class Hashtable is parameterised by Hashable and Item types, and using a pre-existing
Array container class that is parameterised by the type of entry it supports (HashtableEntry).
A HashtableEntry is itself parameterised by the types it accepts for variables to be stored as
the key of the entry, and the data item of the entry.

This is the class after annotating it with aliasing modes:

class Hashtable<arg k Hashable, arg I Item> {
    private rep
    Array<rep HashtableEntry<arg k Hashable, arg I Item>> contents;
    private val int size;
    public void put(arg k Hashable key, arg i Item value) { ... }
    public arg I Item get(arg k Hashable key) { ... }
}

The contents variable should be private within the Hashtable, and no external access to it
permitted. Although the private modifier suggests this is the case, to fully restrict exposure
of references to contents, we must mark it as part of the representation of the Hashtable.
Furthermore, the HashtableEntry objects should be considered as part of the representation
of the Hashtable, so the HashtableEntry parameter to the Array instance is marked rep.
Then, objects received by the Hashtable’s methods from the Array are automatically “rep of
the Hashtable” and may not be exposed.

Suppose that the objects that the Hashtable will store already exist. An individual
HashtableEntry may therefore not consider the Hashable and Item variables that it stores to
be part of its representation, but it may treat them as its arguments. This could be achieved
by making Hashtable’s Array accept entries of type “rep HashtableEntry<arg Hashable, arg
Item>”. But we wish to ensure that when we create a HashtableEntry object in the methods
of Hashtable, and pass it references to the (externally aliased) key and item objects, it does
not ever confuse the logical roles that each object obviously seeks to fulfil. Therefore, we
annotate the Hashable parameter as arg k, and the Item parameter as arg i. The Hashtable’s
put() method must respect the different roles and not attempt to pass a data item object
where a HashtableEntry conceptually expects a key. To facilitate this, we mark the formal
arg parameters of Hashtable.put() (recall that they are aliased elsewhere) with roles
matching those of the types it is expected to create (HashtableEntry). Then, Hashtable.get()
can be passed an arg k key object and search the array for HashtableEntry objects storing it,
and retrieve from the correct HashtableEntry an argument object of a different role (arg i),
indicating that keys are not being returned. Roles thus permit an outer container can store its
representation in an inner container and retrieve it, sure that the inner container has not
substituted an object of any other role, be it another argument or even part of the inner
container’s own representation.

When a generic type such as Hashtable is instantiated, a sequence of actual moded types
must be bound to a sequence of formal moded types. For example, a Hashtable may be
created by some object that says:

    Hashtable<rep PersonID, rep Person> ht =
        new Hashtable<rep PersonID, rep Person>();
There are two aspects to this instantiation. Firstly, the types passed as actual parameters must match the constraints of extending or implementing Hashable and Item classes or interfaces, as usual. Then, flexible alias protection considers each type parameter with an aliasing mode, and checks that can be bound to the modes of the generic type’s formal parameters. For example, an arg formal parameter can be bound to any actual mode (slogan: “benign aliases can always be made”). Objects that are part of the representation of the object instantiating the Hashtable may be put into the Hashtable, and Hashtable.put() will receive them as argument objects. Within put(), it may alias the objects as it wishes and indeed store them in argument objects of the Hashtable; but it is more likely to instantiate a HashtableEntry object and pass on the objects in a constructor. Note that the object instantiating the Hashtable may not ever receive a reference to the Hashtable’s contents representation object, or to the HashtableEntry objects within that.

Three invariants are formulated to summarise these properties of alias-protected aggregates:

- **No representation exposure** – no dynamic or static references to representation objects should exist outside the aggregate.

- **No argument dependence** – a container may use only those parts of an argument object’s interface that do not rely on its state (i.e. public methods and members that do not refer to local mutable state).

- **No role confusion** – an aggregate container should not return an object in one role when it was passed to the container as another.

There are three other aliasing modes, used less frequently. The free mode indicates that the variable which it annotates holds the only reference to an object in the system, and that that object cannot be aliased. A val–annotated expression is simply a value type, and the mode is introduced so that explicit arg roles are not required for instance of value types. The var mode refers to a mutable object which may be aliased freely, as with Java’s normal reference semantics. A var annotation may optionally be supplemented with a role tag, as for arg, that logically groups mutable references such that references with different roles may not be alias each other.

A clean message (annotated in the method signature) is one made up of only clean expressions, where a clean expression either reads a variable of mode arg or val, or sends a clean message. A clean method cannot modify variables, and the only modes which may appear in it are arg, val or free. The requirement that containers may not rely on the mutable state of their arguments is specifically that they may send only clean messages to their arguments. The effect of these messages is always benign; they either return an instance of a value type (which is no danger to anyone in terms of aliasing), or return a reference to an argument object that by definition is able to be aliased.

**Example**

An example is again given from Hashtable:

```java
public arg i Item get(arg k Hashable key) {
    val int hash = key.hashCode();
    val int index = (hash & 0x7FFFFFFF) % contents.length;
    rep HashtableEntry<arg k Hashable, arg I Item> e;
    for (e = contents[index]; e != null; e = e.next) {
        if ((e.key.hashCode() == hash) && e.key.equals(key)) return e.item;
    }
}
```
A number of checks for the mode correctness of expressions must be made:

- The mode of key is arg, so only clean messages may be sent to it. Many other objects may hold references to this Hashable key object, but they, like the get() method as it executes, can only call methods which do not attempt to update the internal state of the key. hashCode() returns a primitive integer, which always has mode val and there is clearly no danger of side-effects breaking the key’s representation if this happens. (If it did, then hashcode could not have been written).

- The % expression involves propagating the val mode for the hash variable through the arithmetic operations; as these all have mode val (including contents.length), the compound expression is mode val, as expected by the assignment.

- The mode of contents[index] must be determined. The contents array holds elements of type rep HashtableEntry<arg k Hashable, arg I Item>, which are mode rep. The local variable e is also mode rep so the assignment in the first clause of the for loop is mode-compatible (and helpfully, type-compatible but that is not our consideration). e.key, e.item and e.next are not rep of HashtableEntry, so can be referenced by the Hashtable’s methods. We can tell that they declared as arg rather than var within HashtableEntry because e.item must have mode arg i as required by the return type of get() - and this is usual for containers implementing flexible alias protection. (var is described in [OT] as a “loophole” to obtain normal reference semantics when required.)

2.5 Current research directions for ownership

Bokowski used his CoffeeStrainer tool [CS] to implement an early version of ownership types, and published a brief report of his progress [IOTO]. CoffeeStrainer is a tool that allows constraints on the AST (Abstract Syntax Tree) of a program to be expressed in Java, and those constraints applied to user code that expresses its wish to satisfy them (via implementation of certain interfaces). In order to check the representation-related invariants required by ownership types, a program was annotated with aliasing modes in comments, detected when the CoffeeStrainer engine walked the AST. A number of interesting experiences were reported on, such as the fact that programs designed with less care or more design patterns prevented flexible alias protection techniques to some degree.

[DAP] deals with adding explicit notions of ownership into a prototype-based language. Such a language allows objects to define themselves, and not rely on classes or other objects for their definitions. However, once created, a tangled web of inter-object references may lead to undesirable aliasing as with any imperative language. In November 1999, Clarke, Noble and Potter presented a paper “Object Ownership for Dynamic Alias Protection” at TOOLS 99 [OODAP]. This builds on the run-time ownership interpretation mechanism presented by [DAP].

Clarke, Noble and Potter have also focused on developing certain formalisms to assist with abstract manipulation of ownership concepts. By extending Abadi and Cardelli’s object calculus [OC], they are able to reason not only about classes, objects, instance variables and inheritance, but crucially, also about exactly which context (in terms of [OT]) objects interact with. Explicit variables are introduced in the calculus to allow manipulation of the representation context of an object, as distinguished from its ownership context. Integration of these concepts is expected in Java in the coming months, but it is not known what form they will take.

Ownership type inference has been mentioned [WA] as being a potential alternative to programmer-supplied annotations, but it is pointed out that classical Hindley-Milner type
inference does not consider the containment relationship between objects ("nested aggregation").

Similarly, change of ownership is often mooted as being desirable for some applications, at least when dynamic alias protection is considered (there is too much to formalise for the static FAP model at the present time). Since the run-time system has an owner field for each object, that field could be exposed to the programmer for direct assignment. It is not at all clear what language additions this would necessitate, and how it would be checked statically.

Finally, an interesting idea is the visualisation of ownership structures using graphing tools from CASE software [IOO]. As notions of what ownership means develop, particularly by [WA], I believe it is helpful to draw out the desired object structure with graphical annotations for ownership, and graphically verify that it preserves relevant structural invariants.
3. The language JOT – Java with Ownership Types

Ownership was suggested for a minimal language, so in this chapter, I apply ownership types to the full Java language. Decisions about which existing Java language features are orthogonal to the ownership type system, and which are not, are discussed in this chapter.

3.1 Language grammar

The original language featuring ownership was a simple object-oriented language without inheritance:

\[
P ::= \text{defn}^* \\
\text{defn} ::= \text{class } c^{<m>^*} \{ \text{field}^* \text{ meth}^* \} \\
\text{field} ::= t \text{ fd} \\
\text{meth} ::= t \text{ md(decl}* \{ \text{local}* \text{ e} \} \\
\text{decl} ::= t \text{ x} \\
\text{local} ::= t \text{ y} \\
e ::= e; e | x.\text{fd} \begin{array}{|c|c|c|c|c|}
\hline
& \begin{array}{c}
= e \\
\end{array} & x.\text{md(e)} & x = e & \\
\end{array}
\begin{array}{|c|}
\hline
\text{new t} & \text{null} & x \\
\end{array}
\]

\[
t ::= c^{<M|M^*>^*} | \text{int} | \text{char} | \text{boolean} | \text{double} | \text{float} \\
M ::= \text{rep} | \text{norep} | \text{owner} | m \\
\]

c = a class name
m = an ownership context
fd = a field name
md = method name
x = variable name
y = variable name, or this

The enhanced grammar for JOT, with changes or additions in bold, are:

\[
P ::= \text{defn}^* \\
\text{defn} ::= \text{class } c^{<m>^*} \{ \text{field}^* \text{ initializer}^* \text{ meth}^* \} \\
\text{field} ::= \begin{array}{|c|}
\hline
\text{[static]} & t \text{ fd} \\
\end{array} \\
\text{initializer} ::= \begin{array}{|c|}
\hline
\text{[static]} & \begin{array}{|c|}
\hline
\text{e} \\
\end{array} \\
\end{array} \\
\text{meth} ::= t \text{ md(decl}* \{ \text{local}* \text{ e} \} \\
\text{decl} ::= t \text{ x} \\
\text{local} ::= t \text{ y} \\
e ::= e; e | x.\text{fd} \begin{array}{|c|c|c|c|c|}
\hline
& \begin{array}{|c|}
\begin{array}{|c|c|}
\hline
\begin{array}{|c|c|}
\hline
\text{new t} & \text{null} & x \\
\end{array} \\
\end{array} \\
\end{array} \\
\end{array}
\]

\[
t ::= c^{<M|M^*>^*} | \text{int} | \text{char} | \text{boolean} | \text{double} | \text{float} \\
M ::= \text{rep} | \text{norep} | \text{owner} | m \\
\]

c = a class name
m = an ownership context
fd = a field name
md = method name
x = variable name
y = variable name, or this
3.2 Applying ownership types to Java

3.2.1 Visibility modifiers

We now discuss how far the role of visibility modifiers is subsumed by ownership contexts (rep, norep, owner and context parameters). The issue is whether a variable declared at an ownership type also be allowed to feature a visibility modifier keyword (“private”, “protected”, “public” or the default, package). These visibility modifiers seemingly restrict access to the reference based on the caller’s “location”, i.e. outside the current package (for package visibility) or outside the class of the current object (for private). The restrictions are actually based on the type of the object performing the access, since an instance of one class can access the private variables of another instance of that class, regardless of “where” in the system those instances reside. Declaring a variable at an ownership type also restricts references to it on the basis of the caller’s type, i.e. referent’s type, as it is the caller’s type (including rep, norep, owner and enclosing class’ contexts) that via context substitution determines the actual type of the reference in question.

I believe that ownership types replace visibility modifiers and shall try to prove this by discussing the various combinations of ownership and visibility annotations.

Because both visibility modifiers and ownership type annotations both work on the basis of the caller’s type, it is not sensible to try to combine them in the declaration of a single variable. For example, a “rep private” variable would be requiring access to be restricted to the “this” object only (because of rep) and to the other instances of the class (by semantics of private). The obvious thing to do would be to take the most restrictive semantics of the declaration, because by using rep, the programmer clearly intended a desire to strongly restrict access to the variable. In fact, it would be more sensible to remove the private keyword, and this suggests a general rule for coding with ownership types.

Also, “rep public” is not sensible either – the semantics are clearly at odds. And because context parameters actually map, via a sequence of context bindings, to the representation of some object higher in the object graph, using a context parameter with the public keyword is as bad as using rep with public. The meaning of rep is “what private ought to have been”.

With “norep public”, both keywords imply “high-visibility” semantics, and if a mapping from public to norep was suggested, I would be asserting that public visibility by any object is the same as ownership by the system. Since “norep <any non-public modifier>” would be insensible, I am minded to rule out norep public because it simply shows weakness of purpose by the programmer. To summarise, declarations with private visibility should be rewritten as rep, and public as norep.

<table>
<thead>
<tr>
<th>Combinations:</th>
<th>rep</th>
<th>class</th>
<th>context</th>
<th>norep</th>
</tr>
</thead>
<tbody>
<tr>
<td>private</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>protected</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>&lt;package&gt;</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>public</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

I personally uses the default facility of package visibility very widely, mostly in conjunction with private, on the grounds that a class’ own package can be trusted and may have good cause to inspect the state of its own classes’ variables. (Only important constants and special state information needs public variables.)

The question then arises when porting existing Java code to Java with ownership types: in terms of ownership, what does package visibility mean? Who owns a package-visible object? If it is only actually accessed by its enclosing class then rep or private will be applicable, and if wide public access is enjoyed to the object, then public will have been
declared anyway. We are thus only left with norep to signify that the natural semantics of package visibility, i.e. access by other classes in the same package only, are in place – and this has visibility consequences surely unintended by the programmer (or he would have used public). So for a declaration “Object x”, we have no suggestion for what to change the (invisible) package keyword to in terms of owner contexts. Indeed, it can be even worse than that: a declaration “Person<n> x” means an object x which will itself have members owned by the context n, and object x is visible across the package. If a method in a class in the same package gets a reference to the enclosing object of this declaration, then it can access the member variable x even though the n context is meaningless to this other class. Only the absolute requirement for role separation would stop the x object, type Person<norep|x>, being assigned to, for example, a y object, type Person<norep|norep>, as this would obviously introduce role confusion.

The only time there is a paradigm clash is therefore with package visibility. The difference between package and protected visibility is that a protected member of a class can be accessed in subclasses in other packages (in addition to all classes in its package), whereas a package-visible member is absolutely restricted to classes in its package; since we have no inheritance, protected has the same visibility restrictions as package.

To avoid this clash arising, we require that all declarations are changed to have a rep, norep or formal context as their owner. The fact that ownership is really about visibility is what prevents a paradigm clash most of the time. A variant of norep that restricts access to objects within the enclosing class’ package would be helpful, such as the “restricted” keyword proposed by Bokowski. [CS]

### 3.2.2 Static methods, initializers and variables

In ordinary Java, a static method can only instantiate static member variables, as there is no current instance of the enclosing class available to hold references to fields should a static method attempt to instantiate them (or indeed, refer to them at all). Then, should a static method be allowed to instantiate static variables at ownership types? In other words, what ownership characteristics may a static variable possess? Alias protection is declared on a per-object basis, but this does not fit the notion of static variables which are in a sense, “global”.

As well as static methods, the discussion below also holds for static {...} initializer blocks.

**The rep context**

There are two aspects to using the “rep” context statically. Firstly, from the point of view of the caller, i.e. the method referring to the rep-moded static field, the “rep” context is impermissible as it implies the representation of the “current object”. Of course, no objects of the enclosing class need exist when a static method executes or when a static field is accessed, and another familiar side-effect of this fact is that a static method may not use the “this” reference. Hence, rep variables may not be accessed within static methods.

Strictly from the point of view of the callee, i.e. the rep variable itself, it makes no sense for it to be static even if the question of static methods never arises and we guarantee that only non-static methods access the static rep variable. A static variable declared with rep that is instantiated during the execution of a method in one object of the enclosing class, is accessible to the methods of all objects of that class (even if the static member is hidden as much possible with Java’s usual visibility modifiers, i.e. declared private). Thus, the meaning of rep as giving “per-object” protection for references would be inappropriate, as
the static rep-declared variable is by definition open to all instances of the enclosing class. Indeed, this is why static variables are known as “class” variables instead of “instance” variables.

Class context parameters

May a static variable feature context parameters from the enclosing class’ definition? No, for the same reason that the rep context cannot feature - because those contexts only make sense when they are formal contexts being bound to actual contexts in the instantiation elsewhere of an object of this class. The context parameters have no meaning as simple textual identifiers representing owning objects, until the enclosing class is instantiated; a static method outside this restriction of instantiation is therefore ill-placed to refer to context parameters.

And as with the rep variable itself, a static variable declared with context parameters but instantiated even only within non-static methods is available to very many real objects (those of the enclosing class). Consider this:

```java
class A<n> {
    static n Object x;
}
rep A<rep>   a1 = new A();
rep A<norep> a2 = new A();
```

Since the enclosing class for such a static variable may have different actual contexts bound to its formal contexts, it would not do for a static variable to use any of those formal contexts, as that would imply the context “n” above being a) the same and b) accessible for all objects that are instances of class A. Static variables may therefore not feature class context parameters; although such parameters may appear to be “class-based” just like class variables, they are not.

The norep context

The “norep” context represents the root context in which objects are owned by the system. A static member variable declared with an owner of norep and featuring actual context parameters that are all norep, is permissible.

The resulting static variable would be widely accessible across instances of the enclosing class and indeed any class in the system. This may not be appropriate for a static variable previously declared as package-visible, but the requirement above (in the Visibility Modifiers section) requires that package visibility be given up if code inside ownership types is to be able to view (as norep) the static variable.

Technically, if we could guarantee that a static member variable was instantiated only within a non-static method, then the availability of bindings for the formal contexts would be available; the static member variable could then use any formal contexts from the enclosing class. Could we use “rep” if this guarantee held? No. A static member variable declared with rep that is instantiated following the invocation of a method on one object of the enclosing class, is accessible to the methods of all objects of that class. Thus, there is no per-object protection available for the static member, and its enclosing class cannot hide the variable as part of its representation.

If, however, a static member variable is only or also instantiated in a static {...} initializer block, then only the norep context is available for the declaration.
3.2.3 Constructors & instance initializers

Although [OT] specifies that constructors are not supported for instances created at an ownership type, this is only for simplicity. They are permitted in the syntax accepted by the pre-processor. Instance initializers, which take the form `{ ... }` and exist in classes outside methods are also supported, and have essentially the same semantics as constructors.

3.2.4 Arrays (one-dimensional only)

The following syntax is supported:

```java
rep A<m,n>[10] x = new A[10];
```

Each element of `x`, `x[0]` through `x[9]`, has the type `A<rep|m,n>` derived from the array’s type. Dereferencing an array reference to obtain a reference to a particular element yields a reference whose type is identical to a singleton declared with the same ownership type as the array but without the `[..]` construct. Hence, for the array above, this is permitted:

```java
rep A<m,n> anA = new A();
x[0] = anA;
```

Similarly, in method calls, an individual array element may be passed where a formal parameter of the singleton’s type is expected:

```java
... f(x[0], x[1]);
...
void f(rep A<m,n> x, rep A<m,n> y) { ... }
```

Naturally, an array reference itself can also be passed as an actual parameter:

```java
... f(x);
...
void f(rep A<m,n>[] x) { ... }
```

3.2.6 Other constructs

For the implementation, the following facilities are available:

- Declarations of types that do not have an annotation for their owner are taken to be norep, i.e. owned by the system.

- Within a class, the formal context parameters in a class declaration must be unique. Instantiations of the class must specify actual ownership modes (rep, norep, owner or a context parameter) with the same arity as the class declaration.

- Research papers typically require field access to have the form `this.fieldname`. In JOT, this is not required, thus motivating the implementation techniques in section 4.3.

- Method signatures may contain formal parameters with and without context annotations, depending on the assignments of the formal parameters to instance variables in the method.
Ownership Types Restrict Aliasing

• package and import statements are supported as they are unaffected by ownership of individual objects; they are simply ignored in the implementation. The synchronized, wait, notify and notifyAll mechanisms for concurrent programming, and exception handling, are omitted for simplicity.

• We do not keep the syntax that fully qualifies ownership types during object instantiation, i.e. new rep A<rep, norep, m, n>() . There is no benefit to supporting this, since declaring a variable already includes all the ownership type information, and the new() expression cannot feature (at least in [OT]; [OOTO] is different) any ownership type different to that in the declaration.

The following facilities are not available:

• There is no support for overloaded methods. Although

• Type-casts are not supported between ownership types. The natural syntax for casting between ownership types would be rep B<m,n> b = (B<rep|m,n>)v.elementAt(0), where v is an instance of a class like Vector that typically returns Objects. However, [OT] does not deal with casting, so the pre-processor will ignore any casting operation.

3.3 Type rules

The basic type rules in JOT are those from [OT]. However, one issue with them is that the (Type) rule only checks that contexts featured in type declarations (at fields, method formals and local variables) are to be bound in the enclosing class’ signature. Declarations at ownership types need not actually use the same number of context parameters in the declared type as there are featured in the corresponding class signature for that type. For example, given a class definition:

```java
class C<m,n> { … }
```

it may be use in a declaration as:

```java
class X<p,q> {
   norep C<rep,norep,p,q> c;
}
```

Such a declaration should be illegal, but as it used only the contexts rep, norep, and p and q from the enclosing class, it satisfies the \( \vdash \) judgement. I redefine two type rules to rectify this.

A check is introduced in checkClassRule() to warn about this. For each method M in a class C, it performs a pairwise comparison between M and the other methods M_1, M_2, etc, of C. If M and another method M_x are found to have the same number of formal parameters, then pairwise comparison occurs between the corresponding parameters of M and M_x. If all pairs of parameters are found to have, between themselves, the same class names, then the method signatures of M and M_x will be identical to javac. A warning is generated for M (and later, for M_x when it is the “master” method M, compared against other methods including the current M) to this effect.

To facilitate this, another dictionary is added to the environments surrounding the type system, that maps a class name to the object context environment generated by the class definition. This dictionary is called CP (“Context Parameters”), a function of type \( \forall c \in \text{classes}(P) . c \rightarrow \wp(M) \), where classes(P) are the names of defined classes in the program and M is the set of context parameters in their class signatures.
In addition, since ownership constructs will eventually be stripped out of a type-correct program, it is important that overloaded methods do not have signatures that are textually identical when stripped of ownership annotations. For example, `void f(norep A a)` and `void f(rep A a)` feature different ownership types, and section 7.2 describes why it is important to consider the owner and other contexts part of the type. Compile-time binding for method invocations is conceptually possible when the full ownership type is considered; but since the norep and rep annotations will be removed before javac sees the program, the pre-processed Java source will feature (illegally) two identical method signatures.

I redefine the existing (Class) rule such that it performs exactly as before, but in addition to setting up the object context environment $\Sigma$, it also updates the CP mapping when it is applied. Then, the existing (Type) rule is redefined so that it checks the basic structure of a declared type against the signature for the type’s class.

\[
\begin{align*}
\text{(Class)} & \quad P, \Sigma \vdash t_1 : i..n \quad P, \Sigma, \{ \text{this} : c<\Theta | M^*> \} \vdash \text{meth}_k : i..p \\
& \quad P \vdash \text{class} c<\Theta | M^*> \{ t_1 \text{ fd}_1 \ldots t_n \text{ fd}_n \quad \text{meth}_i \ldots \text{meth}_p \}
\end{align*}
\]

where $\Sigma = \{ \Theta \} \cup M^*$ and $\text{CP}(c) = \{ M^* \}$

\[
\begin{align*}
\text{(Type)} & \quad M, M^* \in \Sigma \cup \{ \text{rep, norep} \} \quad \text{CPV}(M^*, c) \\
& \quad P, \Sigma \vdash c<\Theta | M^*> \\
& \quad \text{where CPV}(M^*, c) \iff |M^*| = |\text{CP}(c)|
\end{align*}
\]

The predicate CPV expresses that “Context Parameters are Valid”; this is true iff the cardinality of the non-owner contexts in the declared type matches the cardinality of the formal contexts in the class’ signature.

\[
\begin{align*}
\text{(ArrayType)} & \quad P, \Sigma \vdash e : c<\Theta | M^*>[] \\
& \quad P, \Sigma \vdash e[i] : c<\Theta | M^*>[i]
\end{align*}
\]
4. Implementation

This section presents an overview of relevant tools, describes why JavaCC was selected and demonstrates its capabilities. I then show how the standard Java grammar was extended with ownership constructs, and describe the algorithms and data structures that apply the ownership type rules to check type-correctness. It also lists a range of test programs that demonstrate the supported language constructs and rules.

4.1 Tools & techniques

4.1.1 Available tools

A number of parsers and language toolkits were considered that would allow a) straightforward construction of a parser supporting basic Java syntax, b) extension of the grammar with ownership features, and c) creation of an abstract syntax tree encoding the ownership information. These tools fall into two categories, traditional parser generators creating a (usually Java) parser from a grammar, and more advanced tools.

Traditional parsers

CUP [CUP] is one of the most well-known parsers written in Java, and is essentially an updated version of YACC. It provides a simple syntax for specifying productions and non-terminals, allows arbitrary Java code to be embedded in productions, and produces a scanner and parser implemented in Java. It requires both terminals and non-terminals to be explicitly named and possibly typed (with a Java reference type), in which case the productions must feature arbitrary Java code that assigns an object to a special value indicating the parsed value. The parser is parameterised by a scanner that the programmer must supply, either hand-coded or generated by a third-party tool such as JLex; either way, the scanner must implement a particular interface. Lookahead is controlled through explicitly overriding various methods in the generated parser.

BYACC/Java [BYJ] is an extension of the standard YACC tool with an option to generate the parser in Java rather than in C. As a “drop-in” replacement for YACC, it is valuable in existing environments moving from C/C++ to Java, and produces efficient parsers, but is otherwise similar to CUP (though somewhat less object-oriented in terms of internal design, and hence, less easy to extend).

Another parser generator written in Java was Jack, written in 1996 by Sun. It has since evolved into JavaCC [JCC] and is now supported by a company, Metamata, that produces a range of Java programming tools. JavaCC is extremely feature-rich; it generates a complete scanner and parser from productions that use a syntax barely more complicated than CUP, has syntactic and semantic lookahead available within productions, is well-supported (unlike earlier Java parsers whose authors have moved on), and ships with very many example grammars (which CUP does not).

Advanced tools

A system called CoffeeStrainer[CS] allows constraints on the AST (Abstract Syntax Tree) of a program to be expressed in Java, and those constraints applied to user code that expresses its wish to satisfy them (via implementation of certain interfaces). However, those constraints are included as comments in Java source code, so the direct definition of classes and declaration of variables at ownership types is not possible. In programming aggregate objects that use ownership, it is natural to want to follow [OT]’s examples, with ownership
contexts being embedded directly into source code as extended types, so the CoffeeStrainer methodology is seen as a disadvantage.

Vanilla is described as “the open source language toolkit” [VLA] and allows construction of an interpreter by combining components defining the syntax, type-checking and behaviour of language constructs. It is a very interesting tool for building innovative “toy” languages from scratch, although the fact that its source code is open is of no particular advantage.

Sun’s Forté [SF] is an extensible IDE for Java that features a large number of APIs for accessing both the development environment and the parsed source code. It was evaluated but as might be expected for a complete IDE written in Java 1.2 throughout, it is extremely slow on even fast Pentium III machines.

Another tool is EPP, the Extensible Pre-Processor for Java. [EPP] This advertises itself as a “framework which supports implementation of tools... (like) pre-processors which extend the Java language”. It is essentially a source code processor that supports user-defined plug-in modules that can re-write the AST of a Java source file. For example, one plug-in expands “a (+) b” to “a.plus(b)” (and similarly for other operators), whilst another allows actual parameters which are primitive types to be prefixed with a &, and code is written into the source to support call-by-reference semantics. This approach has appeal for some tasks, but advanced type-checking of complex class definitions seems not quite within the scope of EPP.

The source code of the Java2 development kit, including for javac itself, is also available under Sun’s Community Source Licence. [J2SDK] The actual java type-checker is of course embedded in this kit, but the build process for even the development tools is involved (let alone the VM itself, which is extremely involved!), and working to understand the evolving Java source code itself is a very considerable task.

4.1.2 Tool selection

I selected JavaCC due to the fact that full grammars for Java 1.0.2 and 1.1 are supplied with the tool, and a grammar for Java 1.2 is also now available. JavaCC provides the best base of any tool for starting with the standard Java language and extending specific aspects of it. Not only is it able to produce a customisable AST from parsed code, but the AST that it produces is composed of nothing more than ordinary Java objects linked via references, and each implementing an interface that allows easy navigation up and down the tree. (This facility turns out to be the key implementation technique in the type-checker.)

A variety of other solid reasons supports the choice of JavaCC:

- Its origins at Sun and its continuing development give the tool a perception of quality and reliability that other tools – more recent or more feature-packed – do not have.
- A wide range of example grammars is supplied, from simple languages like HTML and the CORBA IDL, to fully-fledged languages like VHDL and C++ (and of course Java).
- Excellent tutorials are supplied on creating a parser, using lookahead specifications and the programmatic interfaces to the tool’s internals.
- Being written in Java (although only the class files are supplied), it allows development on Java-supporting platforms, notably Windows NT and Linux, with minimal effort when changing environment.
- The Vanilla toolkit that supports easy prototyping of new OO languages in fact uses JavaCC for parsing; writing a language specification for Vanilla is essentially writing a JavaCC-compliant grammar. But the extra abstraction layers that Vanilla needs to
support the building of languages in general means extra complexity when all we wish to do is access extended type information in the AST.

The pre-processor will be developed in Java, for its familiarity to the author, natural fit with JavaCC and cross-platform development capability. Integration with a development environment was not required, due to the novel and emergent nature of ownership types, so only a simple command-line interface to the pre-processor is required.

4.1.3 JavaCC

JavaCC takes as input a .jj file that specifies a grammar, and outputs Java source code that implements a recursive descent parser for that grammar, with lookahead configurable on a per-production. Additional Java classes are created that implement lexical scanning from streams, token management, constants and exceptions.

A grammar file starts with a list of options, followed by a Java compilation unit enclosed between "PARSER_BEGIN(name)" and "PARSER_END(name)" syntax. After this is a list of grammar productions. The name that follows "PARSER_BEGIN" and "PARSER_END" identifies the name of the generated parser, which guides the file and class names of the Java source code written out. In section 4.2, we will choose to call our parser JOTParser via this mechanism, so PARSER_BEGIN(JOTParser) and PARSER_END(JOTParser) will generate:

- JOTParser.java – the generated parser itself
- JOTParserTokenManager.java – the generated token manager and lexical analyzer
- JOTParserConstants.java – useful constants

Between the PARSER_BEGIN and PARSER_END constructs is a regular Java compilation unit (i.e. the contents of a Java file). This may be any Java code as long as it contains a class declaration whose name is the same as the name of the generated parser. Hence, in general, this part of the grammar file looks like:

PARSER_BEGIN(parser_name)
...
class parser_name {
    ...
}
...
PARSER_END(parser_name)

The generated parser file contains everything in the compilation unit, followed by the generated parser code enclosed in the class named after the parser. For the above example, the generated parser will look like:

... 
class parser_name {
    // generated parser is inserted here.
}
...

The generated parser includes a public method declaration corresponding to each non-terminal in the grammar file. Parsing with respect to a non-terminal is achieved by calling the method corresponding to that non-terminal. Unlike yacc, there is no single start symbol in JavaCC - one can parse with respect to any non-terminal in the grammar. A typical application taking whole Java source files (i.e. compilation units) as input would probably
wish to start parsing the CompilationUnit production, as this example from the Java1.1 grammar does:

```java
PARSER_BEGIN(JOTParser)

public class JOTParser {
    public static void main(String args[]) {
        JOTParser parser = new JOTParser(new java.io.FileInputStream(args[0]));
        try {
            parser.CompilationUnit();
            System.out.println("Java program parsed successfully.");
        } catch (ParseException e) {
            System.out.println(e.getMessage());
        }
    }
}
PARSER_END(JOTParser)

void CompilationUnit() :

{}    
[ PackageDeclaration() ]
( ImportDeclaration() )*    
( TypeDeclaration() )*    
<EOF>

The CompilationUnit production exemplifies the vast majority of productions in the grammar, as it is a straightforward BNF production. Another (rare) type of production is the "JAVACODE" production that consists purely of Java code utilising the JavaCC API functions to manipulate tokens in the input stream; this is useful when there is the need to recognize something that is not context-free or for whatever reason is very difficult to write a grammar for. Other types of production define tokens and regular expressions for use in BNF productions.

BNF productions have the following syntax:

```
bnf_production ::=  
java_return_type java_identifier "(" java_parameter_list ")" ":"  
java_block  
"{" expansion_choices "}"
```

Each BNF production has a left hand side which is a non-terminal. The BNF production then defines this non-terminal in terms of BNF expansions on the right hand side. The non-terminal is written exactly like a method declaration in Java. Since each non-terminal is translated into a method in the generated parser, this style of writing the non-terminal makes this association obvious: the name of the non-terminal is the name of the method, and the parameters and return value declared are the means to pass values up and down the parse tree.

There are two parts on the right hand side of an BNF production. The first part is a set of arbitrary Java declarations and code (the "java_block"). This code is generated at the beginning of the method generated in the parser for the non-terminal. Hence, every time this non-terminal is used in the parsing process, these declarations and code are executed, and the declarations are visible to all Java code in actions in the BNF expansions. JavaCC does not process these declarations and code, except to skip to the matching ending brace, collecting all text encountered on the way. Hence, a Java compiler may detect errors in this code even after JavaCC has inserted it into the parser.

The expansion_choices block is a list of possible expansions separated by ‘|’; an expansion is a (possibly singleton) list of expansion units. An expansion unit may be a lookahead
void PrimitiveType() :
{}{
"boolean" | LOOKAHEAD(2)
"char" | LabeledStatement()
"byte" | Block()
"short" | EmptyStatement()
"int" | StatementExpression() ";"
"long" | SwitchStatement()
"float" | IfStatement()
"double"
}ReturnStatement()

void WhileStatement() :
{}{
"while" "(" | ContinueStatement()
Expression() ")" | Statement()
}ThrowStatement()
}TryStatement()

The object token (of class Token) holds the last token consumed by the parser and can be used in parser actions. Methods getToken(int i) and getNextToken() can also be used in actions to traverse the upcoming token list. The Token class provides fields for the type of token (according to generated constants class), the beginning and ending line and column numbers for the token, and its “image” (the textual content of the token; defined immediately for a string literal and otherwise blank, unless set by an action).

JJTree

JJTree is a preprocessor for JavaCC that inserts actions to build an abstract syntax tree at various places in a JavaCC grammar. The output of JJTree is run through JavaCC to create the parser as usual. By default, JJTree generates code to construct AST nodes for each non-terminal in the language. This behaviour can be modified so that some non-terminals do not have nodes generated, or so that a node is generated for a part of a production’s expansion.

JJTree defines and generates a Java interface Node that all AST nodes must implement. The interface provides methods for operations such as setting the parent of the node, and for adding children and retrieving them.

public interface Node {
    ...public int jjtGetNumChildren();
    ...}
JJTree operates in one of two modes. In “simple mode”, each AST node is of concrete type `SimpleNode`; in “multi-mode” the type of the AST node is derived from the name of the node (i.e. of the production). The user may provide implementations for the node classes, or JJTree can generate sample implementations that all extend a generated `SimpleNode` class. We shall prefer “multi-mode”, which generates code such as:

```java
public class SimpleNode implements Node {
    protected Node parent;
    protected Node[] children;
    ...
}

public class ASTCompilationUnit extends SimpleNode {
    public ASTCompilationUnit(int id) {
        super(id);
    }

    public ASTCompilationUnit(JOTParser p, int id) {
        super(p, id);
    }
}
```

User actions within a production can access the node under construction by using the special identifier `jjtThis` to refer to the node. This identifier is implicitly declared to be of the correct type for the node, so any fields and methods that the node has can be accessed. This provides an easy way of storing “metadata” about productions directly into their corresponding node objects. For example, a `lineNumber` field may be added to the `SimpleNode` class and set by the Java action `{ jjtThis.lineNumber = token.beginLine }` in a production. With “multi-mode”, each node class extending `SimpleNode` may have particular fields; for example, the class for type declaration nodes may have a flag indicating whether the type has ownership context parameters.

The `jjtThis` object is also how ordinary programmers gain access to the generated AST. Previously, we simply had the line

```
parser.CompilationUnit();
```

whereas with this `CompilationUnit` production:

```
ASTCompilationUnit CompilationUnit() : {}
{
    [ PackageDeclaration() ]
    ( ImportDeclaration() )*  
    ( TypeDeclaration() )*  
    <EOF>
    { return jjtThis; }

we can store a reference to the root of the AST for a Java file with:

```
ASTCompilationUnit cu = parser.CompilationUnit();
```

This `cu` reference may be used to create a print-out of the AST (via the `dump()` method defined in the base `SimpleNode` class), or the `jjtGetChildren()` method may be used to iterate over the children nodes as there may be below the root node, for example, `ASTPackageDeclaration`, `ASTImportDeclaration` and `ASTTypeDeclaration` nodes.

**Visitor support**
JJTree also provides support for the Visitor design pattern, for friendlier processing of the AST than explicit iteration over children nodes. If the VISITOR option is set to true in the input grammar, JJTree will insert an jjtAccept() method into all of the node classes it generates, and also generate a visitor interface that can be implemented and passed to the nodes to accept, i.e.

```java
public interface Node {
    // Methods from ordinary JJTree, for example:
    public int jjtGetNumChildren();

    // The visitor method
    public Object jjtAccept(JOTParserVisitor visitor, Object data);
}

public class ASTCompilationUnit extends SimpleNode {
    public ASTCompilationUnit(int id) {
        super(id);
    }

    public ASTCompilationUnit(JOTParser p, int id) {
        super(p, id);
    }

    /** Accept the visitor. **/
    public Object jjtAccept(JOTParserVisitor visitor, Object data) {
        return visitor.visit(this, data);
    }
}

public interface JOTParserVisitor {
    public Object visit(SimpleNode node, Object data);
    public Object visit(ASTCompilationUnit node, Object data);
    // Many more visitors for nodes, e.g.
    public Object visit(ASTLocalVariableDeclaration node, Object data);
    public Object visit(ASTReturnStatement node, Object data);
}
```

The name of the visitor interface is constructed by appending Visitor to the name of the parser, e.g. the JOTParser parser generates a JOTParserVisitor interface. A class implements this interface by providing overloaded methods that visit each type of node in the AST. The interface is regenerated every time that JJTree is run, so that it accurately represents the set of nodes used by the parser. It is considered a feature of JavaCC that this causes compile-time errors if the implementation class has not been updated for the new node types!

This code instantiates parser and visitor objects for the AST. The JavaVisitor class must implement the JOTParserVisitor interface, because this is the interface that is known by both the programmer and JJTree when it creates the jjtAccept() method in the Node interface and implementing node classes. Note how to use fields in the JavaVisitor class, we have to manually typecast from JOTParserVisitor back to the interface’s implementing class; otherwise, such fields are not in the JOTParserVisitor interface generated by JJTree.

```java
public static void main(String args[]) {
    FileInputStream fis = new FileInputStream(args[0]);
    JOTParser parser = new JOTParser(fis);
    JOTParserVisitor parserVisitor = new JavaVisitor(fis);
    JavaVisitor visitor = (JavaVisitor)parserVisitor;
    try {
        ASTCompilationUnit cu = parser.CompilationUnit();
        System.out.println("Java program parsed successfully.");
        cu.dump("");
        visitor.setFileBeingParsed(fis.getFilename());
        cu.jjtAccept(parserVisitor, null);
    }
```
4.2 Grammar extension

The first requirement for fully supporting ownership types in Java is extension of the Java1.1 grammar to allow ownership contexts in class definitions and variable declarations. Almost nothing has been removed from the standard grammar; this means that any valid Java program can be run through the type-checker and the tool ensures a default mode of "norep" will apply to unannotated variables.

4.2.1 Modifying type declarations

The standard production for a type is:

```java
void Type() :
{
    ( PrimitiveType() | Name() ) ( "[" "]" )?
}
```

A type either expands to a PrimitiveType() production, or to a Name(); then, the literal array indices "[]" may occur zero or one times. PrimitiveType() is simply the choice of one literal for a primitive type name, whereas Name() signifies an arbitrary string with the <IDENTIFIER> production:

```java
void PrimitiveType() :
{
    "boolean" | "char" | "byte" | "short" | "int" | "long" | "float" | "double"
}
```

The <IDENTIFIER> non-terminal in Name() is defined as standard by the Java1.1 grammar, using a special sort of production for tokens, to be a regular expression consisting of a letter followed by any combination of letters and numbers. (<LETTER> and <DIGIT> are also defined as regular expressions ranging over sets of literal ASCII characters.)

```java
TOKEN :
{
    < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
}
```

Anywhere where the Name() production is used in an expansion, such as in fig. 1, the <IDENTIFIER> production would do just as well; but Name() is easier to read. Of course,
overloading its use in this way has necessitated the grammar authors including a lookahead which will slow down all potential expansion of Name().

void PackageDeclaration() : 
{ 
    "package" Name() ";" 
} 

void ImportDeclaration() : 
{ 
    "import" Name() [ "." '"' ** ' ] ";" 
}

Returning to Type(), it is used in a number of productions:

void FieldDeclaration() : 
{ 
    ( "public" | "protected" | "private" | "static" | "final" | "transient" | "volatile" )* 
    Type() VariableDeclarator() ( "," VariableDeclarator() )* 
    ";" 
} 

void LocalVariableDeclaration() : 
{ 
    [ "final" ] 
    Type() VariableDeclarator() ( "," VariableDeclarator() )* 
} 

Hence, by modifying Type() below, all elements of the grammar benefit from the introduction of ownership type syntax:

void Type() : 
{ 
    String s; 
    Token t; 
} 

Many actions are occurring here:

- For clarity, a production ReferenceType() is added, which is nothing more than an <IDENTIFIER> since class names can still be any arbitrary string.
Both the PrimitiveType() and ReferenceType() productions have a return type. PrimitiveType() is easily expanded to return the name of the terminal matched as a String, and ReferenceType returns the <IDENTIFIER> token that has matched the class name.

A variable can be immediately assigned to an expansion when defining the right-hand side of a production. The space for arbitrary Java code at the beginning of a non-terminal’s definition is utilised to declare variables for this assignment.

The jjtThis object is used to invoke methods storing metadata about the matched Type().

As well as a textual identifier, a type may followed by context parameters.

Metadata is added to the AST node if array indices are matched. This will be inspected by the tool at some stage, since the type-checker will need to know if one instance of an ownership type is an array when the other is not, e.g. in method calls.

When inspecting the AST at a later stage, it is as well to simplify its structure as much as possible. Since it is straightforward to store the underlying type name for an ownership type in the node produced by Type(), there is no need for explicit PrimitiveType() and ReferenceType() nodes in the final AST. The #void annotation on those productions expresses that no nodes of the corresponding class should be instantiated and added to the AST.

The new, simpler, expansions for PrimitiveType() and ReferenceType() are:

```java
String PrimitiveType() #void :
{
    { "boolean" { return "boolean"; } |
    "char" { return "char"; } |
    "byte" { return "byte"; } |
    "short" { return "short"; } |
    "int" { return "int"; } |
    "long" { return "long"; } |
    "float" { return "float"; } |
    "double" { return "double"; }
}
```

```java
Token ReferenceType() #void :
{
    Token t;
    t=<IDENTIFIER> { return t; }
}
```

The new ModeDeclarations() expansion in Type() is simply one or mode Mode() productions:

```java
void ModeDeclarations() :
{
    Mode() { "," Mode() }*
}
```

It is in Mode() that actual contexts are matched:
Another common type of production is the “multiple”: FormalParameters() is simply a sequence of FormalParameter() non-terminals, similarly for NameList() and Name(). ModeDeclarations() is in the same style, and the important production is Mode(). This represents a context parameter, and can be one of four tokens (unsurprisingly, <REP> is defined as “rep”, <NOREP> as “norep”, <OWNER> as “owner”).

The aspect of the new Type() production that might be supposed missing is its owner. Although an ownership type is conceptually c<OM*>", it is declared in Java as “O<OM*>”. It is thought desirable that the owner mode to be optional, defaulting to “norep” if omitted. It will be up to the inspection stage to create this default in whatever dictionaries it manages, by discovering a variable declaration node of some kind (e.g. FieldDeclaration) with a Type() child not preceded by a Mode() child. Because an ownership context may be any identifier, the default lookahead of one token cannot distinguish between “m Object...” and “Object x...” - both immediately match the <IDENTIFIER> token. With the desired syntax for ownership types, therefore, it is necessary to attempt a lookahead beyond the potential owner mode and into the ownership type and the variable name declaration. It is the responsibility of the parent node for the Type() to do this, and for variable declarations, the relevant productions are:

```java
void FieldDeclaration() :
{

};

void LocalVariableDeclaration() :
{

};

void FormalParameter() :
{

};
```
Similarly with MethodDeclaration(), the lookahead must be used to distinguish “m Object f(...)” from “Object f(...).” Therfore, ResultType() is unchanged, using only Type(); the MethodDeclaration() production that uses ResultType() must take responsibility for parsing the optional owner context:

```java
void MethodDeclaration() :
{
    Token t;
}
{
    "public" | "protected" | "private" |
    "static" { jjtThis.addMetadata("IsStatic"); } | "abstract" | "final" |
    "native" | "synchronized")

    [ LOOKAHEAD( Mode() ResultType() <IDENTIFIER> "(" ) Mode() ]
    ResultType()
    t=MethodDeclarator() [ "throws" NameList() ]
    { jjtThis.setInfo(t.image, t.beginLine); }
    ( Block() | ";")
}
```

Type-casts are still permitted for variables declared at non-ownership types, and I wish to ignore it in the type-checker. Therefore, an OriginalType() production is added which is identical to the original Type() production (in the grammar before modification), and it is this production used in the definition of productions involved in type-casting:

```java
void CastExpression() #void :
{
    LOOKAHEAD("(" PrimitiveType()
    "(" OriginalType() ")" UnaryExpression()
    | "(" OriginalType() ")" UnaryExpressionNotPlusMinus()
}
```

**A note on JavaCC constants**

The file JOTParserConstants.java defines constants for each keyword defined as a token in the parser, so the constants JOTParserConstants.BOOLEAN, JOTParserConstants.INT, etc, are available for use. JOTParserConstants.tokenImage[JOTParserConstants.INT] holds the value “‘int’” (the double quotes are part of the value). Through use of the Token.newToken(int kindOfToken) method, the PrimitiveType production was initially configured to return a Token:

```java
Token PrimitiveType() #void :
{
| "boolean" { return Token.newToken(JOTParserConstants.BOOLEAN); } |
| "char" { return Token.newToken(JOTParserConstants.CHAR); } |
...
```

The main problem with this technique is that Token.newToken() merely returns a new instance of the Token class, with no instance variables set. Token.java would have to be changed so that the image field of the returned Token objects were set to the JOTParserConstants.tokenImage[kindOf] value, and excess quotes stripped. It is simpler to merely return a String from the PrimitiveType production, especially since we are really only interested in the name of the type. Hence the following is used:
Ownership Types Restrict Aliasing

4.2.2 Modifying class definitions

[OT] does not consider inheritance (dealt with by a substantially new type system in [OOTO]), and the JOT language does not support it. Class definitions are therefore not permitted to use the “extends ...” or “implements ...” syntax; this is the only time when functionality is removed from the grammar and parser. Class definitions are, however, permitted to feature formal context parameters in the class signature, e.g. class C<m,n> { ... }. The relevant production is UnmodifiedClassDeclaration():

```java
void UnmodifiedClassDeclaration() : {
    Token t;
}
|
"class" t=<IDENTIFIER> [ "<" ModeDeclarations() "]" ]
/* [ "extends" Name() ] [ "implements" NameList() ] */
( jjtThis.setInfo(t.image, t.beginLine); )
ClassBody()
}
```

4.2.3 Shaping the AST

The UnmodifiedClassDeclaration() production is in fact used in the expansion of the simpler ClassDeclaration():

```java
void ClassDeclaration() #void :
{
    ( "abstract" | "final" | "public" )* 
    UnmodifiedClassDeclaration()
}
```

However, the keywords abstract, final and public are of no interest to the type-checker, so a node of type ASTClassDeclaration in the AST would be redundant. For this reason, the #void annotation is used so ensure no class representing a ClassDeclaration node is output by JJTree.

We have now seen two examples of #void (no AST node) productions – PrimitiveType() and ClassDeclaration(). There are many more productions which have no impact on the type-checking; for example, the grammar starts with:

```java
ASTCompilationUnit CompilationUnit() :
{
    [ PackageDeclaration() ]
    ( ImportDeclaration() )* 
    ( TypeDeclaration() )* 
    <EOF> 
    ( return jjtThis; )
}
```

```java
void PackageDeclaration() #void :
{
    [ PackageDeclaration() ]
    ( ImportDeclaration() )* 
    ( TypeDeclaration() )* 
    <EOF> 
    ( return jjtThis; )
}
```
And yet the last three productions merely clutter the tree – so they are voided out. Perhaps the worst “offender” is the Expression production, which is widely applicable as it represents any arithmetic operation. The default hierarchy of nodes for an Expression is:

Expression
   ConditionalExpression
      ConditionalOrExpression
      ConditionalAndExpression
      InclusiveOrExpression
      ExclusiveOrExpression
      AndExpression
      EqualityExpression
      InstanceOfExpression
      RelationalExpression
      ShiftExpression
      AdditiveExpression
      MultiplicativeExpression
      UnaryExpression
      PrimaryExpression | PreDecrementExpression
      PrimaryPrefix
      Literal | AllocationExpression | <IDENTIFIER>
      PrimarySuffix
      AllocationExpression | <IDENTIFIER> | Arguments

The problem is that Expression is the parent of PrimaryExpression which represents field access, object instantiation and method invocation through a combination of its PrimaryPrefix and PrimarySuffix children. But PrimaryPrefix permits any bracketed expression as a choice, i.e. "(" Expression() ")", so the PrimaryExpression (a.b).c will generate the entire hierarchy of Expression-related nodes even though the bracketed expression eventually matches to another simple PrimaryExpression field access. Since it is field access and method invocations we are interested in when checking ownership type-correctness, it would be convenient to do away with the structure induced on the AST by operator precedence; we only need the PrimaryExpression in any Expression. As such, every production from Expression down to PreIncrementExpression and PreDecrementExpression is annotated with #void. PrimaryExpression is then left as the concrete root of any field accesses, method invocations or object instantiations, but PrimaryPrefix and PrimarySuffix themselves are voided. This allows the children of a PrimaryExpression to be <IDENTIFIER> nodes representing field access, Literal nodes for primitive values like integers and string literals, AllocationExpression nodes for new objects and Arguments() nodes for actual parameters should the field access end in a method call.
4.2.4 Adding metadata

Every node in the AST (i.e. non-void production in the grammar) has a name and a line number, extracted from the `token.image` and `token.beginLine` variables as soon as possible after a relevant expansion has been matched.

To supply richer information about the parsed source to the inspection stage, I devised a simple scheme for tagging a node with arbitrary information. The class `SimpleNode` is enhanced with methods `void addMetadata(String data)` and `boolean hasMetadata(String data)`, and these are invoked inside `FieldDeclaration()` and `MethodDeclaration()` when matching access and other modifiers:

```java
    ( "public" | "protected" | "private" | "static" { jjtThis.addMetadata("IsStatic"); } | "final" | "transient" | "volatile" )*
```

Also the Java initializer block:

```java
    void Initializer() :
    {
        { jjtThis.setName("static{}"); jjtThis.addMetadata("IsStatic"); } } Block()
    }
```

The name of the block (which is essentially a constructor) is set to “{ }” as the expansion starts. If “static” is matched, then the name of the node changes to “static{}” and a marker is added to the node’s metadata. This is important because the inspection stage needs to know which variable declarations are static, and also in what environment they were made. A field declaration marked as static and featuring ownership modes other than norep is easily identified, but the type-correctness of variable declarations within methods or initializers depends on whether the method or initializer is static.

4.3 Structure of the pre-processor

The modified grammar file is called `JOT1.1.jjt`, which JJTree processes and rewrites to `JOT1.1.jj`. This latter file is then processed by JavaCC to generate the parser, which we name via the `PARSER_BEGIN` construct in the grammar as `JOTParser`.

A Java package, `jot`, holds all classes concerned with the pre-processor (JOT = Java with Ownership Types). The class featuring a main() method is `JOTMain`, and it can be invoked as:

```java
    java jot.JOTMain <parameters>
```

where parameters includes the filename to be checked and various options. A Perl script front-ends this command, since not all DoC machines have the JDK installed (in which case the script reports the problem) and since program options are more easily parsed in Perl. This script also allows only files with a .jot extension through to `JOTMain`.

```perl
    perl jot.pl <parameters> <filename>
```

Parameters:
JOTMain launches the major stages of pre-processing. It takes responsibility for creating a parser as an instance of class JOTParser and instructing a parse to occur for the input file. It then passes through two stages:

- **Inspection** - [OT] defines various dictionaries and functions used in the type rules, so the AST is walked and information collected on class definitions, method signatures, and the ownership types of fields, formal parameters and local variables.

- **Type-checking** - The AST is walked again and the expressions with defined type rules are checked with reference to the dictionaries.

When the JOTMain program eventually finishes, it exits with a return code of zero if the input file was type-correct, non-zero if there were errors. The `jot.pl` script detects this return code and if zero, invokes another Perl script, `stripjot.pl`, with the filename that `jot.pl` itself was passed. `stripjot.pl` is very simple; it iterates over every line in the named file and removes the rep, norep and owner annotations, and any context parameters.

### 4.4 Pre-processor stage #1 - Inspecting the AST

For inspection to occur, JOTMain needs to create an instance of a visitor to the AST as well as instantiate the parser itself. JJTree defined an interface `JOTParserVisitor` (whose name was derived from the identifier `JOTParser` in the grammar’s `PARSER_BEGIN` option), and it is implemented by our class `JOTVisitor`. It is this class that JOTMain must instantiate:

```java
JOTParserVisitor parserVisitor = new JOTVisitor(filename);
```

// Perform the parse
ASTCompilationUnit cu = parserCompilationUnit();
if (showtree) cu.dump("\n");

// Start inspection
JOTVisitor visitor = (JOTVisitor)parserVisitor;
visitor.status = INSPECTION_STAGE;
cu.jjtAccept(parserVisitor, null);
```

The `JOTVisitor` class is show below. Every class representing an AST node has a `jjtAccept()` method, including the top-level class for a CompilationUnit production. The `JOTVisitor`, implementing the `JOTParserVisitor` interface, is passed as the first parameter to these `jjtAccept()` methods, and the relevant method of the interface is invoked on that parameter object by the AST node. Hence, `JOTVisitor.visit(ASTCompilationUnit node, Object data)` will be invoked first.

As well as the methods necessary for the `JOTParserVisitor` interface, the `JOTVisitor` class also maintains three crucial data structures - `masterClassList`, `fieldDictionaries`, and `methodDictionaries` - declared as `public static` so that important dictionaries (described below) can be accessed from anywhere in the pre-processor.

A global variable, `JOTVisitor.status`, is set to `INSPECTION_STAGE` just before visiting any nodes; it will be set to `TYPECHECKING_STAGE` after inspection has completed.
4.4.1 Data structures

A variety of useful classes are defined in the `jot.utils` package.

The field and method dictionaries in [OT] are implemented directly as classes in Java:

- A FieldDictionary provides basic information about a class’ fields.

- A MethodDictionary maintains three FieldDictionary objects, each of which can, for a given method, give information about either the return type of the method, its formal parameters or its local variables. A variety of accessor methods are supplied to provide quick and helpful access to the contents of these FieldDictionary objects.

Other utility classes include:

- MethodID is an abstraction of a method, since more information must be carried around about a method than simply its name. The class implementing an ownership is one of the most important classes in the system, but its contents are straightforward.

- Type is a very important class that encapsulates known information about the type of a particular declaration.

- The Context class is a simple wrapper for textual context parameters found in class signatures and variable declarations. It is primarily used by the Type class.

- The Variable class is designed simply to associate a textual variable identifier with the Type object representing the ownership type it was declared at. This is useful in the FieldDictionary.
Technically, it would also be possible to build the dictionaries inside the parser itself, utilising the support for arbitrary Java code to be inserted into productions. There are two issues with this: a) it would clutter an already busy grammar file, which is understood most easily when the naturally hierarchial productions are laid out one after the other, and b) any change in the logic for building the dictionaries, or in their content, would require the parser sources to be regenerated and recompiled. Separation of concerns demands that inspection occur in the visitor.

```java
FieldDictionary
String enclosingClass;
Vector fields, types;
void addField(String fd, Type t)
void addField(Variable v)
Type getType(String fd)
Enumeration getFields()
Enumeration getTypes()
boolean hasField(String fd)

MethodDictionary
String enclosingClass
Hashtable mapMethodToReturnType
Hashtable mapMethodToFormals
Hashtable mapMethodToLocals
MethodDictionary(String enclosingClassName)
void addMethod(MethodID md, Type returnType,
FieldDictionary formals, FieldDictionary locals)
boolean hasMethod(String methodName)
Type getReturnType(MethodID md)
Enumeration getLocalTypes(MethodID md)
Enumeration getFormalTypes(MethodID md)
Enumeration getLocalVariables(MethodID md)
boolean hasLocalVariable(MethodID md, String var)
boolean hasFormalParameter(MethodID md, String param)
FieldDictionary getFieldDictForLocals(MethodID md)
FieldDictionary getFieldDictForFormals(MethodID md)

MethodID
MethodID(String name, int lineNum)
String getName()
int getLineNum()
String getSignature()
boolean isStatic()
void setStatic()
boolean equals(String s)
boolean equals(String s, int l)
boolean equals(MethodID m)
String toString()

Variable
Variable(Type t, String name)
void setName(String name)
void setType(Type t)
String getName()
Type getType()
String toString()
```
### Type

- **Type()**
- **Type(String class)**
- **Type(String class, Context owner)**

  - void setLineNum(int n)
  - void setClassName(String class)
  - void setOwner(Context o)
  - void setParameters(Vector v)
  - void setStatic(boolean isStatic)
  - void setValidAsStatic(boolean validStatic)
  - void setArray(boolean isArray)
  - void setPrimitive(boolean isPrimitive)

  - int getLineNum()
  - String getClassName()
  - Context getOwner()
  - Vector getParameters()
  - String getParameterList()
  - boolean isStatic()
  - boolean isValidAsStatic()
  - boolean isArray()
  - boolean isPrimitive()

  - Type resolveDefinition()
  - boolean hasContext(Context c)
  - boolean equals(Type t)
  - boolean equalsWithoutOwnership(Type t)
  - Object clone()
  - String toString()

### Context

- **static Context Rep = new Context("rep")**
- **static Context NoRep = new Context("norep")**
- **static Context Owner = new Context("owner")**

- Context(String name)
- String getName()
- boolean equals(Context c)
- Object clone()
- String toString()
4.4.2 Inspecting classes

The structure of the AST for a simple program is shown below.

```java
class A<m,n> {
    rep Object a;
    m A<m,n> f(rep Object x) {
        x.g();
        m A<m,n> z = new A();
        return z;
    }
}
```

Because type-checking is carried out on a per-class basis, visiting a CompilationUnit node simply means visiting the UnmodifiedClassDeclaration children nodes:

```java
public Object visit(ASTCompilationUnit node, Object data) {
    node.childrenAccept(this, data);
    return done("CompilationUnit", "", 0);
}
```

childrenAccept() is defined in the generated SimpleNode class:

```java
public class SimpleNode implements Node {
    protected Node parent;
    protected Node[] children;

    /** Accept the visitor. **/ 
    public Object childrenAccept(JOTParserVisitor visitor, Object data) { 
        if (children != null) { 
            for (int i = 0; i < children.length; ++i) { 
                children[i].jjtAccept(visitor, data); 
            } 
        }
    }
```
An UnmodifiedClassDeclaration node is the first time real work occurs. It sets an instance variable of the Visitor called “currentClass” to the name of the UnmodifiedClassDeclaration node, which is set in that production in the grammar. Then buildClassDictionary() is passed with a reference to the ASTUnmodifiedClassDeclaration object. A new Type() is instantiated and its name set to that of the currentClass variable and its owner set to “owner”. If the first child of UnmodifiedClassDeclaration is a ModeDeclarations node, then a technique is used to gather up the formal contexts.

A ModeDeclarations (and FormalParameters) node is merely an indicator that multiple instances of a particular child node follow, i.e. Mode (and FormalParameter). We use the ability of the visitor class to accept an arbitrary Object as the second parameter of the visit() methods when called back by the nodes themselves.

The parent node, when visited, creates a Vector object. It then orders its children to accept the visitor object and passes as a parameter to the node, a reference to the Vector. When the children nodes call the visitor back, the Vector reference that the child node received is automatically passed to the visitor (this is a feature of the visitor support of JavaCC). As the visitor visits a child node, it extracts relevant information from the it (which may be complex; a FormalParameter is not unlike a local variable declaration) and then calls addElement on the supplied parameter (the Vector).

When all children nodes have been visited, the parent’s original Vector object has been added to by each one because they all received the same Vector reference.

A utility method, extractModeDecls, is called that visits ModeDeclarations. The visitor for ModeDeclarations uses this technique to produce a Vector of the contexts found in its Mode children nodes.

private Vector extractModeDecls(ASTModeDeclarations node) {
    return (Vector)( node.jjtAccept(this, null) );
}

public Object visit(ASTModeDeclarations node, Object data) {  
    Vector v = new Vector();  
    node.childrenAccept(this, v);  // Iterate over Mode()'s  
    return v;  
}

public Object visit(ASTMode node, Object data) {  
    Context c = new Context(node.getName());  
    ((Vector)data).addElement(c);  
    return data;  
}

Having built a Type object representing the class signature, buildClassDictionary() adds it to important instance variables of the visitor:

masterClassList.put(currentClass, currentType);
FieldDictionary myFields = new FieldDictionary(currentClass);
myFields.addField("this", currentType);
MethodDictionary myMethods = new MethodDictionary(currentClass);
fieldDictionaries.put(currentClass, myFields);
methodDictionaries.put(currentClass, myMethods);
Its final action is to visit the children of the UnmodifiedClassDeclaration, i.e. field and method declarations. The visitors for those nodes will need to know information about their environment:

```java
currentFieldDictionary = myFields;
currentMethodDictionary = myMethods;
node.childrenAccept(this, data);
```

### 4.4.3 Inspecting methods and fields

The visitor for a method, `visit(ASTMethodDeclaration node, Object data)`, simply calls `buildMethodDictionary` immediately, passing a reference to the `ASTMethodDeclaration` object. It will be seen from fig. 2 that a `MethodDeclaration()` can have children of `Mode` `ResultType` `MethodDeclarator` `<other nodes>` since the result type was `m A<m,n>`. The result type of just `A<m,n>` is equally acceptable, and leads to children of `ResultType` `MethodDeclarator` `<other nodes>`; this also occurs if the method returns a primitive or is `void`.

`buildMethodDictionary()` therefore sets a `currentMethod` variable:

```java
MethodID mID = new MethodID(node.getName(), node.getLineNum());
mID.setStatic(node.hasMetadata("IsStatic"));
currentMethod = mID;
```

and then gathers the return type from a utility method, `getTypeFromAST()`. This is responsible for checking the existence of an initial `Mode()` child, and returning a `Type()` object with the correct class name, owner context, formal contexts, primitive/reference status, static/non-static status and array/non-array status.

Having determined its return type, `buildMethodDictionary()` builds a `FieldDictionary` of the formal parameters to the method. After the `Mode` and `ResultType` children of `MethodDeclaration`, a `MethodDeclarator` node gives the name of the method and also has a `FormalParameters` node. A utility method visits the `FormalParameters` node and builds a `Vector` of `Variable` objects from its individual `FormalParameter` children, as described for `ModeDeclarations` with its `Mode` children.

A `FieldDictionary` for the local variable declarations in the method is also instantiated, and a utility method called to populate it. It does this by iterating over the `MethodDeclaration`’s children looking for `LocalVariableDeclaration` nodes, and visiting any found; a reference to the `FieldDictionary` object is passed as the second parameter to the `jjtAccept()` method of the node. When the JOTVisitor’’s `visit(ASTLocalVariableDeclaration node, Object data)` method is called back by the node, the `FieldDictionary` object is extracted from `data` and a utility method, `walkVariableDeclaration`, called to inspect the `LocalVariableDeclaration`. At this point, we only inspect the declaration itself, e.g. “`m Object x`” of “`m Object x = new Object();`”.

A `LocalVariableDeclaration` or `FieldDeclaration` node has an optional `Mode` child, a `Type` child and then one or more `VariableDeclarator` children nodes. Within `walkVariableDeclaration`, the `getTypeFromAST` method is called to extract the full ownership type from `Mode` and `Type`, and then any `VariableDeclarator` children are iterated over and fields added to the `FieldDictionary` passed to `walkVariableDeclaration`. This permits the syntax ““`m Object x.y.z;`””, since `x`, `y` and `z` are separate `VariableDeclarator` nodes. A `VariableDeclarator` has two children – a `VariableDeclaratorId` placeholder for the variable name, and an `Expression` that can be any arbitrary `PrimaryExpression`, i.e. a field access, method invocation or object instantiation. Since the right-hand side of a declaration is ignored at this stage, it is possible to say ““`m Object x = new Object(), y = a, z = y;`”” and the
Ownership Types Restrict Aliasing

assignment, represented as Expression nodes, are ignored and x,y,z gathered as fields as before.

When the visitor for local variables has finished, buildMethodDictionary can associate the currentMethod object with FieldDictionary objects for formal parameters and local variables by adding them to the current method dictionary.

Since fields are children of an UnmodifiedClassDeclaration as well as methods, the visit (ASTFieldDeclaration node, Object data) method may be called back by the AST as well as visit (ASTMethodDeclaration node, Object data). The walkVariableDeclaration utility method is immediately called to build a FieldDictionary for the potential sequence of instance variables all declared at the same type; these fields are added to the currentFieldDictionary object of the Visitor.

4.5 Pre-processor stage #2 – Type-checking

When all visiting of AST nodes has completed, control returns to JOTMain. It updates the status of the system and starts a new round of visiting the AST:

```java
visitor.status = TYPECHECKING_STAGE;
cu.jjtAccept (parserVisitor, null);
```

This again results in invocation of visit (ASTCompilationUnit node, Object data) in the JOTVisitor class. This detects that type-checking rather than inspection is now in progress and invokes checkClassRule on the type-checker object. This checks that the pre-conditions for the (Class) rule hold by a) applying the (Type) rule to types used in the class’ field declarations, then b), with this: <cθ|m*> added to the type environment, applying the (Method) rule to each of the class’ methods. To access the class’ methods, the mechanism used is to visit the children nodes of the UnmodifiedClassDeclaration; these are either FieldDeclaration or MethodDeclaration. If a FieldDeclaration is visited in type-checking mode, we have already checked the declaration itself in the (Type) rule; since we follow Java’s facility to directly assigning to fields, the right-hand side of any assignment must be checked for compatibility with the field.

If a MethodDeclaration node is visited, we immediately set a currentMethod variable in the Visitor and invoke the type-checker’s checkMethodRule method. This checks well-formedness of types used in formals, locals and the method’s result, and then needs to check the body of the method. This means visiting the children nodes of the MethodDeclaration node, and because of shaping the AST, these will either be StatementExpression or ReturnStatement nodes. The visit() method for each of these will inspect the node and invoke the appropriate method in the TypeChecker; due to the complexity of the expressions which can arise from a StatementExpression, section 4.5.2 discusses it explicitly.

In general, at the visit() method for a given AST node, the judgements of the relevant type rule are asserted by mapping them directly to method invocations in the type-checker, which throw appropriately descriptive exceptions if their rule cannot be satisfied. Which methods implement which types rules should be clear from the OMT diagram.

Since there is no actual instance of JOTMain (as only its static main() method is executing), and since we may wish to instantiate a new TypeChecker object for every new source file that is parsed, it is more appropriate to consider an instance of JOTVisitor and an instance of TypeChecker together. For this reason, the instance of TypeChecker is a field of JOTVisitor rather than of JOTMain because the type-checker is only ever invoked by the visitor. The JOTVisitor constructor takes responsibility for instantiating the TypeChecker object.
Ownership Types Restrict Aliasing

The `jot.utils` package is augmented with the following new classes:

- **ObjectContextEnvironment**, providing a direct implementation of $\Sigma$, defined by [OT] as a collection of context parameters.

- **TypeEnvironment**, defined in [OT] as $\Gamma$, providing a mapping from variables to ownership types.

- **Substitution**, modelling the $\sigma$ substitution function defined in [OT]:

<table>
<thead>
<tr>
<th>TypeEnvironment</th>
<th>TypeEnvironment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector contexts</td>
<td>Vector contexts</td>
</tr>
<tr>
<td>void addContext(Context c)</td>
<td>void addContext(Context c)</td>
</tr>
<tr>
<td>void addContexts(Vector v)</td>
<td>void addContexts(Vector v)</td>
</tr>
<tr>
<td>boolean hasContext(Context c)</td>
<td>boolean hasContext(Context c)</td>
</tr>
<tr>
<td>Vector getcontexts()</td>
<td>Vector getcontexts()</td>
</tr>
<tr>
<td>Object clone()</td>
<td>Object clone()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TypeEnvironment</th>
<th>TypeEnvironment</th>
</tr>
</thead>
<tbody>
<tr>
<td>String currentFile;</td>
<td>String currentFile;</td>
</tr>
<tr>
<td>Type currentType;</td>
<td>Type currentType;</td>
</tr>
<tr>
<td>MethodID currentMethod;</td>
<td>MethodID currentMethod;</td>
</tr>
<tr>
<td>int errorsFound;</td>
<td>int errorsFound;</td>
</tr>
</tbody>
</table>

```java
TypeChecker(String inputFile)
void generateError(String filename, OwnershipException oe)
void checkClassRule(JOTParserVisitor jp, String currentClass,
                     ASTUnmodifiedClassDeclaration node)
void checkMethodRule(JOTParserVisitor jp, Environments env,
                     MethodID method, ASTMethodDeclaration node)
void checkTypeRule(Environments env, Type t)
Type checkReturnRule(JOTParserVisitor jp, Environments env,
                     ASTPrimaryExpression node)
Type checkNewRule(JOTParserVisitor jp, Environments env,
                  Type instantiatedType)
Type checkLocalAccessRule(JOTParserVisitor jp, Environments env,
                          String varName)
Type checkFieldAccessRule(JOTParserVisitor jp, Environments env,
                         String fd, ASTPrimaryExpression node)
Type checkFieldUpdateRule(JOTParserVisitor jp, Environments env,
                         ASTPrimaryExpression lhsNode,
                         ASTPrimaryExpression rhsNode)
Type checkMethodCallRule(JOTParserVisitor jp, Environments env,
                        String md, ASTArguments args,
                        ASTPrimaryExpression node)
```

### 4.5.1 Data structures

The `jot.utils` package is augmented with the following new classes:

- **ObjectContextEnvironment**, providing a direct implementation of $\Sigma$, defined by [OT] as a collection of context parameters.

- **TypeEnvironment**, defined in [OT] as $\Gamma$, providing a mapping from variables to ownership types.

- **Substitution**, modelling the $\sigma$ substitution function defined in [OT]:
For convenience when type-checking a class, these environments and a class' field and method dictionaries are encapsulated in an Environments object:

```
Environments
ObjectContextEnvironment sigma
TypeEnvironment gamma
FieldDictionary fDict
MethodDictionary mDict
```

### 4.5.2 Type-checking complex expressions

In the grammar, a StatementExpression is:

```java
void StatementExpression() :
{
    PreIncrementExpression() |
    PreDecrementExpression() |
    PrimaryExpression() { jjtThis.setLineNum(currentLine); } |
    "++" |
    "--" |
    AssignmentOperator() Expression()
}
```

`PreIncrementExpression()` and `PreDecrementExpression()` are both void productions, and reduce immediately to a `PrimaryExpression()`, as does `Expression()`. Therefore, and since the literal "++" and "--" do not have nodes representing them in the AST, `StatementExpression` is simply:

```
PrimaryExpression() [ AssignmentOperator() PrimaryExpression() ]
```

i.e. it allows either a `PrimaryExpression` or an assignment between two `PrimaryExpressions`. `PrimaryExpression` is defined in the grammar as:

```
PrimaryPrefix() ( PrimarySuffix() )*
```

but again, these are void nodes, so the direct children of a `PrimaryExpression` will be the Literal, Name, AllocationExpression, ArrayIndex and Arguments nodes featured below (so there is no difference between the “prefix” and “suffix” of the expression):
void PrimaryPrefix() #void :
{
    Token t;
}
{
    Literal()
    "this" { jjtThis.setInfo("this", token.beginLine); } #Name
    "(*) Expression() "*
    AllocationExpression() { currentLine = token.beginLine; }
    t=<IDENTIFIER> { jjtThis.setInfo(t.image, t.beginLine); } #Name
}

void PrimarySuffix() #void :
{
    Token t;
}
{
    LOOKAHEAD(2)
    "." AllocationExpression()
    "." t=<IDENTIFIER> { jjtThis.setInfo(t.image, t.beginLine); } #Name
    Arguments()
}

Fig. 2 shows the AST structure resulting from common expressions:

<table>
<thead>
<tr>
<th>Field access: a.b.c;</th>
<th>Method invocation: a.b.c(d.e);</th>
</tr>
</thead>
<tbody>
<tr>
<td>StatementExpression</td>
<td>StatementExpression</td>
</tr>
<tr>
<td>PrimaryExpression</td>
<td>PrimaryExpression</td>
</tr>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Assignment: a.b = 5;</th>
<th>Assignment: a.b().c = new X();</th>
</tr>
</thead>
<tbody>
<tr>
<td>StatementExpression</td>
<td>StatementExpression</td>
</tr>
<tr>
<td>PrimaryExpression</td>
<td>PrimaryExpression</td>
</tr>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>Name</td>
<td>Name</td>
</tr>
<tr>
<td>AssignmentOperator</td>
<td>Arguments</td>
</tr>
<tr>
<td>PrimaryExpression</td>
<td>Name</td>
</tr>
<tr>
<td>Literal</td>
<td>Name</td>
</tr>
</tbody>
</table>

If StatementExpression has a single child, then it immediately visits it as it will be a PrimaryExpression. This is also the behaviour of the other potential child of MethodDeclaration, ReturnStatement, which is simply the parent of a single PrimaryExpression node. If StatementExpression is the parent for an assignment, then it
invokes `checkFieldUpdate` on the type-checker object, which will in due course visit the left and right hand sides as PrimaryExpressions in their own right.

Clearly, it is PrimaryExpression that will cause the most work for the visitor and type-checker; if we can type a PrimaryExpression, then any node in the body of a method can be typed.

Given a PrimaryExpression consisting of an arbitrary number of children nodes, the `visit(ASTPrimaryExpression node, Object data)` method checks the class of the last node:

- If it is Arguments, then a method invocation is desired, and the preceding node, a Name, is inspected to discover the name of the method.

- If it is Name, then a variable access is occurring. We do not require the programmer to put "this." before instance variables, so the visitor looks up the variable name as a local variable or formal parameter of the current method. (Using the mDict member of the Environments object.) If it is found, then a local access is occurring; if it is instead found in the field dictionary of the current class (the fDict member of Environments), then a Name node with a value of “this” is physically inserted at the head of the PrimaryExpression.

- If it is a PrimaryExpression, then the programmer has used parentheses around some arbitrary PrimaryExpression, so we simply visit it.

- If a literal occurs, then it is either null or a string/numeric/boolean literal.

It is important to realise that the visit() method for PrimaryExpression is unique amongst all node types handled by the Visitor, in that the value it returns may be of great significance. All the visit() methods are constrained by the JJTree-generated interface to be publicly visible, and return an Object, the visitors for ASTUnmodifiedClassDeclaration, ASTMethodDeclaration ASTFieldDeclaration, ASTReturnStatement and ASTStatementExpression return a simple String indicating that a certain expression type (class, method, etc) has been processed at a particular line number (in case this is ever useful for debugging). But because PrimaryExpression is the root of all object access paths, its visitor may either be being called by a top-level StatementExpression, or be being invoked by itself or the type-checker attempting to type a sub-expression.

For example, the (MethodCall) type rule applies to the expression `e.md(e1...en)`, and the first pre-condition for the rule is that the expression e types to some type t. Naturally, if `e.md(e1)` was represented as an AST, it would be a PrimaryExpression consisting of a Name node (e) followed by a Name node (md) followed by an Arguments node (although with no Argument children). The first time the overall PrimaryExpression is visited, the Arguments node is discovered, so the immediately preceding Name node is checked for the method name (md). The entire PrimaryExpression preceding that name (which happens to be just “e”, a single Name node, in this case – but could be a sequence of Name and Argument nodes) will, at some point, need to be type-checked itself.

In order to have a neat interface between the visitor and type-checker, we wish to perform navigation of the AST only in JOTVisitor, and hand off application of the type rules to the type-checker. The type-checker must be supplied with enough high-level program information to not require further access to the physical tree structure, and encapsulating this information is the purpose of the Environments class. An Environments object is initialised in the `checkClassRule` method and then always passed between visitor and type-checker. So any method in the type-checker needing to type a sub-expression (which is needed in most
rules) must be able to visit a PrimaryExpression node and receive back the type of the subexpression, before proceeding with various ownership-related checks.

The general scheme is therefore to visit a node in the Visitor class and invoke the appropriate method in the TypeChecker class, passing the Environments object. This method applies as much of the type rule as possible using the Environments object, and any large sub-components are themselves visited with the Environments object passed as a parameter.

<table>
<thead>
<tr>
<th>Visitor method</th>
<th>TypeChecker method</th>
</tr>
</thead>
<tbody>
<tr>
<td>visit(ASTCompilationUnit)</td>
<td></td>
</tr>
<tr>
<td>visit children</td>
<td></td>
</tr>
<tr>
<td>visit(ASTUnmodifiedClassDeclaration)</td>
<td>checkClassRule</td>
</tr>
<tr>
<td>checkClassRule</td>
<td>perform immediate checking</td>
</tr>
<tr>
<td>visit children</td>
<td></td>
</tr>
<tr>
<td>visit(ASTMethodDeclaration)</td>
<td>checkMethodRule</td>
</tr>
<tr>
<td>checkMethodRule</td>
<td>perform immediate checking</td>
</tr>
<tr>
<td>visit children</td>
<td></td>
</tr>
<tr>
<td>visit(ASTStatementExpression)</td>
<td></td>
</tr>
<tr>
<td>visit(ASTReturnStatement)</td>
<td>checkReturnRule</td>
</tr>
<tr>
<td>visit children</td>
<td></td>
</tr>
<tr>
<td>visit(ASTPrimaryExpression)</td>
<td>checkNewRule</td>
</tr>
<tr>
<td>checkNewRule</td>
<td></td>
</tr>
<tr>
<td>visit children</td>
<td></td>
</tr>
<tr>
<td>visit(ASTPrimaryExpression)</td>
<td>checkLocalAccessRule</td>
</tr>
<tr>
<td>checkLocalAccessRule</td>
<td></td>
</tr>
<tr>
<td>visit children</td>
<td></td>
</tr>
<tr>
<td>visit(ASTPrimaryExpression)</td>
<td>checkFieldUpdateRule</td>
</tr>
<tr>
<td>checkFieldUpdateRule</td>
<td></td>
</tr>
<tr>
<td>visit children</td>
<td></td>
</tr>
<tr>
<td>visit(ASTPrimaryExpression)</td>
<td>checkFieldAccessRule</td>
</tr>
<tr>
<td>checkFieldAccessRule</td>
<td></td>
</tr>
<tr>
<td>visit children</td>
<td></td>
</tr>
<tr>
<td>visit(ASTPrimaryExpression)</td>
<td>checkFieldUpdateRule</td>
</tr>
<tr>
<td>checkFieldUpdateRule</td>
<td></td>
</tr>
<tr>
<td>visit children</td>
<td></td>
</tr>
<tr>
<td>visit(ASTPrimaryExpression)</td>
<td>checkMethodCallRule</td>
</tr>
<tr>
<td>checkMethodCallRule</td>
<td></td>
</tr>
<tr>
<td>visit children</td>
<td></td>
</tr>
<tr>
<td>visit(ASTPrimaryExpression)</td>
<td></td>
</tr>
</tbody>
</table>

4.5.3 Notes on the implementation of type rules

The implementations for checkClassRule, checkMethodRule, checkFieldAccessRule and checkMethodCallRule are now shown.
public void checkClassRule(JOTParserVisitor jp, String currentClass, ASTUnmodifiedClassDeclaration node) {

    currentFile = ((JOTVisitor)jp).currentFile;

    // Set up environments to be passed down the tree of well-formed judgements
    Environments env = new Environments();
    currentType = (Type) JOTVisitor.masterClassList.get(currentClass);

    env.sigma = new ObjectContextEnvironment();
    env.sigma.addContext(Context.Owner);
    env.sigma.addContexts(currentType.getParameters());
    env.fDict = (FieldDictionary)JOTVisitor.fieldDictionaries.get(currentClass);
    env.mDict = (MethodDictionary)JOTVisitor.methodDictionaries.get(currentClass);

    // We want:   P |- class c<m*> { t1 fd1 ... tn fdn   meth1 ... methp }
    env.gamma = new TypeEnvironment();
    env.gamma.addVariable("this", currentType);

    // Requirement (1)
    // P, S |- tj for j=1..n
    Enumeration e = env.fDict.getTypes();
    while (e.hasMoreElements()) {
        try {
            checkTypeRule(env, (Type)e.nextElement());
        } catch (InvalidTypeException ite) {
            generateError(currentFile, ite);
        }
    }

    // Check that without ownership annotations, method signatures are unique

    // Requirement (2)
    // P, S, {this:c<owner|m*>} |- methk for k=1..p
    node.childrenAccept(jp, env);

    currentType = null;
}
public void checkMethodRule(JOTParserVisitor jp, Environments env, MethodID method, SimpleNode node) {
    // We want:  P, S, G |- t md(t1 x1, ..., tn xn) { t1’ y1, ..., tm’ ym   e }
    // With each new method, comes a new set of environments
    env.gamma = new TypeEnvironment();
    env.gamma.addVariable("this", currentType);
    // Requirement (1)
    // P, S |- t
    try {
        Type methodReturnType = env.mDict ReturnType(method);
        checkTypeRule(env, methodReturnType);
        } catch (InvalidTypeException ite) {
            generateError(currentFile, ite);
        }
    // Requirement (2)
    // P, S |- tj for j=1..n
    Enumeration e = env.mDict.getFormalTypes(method);
    while (e.hasMoreElements()) {
        try {
            checkTypeRule(env, (Type)e.nextElement());
            } catch (InvalidTypeException ite) {
                generateError(currentFile, ite);
            }
        }
    // Requirement (3)
    // P, S |- tl’ for l=1..m
    Enumeration e = env.mDict.getLocalTypes(method);
    while (e.hasMoreElements()) {
        try {
            Type t = (Type) e.nextElement();
            checkTypeRule(env, t);
            } catch (InvalidTypeException ite) {
                generateError(currentFile, ite);
            }
        }
    // Requirement (4)
    // P, S, G[x* : t*, y* : t1*] |- e : t
    // Now that local and formal variables have been typed correctly,
    // add them to the typing environment
    env.gamma.addVariables(env.mDict.getFormalParameters(method));
    env.gamma.addVariables(env.mDict.getLocalVariables(method));
    // Propagate type-checking down to all expressions
    currentMethod = method;
    node.childrenAccept(jp, env);
    currentMethod = null;
}
public Type checkFieldAccessRule(JOTParserVisitor jp, Environments env,
String fd, ASTPrimaryExpression node) {

try {
  // Requirement (1)
  // P, S, G |- e : t
  Type t = (Type) node.jjtAccept(jp, env);

  // Standard sanity checks when typing a subexpression
  if (t == null)
    throw new InvalidTypeException("Unable to type field access");
  if (t.isPrimitive())
    throw new InvalidTypeException("Attempt to access field on primitive");
  if (t.resolveDefinition() == null) return UnknownType;

  // Requirement (2)
  // O = V(env, t)
  Substitution O = V(env, t);

  // Requirement (3)
  // At(fd) = t1
  Type t1 = ((FieldDictionary) JOTVisitor.fieldDictionaries.
    get(t.getClassName())).getType(fd);
  if (t1 == null)
    throw new InvalidTypeException("Field " + fd + " does not exist in " + t);
  if (t1.isPrimitive())
    return PrimitiveType;

  // Requirement (4)
  // SV(e, t1)
  checkStaticVisibility(node, t1);

  // Now we can say: e.fd : O(t1)
  Type t2 = O.applySubstitution(t1);
  return t2;
} catch (InvalidTypeException ite) {
  generateError(new OwnershipException("...");
  return null;
} catch (NoPossibleSubstitutionException npse) {
  generateError(new OwnershipException("...");
  return null;
} catch (SVIsFalseException svife) {
  generateError(new OwnershipException("...");
  return null;
}
public Type checkMethodCallRule(JOTParserVisitor jp, Environments env,
   String md, ASTArguments args,
   ASTPrimaryExpression node) {

   // We want: P, S, G |- e.md(e1, ..., en) : O(t1)
   try {
      // Requirement (1)
      // P, S, G |- e : t
      Type t = (Type) node.jjtAccept(jp, env);
      // Standard sanity checks when typing a subexpression
      if (t == null)
         throw new InvalidTypeException("Unable to type object being invoked");
      if (t.isPrimitive())
         throw new InvalidTypeException("Attempt to invoke method on primitive");
      if (t.resolveDefinition() == null) return UnknownType;

      // Requirement (2)
      // O = V(env, t)
      Substitution O = V(env, t);

      // Requirement (3)
      // M(t) = \langle t^* -> t' \rangle, _,..., _
      Type classSignature = (Type) JOTVisitor.masterClassList.
         get(t.getClassName());
      MethodDictionary mDict = (MethodDictionary) JOTVisitor.methodDictionaries.
         get(classSignature.getClassName());
      MethodID mID = (MethodID) mDict.getMethodID(md);
      Enumeration ti = mDict.getFormalTypes(mID);
      int actualNumOfTypes = args.jjtGetNumChildren(),
         desiredNumOfTypes = countEnumeration(ti);
      if (actualNumOfTypes != desiredNumOfTypes)
         throw new IncorrectParametersException("…");
      int currentArgument = 0;
      ti = mDict.getFormalTypes(mID);
      while (ti.hasMoreElements()) {
         Type literalExpectedType = (Type) ti.nextElement();
         Type substitutedExpectedType = O.applySubstitution(literalExpectedType);
         Type actualType = (Type) args.jjtAccept(jp, env, currentArgument++);
         Type supposedType = (Type) ti.nextElement();
         if (!supposedType.equals(actualType))
            throw new InvalidTypeException("…");
      }

      // Requirement (4)
      // SV(e', t')
      Type returnType = mDict.getReturnType(mID);
      if (!returnType.isPrimitive())
         checkStaticVisibility(node, returnType);

      // Requirement (5)
      // SV(e, t)
      Type return(s) = mDict.getReturnType(mID);
      if (!returnType.isPrimitive())
         checkStaticVisibility(node, returnType);

      // Requirement (6)
      // SV(e, ti) for i=1..n
      ti = mDict.getFormalTypes(mID);
      while (ti.hasMoreElements()) {
         Type formalType = (Type) ti.nextElement();
         if (!formalType.isPrimitive()) checkStaticVisibility(node, formalType);
      }

      // Now we can say: e.md(e1, ..., en) : O(t')
      if (returnType.isPrimitive())
         return return(s);
      return O.applySubstitution(returnType);
   } catch (InvalidTypeException ite) {
      generateError(new OwnershipException("…"));
   }
return null;
} catch (IncorrectParametersException ipe) {
    generateError(new OwnershipException("..."));
    return null;
} catch (NoPossibleSubstitutionException npse) {
    generateError(new OwnershipException("..."));
    return null;
} catch (SVIsFalseException svife) {
    generateError(new OwnershipException(".."));
    return null;
} catch (NoPossibleMethodException npme) {
    generateError(new OwnershipException(".."));
    return null;
}
Auxiliary methods

Other than the methods performing checking of the type rules, the TypeChecker class features a number of methods implementing predicates used by the rules:

Substitution V(Environments env, Type t)
throws NoPossibleSubstitutionException {
// Get context parameters from the ownership scheme
Type classSignature =
(Type)JOTVisitor.masterClassList.get(t.getClassName());
Vector formalContexts = classSignature.getParameters();

// Get context parameters for the actual expression we have
Vector actualContexts = t.getParameters();

if (formalContexts.size() != actualContexts.size())
    throw new NoPossibleSubstitutionException(t.getLineNum(),
        "Unable to create a substitution from "+classSignature+" to "+t);

Substitution s = new Substitution();
s.addMapping(Context.NoRep, Context.NoRep);
s.addMapping(Context.Owner, t.getOwner());
s.addMappings(formalContexts, actualContexts);
return s;
}

void checkStaticVisibility(ASTPrimaryExpression e, Type t)
throws SVIsFalseException {
    boolean eEqualsThis = checkExpressionForThis(e);
    boolean repWithinContexts = checkContextsForRep(t);
    // A -> B  ==  not(A and not(B))
    if (!(!(eEqualsThis & repWithinContexts) == false)
        throw new SVIsFalseException(t.getLineNum(),
            "Static visibility does not hold");
}

checkExpressionForThis() takes an instance of an ASTPrimaryExpression and checks if it is specifically an ASTName containing the value “this”.

CheckContextsForRep() takes an instance of a Type and checks the owner and other context parameters for equality with Context.Rep. The Context.equals() method is used to perform this check.
4.5.4 Error handling

An important aspect of programmers’ tools, including type-checkers, is their ability to report programmer errors accurately. Modelling semantic errors of ownership as exceptions thrown by various phases of the type checker is neat and simple. These exceptions are caught close to their origin in the type checker and form the basis of useful error messages reported to the user. This mechanism saves cluttering the important type-checking methods with conditionals checking the success of various predicates and other pre-conditions for the rule. The exception classes from which exception objects are instantiated are part of the jot.exceptions package.

![Exception Classes Diagram]

But the tendency is always to throw an exception as soon as some out-of-place or unexpected detail is found, and this possible leads to multiple type-checking runs to fix all the errors on even a single line of code. So instead of throwing an exception immediately, flag that one will need to be thrown and add the error’s details to a local list; then pick strategic points within a method implementing a type rule and generate an “aggregate” exception there. Literature has referred to the problem of a left-to-right bias in Algorithm W, although ours is a rather more straightforward situation.

4.5.5 Implementation notes

Because JavaCC generates a large number of .java files for the AST node classes, we would prefer to have them in their own sub-package, jot.parser.ast. To do this, JavaCC supports an option, NODE_PACKAGE, that specifies which package the generated files should declare themselves in. Without NODE_PACKAGE, generated files go into the same package as the generated parser, which is jot.parser. However, there is no way to tell jjtree, which actually creates the .java files for AST nodes, which directory to physically put the files in,
and it creates them in the same directory as the grammar on which it runs. Our JOT1.1.jjt grammar file is in a directory jot/parser, following the package name of the generated parser (jot.parser); but javac does not accept our generated files featuring a "package jot.parser.ast;" declaration being located in the jot/parser directory.

In addition, although it is tidy to have a jot.parser.ast package for the generated .java files and miscellaneous other JJTree-generated support files, some of these classes (e.g. jot.parser.ast.JJTJOTParserState) are not declared as public when generated by JJTree. They would therefore only be visible within their package, jot.parser.ast, and this is not acceptable because JJTree insists on inserting code into JOT1.1.jj (when it creates it from JOT1.1.jjt) — in jot.parser - that accesses JJTJOTParserState. Although we move JOT1.1.jj from jot/parser/ast (there according to NODE_PACKAGE) into jot/parser (to keep the non-AST parser code separate from the AST support code), the JOTParser.java file generated by JavaCC ends up in jot/parser due to being part of package jot.parser. Then, in the main "javac JOTParser.java" stage, jot.parser.JOTParser is unable to access the class jot.parser.ast.JJTJOTParserState, because they're in different packages and JJTJOTParserState is non-public. This stops compilation of the parser.

The effect of this is that JJTree and JavaCC's output must go into the same package/directory! This package is unsurprisingly chosen as jot.parser. (This of course saves having to move, for "neatness", the JJTree-generated JOT1.1.jj, left in jot/parser/ast, up into jot/parser for JavaCC to find it).

4.6 Test plans

The test suite, and a log of the output from the type-checker for that input, is given in Appendix A.
5. Extending ownership types in JOT+*

This chapter explains the novel ownership constructs developed in the course of the project. I suggest two extensions to the type system of the language JOT: anonymous and polymorphic ownership. This chapter gives a rationale for each of these extensions, discusses their implications on the type rules in JOT and develops new type rules. These are incorporated in a new language, JOT+*, that features an augmented grammar to JOT. Changes in the implementation of the pre-processor to support JOT+* are also discussed.

5.1 Anonymous ownership contexts

Fields, local variables and formal parameters typically require that any context parameters used in their declaration are either rep, norep, owner or those formal context parameters mentioned in the enclosing class’ definition (i.e. a, b, c in the case of class C<a,b,c> {...}).

The meaning of an "anonymous context" is that anyone (i.e. any object) can own the method parameter in question, as opposed to the usual situation of formal parameters being either norep or mentioning only those contexts appearing in the enclosing class’ definition. This usual state of affairs is very safe because any method in any class that can type an object A (i.e. it can produce a substitution between the context parameters defined for class A and declared at object A) is able to supply parameters to any method M in A that are guaranteed to be typeable via the context substitution. A method whose formals feature anonymous context brings restrictions upon itself because of its potentially dangerous lack of knowledge of any ownership structure that might arise from the program.

But some methods may wish to receive objects as formal parameters and operate on only some aspects of those objects. (This is closely related to the “owner-different” methods from [WA].)

For example, consider a Student object that holds many important details of the student, including their raw exam marks and the marks after moderation. It also stores various other departmental information as well as a widely-accessible (through norep) final award of degree:

```plaintext
class Student<d, r, m> {
    d DepartmentalRecord deptRec;
    r RawMark[] rawExamMarks;
    m Mark[] moderatedExamMarks;
    norep Grade degreeClass;
}
```

An individual marking a student’s exams need not have access to either their overall departmental record nor their moderated marks (which are indeed not yet know; but the marker should not be able to set them). The class representing people who mark exams, Marker (which would probably extend StaffMember or somesuch in real life), will have a mark() method that accepts, amongst other things, a Student object. But any declaration of a Student reference must feature its owner and three formal context parameters; how can we avoid having to give explicit contexts representing d, r and m to an instance of a Marker?

To remedy this, we propose the introduction of “anonymous” contexts. The use of “_” for a context parameter indicates that we acknowledge there is an owner object involved, but we do not care who it is. Furthermore, we will respect the fact that the owner could be any object in the system, and that that object must still dominate all access to the anonymously-owned object. Access to fields of an object that, through context substitution, are typed to have an anonymous owner, is severely restricted.
Hence, the Marker class may be declared only with context parameters relevant to the actual operations it will perform on objects passed to its methods. The formal parameter for a Student in Marker.mark() can feature "_" for those contexts in Student’s ownership scheme which Marker is not interested or permitted to see. The type rules developed in this chapter ensure that the field accesses shown are type-incorrect or type-correct depending on the contexts used in formal parameters:

```java
class Marker<s, r> {
    public void mark(s Student<_,r,>_ candidate, norep Exam e) {
        ... candidate.deptRec ...  // type-incorrect
        ... candidate.moderatedExamMarks ...  // type-incorrect
        ... candidate.rawExamMarks ...  // type-correct
    }
}
```

### 5.1.1 Anonymous owner

An anonymous context can be used to signify that Y.doWork() takes an object of class A as a formal parameter, but does not care who the owner of that object is. Since an object of A owned by anyone can be passed as an actual parameter, it is not possible or necessary to parameterise class Y with contexts that by their very nature are specific to some individual object. Of course, this restricts what Y.doWork() can do with the instance of class A; not only are statically-declared "rep" members out of bounds as usual, so too are members statically declared as being owned by "owner":

```java
class Y {
    void f(_ A a) {
        a.doSomething();
    }
}
```

In method f(), a.x clearly cannot be referred to. a.z is owned by the owner of the instance of a, which is specifically defined as anonymous, so may not be referred to either. But a.y may well be the only member of interest anyway, and can be accessed by virtue of being owned by norep.

The issue arises that if the owner of formal parameter a is unknown, how can the method body touch the reference at all? Presumably, the caller intends that Y.f() does something useful with the parameter, such as invoking methods on it or accessing norep-owned member variables declared inside it. Consequently, the formal parameter reference can be accessed essentially as if it was norep, but with restrictions as to its use.

#### Restrictions on anonymously-owned objects

It would not be correct to allow a formal parameter with an anonymous owner to be statically aliased (i.e. assigned to a class’ instance variable) by a method body. There is some object that actually owns the object referred to by the formal parameter, and the method is declaring that it does not know who that owner is. Specifically, the method neither knows nor cares about having access to the representation context of the owner object. If the method did want access to that context, then the method’s enclosing class declaration should have been passed the context.

As it is, by quoting _ as the owner, the method gives up all claim on being an interested party to the owner of an object, and hence any instance variables within the formal parameter’s class definition that are owned by "owner". The method is proposing that only a
minimal set of functionality will be extracted from the formal parameter, and the access is such that conceptually, the owner should not mind being circumvented.

5.1.2 Anonymous context parameters

The anonymous mechanism extends to context parameters (those not in the owner position) of formal parameters:

```java
class A<m,n> {
    m Object x;
    n Object y;
}

class X {
    void f() {
        norep A<rep, norep> a = new A();
        norep Y y = new Y();
        y.doWork(a);
    }
}
class Y {
    void g(_ A<_, norep> a) { .. }
}
```

The body of `Y.g()` should clearly be permitted to refer to those instance variables of class `A` which are owned by the second context mentioned in the class’ signature, as that context is bound to `norep` in the formal parameter. If we had:

```java
class X {
    void f() {
        norep A<rep, norep> a = new A();
        norep Y<rep> y = new Y();
        y.g(a);
    }
}
class Y<m> {
    void g(_ A<m, _> a) {
        System.out.println(a.x); // OK
        System.out.println(a.y); // Not permitted
    }
}
```

Then we expect the actual parameter passed to `Y.g()` to feature a context `m` that matches that used in the declaration of the corresponding `Y` instance. As for those instance variables of `A` owned by the second context parameter for the class, we do not care about their ownership so revoke the right to access them by declaring that context as anonymous.

Restrictions on anonymous context parameters

A formal parameter with anonymous contexts implies that the method does not care about the ownership of some instance variables of the parameter object, and hence it cannot safely be allowed to access them. Because even access to such anonymously-owned fields is impossible, so too is assigning one anonymously-owned field to another.
5.1.3 Multiple anonymous contexts

When we write \(A^{m,n}\) as the type of the formal parameter above, it is syntactic sugar for the ownership type \(A^{<\_|m,,>_}\), i.e. we are already dealing with types featuring more than one anonymous context. Hence the reference \(a.x\) in the example below is untypeable, as is \(a.y\).

```java
class A^{m,n}\ {  
m Object x;  
n Object y;  
}

class Y\ {  
void g(_ A^{<,>_} a) \{  
    System.out.println(a.x);  
\};
\}
```

Each anonymous context in a formal parameter of \(Y.g()\) is considered different from all other anonymous contexts in the same method signature. So with:

```java
class Y\ {  
void g(_ A^{<,>_} a1, _ A^{<,>_} a2) \{  
    a1.x = a2.x; // Neither side of the assignment is typeable  
\};
\}
```

The 6 \(_\) contexts are really \(_1, _2, _3, _4, _5, _6\), which are textually distinct and assignment-incompatible with each other.

5.1.4 Anonymous contexts in actual parameters

In the example below, class \(Y\)'s context parameter \(m\) is bound in a higher-level instantiation of \(Y\) to some concrete object's context. The formal for \(Y.g()\) is restricted to an instance of \(A\) that uses that same context to own its member variable \(x\). When \(Y.f()\) passes its formal parameter \(a\), it is claiming that type \(A^{<,|>_}\) is compatible with the type \(A^{<|m>}\) - but clearly, \(f()\)'s reference \(a\) could have the representation context of any object bound to its non-owner context parameter, whereas \(g()\) expects a specific context (precisely which depends on the instantiation of class \(Y\), but there is unquestionably one). There is no requirement in \(f()\) to deal specifically with instances of \(A\) that use the context bound to \(Y\)'s \(m\) to own its \(A.x\) field, and hence we cannot statically decide if the non-owner context of the formal parameter \(a\) in \(f()\) matches the \(m\) context expected by \(g()\).

```java
class A^{n}\ {  
n Object x;  
}

class Y^{m}\ {  
void f(_ A^{<>_} a) \{  
    g(a);  
\}  
void g(_ A^{<m>}_ a) \{ _ \}  
}
```

Conversely, passing an instance of \(A^{<|>_}\) is acceptable where the formal to be bound to is \(A^{<|>_}\). We are losing information, but safely, due to the restrictions then placed on the formal parameter in the invoked method. So passing formals with anonymous contexts to other methods may be permissible, depending on how the actual anonymous contexts relate to the formal contexts.
5.1.5 Instantiating objects with anonymous contexts

Instantiating objects as local or instance variables at ownership types featuring anonymous contexts is not permitted. An anonymous context implies that the owner of a object’s field is unknown, but this is unimportant because such anonymously-owned fields are never accessed. (The case of a method formal with an anonymous owner is different, as explained previously; it ought by rights to be inaccessible, but requiring this would make passing the parameter at all an insensible proposition.)

It might be asked, if the owner of such fields is not accessed, why not just allocate norep as their owner context? (or rep, for that matter) This would mean that at some point deeper in the object graph, two objects might be ownership-type-compatible as a by-product of the bindings resulting from the decision to use rep or norep. On the other hand, by declaring an object with an anonymous owner, it is ensured that those methods to which it is passed must treat it as anonymously-owned, and not statically alias it. This applies to member variables owned by anonymous contexts of the instantiated variable too, e.g.

```java
class A<m,n> {  
  m Object x;
  n Object y;
}

_ A <rep, _> a = new A();
```

As well as invoking methods on object `a`, its `y` member alone may be accessed. Indeed, it seems that such a local variable declaration is indistinguishable from a method formal of the same type; it just so happens that the instantiation is inside our method already rather than being passed in. Our treatment of the object is still restricted with respect to its anonymously-owned (and hence invisible) members.

However, if we allowed instantiations of objects with truly anonymous members, then some methods which are statically legal will lead to role confusion at run-time. It is the case that formal parameter types with anonymous contexts do actually map to some real object in the caller’s environment; there cannot be the concept of a truly unknown owner in a static type system.

In the example below, the formal context `m` in class `Y`’s signature that gets bound to “” is of course usable as a formal context in `Y`’s method signatures (such as in `Y.g()`). When typing with the (MethodCall) rule for method invocations on the `y` object, we would allow any actual parameter’s owner to bind to the `g` method’s formal `m` context.

```java
class Y<m> {  
  m A a1;
  void g(m A a) {  
    a1 = a.z;
  }
}

class X {  
  void f() {  
    rep A a1 = new A();
    norep A a2 = new A();
    norep Y<_> y = new Y();
    y.g(a1);
    y.g(a2);
  }
}
```

When type-checking class `Y` alone, the assignment `a1=a.z` is permissible. `a1` types to `A<m|>` in `g()`, and `C` types to `A<owner|>` with a substitution from the owner context to `m`, i.e. `a.z`’s final type is `A<m|>`. Hence `a1:A<m|> = a.z:A<m|>`
However, varying the behaviour allows us to see that role confusion can result because many contexts in the caller may legitimately bind to the m context in the callee’s (i.e. Y’s) methods. The example below shows how successive calls to \(Y.g()\) store the formal parameter in different instance variables, both of which are legitimately in the ownership context \(m\):

```java
class X {
    void f() {
        rep A a1 = new A();
        norep A a2 = new A();
        norep Y<_> y = new Y();
        y.g(a1);
        y.g(a2);
    }
}

class Y<m> {
    m A a1;
    m A a2;
    void g(m A a) {
        if (a1 == null)
            a1 = a;
        else
            a2 = a;
    }
    m A getA1() { return a1; }
    m A getA2() { return a2; }
}
```

In the caller’s context, we have to type \(Y.getA1()\) and \(Y.getA2()\). In class X, we stated with \(Y<_>\) that any context could be bound to Y’s \(m\) context. It is true that because \(y\) is a local variable in \(X.f()\), the only objects that could possibly end up being passed as actual parameters to \(y\), and hence referred to by \(y.a1\) and \(y.a2\), are those within the scope of \(X.f()\). However, this is not a very interesting observation, because amongst all the contexts visible in \(X.f()\), the object coming back from \(Y.getA1()\) could still belong to any one of them. So to claim knowledge of the specific context coming back from a method of \(y\), when that context is bound to an actual \(_\) context in the caller’s declaration of \(y\), is not permissible. This is similar to what happens when an object featuring members at ownership types, is passed into a Vector and up-cast to an Object – upon retrieval of the object, all information about the owners of its instance variable members has been lost. Here, the instance of \(Y\) is the Vector, taking specifically-owned parameters but then not maintaining the information during internal storage, nor when returning the data later on.

Note how in the situation where we are checking the callee only (i.e. class Y), the methods appear perfectly acceptable; class Y does not know of the situations when any context is acceptable as a partner for its \(m\) context. This is a flaw in the type system, and creating objects explicitly with anonymous contexts is forbidden. It would be possible to allow it, and then for method invocations that seek to bind a specifically-owned actual parameter to an anonymous formal parameter (even though the formal may have a non-\(_\) owner, as \(Y.g()\) does), require that the method does not statically alias that formal parameter. But this means analysing the body of the method more often, and is more complex, so is rejected.

### 5.1.6 Anonymous contexts in return types

A method may not use anonymous contexts in its return type. This is again the theme that losing information is possible, but gaining information by trying to name specific contexts for previously anonymously-owned objects is not permissible.

```java
class A {
    norep Object x;
    rep Object y;
    owner Object z;
}

class Y {
    void g(_ A a) {
        a = h(a);
    }
    _ A doSomething(_ A a2) {
        return a2;
    }
}
```
The question is, to what objects may results featuring anonymous contexts be assigned? The caller of \texttt{Y.g()} must assign a variable to the value of the \texttt{A<\_|>} instance returned by \texttt{g()}, so what is the type of that variable? Since we cannot declare variables at anonymously-owned types explicitly, the only candidates are those method formals we received that use anonymous contexts. The example above uses the convenient reference “\texttt{a}” as being compatible with the return type of doSomething(). But it is not usual programming practice to take a formal parameter and then assign it a value, and being forced to do this to make anonymous returns acceptable, seems incorrect.

```java
class A<m> {
    norep Object x;
    norep A<m> y;
}

class Y {
    void g(_ A<_> a) {
        a.x = h(a);
    }

    norep A<_> h(_ A<_> a2) {
        return a2.x;
    }
}
```

The only variable that \texttt{Y.g()} can return which features anonymous contexts is one that it is received as a formal parameter (since we cannot declare new variables at anonymous ownership types). But \texttt{Y.g()} cannot access anonymously-owned member variables in the formal parameters that it is passed – that is the point of having them anonymous! It can, however, access the \texttt{y} instance variable in class \texttt{A}, which types to \texttt{A<norep|_>} in the scope of \texttt{Y.g()}. If it was to return this, then its return type would be \texttt{A<norep|_> -} and we still have the problem of typing the method call \texttt{h(a)} in the caller, i.e. assigning the method’s result to a variable of type \texttt{A<norep|_>} (or even a type with less information, \texttt{A<\_|>}). This means we are still reduced to assigning some formal parameter of \texttt{g()} to the result of the \texttt{h()} invocation; but a formal parameter with the desired ownership structure is not guaranteed to be available. The point of parameters is that they form part of the interface between caller and callee; setting up caller-visible parameters just to accommodate the method body of the callee is not acceptable. For this reason, return types may not feature anonymous contexts in any location.

Notwithstanding this, whilst it is impossible for a method’s return type to have an anonymous owner, there would be no problem with it featuring anonymous contexts not in the owner position. But still we fall foul of matching the return type in the caller.

5.1.7 Anonymous contexts and subtyping

Because the use of anonymous contexts in non-owner positions has the effect of shrinking the visible portion of a class (\texttt{a.x} is invisible in the last example), we are close to dealing with subtyping, and specifically subsumption, when we consider to what methods may \texttt{A<\_|rep,\_>} be passed (i.e. what formal parameters can \texttt{A<\_|rep,\_>} bind to?)

In addition, anonymous contexts share the transitivity aspect of the “\texttt{arg}” mode from [FAP].
Anonymous contexts and static visibility

rep is permissible as the owner mode for a method’s formal parameter, and this is important because a simple unification of a formal parameter B<rep> and an actual parameter B<rep> will indicate that the types are the same. The type system is able to distinguish between rep-owned variables passed to the method from the this object and from other objects, i.e. work out if the “rep” for the actual parameter signifies a different object to the “rep” for the formal. This is done by checking the Static Visibility (SV) property in the (MethodCall) rule, for all the parameters (and return type) of a method. For the invocation b.f(b) below, although the formal x:Object<rep> can unify with the actual o:Object<rep>, the property SV(b, Object<rep>) does not hold – we are using a type in object x, and this type has “rep” in it; but since that is the representation context of the caller and not the current instance of B, we cannot interpret the type Object<rep>.

```java
class A {
    void d() {
        rep B b = new B();
        rep Object o = new Object();
        b.f(o);
    }
}
class B {
    void f(rep Object x) {
    }
}
```

However, another approach to providing this object-based protection arises with anonymous contexts. Normally, having typed the objects in the access path of the method invocation b.f(), we end up with an expression e:t, which is b:B<rep> here. Then we make a substitution, σ, from the formal contexts of class B to the actual context, which is simply:

\[
\sigma = \psi(B<rep>) = \{ \Theta \rightarrow \text{rep} \}
\]

We then apply this substitution to the types of the formal parameters for the invoked method, i.e. Object<rep>, and by the definition of σ, σ(n) = n if n \( \notin \text{dom}(\sigma) \). So we end up with Object<rep> again, and it is static visibility that realises that in object b, this Object<rep> is not typeable since b \( \neq \text{this} \). But we could realise that b \( \neq \text{this} \) earlier, and if so, add a mapping to the substitution σ that says rep \( \rightarrow _{\text{con}} \). Then, if a formal parameter has a rep mode anywhere as a context parameter, applying the substitution will result in that context parameter being anonymous. In this case, the formal Object<rep> will become Object<_> and unification between that and the actual expression’s type (o:Object<rep>) will “fail”. There is then no need to check the static visibility in the (MethodCall) rule at all.

The use of the anonymous context as a special marker in the formal parameter’s contexts is claimed to be a valid use for it, since in this situation, it is indicating that a mode of the formal parameter is unknown (of course there is one, but it is anonymous) at the time of the method call – and since the object access path to the method is not “this”, it is precisely the case that the rep context of the invoked method’s object is unknown.

It must be said that this scheme leads to slightly obtuse error reports: because the physical substitution for the formal parameter turns a Object<rep> into a Object<_>, passing a Object<rep> results in an error message to the effect that the the formal parameter type Object<_> cannot be unified with the actual parameter type Object<rep> - but an inspection of code will never reveal the former type in the method declaration. It would be up to the programmer, or a more intelligent type rule, to recall that the problem is the mapping from one rep to another, or not an incompatible pair of types.
5.1.9 Type rules

(Type+)

This is as (Type), but allows the _ anonymous context. This will be used in judgements about method formals.

(New+)

The current (New) rule uses the $\overline{t}$ judgement as the pre-condition for being able to type the "new $t$" expression. Because this permits only rep, norep and context parameters from class declarations to feature in objects instantiated at an ownership type, this is what we require when anonymous contexts exist. Since they cannot be instantiated at, using the $t+$ rule in the pre-condition would be a mistake.

(Class+)

This is identical to (Class), except that it uses the $\overline{m}$ judgement to type method definitions. As with (New+), it uses the $\overline{t}$ judgement for typing field declarations, as anonymous contexts are not permitted within them.

(Method+)

The only difference between (Method) from [OT], and (Method+) here, is that we allow method formal parameters to type according to the (Type+) rule, rather than (Type).

(FieldAccess+)

We show the process of applying the usual (FieldAccess) rule to the code below.

```
class A<a,b,c> {
    a X x;
    b Y y;
    c Z z;
}
class B<m,n> {
    void doWork(m A<_, n, _> a) {
        a.x...
    }
}
```

(FieldAccess)

$\text{P, } \Sigma, \Gamma \vdash e : t$

$\sigma = \psi(t)$

$A'(fd) = t'$

$SV(e, t')$

- $e = a : A<m | _, n, _> \text{ by lookup of object } a \text{ (formal parameter) in the type environment.}$
- $\sigma = \psi(A<m | _, n, _>) = \{ \Theta \rightarrow m, a \rightarrow _, b \rightarrow n, c \rightarrow _ \}$
- $A'(X) = t' = X<_|>$
- $SV(a, X<_a>) \text{ holds}$

So can we type $a.x : \sigma(t') = \sigma(X<_a>) = X<_|>$? This is clearly unacceptable; the a.x instance variable is shown to be anonymously owned, so access to it should not be allowed. In the type system, we must consider the field access untypeable if application of the $\sigma$ substitution results in an anonymously-owned object. This can achieved with a predicate similar to $SV$, called $AV$ ("Anonymous Visibility") -

"$AV(t, \sigma)$ holds if by applying the substitution $\sigma$ to type $t$, the owner context of the resulting type is not anonymous"

$AV(t, \sigma) \iff \text{owner}(\sigma(t)) \neq _$
The clause $AV(t', \sigma)$ must be added to (FieldAccess+) as a pre-condition.

(\textbf{FieldUpdate+})

Again, we apply the standard (FieldUpdate) rule to the code below.

\begin{align*}
\text{P, } \Sigma, \Gamma \vdash e : t & \quad \sigma = \psi(t) \quad A'(fd) = t' \quad SV(e, t') \quad \text{P, } \Sigma, \Gamma \vdash e' : \sigma(t') \\
\text{P, } \Sigma, \Gamma \vdash e.fd = e' : \sigma(t')
\end{align*}

Naively, we obtain an ownership-type-compatible assignment of $a1.x : X<\text{norep}_1>$ and $a2.x : X<\text{norep}_2>$. But because of the basic incompatibility between the _ context in $a1.x$ and the similar-looking _ context in $a2.x$, we do not wish this assignment to succeed.

If we imagined that all _ contexts in the method signature had subscripts, then substitution would produce $a1.x : X<\text{norep}_1>$ and $a2.x : X<\text{norep}_2>$ which are more obviously incompatible. Alternatively, we shall simply say that for two ownership types to be compatible, not only must their owners both be non-anonymous ("specific"), but all their non-owner context parameters must be specific too. Then, the "syntactically identical" rule for context comparison comes into play as usual, to check that contexts are statically compatible. We could actually get by with the existing rule if we had to compare $X<\text{n}>$ against $X<_>$ for example, because $n$ and _ are clearly textually distinct. But to handle assignments of $X<_>$ to $X<_>$ (or rather, to disallow such an assignment), we need to impose this constraint of "no anonymous contexts on either side of the =". What we are actually trying to formulate is the idea that "an anonymous context is never equal to any other context, anonymous or specific".

We formulate the NAC ("No Anonymous Contexts") predicate:

"NAC(t, \sigma) holds if by applying the substitution \sigma to type t, there are no _ contexts anywhere in the resulting type, either in the owner position or in non-owner positions".

\begin{align*}
\text{NAC}(t, \sigma) & \iff _ \notin \text{contexts}(\sigma(t))
\end{align*}

NAC(t', \sigma) is required in (FieldUpdate+) so that $\sigma(t')$ does not use any _ contexts. Then, if $\sigma(t')$ (which is the type of the left-hand side) features a _, the assignment is untypeable; if it does not feature a _, then it is up to the right-hand side to type to it (and by implication, the right-hand side will not succeed if it features a _).

(\textbf{MethodCall+})

With the following code:

\begin{verbatim}
class B<a,b,c> {
  a X x;
  b Y y;
  c Z z;
}

class A<m,n> {
  void f(m B<_> b) {
    // ...
  }
}
\end{verbatim}
this.g(b1);
);
void g(m B<_ , _, _> b2) { }
}

Apply the standard (MethodCall) rule:

\[
\begin{array}{c}
\Gamma \vdash \sigma = \psi(t) \\
\Gamma \vdash \sigma = \sigma(t_i) \quad i:1..n \\
\Gamma \vdash \sigma = \sigma(t_i) \quad i:1..n
\end{array}
\]

\[
P, \Sigma, \Gamma \vdash e : t
\]

\[
\sigma = \psi(\sigma(t) \rightarrow \sigma(t'))
\]

\[
M'(md) = (B\langle m|_,_,_\rangle \rightarrow \text{void}), .., >
\]

\[
P, \Sigma, \Gamma \vdash e: (B\langle m|_,_,_\rangle \rightarrow \text{void}), .., >
\]

\[
\sigma = \psi(A\langle \Theta|m,n\rangle) = \{ \Theta \rightarrow \Theta, m \rightarrow m, n \rightarrow n \}
\]

\[
\Gamma \vdash x : c<N | N*>
\]

We now require that \( b_1 : \sigma(B\langle m|_,_,_\rangle) = B\langle m|_,_,_\rangle \). But \( b_2 \) is actually of type \( B\langle m|_,n,_\rangle \) - can it be claimed that it has the type \( B\langle m|_,_,_\rangle \)? Since \( b_1 \) is a formal parameter of \( f() \), it was added to the type environment by the (Method) type rule, so we apply the (LocalAccess) rule to judge if \( b_1 : B\langle m|_,_,_\rangle \). We must modify the rule to permit \( b_1 \) to be looked up in the type environment as \( B\langle m|_,n,_\rangle \), and then check acceptably against a judgement of \( b_1 : B\langle m|_,_,_\rangle \). If \( b_1 \) was not a local variable (or formal parameter), then a) it could not feature anonymous contexts and b) the ordinary (FieldAccess) rule would be applicable.

We take the function \( \psi \) and extend it to \( \psi' \), which takes two ownership types as parameters and returns a substitution function \( \sigma' \). The types are of the same class \( c \), but have different owner and context parameters. \( \sigma' \) is a mapping between these context parameters:

\[
\sigma' = \psi'(c\langle M | M*> , c\langle N | N*\rangle) = \{ M \rightarrow N , M_1 \rightarrow N_1 , ..., M_n \rightarrow N_n \}
\]

where \( M* \) and \( N* \) are lists of context parameters \( M_1..M_n \) and \( N_1..N_n \).

Just as the \( \sigma \) substitution maps an ownership scheme to an ownership type, so the \( \sigma' \) substitution maps formal parameters’ contexts to actual parameters’ contexts, for some method invocation. For example, if we pass a variable of type \( B\langle m|n\rangle \) to a method with formal parameter type \( B\langle x|_\rangle \), the \( \sigma' \) function is \( \{ x \mapsto m , \_ \mapsto n \} \). Passing such a variable is acceptable, because we always have specific contexts in the co-domain of \( \sigma' \), and essentially throw away information inside the invoked method by unifying \( \_ \) with these specific \( m \) and \( n \) contexts.

But if we pass a variable of type \( B\langle _,n\rangle \) to a method expecting a formal \( B\langle x|_\rangle \), then \( \sigma' = \{ x \mapsto _ , _ \mapsto n \} \). Having an anonymous context in the co-domain of \( \sigma' \) that is bound to a specific context indicates that the actual parameter cannot be typed according to the formal parameter’s type; the formal parameter demands a specific context which the actual parameter cannot supply.

Consequently, the rule’s judgement holds that \( x : c\langle N | N*\rangle \), the type of the formal parameter, if this mapping is produced and is valid. This requires that a specific context in a formal parameter is bound to a specific context in the actual parameter. If \( _ \) appears in the formal parameter, then we have no preference for a specific or an anonymous context in the actual parameter.

\[
\begin{array}{c}
x \in \text{dom}(\Gamma) \\
\Gamma(x) = c\langle M | M*> \\
\sigma' = \psi'(c\langle N | N*\rangle , c\langle M | M*\rangle)
\end{array}
\]

\[
P, \Sigma, \Gamma \vdash x : c\langle N | N*\rangle
\]

\[
\Gamma(x)\vdash c\langle M | M*\rangle
\]

\[
\Gamma\vdash \sigma' = \psi'(c\langle N | N*\rangle , c\langle M | M*\rangle)
\]

\[
\Gamma\vdash \text{Valid}(\sigma')
\]

\[
P, \Sigma, \Gamma \vdash x : c\langle N | N*\rangle
\]

\[
\Gamma\vdash c\langle M | M*\rangle
\]

\[
\Gamma\vdash \sigma' = \psi'(c\langle N | N*\rangle , c\langle M | M*\rangle)
\]

\[
\Gamma\vdash \text{Valid}(\sigma')
\]

\[
P, \Sigma, \Gamma \vdash x : c\langle N | N*\rangle
\]

\[
\Gamma\vdash c\langle M | M*\rangle
\]

\[
\Gamma\vdash \sigma' = \psi'(c\langle N | N*\rangle , c\langle M | M*\rangle)
\]

\[
\Gamma\vdash \text{Valid}(\sigma')
\]

\[
P, \Sigma, \Gamma \vdash x : c\langle N | N*\rangle
\]
Ownership Types Restrict Aliasing

\[
\text{Valid}(\sigma') = \forall x \in \text{dom}(\sigma) \ [ x \neq _- \Rightarrow \sigma'(x) \neq _- ]
\]

In fact, we can also pass instance variables as actual parameters where the formal parameters expects anonymous contexts, e.g.

```java
class A {
    norep B<norep, rep, norep> b;

    void f() {
        this.f(this.b);
    }
    void g(m B<_, _, _> b2) { }
}
```

This means that when we try to type \( e : \sigma(t_i) \) in (MethodCall), to check the actual parameter according to the anonymously-contexted formal, we will apply the (FieldAccess) rule. Thus the unification of formal contexts with actual contexts is a shared requirement of both (LocalAccess) and (FieldAccess). It is thus incorporated in a new type rule that encapsulates the idea that we say a variable has a particular type \( X \) if it has a declared type \( Y \) and \( Y \) features more specific context information than \( X \). (i.e. \( Y \) is a better type than \( X \).)

\[(\text{BetterType})\]

\[
P, \Sigma, \Gamma \mid e : c<M | M*> \quad \sigma' = \psi'(c<N | N*> , c<M | M*>) \quad \text{Valid}(\sigma')
\]

The appropriate type rule can then be applied for the first clause, and find the type of the actual parameter either from the type environment (for a local access), or by context substitution and static visibility checking (for field access). With this actual type \( c<M | M*> \) discovered, we can check its validity against the formal parameter’s requirement for a type \( c<N | N*> \); essentially checking if \( c<N|N*> \) can subsume \( c<M|M*> \). The (MethodCall+) rule simply uses (BetterType) to type the actual parameters for the method invocation; the fact that the formals must be of types that are statically visible in the caller (i.e. do not feature the callee’s rep context) is still in force.

\[(\text{MethodCall+})\]

\[
P, \Sigma, \Gamma \mid e : t \quad \sigma = \psi(t) \quad M'(md) = (t^* \rightarrow t'), .. . >
\]

\[
P, \Sigma, \Gamma \mid e_i : \sigma(t_i) \quad i:1..n \quad \text{SV}(e, t^*) \wedge \text{SV}(e, t_i) i:1..n
\]

\[
P, \Sigma, \Gamma \mid e/md(e_1, ..., e_n) : \sigma(t')
\]
Ownership Types Restrict Aliasing

(Type+)

\[ M, M^* \in \Sigma \cup \{ \text{rep, norep, _} \} \quad \quad \text{CPV}(M^*, c) \]

\[
P, \Sigma \vdash t : M \quad c < M | M^* >
\]

where \( \text{CPV}(M^*, c) \iff | M^* | = | \text{CP}(c) | \)

(Class+)

\[
P, \Sigma \vdash t_j : 1..n \quad P, \Sigma, \{ \text{this : } c < \Theta | M^* > \} \vdash \text{meth}_k : 1..p
\]

\[
P \vdash \text{class } c < M^* > \{ t_1, \ldots, t_n \quad \text{fd}_n, \text{meth}_1, \ldots, \text{meth}_p \}
\]

where \( \Sigma = \{ \Theta \} \cup M^* \) and \( \text{CP}(c) = \{ M^* \} \)

(New+)

\[
P, \Sigma \vdash \text{t}
\]

\[
P, \Sigma, \Gamma \vdash \text{new t : t}
\]

(Method+)

\[
P, \Sigma \vdash t
\]

\[
P, \Sigma, \Gamma \vdash t_j : 1..n
\]

\[
P, \Sigma \vdash t'_j : 1..m
\]

\[
P, \Sigma, \Gamma \{ x : t, y : t' \} \vdash e : t
\]

\[
P, \Sigma, \Gamma \vdash \text{md}(t_1, x_1, \ldots, t_n, x_n) \{ t'_1, y_1, \ldots, t'_m, y_m \quad e \}
\]

(FieldAccess+)

\[
P, \Sigma, \Gamma \vdash e : t
\]

\[
\sigma = \psi(t)
\]

\[
A'(fd) = t'
\]

\[
\text{SV}(e, t') \quad \text{AV}(t', \sigma)
\]

\[
P, \Sigma, \Gamma \vdash e : t \quad e.\text{fd} : \sigma(t')
\]

(FieldUpdate+)

\[
P, \Sigma, \Gamma \vdash e : t
\]

\[
\sigma = \psi(t)
\]

\[
A'(fd) = t'
\]

\[
\text{SV}(e, t') \quad \text{NAC}(t', \sigma)
\]

\[
P, \Sigma, \Gamma \vdash e : t \quad e.\text{fd} = e' : \sigma(t')
\]

(BetterType)

\[
P, \Sigma, \Gamma \vdash e : c < M | M^* >
\]

\[
\sigma' = \psi(c < N | N^*> , c < M | M^* > )
\]

\[
\text{Valid}(\sigma')
\]

\[
P, \Sigma, \Gamma \vdash e : c < N | N^*>
\]

(MethodCall+)

\[
P, \Sigma, \Gamma \vdash e : t
\]

\[
\sigma = \psi(t)
\]

\[
M'(md) = < (t^* \rightarrow t'), \ldots, >
\]

\[
P, \Sigma, \Gamma \vdash e_i : \sigma(t_i) \quad i : 1..n
\]

\[
\text{SV}(e, t') \land \text{SV}(e, t) \land 1..n
\]

\[
P, \Sigma, \Gamma \vdash e.\text{md}(e_1, \ldots, e_n) : \sigma(t')
\]

MEng. Computing, Imperial College
5.2 Polymorphic ownership contexts

Usually, a class can be declared with one or more context parameters that are used in field and local variable declarations within the class. Instantiations of the class can pass ownership contexts in existence at the point of instantiation, allowing a rich ownership structure to be set up that parallels the direct references held by objects.

A polymorphic ownership context, written *, **, ***, etc, may be attached to any formal parameter and indicates that the method is able to operate safely on objects whose ownership characteristics do not necessarily match up with those formal context parameters declared in the class header.

Consider a class Garage that provides an inspect() method for any member of the public to have their car quickly checked. The formal parameter in inspect() allows the actual parameter at invocation to be owned by anyone, and the method performs an innocent operation on it.

```java
class Garage<m> {
    void inspect(* Car<**> c) {
        if (c.engine.getAge() > 10) c.engine.refillOil();
    }
}

class Car<n> {
    rep ServiceHistory sh;
    n   Engine engine;
}
```

Without the * and ** contexts, a context from Garage’s class signature would have to specify an owner for inspect()’s parameter. Specifying norep may not be appropriate, and specifying rep would limit the potential actual parameters to be field or local variables within the current object (the object on which inspect() is being invoked). Since inspect() requires access to the Car’s engine field to invoke methods on it, using an anonymous context is not sufficient. Garage’s formal context parameter m may well be for unrelated fields and refers to the ownership context of a specific object somewhere higher in the object graph; that object need not be the owner of the actual parameter to inspect().

Indeed, the Garage may wish to replace the engine entirely, in which case it could declare an object of class Engine owned by *, and assign that to c.engine. It may then seem that representation containment is broken; but the new Engine object is owned by the same object’s context as the previous object. There is thus no change in the ownership structure of the owner of the engine field of a Car and the engine object itself.

With a polymorphic ownership context, we are not claiming that, conceptually, we have no idea who the owner is, as is the case with anonymous contexts. Rather, we claim that the owner is a particular object whose representation context may merely have not been passed down the object graph, such that no formal context parameter for the class of the object being sent a message is going to match this representation context.

Naturally, polymorphic contexts may be used as both the owner mode for a formal parameter, and as formal context parameters needed by the class of a formal parameter, i.e. * B<**, ***> is an acceptable type for a formal parameter, where class B<m,n> is defined somewhere.

We may, as with templatised classes from C++, wish to specify that multiple ownership contexts own the formal parameters to a method. As such, each ownership context appearing in a formal parameter may use a different *-form context, with as many *’s as necessary to
differentiate the contexts: *, **, ***, ****, *****, etc. Note that this does mean that each
ownership context must be different; only that they may be different. This is analogous to
the results of type inference on this code:

```plaintext
let
twice = ( fn f => fn x => f (f x) )
in
let
  inc = (fn y => y + 1)
in
(twice inc) 3
```

The twice function takes a function (f→inc) and an integer (x→3), and applies the function
to the integer, then again to the result of the application. The type of twice is naively:

`(*→**) → (**→**)`

The first part is the type of the function taken as a parameter (*→**), and this is turned into
a new function that is the equivalent of applying the function twice (i.e. twice inc 3 == inc
inc 3). The new function takes as input, the variable x, which may supposedly be a third type
***. We do know that the new function returns the same type as the original function, **.

Because x must be applicable to the function f on its own, and f has type *→**, x must have
type * . The type of twice is then (**→*)→(**→*). And because the result of applying
the function once produces a value whose type is acceptable as the input for the second
invocation, it must be that the return type ** is the same as the input type.

Therefore, we have typed `twice` as `(*→*)→(*→*)` – a function from * to * is transformed
into a function from * to *. A compiler not performing this depth of type analysis may
conclude that different types are expected to fill the various parameter positions to twice,
when in fact some of those types may be the same.

The analogy is that a number of polymorphic ownership parameters (*, **, etc) may be
textually different, but refer to the same object in the object graph. Here, both
`y` and `z` members of the instance of B received by `A.f()` are owned by norep, but the author of
`A.f()` considers that they may have different ownership.

```plaintext
class A {
  void f(* B<**, ***> b) {
    System.out.println(b.y);
    System.out.println(b.z);
  }
}

class B<m, n> {
  m Object y;
  n Object z;
}

A a = new A();
rep B<norep, norep> b = new B();
a.f(b);
```

### 5.2.1 Unification for polymorphic contexts

Below, the `A.f()` method is privileged in that the actual parameter unified with the formal `b`
is within the representation context of the some higher level object. Furthermore, `f()` is
oblivious to the fact that its ** context is the representation context of some other object, so
we must ensure that it does not “leak” references in a way that would break the
representation containment property (“all access paths to an object pass through its owner”).
Ownership Types Restrict Aliasing

This means imposing restrictions on methods with formal parameters featuring polymorphic contexts; but they are unsurprising restrictions. (If a method was aware that its caller was passing actual parameters declared with rep, then it could relax the restrictions, but at the cost of breaking modularity of type-checking.) Because a polymorphic context might be the representation context of some object (in fact, it is almost certain to be one since it is either that or norep), we cannot assign a *-owned object to a **-owned object, for all *, **. This is essentially the property of role separation, which is dealt with later.

There is an additional constraint we can impose specifically upon the caller of a method. This is that the distinct contexts of actual parameter objects must be bound consistently to distinct polymorphic contexts in formal parameters. The code below shows rep being bound to * and norep to ** for the first parameter of \( f() \), and norep bound to ** and rep to ** for the second parameter. Binding rep to both * and ** is clearly nonsensical, since it allows the construction of object \( r : B<*|**> \) which references the \( f2.z \) object owned by the method’s caller; combined with norep binding to ** in the caller, this allows \( b3 \) to expose this representation context.

5.2.2 Using polymorphic contexts

Polymorphic contexts are assignment-compatible with one another iff the “*” contexts are identical on both sides of the expression. An assignment of \( b1 : B<* | **> \) to \( b2 : B<* | **> \) is acceptable:

```java
class B<m> {
    m Object z;
}

class X {
    ** B<**> f ( * B<**> f1, ** B<**> f2 ) {
        ** B<**> r = new B();
        r.z = f2.z;
        return r;
    }
}

rep B<norep> bl;  
norep B<rep> b2;  
X x;            
norep B<norep> b3 = x.f(bl, b2);
```
Whatever the actual contexts are that here are denoted by * and **, it is the case that \( b_1 \) and \( b_2 \) are both owned by the same object, and the \( x \) fields of the \( b_1 \) and \( b_2 \) objects are also owned by the same object. It is possible that the owner of \( b_1 \) and \( b_2 \) is the same object that owns \( b_1.x \) and \( b_2.x \), i.e. that contexts * and ** are actually the representation context of the same object, but this cannot be guaranteed in general. To preserve the role separation property, the polymorphic contexts are assignment-incompatible. This code shows the result of ignoring this:

```c
rep B<norep> a1;
rep B<rep> a2;
class A {
    void swap(* B<<*> b1, * B<***> b2) {
        b1 = b2;
    }
}
```

\( b_1 : B<* | **> \) is assignment-incompatible with \( b_2 : B<* | ***> \), because the ** context featured in \( b_1 \) is incompatible with *** in \( b_2 \). If the assignment was allowed to proceed, then \( b_1.x \), which is owned by norep according to the declaration of \( a_1 \), would end up referring to the rep-owned \( b_2.x \) object – and if \( b_1 \) was returned to the caller, then the rep-ness of \( a_2 \)'s \( x \) field will be subverted via \( a_1.x \).

If \( a_1 : B<rep | norep> \) had been declared as \( B<rep | rep> \), then even though ** and *** both mean the representation context in \( a_1 \) and \( a_2 \)'s enclosing object, this fact is hidden during analysis of the \( swap() \) method alone. Unsurprisingly, for safety, the incompatibility of distinct polymorphic contexts is assured.

### 5.2.3 Instantiating objects with polymorphic contexts

There is no problem with instantiating objects to be owned by a * context featured in a formal parameter, and using polymorphic contexts as actual context parameters in an object declaration. These instantiations, however, must not persist outside the scope of execution of the method. However, since polymorphic contexts are only available to local variable declarations within methods, and not for field declarations within the class body, it is not possible to dynamically alias a polymorphically-owned object anyway, as the left-hand side of such an assignment (i.e. an instance variable without * contexts) would never be able to match the type of the right-hand side (a formal parameter or local variable with * contexts).

The fact that a method may feature *, ** and *** context parameters does not mean that a **** context exists (or ***** or ******, etc). In fact, these “-*forms*” of polymorphic ownership may be viewed as merely a labelled form of anonymous contexts (\(_1\) \(_2\) \(_3\) are simply written as *, **, ***), where although the physical object representing the ownership context is unknown, each context is still known to exist. The difference between anonymous and polymorphic context is that “less” information is considered to exist about the anonymous context – two objects that are owned by an anonymous context cannot be assigned to each other, but two objects owned by the same *-form polymorphic context are assignment-compatible.

In this sense, we have direct access to another object’s representation context, able to instantiate new objects and assign them to fields in that object as long as we have a *-form polymorphic context available for that field.
Another interpretation of contexts *, **, ***, etc, is that they are aliases for the actual ownership contexts; ironic then, since we seek to control aliasing of variables, that aliasing of ownership is allowed to occur freely! (Of course, these “aliases” to ownership occur only during the invocation of a method and hence only “statically” alias it. This is traditionally seen as a controlled form of aliasing, in contrast to “dynamic” aliasing that continues after a method returns via references on the heap, and to which Java is particularly susceptible.)

5.2.4 Polymorphic contexts in actual parameters

Once a method has received formal parameters with polymorphic ownership, it may pass them as actual parameters in its own, deeper, method invocations as long as the receiving method itself expects polymorphically-owned formal parameters. Naturally, a * context in an actual parameter passed to a method is not the same as a * context featured in a formal parameter of that method, and the usual rules apply for preventing role confusion within the receiving method.

This example shows the formal \( b_2 : B<** | *> \) unified with \( b_1 : B<* | **> \). There is no importance attached to the fact that the owner of \( b_2 \) is ** and its context parameter is *; within \( g() \), they are merely placeholders for the true ownership contexts. As with anonymous contexts, polymorphic ownership is transitive: given \( B<*|> \) that is polymorphically owned, and a binding of \(* \rightarrow **\), then \( B<**|> \) is also polymorphically owned.

```java
class A {
    void f(* B<**> b1) {
        this.g(b1);
    }

    void g(** B<*> b2) {
    }
}
```

It is not possible to bind actual parameters whose ownership is polymorphic, to formal parameters whose ownership is specified through concrete ownership parameters, because there is no guarantee that the de-facto owner of actual parameters corresponds to the physical contexts bound to the concrete ownership parameters of the receiving object.

There is no problem binding actual polymorphic ownership contexts to formal anonymous contexts; as usual, the potentially different owners of the actual parameters and their members are lost when all are “anonymised” to the simple _ ownership context. The other way around – passing anonymously-owned objects to formal parameters featuring polymorphic ownership – is not permitted. It might be supposed that the _ context could map to a single * context, and as long as all anonymous contexts in the actual parameters mapped to the same * context in the formals, then there would be no ownership conflict. But of course, it may well be that each instance of the _ context actually represents a different owner, whereas multiple instances of a * context demand the same owner object for each.

In other words, the _-form provides for mapping multiple physical contexts into a single symbolic context (i.e. anonymising them); the *-form provides for multiple symbolic contexts (*, **, ***, ... ) representing the same physical context.

A method using polymorphic contexts in its formal parameters is able to use them quite freely, and if such contexts are in its return type, then the caller of the method is constrained in the type of the reference receiving the returned object. With anonymous contexts,
however, the restrictions are all on the invoked method, because of the very many restrictions imposed on formal parameters featuring anonymous contexts.

5.2.5 Polymorphic contexts in return types

Polymorphic ownership contexts are permitted in return types. As with templates in C++, any polymorphic scheme for the formal parameters may be adopted, but polymorphic contexts used in the return type may only be those found in formal parameters. Suppose:

```cpp
template<class T> { ... }
```

Then the following C++ function is acceptable in the template definition:

```cpp
T max(T x, Ty) {
    if (x>0) return x; else return y;
}
```

Generally, the actual type returned must be able to be inferred from the unification of actual types to type parameters T and U, so

```cpp
template<class T, class U> {
    T f(T x, U y) { ... }
}
```

is permitted, but

```cpp
template<class T, class U> {
    T f(U y) { ... }
}
```

is not.

Analogously, because the binding of formal polymorphic contexts to actual parameters’ contexts is known for the method call, then as long as the polymorphic contexts returned by a method are featured in that binding, they can be successfully bound back to physical contexts in the caller. This code is legal as the binding applies correctly to the return type:

```cpp
class A {
    * B<*> f ( * B<*> b1, ** B<*> b2 ) {
        * B<*> b = new B();
        return b;
    }
}
```

A a = new A();
rep B<norep> b1 = new B();
norep B<rep> b2 = new B();
rep B<rep> b3 = a.f(b1, b2);

In addition, the declared type of the variable which receives the result of this method invocation, must be mapped to a type consistent with the bindings from actual to formal contexts that occurs when parameters are passed.
5.2.6 Type rules

(Type*)

This is like (Type+), but allows polymorphic contexts instead of anonymous contexts. The CP and CPV (“Context Parameters Valid”) definitions from anonymous contexts are used.

(New*)

Unlike (New+), it is possible to use to polymorphic contexts in instantiations, as long as that instantiation is for a local variable. (New*) supplies this judgement, and will be used in the updated rule for typing methods.

(Class*)

This is identical to (Class) and (Class+), as it uses the judgement to permit fields to use only contexts from the enclosing class definition, not polymorphic (or anonymous) contexts.

(Method*)

This incorporates the requirement that polymorphic contexts in formal parameters totally determine the vocabulary of polymorphic contexts that may feature in a method’s return type. A polymorphic contexts dictionary, PCD, is created that for each method, lists the permissible polymorphic contexts (featured in formal parameters) available to appear in the return type and local variable declarations:

\[
\text{PCD}(md) = \bigcup \{ \text{PC}(t_j) : j:1..n \}
\]

PC(t) returns the polymorphic contexts featured in an actual type t, e.g.

\[
\text{PC}(A<*|m,**,n,***>) = \{*, **, ***\}
\]

(FieldAccess*)

Because polymorphic contexts are considered just like the textual formal contexts appearing in the class header (class A<msg, n>), the (FieldAccess) rule remains the same. Consider typing expressions b.y and b.z in the code below:

```java
class A {
    void X (* B<**, ***> b) {
        ... b.y ...
        ... b.z ...
    }
}

class B<m, n> {
    m X y;
    n X z;
}
```

Apply the standard FieldAccess rule:

\[
P, \Sigma, \Gamma \vdash e : t \quad \sigma = \psi(t) \quad \Lambda'(fd) = t' \quad SV(e, t')
\]

- \(e = b : B<* | **, ***>\) by lookup of b (formal parameter) in the type environment
- \(\sigma = \psi(B<* | **, ***>) = \{ \Theta \rightarrow *, m \rightarrow **, n \rightarrow *** \}\)
- \(\Lambda'(y) = t' = X<m>\) \quad \(\Lambda'(z) = t' = X<n>\)
• SV(b, X<m|>) holds \quad SV(b, X<n|>) holds

So can we type b.y : σ(t') = σ(X<m|>) = X<**|> and b.z : σ(t') = σ(X<n|>) = X<***|> ?
Indeed so; we were passed the *** context along with the formal parameter, and fully expect that the relevant members of the parameter object are accessible. This is in contrast to typing to an anonymous context, which means that access should be denied to a parameter object's member.

(FIELDUPDATE*)

Just as an instance of type A<m|h,o> is compatible only with an instance of A<m|j,h,o> because of the exact textual equivalence of context parameters, so too is A<*>m,***> compatible only with A<*>n,***>. Therefore, no change needs to be made to (FIELDUPDATE) from [OT].

(METHODCALL*)

A mechanism was developed for anonymous contexts whereby a mapping was created between the contexts of formal parameter types and the contexts of the corresponding actual parameter types. This is required here also, and we change the particular definition of validity that held when unifying with anonymous contexts. The scheme was:

\[\sigma' = \psi'(c<M|\, M*>), c<N|\, N*>) = \{ M \to N, M_1 \to N_1, ..., M_n \to N_n \}\]

Valid(\sigma') = \forall x \in \text{dom}(\sigma') [ x \neq _ \Rightarrow \sigma'(x) \neq _ ]

Our semantic requirement is that non-polymorphic formal contexts may not map to polymorphic actual contexts:

Valid(\sigma') = \forall x \in \text{dom}(\sigma') [ \neg \text{isPoly}(x) \Rightarrow \neg \text{isPoly}(\sigma'(x)) ] \quad \text{isPoly(t) iff t = *}

It is permissible to have an actual context M map to both N1 and N2 formal contexts, i.e. within a \sigma' mapping, if M \to N1 and M \to N2 then N1 must equal N2 as long as M\neq_ (in which case, any number of different actual contexts may map to it). But we cannot allow two different actual contexts to map to the same formal context. This invariant must hold within a unification of context parameters for a particular formal/actual parameter pair, and also across all the unifications created for a particular method invocation. Call this invariant "No Context Overloading" (i.e. the formal context cannot be overloaded with multiple actual contexts, unless the formal is anonymous):

NCO(\sigma) \text{ iff } \forall x \in \text{dom}(\sigma), x\neq_ [ \sigma(x) = y_1 \land \sigma(x) = y_2 \Rightarrow y_1 = y_2 ]

By considering that all the substitutions’ mappings are unioned into an overall substitution \sigma’, then NCO(\sigma’) must also hold.

The (BETTERTYPE) rule therefore remains the same, and (METHODCALL*) takes exactly the same form as (METHODCALL+) in exploiting it.

Any additional substitutions made from formal to actual contexts in the method call (i.e. the set of \sigma’ substitutions made by applying the (BETTERTYPE) rule) must be applied in reverse to the method’s return type.
Ownership Types Restrict Aliasing

<table>
<thead>
<tr>
<th>Ownership Types Restrict Aliasing</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M, M^* \in \Sigma \cup { \text{rep, norep, } <em>^</em> } )</td>
</tr>
<tr>
<td>( P, \Sigma \vdash_{h^*} c &lt; M</td>
</tr>
</tbody>
</table>

(New*)

<table>
<thead>
<tr>
<th>(New*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P, \Sigma \vdash_{h^*} t )</td>
</tr>
<tr>
<td>( P, \Sigma \vdash_{f^*} \text{new } t : t )</td>
</tr>
</tbody>
</table>

(Class*)

<table>
<thead>
<tr>
<th>(Class*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P, \Sigma \vdash_{f^*} t _j : 1..n )</td>
</tr>
<tr>
<td>( P, \Sigma, { \text{this : } c &lt; \Theta</td>
</tr>
<tr>
<td>( P \vdash_{\Theta} \text{class } c &lt; M^* &gt; { t_1, \ldots, t_n, \text{fd}_1, \text{meth}_1, \ldots, \text{meth}_p } )</td>
</tr>
</tbody>
</table>

| \( \Sigma = \{ \Theta \} \cup M^* \) |
| CP(c) = \{ M^* \} |

(Method*)

<table>
<thead>
<tr>
<th>(Method*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P, \Sigma \vdash_{f^*} \text{t}_j : 1..n )</td>
</tr>
<tr>
<td>( P, \Sigma, \Gamma \vdash_{f^*} \text{t}_j' : 1..m )</td>
</tr>
<tr>
<td>( P, \Sigma, \Gamma \vdash_{f^<em>} { x : t, y : t' } \vdash_{f^</em>} e : t )</td>
</tr>
<tr>
<td>( PC(t) \subseteq PCD(md) )</td>
</tr>
<tr>
<td>( PC(t_j') \subseteq PCD(md) )</td>
</tr>
<tr>
<td>( t_{md}(t_1, \ldots, t_n, x_1, \ldots, x_n) { t_1', y_1, \ldots, t_m', y_m, e } )</td>
</tr>
</tbody>
</table>

(FieldAccess*)

<table>
<thead>
<tr>
<th>(FieldAccess*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P, \Sigma, \Gamma \vdash_{f^*} e : t )</td>
</tr>
<tr>
<td>( \sigma = \psi(t) )</td>
</tr>
<tr>
<td>( A'(fd) = t' )</td>
</tr>
<tr>
<td>( SV(e, t') )</td>
</tr>
<tr>
<td>( P, \Sigma, \Gamma \vdash_{f^*} \text{e.fd : } \sigma(t') )</td>
</tr>
</tbody>
</table>

(FieldUpdate*)

<table>
<thead>
<tr>
<th>(FieldUpdate*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P, \Sigma, \Gamma \vdash_{f^*} e : t )</td>
</tr>
<tr>
<td>( \sigma = \psi(t) )</td>
</tr>
<tr>
<td>( A'(fd) = t' )</td>
</tr>
<tr>
<td>( SV(e, t') )</td>
</tr>
<tr>
<td>( P, \Sigma, \Gamma \vdash_{f^*} e' : \sigma(t') )</td>
</tr>
<tr>
<td>( P, \Sigma, \Gamma \vdash_{f^*} \text{e.fd = } e' : \sigma(t') )</td>
</tr>
</tbody>
</table>

(BetterType)

<table>
<thead>
<tr>
<th>(BetterType)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P, \Sigma, \Gamma \vdash_{f^*} e : c &lt; M</td>
</tr>
<tr>
<td>( \sigma' = \psi(c &lt; N</td>
</tr>
<tr>
<td>( \text{Valid}(\sigma') )</td>
</tr>
<tr>
<td>( P, \Sigma, \Gamma \vdash_{f^*} e : c &lt; N</td>
</tr>
</tbody>
</table>

(MethodCall*)

<table>
<thead>
<tr>
<th>(MethodCall*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P, \Sigma, \Gamma \vdash_{f^*} e : t )</td>
</tr>
<tr>
<td>( \sigma = \psi(t) )</td>
</tr>
<tr>
<td>( M'(md) = &lt; (t^* \rightarrow t'), \ldots, &gt; )</td>
</tr>
<tr>
<td>( P, \Sigma, \Gamma \vdash_{f^*} e_i : (\text{td}) : i : 1..n )</td>
</tr>
<tr>
<td>( SV(e, t') )</td>
</tr>
<tr>
<td>( \text{Valid}(\sigma') )</td>
</tr>
<tr>
<td>( P, \Sigma, \Gamma \vdash_{f^*} e_{md}(e_1, \ldots, e_n) : \sigma^{-1}(\sigma(t')) )</td>
</tr>
</tbody>
</table>

\[ \text{MEng. Computing, Imperial College} \]
5.3 Formalising JOT+*

5.3.1 Augmented grammar

Because of anonymous and polymorphic contexts, the contexts which may be featured by ownership types depend on the circumstances in which the type is used:

- Field declarations may use class-supplied contexts only.
- Formal parameters may use class-supplied, anonymous and polymorphic contexts.
- Return types may use class-supplied and polymorphic contexts.
- Local variable declarations may also class-supplied and polymorphic contexts.

The following grammar therefore extends that used by JOT. For implementation reasons, we use ` as the basic polymorphic context instead of "*" because "*" is already widely applicable in most grammars, especially that used in this project. The identifier m used in the previous expansion of an ownership type t is taken to mean any alphabetic identifier other than _ and a sequence of `s.

\[
P ::= \text{defn*}
\]

\[
defn ::= \text{class } c\langle m\rangle \{ \text{field* initializer* meth* } \}
\]

\[
\text{field} ::= [\text{static}] \ t \ \text{fd}
\]

\[
\text{initializer} ::= [\text{static}] \{ \ e \}
\]

\[
\text{meth} ::= t' \ \text{md}(\text{decl*}) \{ \ \text{local* } \ e \}
\]

\[
\text{ decl} ::= t'' \ \text{x}
\]

\[
\text{local} ::= t' \ y
\]

\[
e ::= e; e \ | \ x.\text{fd} [ = e ] \ | \ x.\text{md}(x', y') \ | \ x = e \ | \ \text{new } t \ | \ \text{null} \ | \ x
\]

\[
t ::= c\langle M|M*\rangle["[\]" ] \ | \ \text{int} \ | \ \text{char} \ | \ \text{boolean} \ | \ \text{double} \ | \ \text{float}
\]

\[
M ::= \text{rep} \ | \ \text{norep} \ | \ \text{owner} \ | \ m
\]

\[
t' ::= c\langle P|P*\rangle
\]

\[
P ::= \ M \ | \ * \ | \ *** \ | \ **** \ | \ ***** \ | \ *****
\]

\[
t'' ::= c\langle A|A*\rangle
\]

\[
A ::= P \ | \ _
\]

\[
c = \text{a class name}
\]

\[
m = \text{an ownership context}
\]

\[
\text{fd} = \text{a field name}
\]

\[
\text{md} = \text{method name}
\]

\[
x = \text{variable name}
\]

\[
y = \text{variable name, or this}
\]

5.3.2 Type rules

The complete set of type rules for JOT+* is shown:

- The (Class) rule is as discussed in section 3.3.
- Two sorts of New rules are needed: one that permits only class-supplied contexts to be used, and one that permits class-supplied and polymorphic contexts.
- (Method+) and (Method*) combine to (Method+*).
- (FieldUpdate*) is merely (FieldUpdate+).
- For consistency with the other rules, (BetterType) is renamed (BetterType+*) and (MethodCall*) is renamed (MethodCall+*).
Ownership Types Restrict Aliasing

(Class)

\[ \frac{P, \Sigma \vdash \forall \ t_1: 1..n \ P, \Sigma, \{\text{this} : c<\Theta \mid M^*\} \ \forall \ t_m^* \ \text{meth}_k : 1..p}{P \vdash \text{class} c<\Theta \mid M^* > \{ t_1 \ \text{fd}_1, \ldots, t_n \ \text{fd}_n,\ \text{meth}_1, \ldots, \text{meth}_p \} \]

where \( \Sigma = \{\Theta\} \cup M^* \) and \( \text{CP}(c) = \{ M^* \} \)

(Type)

\[ \frac{M, M^* \in \Sigma \cup \{\text{rep, norep}\} \ \text{CPV}(M^*, c)}{P, \Sigma \vdash \forall \ c<\Theta \mid M^*>}{P, \Sigma \vdash \exists \ c<\Theta \mid M^*>} \]

where \( \text{CPV}(M^*, c) \iff |M^*| = |\text{CP}(c)| \)

(Type+)

\[ \frac{M, M^* \in \Sigma \cup \{\text{rep, norep, _}\} \ \text{CPV}(M^*, c)}{P, \Sigma \vdash \forall \ c<\Theta \mid M^*>}{P, \Sigma \vdash \exists \ c<\Theta \mid M^*>} \]

where \( \text{CPV}(M^*, c) \iff |M^*| = |\text{CP}(c)| \)

(Type*)

\[ \frac{M, M^* \in \Sigma \uplus \{\text{rep, norep, ^}\} \ \text{CPV}(M^*, c)}{P, \Sigma \vdash \forall \ c<\Theta \mid M^*>}{P, \Sigma \vdash \exists \ c<\Theta \mid M^*>} \]

(Type+*)

\[ \frac{M, M^* \in \Sigma \cup \{\text{rep, norep, _}, ^\ast\} \ \text{CPV}(M^*, c)}{P, \Sigma \vdash \forall \ c<\Theta \mid M^*>}{P, \Sigma \vdash \exists \ c<\Theta \mid M^*>} \]

(New)

\[ \frac{P, \Sigma \vdash \exists \ t}{P, \Sigma, \Gamma \vdash \exists \ t : t} \]

(New*)

\[ \frac{P, \Sigma \vdash \exists \ c<\Theta \mid M^*>}{P, \Sigma, \Gamma \vdash \exists \ c<\Theta \mid M^*>} \]

(Method+)

\[ \frac{P, \Sigma \vdash \exists \ t \ P, \Sigma \vdash \exists \ t_1: 1..n \ P, \Sigma \vdash \exists \ t_i: 1..m \ P, \Sigma, \Gamma[x : t, y : t'] \vdash e : t}{P, \Sigma, \Gamma \vdash \exists \ t \text{md}(t_1, x_1, \ldots, t_n, x_n) \{ t'_1, \ldots, t'_m, y_m, e \}} \]

(FieldAccess+)

\[ \frac{P, \Sigma, \Gamma \vdash e : t \quad \sigma = \psi(t) \quad A'(fd) = t' \quad \text{SV}(e, t') \quad \text{AV}(t', \sigma)}{P, \Sigma, \Gamma \vdash e.f.d : \sigma(t')} \]
Ownership Types Restrict Aliasing

(FieldUpdate\(^+\))

\[
\begin{align*}
\Gamma, \sigma \vdash e : t & \quad A'(fd) = t' \quad SV(e, t') \quad P, \Sigma, \Gamma \vdash e' : \sigma(t') \quad NAC(t', \sigma) \\
\end{align*}
\]
\[
\begin{align*}
P, \Sigma, \Gamma \vdash e.fd = e' : \sigma(t')
\end{align*}
\]

(BetterType\(^+\))

\[
\begin{align*}
P, \Sigma, \Gamma \vdash e : c<M | M^*> & \quad \sigma' = \psi'(c<N | N^*>, c<M | M^*>) \quad \text{Valid}(\sigma') \\
\end{align*}
\]
\[
\begin{align*}
P, \Sigma, \Gamma \vdash e : c<N | N^*>
\end{align*}
\]

(MethodCall\(^+\))

\[
\begin{align*}
P, \Sigma, \Gamma \vdash e : t & \quad \sigma = \psi(t) \quad M'(md) = <(t^*\rightarrow t'), .. , > \\
\end{align*}
\]
\[
\begin{align*}
P, \Sigma, \Gamma \vdash e \in_{i:1..n} \sigma(t_i) \quad \text{Valid}(\sigma(t_i)) \\
\end{align*}
\]
\[
\begin{align*}
P, \Sigma, \Gamma \vdash e.md(e_1, .. , e_n) : \sigma^{-1}(\sigma(t_i))
\end{align*}
\]
5.4 Implementing JOT+*

5.4.1 Changes to the grammar

In JavaCC, it is easy to parameterise a production since they are merely turned into Java methods. Hence, the production’s declaration can be treated as a method signature, e.g.

```
"ASTCompilationUnit CompilationUnit() :
[ "<" ModeDeclarations() ""> "
[ "[ "*" ]" ]*
"
"
```

Therefore, instead of implementing the three legal sorts of type production as Type(), TypeB() and TypeC() productions, it is preferable to use just one Type(int x) production, called appropriately by in the productions for LocalVariableDeclaration, FieldDeclaration, FormalParameter and ResultType. Each of these productions would pass a value of x to the Type() expansion to determine which context parameter identifiers should be available. However, it seems at first that this is not useful because even though the value of a production’s formal parameter can be examined in a { ... } Java action, there seems to be no way to interface that Java code to the chosen production.

A simplified production for Type() is:

```java
void Type() :
{ }
{ s=PrimitiveType() ─ t=ReferenceType() }
[ "<" ModeDeclarations() ""> "
[ "[ "*" ]" ]*
}
```

So TypeB() and TypeC() would have to use ModeDeclarationsB() and ModeDeclarationsC() that themselves use ModeB() and ModeC() as well as Mode() (that only allows rep, norep, owner and identifiers as contexts).

```java
void ModeB() :
{ }
{ <REP> | <NOREP> | <OWNER> | t=<IDENTIFIER> |<ANONYMOUS_CONTEXT>
}
```

```java
void ModeC() :
{ }
{ <REP> | <NOREP> | <OWNER> | t=<IDENTIFIER> |<ANONYMOUS_CONTEXT> | <POLYMORPHIC_CONTEXT>
}
```

These productions use terminals defined in the TOKEN: section of the grammar:

```java
TOKEN :
{ 
< INTEGER_LITERAL:<DECIMAL_LITERAL> (["l","L"])? 
< HEX_LITERAL> (["x","X"])? 
< OCTAL_LITERAL> (["o","O","l","L"])? 
> 
| 
< ANONYMOUS_CONTEXT: "_" > 
| 
< POLYMORPHIC_CONTEXT: ("\"\")> 
}
```
Because “*” is already declared as a token describing an arithmetic operator, it was decided to use `*` as the basic polymorphic context for simplicity.

The classes representing Mode(), ModeB(), ModeC(), ModeDeclarations(), ModeDeclarationsB(), ModeDeclarationsC() all inherit from SimpleNode, and one possibility is to manually extend the inheritance hierarchy of these generated classes. Without this, a visitor method is needed to handle each individual class ModeDeclarationsX nodes. With this, many methods are needed for the different ASTxxx types whose corresponding productions are used by FieldDeclaration, LocalVariableDeclaration, FormalParameter and ResultType.

```java
public Object visit(ASTModeDeclarations node, Object data) {
    if (status != JOTMain.INSPECTION_STAGE) return null;
    Vector v = new Vector();
    node.childrenAccept(this, v);  // Iterate over Mode()’s
    return v;
}

public Object visit(ASTModeDeclarationsB node, Object data) {
    if (status != JOTMain.INSPECTION_STAGE) return null;
    Vector v = new Vector();
    node.childrenAccept(this, v);  // Iterate over Mode()’s
    return v;
}

public Object visit(ASTModeDeclarationsC node, Object data) {
    if (status != JOTMain.INSPECTION_STAGE) return null;
    Vector v = new Vector();
    node.childrenAccept(this, v);  // Iterate over Mode()’s
    return v;
}

public Object visit(ASTMode node, Object data) {
    if (status != JOTMain.INSPECTION_STAGE) return null;
    ((Vector)data).addElement(node.name);
    return data;
}

public Object visit(ASTModeB node, Object data) {
    if (status != JOTMain.INSPECTION_STAGE) return null;
    ((Vector)data).addElement(node.name);
    return data;
}

public Object visit(ASTModeC node, Object data) {
    if (status != JOTMain.INSPECTION_STAGE) return null;
    ((Vector)data).addElement(node.name);
    return data;
}
```

In fact, there is a way to implement parameterised productions which do allow the Java variables acting as parameters to have an influence on the production’s expansion. JavaCC provides two sorts of lookahead specification. Ordinarily, we use syntactic lookahead to specify an expansion of terms to try out, and if it succeeds, then the following choice is taken. Semantic lookahead, on the other hand, allows an arbitrary boolean expression to be specified, whose truth evaluation determines whether to choose the following expansion. The expression is written in Java and may use the public data structures of the JavaCC token manager or any user-defined variables set up by the Java body of the parser.
So, we start by parameterising the Type(), ModeDeclarations() and Mode() productions:

```java
void Type(int x) :
{} {
  ( t=PrimitiveType() | t=ReferenceType() )
  jjtThis.setInfo(t.image, t.beginLine);
  [ "<" ModeDeclarations(x) ">" ]
  [ "[ ""] *
}

void ModeDeclarations() :
{} {
  Mode() ( "," Mode() )*
}
void Mode(int x) :
{
...
}
```

Unfortunately, whilst this production translates directly into a Java method that has (apart from a visibility modifier) a signature identical syntactically to the term ("void Type(int x)"), the generated parser includes auxiliary methods for each production that do not have the x variable available. To get round this problem, it was attempted to assign the x variable to a static or instance variable belonging to the JOTParser class in the Java header of the production ({} above); this is naturally available at any point in the generated JOTParser.java program.

There is still a problem. The parser performs a lookahead before entering the body of the production, and it is during this lookahead that the x value is needed. Tracing the output of the lookahead manager reveals that the {} header of the production has not yet been executed when lookahead occurs, so when the Mode(x) production is reached, the relevant options are not available. This means that semantically legitimate code is rejected as unparseable.

The problem is solved by having productions concerned with variable declaration themselves set a global variable in the parser, `permittedContexts`, specifying legitimate contexts. The values that productions use for the assignment are on of:

```java
public static int BASIC = 0;
public static int POLY_ANON = 1;
public static int POLY_ONLY = 2;
```

For example:

```java
void FieldDeclaration() :
{} {
  { permittedContexts = BASIC; }
  [ LOOKAHEAD ( Mode() Type() VariableDeclaratorId() ) Mode() ]
  Type() { jjtThis.setLineNum(token.beginLine); }
  VariableDeclarator() { "," VariableDeclarator() } ";
}
```

```java
void LocalVariableDeclaration() :
{} {
  { permittedContexts = POLY_ONLY; }
  [ LOOKAHEAD( Mode() Type() VariableDeclaratorId() ) Mode() ]
  Type() { jjtThis.setLineNum(token.beginLine); }
}
```
Ownership Types Restrict Aliasing

A MethodDeclaration needs to set permittedContexts twice, once for its lookahead involved the return type (which may feature only class-supplied and polymorphic contexts) and again for formal parameters (which may feature any legitimate contexts):

```java
void MethodDeclaration() : 
{ 
    Token t; 
}

{ permittedContexts = POLY_ONLY; } 
[ LOOKAHEAD( Mode() ResultType() <IDENTIFIER> "(" Mode() ) ] ResultType()

{ permittedContexts = POLY_ANON; } 

t=MethodDeclarator() [ "throws" NameList() ] 
( Block() | ";" )
```

The MethodDeclarator() node that expands to FormalParameters(), and that then to zero or more FormalParameter() nodes, has the correct permittedContexts set up.

The Mode() production can use permittedContexts in a Java expression to decide on which semantic lookahead to use, thus restricting which possible contexts may be chosen. It is then easy to add metadata to the generated instance of ASTMode that is inserted into the AST.

```java
void Mode() : 
{ 
    Token t; 
}

{ <REP> | <NOREP> | <OWNER> | t=<IDENTIFIER>
 |
    LOOKAHEAD( ( (JOTParser.permittedContexts==JOTParser.POLY_ONLY) ) ) 
    t=<POLYMORPHIC_CONTEXT> { jjtThis.addMetadata("Polymorphic"); } 
 |
    LOOKAHEAD( ( (JOTParser.permittedContexts==JOTParser.POLY_ANON) ) ) 
    { t=<POLYMORPHIC_CONTEXT> { jjtThis.addMetadata("Polymorphic"); } 
    | t=<ANONYMOUS_CONTEXT> { jjtThis.addMetadata("Anonymous"); } 
 )
```

Note that it is not always the case that parameterised productions can reduce the number of different AST nodes, or even that reducing AST nodes where they have aspects in common is desirable even though it makes for a “simpler” AST. Constructors, initializer blocks and methods are similar, yet defined in the Java grammar as different productions (and since any of them may occur in a class declaration, various lookaheads incorporate the syntax expected in each as well as the productions themselves being different). Since redesigning the grammar is not desirable, it is necessary to have different visitor methods for each type: ASTInitializer, ASTConstructor and ASTMethodDeclaration (in order of increasing functionality in terms of return types and acceptance of parameters). Although it would be possible to use the JavaCC construct that allows any production to take the type of any other:

```java
void ConstructorDeclaration #MethodDeclaration : 
{ }
{ ... 
}```
this would again mean rewriting a number of already well-defined productions, and the
initial inspection stage would become more convoluted as it determined what sort of method
was being visited based on metadata in the node.

5.4.2 Changes to the visitor

With the extended grammar now able to reject variable declarations using inappropriate
contexts, the inspection stage of the pre-processor is developed to set up field dictionaries
with Type objects featuring the new contexts. The requirement for the enhanced inspection
is that the type-checker must easily be able to detect if a type features anonymous and/or
polymorphic contexts, or if it does not. One possibility would be to extend the Context class
to ContextWithAnon, ContextWithAnonAndPoly, and ContextWithPolymorphic classes,
and via overloaded methods, have the type-checker reject incorrect contexts for a
given environment (e.g. when checking a field declaration). But this was rejected as altering
the structure of the type-checker too much; instead, the Context class is given
isAnonymous() and isPolymorphic() methods, with corresponding setAnonymous() and
setPolymorphic() methods. Inspection uses them to customise the Context objects that it
creates for Type objects in field dictionaries.

Whenever metadata is detected in an AST signifying an anonymous or polymorphic context,
the relevant context parameter in a Type object is set to a Context marked as anonymous
or polymorphic as appropriate. The type-checker can then use isAnonymous() and
isPolymorphic() when implementing methods representing the Valid() predicate from 5.1
and 5.2.

5.4.3 Changes to the type-checker

The Substitution class is extended to UnoverloadableSubstitution. This simply overrides the
addMapping() method to throw a NoPossibleSubstitutionException if the source context
already maps to a destination which is not that specified in the invocation.

An UnoverloadableSubstitution instance is built up as the MethodCall rule checks actual
parameter types. Requirement (4) in MethodCall changes from:

```cpp
// Requirement (4)
// P, S, G |- ei : O(t) for i=1..n
```

```cpp
int currentArgument = 0;
ti = mDict.getFormalTypes(mID);
while (ti.hasMoreElements()) {
    Type literalExpectedType = (Type) ti.nextElement();
    Type substitutedExpectedType = O.applySubstitution(literalExpectedType);
    Type actualType = (Type) args.jjtAccept(jp, env, currentArgument++);
    Type supposedType = (Type) ti.nextElement();
    if (!supposedType.equals(actualType))
        throw new InvalidTypeException("...");
}
```

to:

```cpp
// Requirement (4)
// P, S, G |- ei : O(t) for i=1..n
```

```cpp
int currentArgument = 0;
ti = mDict.getFormalTypes(mID);
while (ti.hasMoreElements()) {
    Type literalExpectedType = (Type) ti.nextElement();
```
The important method, checkBetterTypeRule() below uses auxiliary methods checkValidSubstitution. The policies for permissible context unification are important and should be kept in the type-checker rather than being hived off to a method such as Substitution.isValid(). Indeed, it is not up to the Substitution class to judge whether it is valid or not; this is the role of the type-checker, utilising the Substitution’s methods as appropriate. A Java2 Map object would have done just as well as the explicit Substitution class, and then the type-checker “policy” methods would certainly be needed.
public void checkBetterTypeRule(JOTParserVisitor jp, Environments env,
ASTPrimaryExpression node, Type expectedType,
int paramNum,
UnoverloadableSubstitution globalCall)
throws InvalidTypeException {
    // Requirement (1)
    // P, S, G |- x : c<M | M*> 
    Type actualType = (Type) node.jjtAccept(jp, env);
    // Standard sanity checks
    if (actualType == null) throw new InvalidTypeException(...);
    if (actualType.isPrimitive() && expectedType.isPrimitive()) return;
    if ((actualType.isPrimitive() && !expectedType.isPrimitive()) 
        || (!actualType.isPrimitive() && expectedType.isPrimitive()) 
        || actualType.isStatic() != expectedType.isStatic() 
        || actualType.isArray() != expectedType.isArray() )
        throw new InvalidTypeException(...);
    try {
        // Requirement (2)
        // O' = V'(c<M | M*>), c<N | N*>)
        Substitution Oprime = Vprime(expectedType, actualType);
        // Add this substitution to the global bindings for the method call
        Enumeration all = Oprime.getFrom();
        while (all.hasMoreElements()) {
            Context c = (Context) all.nextElement();
            globalCall.addMapping(c, Oprime.get(c));
        }
        // Requirement (3)
        // Valid(O')
        checkValidSubstitution(Oprime);
    } catch (NoPossibleSubstitutionException npse) {
        throw new InvalidTypeException(...);
    }
}

Substitution Vprime(Type c, Type d) throws NoPossibleSubstitutionException {
    if (c.getParameters().size() != d.getParameters().size())
        throw new NoPossibleSubstitutionException(0,
            "Unable to create a substitution from "+c+" to "+d);

    Substitution s = new Substitution();
    s.addMapping(c.getOwner(), d.getOwner());
    s.addMappings(c.getParameters(), d.getParameters());
    return s;
}

void checkValidSubstitution(Substitution o)
throws NoPossibleSubstitutionException {
    Enumeration e = o.getFrom();
    while (e.hasMoreElements()) {
        Context from = (Context) e.nextElement();
        Context to = o.get(from);
        if ( (from.isAnonymous() && to.isAnonymous()) 
            || (!from.isPolymorphic() && (to.isPolymorphic() || to.isAnonymous()))
            || (!from.equals(to) && 
            !from.isPolymorphic() || to.isPolymorphic())))
            throw new NoPossibleSubstitutionException(...);
    }
}
6. Ownership at runtime

This chapter describes the rationale for knowing ownership at runtime, and what information needs to be collected at compile-time in order to support this. Further uses for the information known at run-time are discussed.

6.1 Dynamic ownership support

The ownership type system allows role separation and restricted visibility to be enforced statically, without information at run-time. However, this enforcement cannot occur in general when type-casting is used, and casting is necessary for two reasons.

Firstly, the use of external classes gives “untypeable expressions” because it is impossible, in the absence of source code for those classes, to statically check their class definitions and method signatures. Type-casting is available to the programmer to remove these type errors by claiming that the type of the object available from an external class is in fact compatible with the environment in which it. For example, below, when the reference c is to an object whose class definition is not available at compile-time, the type of c.d() is unknown and the type-checker considers that expression untypeable:

```java
norep B<m,n> b = c.d();
```

To assign the object returned by this method to some variable in the caller, without a type error, requires a casting construct like this:

```java
norep B<m,n> b = (B<m,n>)c.d();
```

One important category of external classes utilised by the programmer but typically written without ownership features, are container classes such as Vector and Hashtable. They use casting because Java has no facility for parametric types (at least not officially; but see [GJ] for a popular implementation). Therefore, even in the most recent Java2 Collections Framework, it is necessary to upcast (implicitly) an instance of an ownership type to Object when the instance is added to a container, and downcast from Object to an ownership type to retrieve it:

```java
rep B<m,n> b = new B();
Vector v = new Vector();
v.addElement(b);
norep B<m,n> b2 = (B<m,n>)v.elementAt(0);
```

In fact, the static ownership type system in [OT] has no support for type-casting, and the type-checker does not go as far as support the casting construct above. Even if such a construct was implemented, it would still be necessary to check the ownership of objects retrieved from containers at runtime. The property of role separation guaranteed by the type system – that two textually distinct contexts in instantiated ownership types are never assignment-compatible – cannot be verified at all, since there is no context information available about the downcast object.

Java demonstrates that it is not always possible to statically guarantee the type-safety of an operation, due to upcasting. This code compiles successfully but throws an ArrayStoreException at the assignment `ys[0] = y`, since it is trying to widen a reference to a String (which is the object that `ys[0]` really refers to) to a Byte, and this is clearly inappropriate. The exception is thrown by the Java run-time system which checks that both sides of the expression have a compatible type:
String[] xs = new String[1];
Object[] ys = xs;
ys[0] = new Byte(1);

In a similar vein, if the owner of an object and the contexts used in its declaration could be known at run-time, then even in the presence of downcasting, a scheme could be devised to check compatibility between the owners. Specifically, the fact that Java has a unique inheritance hierarchy starting at `java.lang.Object` means that a) all objects have a unique identifier, and b) if a reference to an object is stored, then even if it is type-cast, added and retrieved from containers and passed and returned from any other methods, equality of the modified reference with the stored reference will still hold (because the `==` test compares references to the object, rather than any aspect of the object such as its class).

It is important to realise that knowing the owner of an object is not necessarily enough. Objects of the ownership type `C<m,n>` have three contexts associated with them – owner, `m` and `n` (and implicitly rep also; but that does not affect this discussion). Any instance of `C<m,n>` needs to be associated with the object that owns the instance, the object represented by the `m` context, and the object represented by the `n` context. These are guaranteed to be objects higher in the object graph than the relevant instance of `C<m,n>`. In the absence of a mechanism for casting to ownership types, a programmer would have to cast from `Object` to `C` when retrieving the instance from a container and assigning it to some variable; at this point, the owner object and the two objects backing the `m` and `n` contexts should be compared with the owner object and objects backing formal contexts for the variable being assigned to.

Whereas an ownership scheme, such as `C<m,n>`, is a template for declaration of ownership types, like `C<rep, norep>`, [OT] describes an ownership structure as the resolution of an ownership type’s contexts to physical object identifiers. We shall therefore talk of ownership structures as being the dynamic variant of ownership types; in fact, the runtime support should be a precise implementation of the semantic interpretation function that takes ownership types to ownership structures. Static ownership typing checks whether access paths are permitted; dynamic ownership checks whether access is permitted.

### 6.2 Design

#### 6.2.1 Requirements

For each object instantiated, we wish to identify a) the formal contexts in its ownership scheme, and b) the actual contexts used in its declaration. Then the actual contexts must be mapped to physical objects depending on the environment in which the declaration occurred. For example, consider:

```java
class C<m,n> {
    void f() {
        m B<n,norep> y = new B();
        Vector v = new Vector();
        v.addElement(y);
        rep B<n,norep> y2 = (B)v.elementAt(0);
    }
}
```

An instantiation of an object of class `C<m,n>` will bind `m` and `n` to contexts representing specific objects. When the `f()` method is invoked on this object, and the `y` variable instantiated, the `m` context must be resolved in the environment of the current instance of `C`. This means passing `this` and the context `m` to some data structure built when the current instance of `C` was instantiated. The physical object – or rather a reference to it – that is returned from the lookup must be stored in association with the `y` reference and a marker for “owner”. Similarly, the `n` context for the `this` object must be resolved and stored. The `norep`
context always means the notional “system” object wherever it appears, so it is suggested that some object is always available to resolve to the “root”, as the norep context is also described in [OT] as representing.

The retrieval of the first element of the Vector (the object actually referred to by $y$), into a variable $y_2$ declared as rep-owned would, without dynamic ownership support, mean that the $y_2$ object considered by the programmer as part of the current object’s representation is in fact already owned and almost certainly aliased by objects above the current object in the object graph.

The immediately obvious way to implement ownership structures – indeed any dynamic facility - is to modify the Java Virtual Machine. However, this is an extremely onerous task, and was rejected because modifying the evolving, complex runtime is simply too large an assignment for one part of this project. The specific requirement would be associating with every object, a mapping from its formal contexts to object identifiers. A simpler technique of code insertion at relevant points in the original source is therefore preferred.

Ownership information is necessary at run-time when an assignment takes place or a method call features actual parameters, or where explicit casting occurs (typically in one of these situations). The runtime support must consist of actions that build the ownership structures for all objects declared in code that the pre-processor sees, and compares the ownership structures of either a) objects on the left and right hand sides of an expression, or b) used as actual parameters against the requirements for ownership of formal parameters.

6.2.2 Design

Runtime support is added for all object instantiations and assignments. This involves adding standard code around existing statements that uses suitable classes to build ownership structures for the program’s objects. No account is taken of whether those assignments are type-correct; indeed, it should be possible to use the pre-processor to add runtime support without type-checking a program, as this may be useful during debugging.

To support this, a third stage, RUNTIME_STAGE, is added to the pre-processor after the type-checking stage. Adding runtime support is optional, but since the dictionaries built in the inspection stage are adequate for what the runtime support needs to know, type-checking the program when runtime support is desired is also optional. (In fact, although it is not very useful, it becomes possible to neither type-check nor add runtime support to a program, but merely inspect its source code.)

Two new parameters are supported by jot.pl:

```
--notypecheck Do not perform type-checking stage
--runtime Perform insertion of runtime support code
```

jot.pl invokes the JOTMain program with additional parameters signifying whether, as with the inspection and type-checking stages, the AST should be visited to look for relevant statements. It also copes with the type-checking stage being optional:

```
visitor.status = INSPECTION_STAGE;
cu.jjtAccept(parserVisitor, null);
if (typecheck) {
    visitor.status = TYPECHECKING_STAGE;
cu.jjtAccept(parserVisitor, null);
}
if (visitor.errorsFound() == 0) {
    if (runtime) {
```
The high-level AST nodes are visited and delegate responsibility to their children by ordering the children nodes to accept the visitor. The concept of a “runtime worklist” is introduced, which is an object holding details of the actions which will need to be inserted into the original source. The runtime worklist is eventually written out to a simple text file. If JOTMain exits with a zero return code (no errors) and the --runtime parameter was specified, then the controlling jot.pl script invokes stripjot.pl as before (to transform a .jot file into a .java file), and then invokes another script, addruntime.pl, to insert Java code into the newly generated .java file, according to directives in the text file.

An object declared at an ownership type is modelled at run-time as an OwnedObject. This holds references to both an object (OwnedObject.ref) and the object that owns it (OwnedObject.owner). Also, an OwnedObject is able to map from textual contexts appearing in the ownership scheme of an object’s declared type, to the objects that back those contexts.

A symbol table manages the OwnedObjects at runtime. The SymbolTable class provides static methods for adding an object, mapping its formal contexts to the objects that back them, and retrieve objects and their owners. Its methods are static so that its facilities are available at any point in a program for which runtime support code is added; there is no need to obtain a reference to a SymbolTable object via a static factory object, for example. For simplicity, an array of OwnedObjects is used to minimise the number of references held by the “management layer” to the actual objects. The fact that this limits the number of objects under management is not a serious problem for the small programs on which it is tested.

OwnedObject, SymbolTable and RuntimeWorklist are declared in the jot.runtime package. Exceptions that can be thrown at run-time due to incompatible ownership at assignments are placed in jot.runtime.exceptions.

### OwnedObject

- int lineNum
- String name
- Object ref
- Object owner
- Hashtable contexts  // String -> Object

### SymbolTable

- static Object norepObject
- static OwnedObject[] objects

- static int addObject(String name, Object o, Object o_owner)
- static void addContext(int handle, String context, Object i)
- static OwnedObject find(Object o)
- static Object getObjectOwningContext(String context, Object i)
- static Object getObjectOwningObject(Object i)
- static void compareObjects(Object a, Object b)
### 6.3 Implementation

#### 6.3.1 Visiting the AST

The top-level visitor for a CompilationUnit takes responsibility for creating the worklist:

```java
public Object visit(ASTCompilationUnit node, Object data) {
    if (status == JOTMain.RUNTIME_STAGE)
        rtInstructions = new RuntimeWorklist();
    // Push the class declarations into inspection, type-checking or
    // runtime support activities
    node.childrenAccept(this, data);
    if (status == JOTMain.RUNTIME_STAGE) rtInstructions.finish();
    return done("CompilationUnit", "", 0);
}
```

The `visit()` methods for UnmodifiedClassDeclaration and its child MethodDeclarations simply call upon their children nodes to accept the visitor when in runtime support mode, as it is only at the level of individual statements that code needs to be inserted. Dealing with FieldDeclarations, the other potential children of an UnmodifiedClassDeclaration, is tricky and will be covered later.

A LocalVariableDeclaration node signifies a new reference which must be added to the symbol table. It is handled by checking its child VariableDeclarator nodes. If no assignment takes place immediately (which means there is no PrimaryExpression child of a VariableDeclarator, i.e. no “right-hand side” expression), then the runtime worklist is updated with a requirement to set that particular variable name to null. If there is an immediate assignment, then the variable declaration is like any other arbitrary assignment. This is the complex case, dealt with by a method `handleAssignmentAtRT()` that takes parameters for the variable’s name, declared type and expression on the right-hand side.

`handleAssignmentAtRT()` needs to know the declared types of both the left and right hand sides of an assignment. It is always passed the left-hand side, so to resolve the type of the right-hand side, calls a utility method `computeType()`. This does not perform the ordinary application of type rules.

Assume we have the expression `c.a.b`, where `c` was declared as `rep C<rep, norep> c`, with these class declarations:

```java
RuntimeWorklist
FileWriter fw
MethodID currentMethodName
int supportRequired
static final int LEVEL1, LEVEL2

RuntimeWorklist()
void addSetNull(int LineNum, String varName)
void addNewObjectSupport(int lineNum, String varName, Type varType)
void addFullOwnershipChecks(int lineNum, String lhsName, Type lhaType,
                          int startRHSName, int endRHSName, Type rhsType)
void addOwnerCheckOnly(int lineNum, String lhsName,
                        int startRHSName, int endRHSName)
void finish()
void setCurrentMethod(MethodID method)
void writeMethodSupport()
```
The symbol table is able to tell us what the objects backing contexts \( m \) and \( n \) are for \( c \). When a context is used in the declaration of an object, we say that an object backs the context if that object’s representation context is bound to the context via a series of object instantiations and unification of parameters in method calls.

When instantiating \( \text{rep } A^{<n>} \ a \) in a method invocation on \( c \) (not shown), the symbol table became able to tell us which object backs the owner of the reference \( a \), and which object backs the context \( n \) for the instance of \( A \). Then, when \( m \ A^{<m>} \ b \) is instantiated by a method of \( a \) (not shown), the symbol table already knows the owner of \( b \) and the object backing its context parameter \( m \). Because the symbol table knows not only the owning objects for any given object, but also what contexts those owning objects are actually backing, all we need to know to determine the ownership of \( c.a.b \) is the declared type of \( b \) within its enclosing class. We do not need to perform context substitution on subexpressions in the \( c.a.b \) access path, as the ordinary ownership type system does, because this was done for each object in the path when it was instantiated. (This is again the fact that the semantic interpretation function, and the operation of the symbol table, allows immediate resolution of a context to the object backing it).

\textit{computeType()} therefore takes a \texttt{PrimaryExpression} and walks it from the start to the finish, unlike the ordinary \textit{visit()} method which inspects it in reverse order. For each \texttt{Name}, \texttt{Name+Arguments} (i.e. a method invocation), \texttt{PrimaryExpression} (bracketed subexpression) or \texttt{AllocationExpression} discovered, the static type declaration is retrieved from the high-level method and field dictionaries maintained by the Visitor. This \texttt{Type} object is then used to retrieve the field and method dictionary for the next part of the access path.

The computed literal type for the right hand side is then known. If either side resolves to a primitive type, then no further consideration of runtime support is needed. If the right hand side is in fact an \texttt{AllocationExpression}, then the runtime worklist is notified that “new object support” is desired at a particular line number for a variable name and type. The more usual and complex situation is when the assignment is between two variables, either of which may be typeable or untypeable, e.g. if one of the elements in the object access path involves a class which is not available to the pre-processor through the \texttt{Type.resolveDefinition()} mechanism.

If both sides are fully typed, then the runtime worklist is instructed to perform “full” owner and context parameter checks for both expressions. The full ownership types and variable names concerned are supplied to the worklist along with the line number of the expression. If the left-hand side is typeable, but the right-hand side is not, then the expression on the left-hand side is expecting ownership to be respected, regardless of whatever the right-hand side is doing. What must be validated at runtime is that the unknown object on the right-hand side has the same owner and parameter details as the object on the left-hand side, i.e. the owner and context parameters from the left-hand side apply also to the right-hand side. Thus, the worklist is instructed to perform full ownership checks for both sides at this point in the code, because we specify that the full type of the left-hand side also holds for the right-hand side. If this in fact does not hold at runtime, then there is an ownership violation and the expression on the left-hand side should not be assigned to.
If the left-hand side is unknown but the right-hand side is known, then the runtime system is similarly “tricked” by us mandating that the known type of the right-hand side is also that occurring on the left-hand side. This ensures that the actual ownership type on the left-hand side cannot “give ownership away” by having its widely aliased contexts considered assignment-compatible with potentially different contexts from the right-hand side reference. Again, full ownership checks are inserted to the worklist with the right-hand side’s ownership type specified for both left-hand side and right-hand side expressions. The case of an unknown left-hand side and known right-hand side may be rare, but is covered explicitly since it is easy to implement and may be useful during some degenerate debugging session.

If neither side is typeable, then the best effort of the runtime system is to check equivalence of the owner objects for the actual objects accessed by the left and right-hand expressions.

The handleAssignmentAtRT() method is overloaded to accept a StatementExpression as its only parameter; the visit() method for this node, when in RUNTIME_STAGE, will invoke handleAssignmentAtRT() if the StatementExpression in fact has three children (indicating PrimaryExpression AssignmentOperator PrimaryExpression). This specific handleAssignmentAtRT() method calls computeType() on the left-hand side since any arbitrary PrimaryExpression may occur there; this is of course not necessary when in the situation of a LocalVariableDeclaration being visited in the run-time stage, as the literal type of variable is available immediately from a field dictionary.

Handling an immediate assignment to a field declaration would require the inserted code to go in the enclosing class’ constructor, as that is where javac actually generates the code to perform the field assignment. This is not done at present; if an instance variable is assigned in any method, then it will be handled as any assignment to a local variable would be. It is therefore not the case that fields are not ownership-checked at runtime; they are, as long as the programmer does not use assignments outside methods. The ordinary type-checking stage can, however, check such field assignments.

6.3.2 Building the Runtime Worklist

The actual behaviour of methods is to write out directives to a text file, that simply describes important parameters for each activity. This file is always called rt.log and is opened in the constructor of the RuntimeWorklist class. The addSetNull() method is invoked when a simple local variable declaration is found, and results in this line in the log:

<line number> : SETTONULL : <variable identifier>

An object instantiation becomes:

<line number> : ADDNEWOBJECT : <variable identifier> : <ownership scheme> : <context parameters>

The addFullOwnershipChecks() method does nothing more than write out:

<line number> : FULLCHECK : <LHS variable identifier> : <LHS type> : <Details of RHS>

And the addOwnerCheckOnly() method writes:

<line number> : OWNERCHECK : <LHS variable> : <RHS variable>

Actual insertion of a standard code template parameterised with the variables and types discovered in the runtime stage is performed by a Perl script. The addruntime.pl script
parses the file written out by the RuntimeWorklist object, builds a list associating important line number with the directives specifying action for that line, and then rewrites the original .java file to one featuring significant actions associated with the data structures for ownership, instead of the original instantiations or assignments at those lines.

As the code examples earlier demonstrated, there is a need for some local variables used by the inserted code. If objects are instantiated or nulled, then only “level 1” support for the enclosing method is needed. This means the int handle variable in case an object is added to the symbol table and then further context->object mappings need to be associated with it.

If assignments demand that we check the ownership of the objects and possibly their contexts, then “level 2” support is needed. The variables are then:

- ownerlhs, ownerrhs - Objects to which the symbol table resolves owner contexts
- ownerlcp[], ownerrcp[] - the Object arrays for context parameters that the symbol tables resolves for expressions on, respectively, the left-hand side and the right-hand side of the assignment. These are instantiated with 10 elements, so an object may feature no more than 10 context parameters in its ownership scheme. This is not believed likely to pose any practical restriction.
- lcpCount, rcpCount - a count of the context parameters in the types on the expressions on the left and right-hand sides.

The visit() method for MethodDeclaration takes responsibility for telling the worklist which method it is now visiting, so that as variable declarations and assignments are found, a record can be kept of which type of support will eventually be inserted into the Java source. When the next MethodDeclaration node is visited and the worklist’s setCurrentMethod() invoked, the level of support needed for that last visited method written out; the final method has its support level written out by the WorkList as it closes the log file. This support level is simply one of:

<line number>:METHODSUPPORT1
<line number>:METHODSUPPORT2
<line number>:METHODSUPPORT3  (if both are needed)

### 6.3.3 Compile-time program transformation

**Setting up the symbol table**

Suppose we have this code:

```java
rep C<rep, norep> c = new C();
```

and C<m,n> is the relevant ownership scheme. The general scheme is to add the reference “c” to the symbol table, owned by whichever object is represented by “rep” in the environment of the “this” object; and then for each actual context parameter (rep, norep) used in c’s declaration, add a mapping from the context to a resolved object. Since these mappings should be associated with the particular c reference just added to the table, a simple handle is returned when a reference is added to the symbol table, to be used when associating that reference with contexts.

Thus, the code above is enriched to:

```java
int handle;
C c = null;
c = new C();
```
Ownership Types Restrict Aliasing

The original ownership type will be removed by a Perl script invoked by the main program after the pre-processor has run, but before addrt.pl runs.

Handling assignments

All assignments, even those which would be type-correct if type-checked, are enriched with support code. There are two problems with assignment. Firstly, the LHS and/or RHS may not have been assigned or instantiated yet, merely declared. So use what the types would be if real objects were available, and ensure that declarations which don’t assign immediately are set to null by the code inserter. Secondly, even if they have been assigned, we must be careful never to quote the original textual expressions more than once, in case they have side-effects. For example, the left-hand side of “a.b[i++] = ...” must only be evaluated once.

Suppose having set up the object c above, we invoke c.f() to ensure that the c.a field is instantiated, and also that the c.a.b field is too.

```java
class C {
    A a;

    void f() {
        a = new A();
        int handle = SymbolTable.addObject(a, /* owner is */ SymbolTable.getObjectOwningContext("rep", this));

        SymbolTable.addContext(handle /* for reference c *//>
            "m",
            SymbolTable.getObjectOwningContext("rep", /* in object */ this));

        SymbolTable.addContext(handle /* for reference c *//>
            "n",
            SymbolTable.getObjectOwningContext("norep", /* in object */ this));

        a.g();
    }
}
```

```java
class A /*<m>*/ {
    //m A<m> b;
    A b;

    void g() {
        b = new A();
    }
}
```
int handle = SymbolTable.addObject(b, /* owner is */ SymbolTable.getObjectOwningContext("m", this));

// A<m> -> A<m>
SymbolTable.addContext(handle /* for object b */, "m", SymbolTable.getObjectOwningContext("m", this));

Returning to the top level of the program, the assignment c.a = c.a.b would become:

try {
    // c.a = c.a.b;
    // A<rep|n> = A<m|m>
    Object ownerlhs = null, ownerrhs = null,
    ownerlcp[] = new Object[10], ownerrcp[] = new Object[10];
    int lcpCount = 0, rcpCount = 0;
    // Evaluate both sides of the assignment
    Object lhs = c.a;
    Object rhs = c.a.b;
    if (lhs != null) {
        ownerlhs = SymbolTable.getObjectOwningObject(24, "c.a", lhs);
        ownerlcp[lcpCount++] = SymbolTable.getObjectOwningContext(24, "m", lhs);
    } else {
        if (c != null) {
            ownerlhs = SymbolTable.getObjectOwningContext(24, "rep", c);
            ownerlcp[lcpCount++] = SymbolTable.getObjectOwningContext(24, "n", c);
        }
    }
    if (rhs != null) {
        ownerrhs = SymbolTable.getObjectOwningObject(24, "c.a.b", rhs);
        ownerrcp[rcpCount++] = SymbolTable.getObjectOwningContext(24, "m", rhs);
    } else {
        if (c.a != null) {
            // Make an attempt at validating context owners on rhs
            ownerrhs = SymbolTable.getObjectOwningContext(24, "m", c.a);
            ownerrcp[rcpCount++] = SymbolTable.getObjectOwningContext(24, "m", c.a);
        }
    }
    // Compare owners: statically, rep -> m
    SymbolTable.compareObjects(24, "c.a", ownerlhs, "c.a.b", ownerrhs);
    // Compare parameters: statically, n -> m
    SymbolTable.compareObjects(24, "c.a", ownerlcp[0], "c.a.b", ownerrcp[0]);
    // Now the assignment itself
    c.a = rhs;
} catch (UnknownObjectException uoe) {
    System.out.println(uoe.getMessage());
} catch (UnknownOwnershipException uoe2) {
    System.out.println(uoe2.getMessage());
} catch (IncompatibleOwnershipException ioe) {
    System.out.println(ioe.getMessage());
}

Assignment with type-casting

Suppose the original code is:

Vector v = new Vector();
v.addElement(c);
rep C<rep, norep> c2 = (C)v.elementAt(0);

This becomes:

```java
try {
    C c2 = null;
    Object ownerlhs = null, ownerrhs = null,
        ownerlcp[] = new Object[10], ownerrcp[] = new Object[10];
    int lcpCount = 0, rcpCount = 0;
    Object lhs = c2;
    Object rhs = v.elementAt(0);
    if (lhs != null) {
        ownerlhs = SymbolTable.getObjectOwningObject(lhs);
        ownerlcp[lcpCount++] = SymbolTable.getObjectOwningContext("n", lhs);
        ownerlcp[lcpCount++] = SymbolTable.getObjectOwningContext("o", lhs);
    } else {
        ownerlhs = SymbolTable.getObjectOwningObject(79, "rep", this);
        ownerlcp[lcpCount++] = SymbolTable.getObjectOwningContext("rep", this);
        ownerlcp[lcpCount++] = SymbolTable.getObjectOwningContext("norep", this);
    }
    if (rhs != null) {
        ownerrhs = SymbolTable.getObjectOwningObject(rhs);
        ownerrcp[rcpCount++] = SymbolTable.getObjectOwningContext("n", rhs);
        ownerrcp[rcpCount++] = SymbolTable.getObjectOwningContext("o", rhs);
    }
    SymbolTable.compareObjects("c2", ownerlhs, ", (C)v.elementAt(0)", ownerrhs);
    SymbolTable.compareObjects("c2", ownerlcp[0], "(C)v.elementAt(0)",
                                ownerrcp[0]);
    SymbolTable.compareObjects("c2", ownerlcp[1], "(C)v.elementAt(0)",
                                ownerrcp[1]);
    c2 = rhs;
} catch (UnknownObjectException uoe) {
    System.out.println(uoe.getMessage());
} catch (UnknownOwnershipException uoe2) {
    System.out.println(uoe2.getMessage());
} catch (IncompatibleOwnershipException ioe) {
    System.out.println(ioe.getMessage());
}
```

6.3.4 Error handling

Part of the work of addruntime.pl, the Perl script that physically adds ownership management code to .java files, is prepending them with:

```java
import jot.runtime.*;
import jot.runtime.exceptions.*;
```

so that the relevant classes are available to javac when it compiles the final Java source.

Three classes of error can occur at runtime. A method using only level 1 support will not need to catch exceptions, but SymbolTable facilities used by level 2 support, `Object getObjectOwningContext()` and `Object getObjectOwningObject()`, may not be able to resolve an object to its owner or a context to an object, so throw an `UnknownOwnershipException` to the main program. These methods may themselves have to deal with an `UnknownObjectException` or `UnknownContextException` thrown by either the `SymbolTable` (if an alien object which was never added to it is looked up) or by a retrieved OwnedObject (if for some reason, an illegal textual context is looked up for an object). Also, level 2 methods using the `SymbolTable`’s `compareObjects()` method may discover that its two parameters are not equal by the `==` test, so it may throw an
IncompatibleOwnershipException. To handle these exceptions, level 2 methods – featuring assignments – have the managed assignment enclosed in a try {...} catch {...} block so that the straight-line flow of the context-backing object comparisons goes smoothly unless an exception is generated.

6.4 Further work

6.4.1 Garbage collection issues

Suppose an object has a context which is owned by some higher object in the object graph. Then, when all other references to the owning object become dead, there is still an OwnedObject entry in the SymbolTable that has a reference to it. Without modifying the JVM’s garbage collector to ignore the SymbolTable when considering unreachable objects, all we can say is that a programmer must manually indicate that an object is ready for garbage collection and that it should be removed from the SymbolTable in preparation for this. The finalize() method inherited from java.lang.Object provides a good location for performing this; all Java classes derived from ownership types could have a finalize() method added that nulls out the OwnedObject entry representing the current instance (“this”) in the SymbolTable.

The issue then arises of what to do with the objects owned by the unreachable object. We may choose to ignore them and let the garbage collector deal with them as appropriate, but the Representation Containment invariant tells us that the unreachable object dominates access to them – can this fact be used to aid garbage collection? The objects that it owns need not necessarily be referred to directly from fields in the unreachable object. (As with the second-lowest object in the graph is below) Furthermore, although the unreachable object dominates access to objects that it owns lower down the graph, it is not necessarily their only dominator since they may feature multiple formal contexts in their ownership schemes. It is not necessarily even the dominator nearest to its owned objects; objects lower than it may pass their own rep down the graph for deeper object declarations to use, so although our unreachable object owns and dominates those deeper objects, so too do the other deeper objects who passed their rep context down.
It seems natural to say that because the unreachable object X dominated access to objects that it owned (such as O and B) and because their owner is now unreachable, they too are now just as unreachable as it is. This would allow the garbage collector to collect objects not even directly referenced from the unreachable object, on the grounds that they could only be reached via it.

Even the convoluted situation above allows this. The unreachable object X has instantiated object A and passed its rep context to it, so that A could instantiate object B owned by X. X could not have obtained object A from elsewhere, because then there would be no way for A to have a context parameter that corresponded to X’s rep. When X instantiated object A, it set it as owned by object C, whose rep context is available to X through one of X’s class’ context parameters. Having now set up objects A, B and C, we can see why garbage-collecting B along with X is legitimate. If we do this, then A will wonder where one of its fields has gone. Any access to B must by definition go through the now unreachable object; even if X has previously returned C to other objects higher than X in the graph, if any of them try to access B by traversing a path through C, then A, and to B, they will find that they cannot type B; within the definition of A’s class, the context parameter used for the owner of the field pointing to B is different to A’s owner. However, when X uses A to obtain a reference to its owned object B, it can successfully type B.

Therefore, garbage collection may collect all objects owned by an unreachable object even if it has no direct references to them. Naturally, objects directly referred to by the unreachable object but not owned by it (such as D) may be referred to also by other objects within the scope of their common owner, so garbage collection must not touch them. Implementing a garbage collector aware of ownership, to take advantage of this observation, is of course a significant task.

6.4.2 Enhancing the symbol table

The Collections framework in the JDK1.2 [J2COL] introduces a WeakHashMap. This is an implementation of the Map interface that surjectively (?) maps keys to values, of interest because it stores weak references to its keys. Weak references are provided by the new (in JDK1.2) package java.lang.ref, which provides classes for reference objects. A reference
object encapsulates a reference to some other object so that the reference itself may be examined and manipulated like any object. Typically a program uses a reference object to maintain a reference to some other object in such a way that the latter object may still be reclaimed by the garbage collector.

The WeakHashMap’s use of weak references allows key-value pairs to be garbage-collected when the key is no longer referenced from outside the WeakHashMap. Specifically, the object referred to by the key is said to be strongly reachable as long as it can be reached by a thread without traversing any reference objects, i.e. when it is referred to by fields in other objects. When those become dead for whatever reason, the object referred to by the key reference becomes weakly reachable, because it can only be reached by traversing a weak reference (namely the key reference); hence, it becomes eligible for finalization. The JDK1.2 documentation states “It is useful for implementing "registry-like" data structures, where the utility of an entry vanishes when its key is no longer reachable by any thread.”, and this is an excellent description of a symbol table.

If we were prepared to require that Java programs be run only under the Java2 VM, then a WeakHashMap could be used in the SymbolTable to map objects (the keys) to the objects which are their owners (the values). When an object became unreachable from all other objects in the system, the weak reference in the SymbolTable would be the only way of reaching it, so would become finalizable and be garbage-collected in due course.

Such a scheme still uses ordinary references to the owning objects, and as noted earlier, these may be garbage collected in due course. Naturally, the SymbolTable’s references stop this. If a different implementation of WeakHashMap was used in which both keys and values were weak references, then it would be possible for an object to find itself without an owner object at all (literally, the key would map to null), and this will be useless for deciding whether ownership is respected in assignments. For this reason, the standard WeakHashMap is suitable for a Java2-specific implementation of the SymbolTable.

### 6.4.3 Visualising ownership structures

On another point, since we have detailed information about all objects in the ownership structure for the program as a whole, a public static method could be provided by the SymbolTable class that writes out an XML representation of objects and their references. The fact that the owner of an object must itself be an object in the system, i.e. that the ownerID element is in fact equal to the <uniqueID> element within some other <object>, is easily expressible with the XML-data schema language.

```xml
<object>
  <name>...</name>
  <uniqueID>...</uniqueID>
  <ownerID>...</ownerID>
  <context name="..." ownerID="..."/>
</object>
```
7. Discussion

7.1 Using Generic Java type parameters to synthesise contexts

Generic Java is a popular implementation of parametric types for Java. A program written to use parametric types contains more type information than a usual Java program (and fewer type-casts). An investigation was made into the possibility of mapping Java source with ownership annotations (JOT) into Generic Java, such that GJ could check the type-correctness of the transformed ownership constructs. That is, is there a function f: JOT->GJ, where given a program p in Java+Ownership, p is type-correct <= f(p) is type-correct? This mapping appeared to have promise; in fact it turned out to be inadvisable. The rationale is presented here along with its problems.

An ownership type parameterised by an owner context and possibly further context parameters would map to a class parameterised by particular types denoting “contexts”. The ability of GJ to bound type parameters as implementing a certain interface (or extending a certain class) would be used to make such “context parameters” be classes implementing Context. (The Context interface would not necessarily do anything.) The context classes could signify the containment relationships through inheritance.

However, there is a problem with representing the concept of "rep". Suppose we made a class RootContext and had specific context classes extending it. What context class could we parameterise a rep-owned field with?

For example, taking a popular programming example from [MS], could we rewrite the field declaration rep Engine engine as Engine<RepContext> engine? With static visibility, types containing rep are only visible to the current instance of the class in which they are defined; but with two Car objects, the type Engine<RepContext> is equally visible in both, and there would be no practical enforcement by Generic Java against assigning the supposedly "rep" member of one object to the "rep" member of the other object. (This is regardless of whether class RepContext extends RootContext or not.) Therefore something more is needed.

To simulate rep, a new class must somehow be declared for every instantiation of an Engine object. One possibility is a new RepContext class for every object ever instantiated from a class featuring rep instance variables. (A static rep variable in a class, that code inside any instance of the class can access, would be instantiated with the same actual RepContext parameter across all instances of the class).

Suppose this is original annotated code:

```java
class User {
    rep Object myData;
}
User x = new User();
User y = new User();
```

We would rewrite it as:

```java
class User<R> {
    Object<R> myData;
}
User<RepContext1> x = new User<RepContext1>();
User<RepContext2> y = new User<RepContext2>();
```
x = y;  // Ideally, this would be valid; but the practical requirement for
the RepContext1/2 parameters means that Generic Java considers x and y as
different types.

Another way to get a unique class representing a specific object’s representation context is
to use an inner class, i.e.

```java
Class Car {
    Class Rep { }
    Driver<Rep> d;
}
```

If the inner class is private (and happened to be uniquely generated for each object), then it
exhibits some of the desired properties of "rep": types containing "rep" are not visible
outside the class. But in fact, the inner Rep class is just an ordinary class whose full name is
Car$Rep, and the Java 1.1/2 compilers hack it up… GJ would actively have to be changed to
create unique per-object inner classes for denoting rep.

Alternatively, if Generic Java was extended to support instantiation of objects whose type is
parameterised by instance variables rather than class names, then a type checker could
instantiate enough concrete "representation context" objects to parameterise every normal
object with.

It would therefore appear that for every object instantiated from a class with rep members,
the overhead is either a class definition or a separate object. Clarke noted that “an
implementation trick which does not require new objects to denote every new rep is to use
the object’s id for the rep. This is unique per object, though it does not capture any of the
necessary containment relationships.” Naturally, this is essentially what the runtime support
system does when it checks object references for equality between the actual object backing
the owner context of some object, and the possible owner object.

Clarke also suggested that inheritance relationships can be specified wrt to type parameters.
The problem (apart from rep not being unique) is trying to use inheritance to capture the
nesting relationship between the different contexts. This is because we need to impose
“implements” relationships between the variables, for example:

```java
class Car {    // With an implicit "owner"
    rep Driver d;
}
```

would become

```java
class Car<Owner> {
    class Rep implements Owner {}
    Driver<Rep> d;
}
```

The line `class Rep implements Owner` attempts to use GJ’s type system to enforce the
nesting relationship between Rep and Owner [DC]. Unfortunately, this is not allowed by GJ.

### 7.2 A monotone framework for ownership

#### 7.2.1 Owner Hierarchy Analysis

Using ownership types allows a programmer to identify the ownership of each object
through its type, in terms of other objects in the system. These other objects are represented
through contexts, which may be passed around in order to create complex hierarchies of ownership. At run-time, it is possible to directly identify which physical object in the heap is the “owner” for any other object. A static analysis of this is identifying which possible contexts may be the true owner for a particular declared variable.

For example, take this simple program:

```java
class C {
  rep Object x = new Object();
}
```

The instance of class C at run-time refers to the instantiated object `this.x`; more interestingly, the object `this.x` is owned by the instance of class C. Specifically, the ownership relation from `this.x` to the instance of C is direct – it might be said that the instance of C is the parent of the object referred to by `this.x`.

Now consider this class:

```java
class D<m> {
  m Object y = new Object();
}
```

An object `c` (of class C) instantiates `d`, an instance of class D; then, `c` is the parent of `d`. For example, the declaration `norep D<rep> d = new D();` in a method of C will produce a variable `d`, owned by `norep` (the “system”, i.e. no-one specific) and passing the context of the `c` down into `d` as the owner of `d`’s `y` field. Notice how the declaring object `c` need not create or otherwise hold a direct reference to this `y` field. The situation is shown in the graph, references are shown with solid lines and ownership with dashed lines.

For the field `y`, it might be said that its owner is not the “parent” object that holds a reference to it, since the object `d` that refers to `y` is not `y`’s owner. Instead, because `d`’s declaring object, `c`, passed its own “rep” (representation context) into `d`, it is `c` that owns the `y` field. Since an object’s ownership cannot change after its declaration, it is valid to say that after a method in the enclosing `c` object has instantiated `d` (and hence `y`), `y`’s owner is its parent’s parent. Even if `c` loses its reference to `d` or `d` loses its reference to `y`, it is still the case that the object `y` is notionally owned by object `c` at run-time. (Although whether anything useful may be done to `y` if this happens is another matter.)

With deeper object graphs, an object may be owned by an object many levels higher, and notion of parent generalises to a parent’s parent, or a parent’s parent’s parent, etc. Such an “owner hierarchy” will exist for the owner context, \( \Theta \), of each object declaration, and for any contexts in its ownership scheme. We represent a hierarchy which is a parent’s parent as \( \text{parent}^2 \), parent\(^1\) is simply “parent”, and the representation can be bounded by parent\(^n\) for some \( n \). Above this \( n \), we will be dealing with owner hierarchies that extend many levels down the object graph. The complexity of the ownership structure of a program as measured by owner hierarchies is interesting, but a limit on the programmer’s understanding of who owns what will probably restrict \( n \) to, say, four.

A data-flow analysis reveals, for a program point, what the ownership hierarchy may be for each context in each variable declaration made up that point. Where multiple contexts have the same owner hierarchy, e.g. contexts \( m \) and \( n \) might both be only parent\(^2\), it might be asked why both contexts are required.
7.2.2 Definitions

The set ID of abstract identifiers is built from the textual declarations of variables in a program, and is simply the identifiers of those variables:

$$\text{ID} = \{ \text{fields, method formals, local variables} \}$$

An abstract hierarchy is a set of pairs for possible (context, owner hierarchy) combinations:

$$\text{AH} = \wp( \{ \Theta, M^* \} \times \{ \text{norep, parent, parent^2, parent^3, parent^n} \} )$$

Given $\Theta$ for the owner context and $M^*$ as the set of all formal contexts used in classes in the program, an identifier that has an owner and two context parameters, $m$ and $n$, might have an abstract hierarchy which is the set $\{ (\Theta, \text{norep}), (m, \text{parent}), (m, \text{parent}^2), (n, \text{parent}^n) \}$. The symbol $\perp$ is available to show that no hierarchy is yet known for a particular context.

The set $\wp(\text{ID} \times \text{AH})$ is thus a set of pairs (abstract identifier, abstract hierarchy) indicating that such an abstract hierarchy is possible for that abstract identifier.

7.2.3 Refinements

The analysis will be improved by refining abstract identifiers to be elements of the set $\wp(\text{Id} \times \wp(\text{Cl} \times \text{Md}))$, which is isomorphic to $\wp(\text{Id} \times \text{Cl} \times \text{Md})$. This is the new set entitled ID.

$$\text{Id} = \text{all variables declared in the program}$$
$$\text{Cl} = \text{set of all class names declared in the program}$$
$$\text{Md} = \text{set of all method names declared in the program, plus } \perp \text{ for identifiers not in a method}$$

All variables occur in a class, so no abstract identifier need encode that it (the textual identifier) is not in an enclosing class definition. But fields are not in a method, so their method item is $\perp$. Note that no type information about the $x$ or $y$ identifiers is held, and there is no differentiation between formal parameters and local variables (both are simply identifiers within a method.).

This definition of abstract identifiers is an improved definition for the "textual point" [CTA] at which a declaration occurs. For example, an field $x$ in class $C$ would have an abstract identifier of $(x, C, \perp)$, since it is not in a method. A local variable $y$ in method $md$ of class $C$ would be $(y, C, md)$.

Also, AH is refined by changing the set of possible contexts $\{\Theta, M^*\}$ to a set that is (class,context parameter) pairs. This set is built from inspecting the actual program, so that only those formal contexts actually declared in a class signature are possible.

7.2.4 Formulation as a monotone framework

Since the set ID is finite, and AH is finite also, $\wp(\text{ID} \times \text{AH})$ is finite and trivially a complete lattice. The analysis is specified as an instance of a Monotone Framework with the complete lattice of properties, $L_{O_H}$, being $\wp(\text{ID} \times \text{AH})$.

$$\text{OH}(I) = \begin{cases} \{ t \} & \text{if } t \in E \\ \cup \{ \text{OH}(I') | (I', I) \in F \text{ or } (I', I) \in F \} & \text{otherwise} \end{cases}$$
The analysis is forwards so the set of extremal labels $E = \text{init}(S')$ and the set of legal sequences in the interprocedural control flow graph $F = \text{flow}(S')$ or $\text{interflow}(S')$, according to the usual definitions.

- We require the smallest sets of owner hierarchies (“may own”) for identifiers, so employ the $\cup$ operator.
- The initial state (extremal value $i$) is a set of pairs $= \forall x \in \text{ID}, y \in \text{AH} \ [ (x, (y, \bot)) ]$

Introduce a special “program object” $P$, representing the initial object that, in a special method, creates other objects and invokes their methods. Context $C$ at any program point is a stack consisting of a set of stack frames. Each stack frame has the type $\phi$ (formal context $x$ owner hierarchy), e.g. $(m, \text{parent}^2), (m, \text{parent}^3), (n, \text{parent}^n)$. The initial stack, $c_0$, is empty, since the special method in $P$ may use only contexts “rep” and “norep”, and these are not looked up from the stack.

Adding context to elements of the property produces the embellished property space $L_{\text{OH}}' = C \rightarrow L_{\text{OH}}'$, isomorphically equivalent to $L_{\text{OH}}' = \varphi(C \times (\text{ID} \times \text{AH})) = \varphi(C \times \text{OH})$. The embellished extremal value $i'$ is $({}, i)$ because of the empty stack in $P$. It is observed that the structure of the context – pairs of context and owner hierarchy values – is very similar to (part of) the elements in the property space itself.

The transfer function $f'_l : \varphi(C \times \text{OH}) \rightarrow \varphi(C \times \text{OH})$ has the form:

$$f'_l(C, \text{OH}) = \cup \{ f'_l(C, (\text{ID}, \text{AH})) \mid (\text{ID}, \text{AH}) \in \text{OH} \}$$

### 7.2.5 Transfer functions

Only object instantiation updates the owner hierarchy, and bindings from the formal contexts of the executing method’s enclosing class to abstract hierarchies is updated only when the enclosing class changes, i.e. on method call. The following auxiliary functions are used:

Suppose we have “class $A<m,n,o> \{ ... \}$” with a declaration $\text{rep A<rep,norep,rep> x;}$

Then:

- $\text{typeof(x)} = A<\Theta | m,n,o>$ // $\Theta$ is the owner context
- $\text{contexts(typeof(x))} = \{ \Theta, m,n,o \}$
- $\text{unify(t,x)} = \{ \Theta \rightarrow \text{rep}, m \rightarrow \text{rep}, n \rightarrow \text{norep}, o \rightarrow \text{rep} \}$
- $\text{context(m,x)} = \text{unify(typeof(x),x)}(m)$

The function $\text{context(m,x)}$ retrieves, from the declaration of $x$, the actual context bound to the formal context $m$ from $x$’s ownership scheme. For the declaration $x$ above, $\text{context(m,x)}=\text{rep}$, $\text{context(n,x)}=\text{norep}$, $\text{context(o,x)}=\text{norep}$ and $\text{context(\Theta,x)}=\text{rep}$. 
Instantiation of local objects - \([x := \text{new } t]\)

\[
F_i(C,OH) = (C,OH')
\]

\[
OH' = \forall z \in \text{contexts}(t) \quad \left[ \begin{array}{l}
\text{if context}(z,x) = \text{rep then} \\
\quad = \text{OH}(x,(z,\bot)) \cup (x,(z,\text{parent})) \\
\text{if context}(z,x) = \text{norep then} \\
\quad = \text{OH}(x,(z,\bot)) \cup (x,(z,\text{norep})) \\
\text{else} \\
\quad = \text{OH}(x,(z,\bot)) \cup (x,(z,Z)) \quad \text{where } (z,Z) \in c_n
\end{array} \right]
\]

where \(c_n\) is the top stack frame of \(C\).

The identifier \(x\) may be any abstract identifier in the refined set, i.e. if a field, local or formal variable is (re)instantiated, then the new possibilities for its formal declaration are added.

Method call - \([x.md(...)]\)

\[
F_i(C,OH) = (C,OH)
\]

\[
C' = [c_1..c_{n-1},c_n] + c_{n+1}
\]

\[
c_{n+1} = \forall z \in \text{contexts}(\text{typeof}(x)) \left[ (z,Z) \right]
\]

where \(Z = \)

\[
\begin{array}{l}
\text{if context}(z,x) = \text{rep} \\
\quad \text{parent}^{\text{n} \times 2} \\
\text{if context}(z,x) = \text{norep} \\
\quad \text{norep} \\
\text{otherwise} \\
\quad (\text{context}(z,x), y) \in c_n \\
\quad \text{if } y = \text{parent}^{\text{n} \times 3} \text{ then} \\
\quad \quad \text{parent}^{\text{n} \times 3} \\
\quad \text{else if } y = \text{parent}^{\text{n} \times q} \text{ then} \\
\quad \quad \text{parent}^{\text{n} \times (q + 1)} \quad \text{where } 1 \leq q < 3
\end{array}
\]

\[
F_i((C,OH),(C',OH')) = (C,OH')
\]

Returning from a method call means taking the stack from before the invocation, but preserving owner hierarchies which may have been updated during the invocation.

\[
F_{ln}(C,OH) = (C,OH)
\]

\[
F_{lx}(C,OH) = (C,OH)
\]

Also, we need to update the abstract hierarchies of those abstract identifiers which are formal parameters for the invoked method. This could usefully be done (but is not shown) in \(F_{ln}(C,OH)\), for which the new stack frame is available from the transfer function \(F_{lc}\).

If a field declaration uses a context, but the field is never instantiated, then pairs in the property space involving the field’s abstract identifier will never be touched by the data-flow analysis. But then, at the end of the analysis, no owner hierarchy will have been associated with those abstract identifier, so they will still have \(\bot\) for all contexts.


Example

class C<m,n> {
    rep D<rep,norep> d1;
    norep D<m,n> d2;

    void f() {
        d1 = new D();
        d2 = new D();
        d1.g();
        d2.g();
    }
}

class D<p,q> {
    void g() {
        rep C<p,q> c1 = new C();
    }
}

Initial state:
Abstract identifiers: (d1,C,⊥), (d2,C,⊥), (c1,D,g)
Abstract hierarchies:  For (d1,C,⊥): (Θ, ⊥), (p, ⊥), (q, ⊥)
                      For (d2,C,⊥): (Θ, ⊥), (p, ⊥), (q, ⊥)
                      For (c1,D,g): (Θ, ⊥), (m, ⊥), (n, ⊥)

Assume an initial program object P now instantiates a variable of type C<rep,rep,norep>, and invokes f() on it. The initial stack frame will contain a mapping from contexts in the enclosing class of the invoked method: Θ → parent^2, m → parent^2, n → norep.

The instantiation of d1 will leave the stack alone, but update the owner hierarchy information. Then ((d1,C,⊥), (Θ,⊥)) will be removed from the element OH of the property space, and ((d1,C,⊥), (Θ,parent)) will be added. Also, ((d1,C,⊥), (p,parent)) and ((d1,C,⊥), (q,norep)) will be added.

The instantiation of d2 will add ((d2,C,⊥), (Θ, norep)) obviously. The contexts m and n used in d2’s ownership scheme are looked up in the c0 stack frame, and ((d2,C,⊥), (p,parent^2)) and ((d2,C,⊥), (q,norep)) added to OH.

For the invocation d1.g(), a new stack frame c1 will be created that maps: Θ → parent^2, p → parent^2, q → norep. The instantiation of c1 will remove ((c1,D,g), (Θ,⊥)) from OH and add ((c1,D,g), (Θ,parent)), ((c1,D,g), (m,parent^2)) and ((c1,D,g), (n,norep)).
8. Evaluation

This chapter considers the key achievements and limitations of the work undertaken, and what insights it has provided into both the ownership type system and the general problem of alias control. It also suggests further work related to both the implementation as it stands and related concepts from software engineering.

8.1 Conclusions

"Is ownership a helpful concept?"

This project has been concerned with evaluating ownership features as they stand, not comparing them to alternative means of alias prevention. It must be said that ownership types take the approach of specifying what can be done rather than what cannot, which is perhaps not the most natural approach when designing a scheme to restrict operations. Therefore, understanding how restricted visibility and representation containment govern the control of aliasing is not immediately obvious. Many schemes have been presented which govern aliasing, notably [BALL] and [ISL], and this work on ownership types has not been concerned with them. But patently, ownership is a strong mechanism for alias control as long as the programmer is fully capable and willing to apply it throughout large portions of a program, if not the whole program.

"Is the implementation correct?"

By starting from the actual Java grammar developed at Sun, and designing a type-checker and auxiliary classes in the closest terms possible to the structures used in [OT], I believe that the implementation follows a correct manual application of the type rules simply and efficiently.

"Are the extensions relevant and correct?"

Because anonymous contexts are so restrictive – role separation is strictly enforced between anonymous and non-anonymous contexts, and restricted visibility is only enhanced by using anonymous contexts - I am informally satisfied that they do not break soundness of the ownership type system.

Introducing polymorphic contexts affects the representation containment invariant, which states that for a well-typed program, an object’s owners (for both the object and its declared contexts) dominate access paths to it in the object graph. A lemma is used in the inductive proof of this invariant, that describes the possible owners of an object o’ (and its contexts) when another object o has a reference to it; crucially, these owners of o’ are specified in terms of the owners of o. When a method is invoked on an object and an actual parameter is a reference featuring polymorphic contexts in its declaration, then the invoked object – equivalent to o - has a reference (via a method formal) to an object – equivalent to o’. But the owners of o’ are not directly related to the owners of the invoked object o, since the method of object o received its parameters with polymorphic contexts. Again, an informal demonstration that representation containment cannot be broken – due to the strong role separation between polymorphic and non-polymorphic contexts – was conducted. A scheme for updating the semantic interpretation function (used in subject reduction to take ownership schemes to ownership structures) to handle polymorphic contexts, which is not appropriate or feasible at first glance, was also devised.
As noted in section 5, anonymous contexts are a strong variation on static visibility, and can subsume it. Polymorphic contexts are orthogonal to static visibility.

More generally, developing anonymous and polymorphic contexts has led to consideration of more expressive forms of ownership for the future. Context parameters featured in a method’s formal parameter could be augmented with descriptions of how the method is allowed to access fields of the formal parameter owned by these contexts, for example, members owned by a readonly context. In general, attaching capabilities to contexts would allow many approaches taken in literature related to [OT] to be described as extensions to ownership types. The main challenge is specifying, in a type system, the invariants for a method’s behaviour that derive from contexts’ capabilities.

Semantic interpretation (used by [OT] in subject reduction), runtime support and the data-flow analysis all aim to describe ownership structures in a program, and there is a strong similarity in their basic operation. Specifically, each can take some identifier for an object (either its dynamic object identifier or a static, textual identifier) and describe who its (possible) owner is in terms of the identifiers of other objects in the system. The similarity between the schemes is apparent only after detailed study of their formulation/implementation.

8.2 Achievements

The ownership type system is not a straightforward system to understand at first glance, and some time was spent in discussion with David Clarke, the primary author of [OT]. However, a wide variety of work has been conducted, largely implementation-based but with some theoretical treatment also.

8.2.1 Application of ownership types to Java

- Grammar based on full Java – Instead of starting with a “toy” language featuring ownership types and only simple expressions, the standard Java 1.1 grammar (as developed by a team at Sun) was extended with ownership constructs. This means that any type-correct program written in standard Java is, without annotation, also a type-correct JOT program.

- Extensions to the type rules – Type rules for class definitions and declaration at ownership types were enhanced to increase the safety of the static analysis, by checking the structure of the declared ownership type against its class definition.

- Visibility modifiers – Whereas Java’s standard visibility modifiers seek to restrict access to a class’ fields and methods depending on purely on the class and/or package of the caller, ownership types restrict access by making fields untypeable unless the caller can effectively demonstrate that it has the correct contexts available. These conflicting approaches were considered and the visibility modifiers discarded as a result.

- Static variables and methods – By considering the semantics and typical use of these features, suitable restrictions were imposed on the use of ownership types for static variable or within static methods and initializers.

- Arrays – It is important to have a simple collection resource available, and consider if there were any issues with declaring arrays at ownership types.
• **Type-casting** – This was considered and an appropriate implementation decision made to not handle it in the supported language grammar.

• **Orthogonal language features** – A number of Java language features were discovered to be orthogonal to the ownership type system, notably constructs such as the package/import facility and concurrency mechanisms.

• **Untypeables policy** – To maintain safety when external library code is used without its source, and hence (ownership) type information being available, the type-checker considers that object access paths featuring declarations at unknown types are untypeable. Therefore, they are incompatible with any references whose type is known and we ensure that ownership cannot be “given away”.

### 8.2.2 Software engineering

• **Techniques for JavaCC** - Various techniques for customising the grammar of a language and the AST to a particular application’s needs have been developed. This is not a parochial observation; JavaCC’s capabilities are so strong, particularly with respect to Java code integration inside the grammar, that its application to many other parsing problems will be helped by the work carried out here.

• **Visitor pattern** – I exploit the facility of JJTree to generate an AST supporting the visitor pattern, so there is a separation of concerns between code navigating the AST and code performing application-specific operations such as type-checking. The interface between such code is clean yet semantic information can be passed from one side to another where necessary.

• **Shaping the AST** – Starting from a non-trivial grammar like Java’s and ensuring that my application considers only those productions which are relevant led to considerable customisation of the AST generated by the parser, to simplify operation of the visitor.

• **Use of exceptions** – The nature of a type-checker is to consider a particular expression and consider whether a number of pre-conditions and predicates hold in the static environment. By implementing the type rules as a series of method invocations that complete silently if successful, and throw descriptive exceptions otherwise, it is straightforward to understand the general operation of the type-checker.

• **Data structures** – The data model is a very clean implementation of the functions and structures defined in [OT].

### 8.2.3 Extensions to ownership

• **Anonymous and polymorphic contexts** – These are novel concepts that to some extent arise fairly naturally to overcome the restrictions of the basic ownership type system in implementing more advanced data structures. We have found when experimenting with ownership models in programs that these two sorts of context can describe them, sometimes to our surprise.

• **Runtime support** - Motivated by both the desire for a complement to the static analysis provided by the type system, and by the need to check object ownership at run-time because of type-casting, the implementation is a very neat model of the ownership structures described in [OT]. Many possibilities for further work arise from this area.
Ownership Types Restrict Aliasing

• Data-flow analysis – As a final piece of analysis, a monotone framework was described to statically model a restricted version of the ownership structures that will be built at run-time. Due to the information required from the analysis, interesting observations can be made about the nature of the framework itself, although time did not permit these to be fully explored.

8.3 Further work

The major facility missing from the JOT+* language is inheritance. [OOTO] develops this and provides type rules that subsume the existing type system and add much more. This was not implemented because it would have been coding for coding’s sake; much time would have passed while a lengthy but fundamentally straightforward implementation took place.

Some areas remain under-developed; a proof of soundness for ownership types with anonymous and polymorphic contexts was considered but the timetable did not permit it. Clarke’s theories of ownership is still developing, having extended Abadi & Cardelli’s object calculus [OC] to support a more explicit notion of representation context. On the assumption that hierarchial contexts are applicable to OO programs (which seems reasonable), implementing suitable mechanisms in a programming language that capture the essence of the constructs from the calculus could be attempted. Clarke has considered this but comments, “the mathematical techniques border on the uncomputable; inheritance tends to destroy the modularity of the static analysis, thus requiring global program analysis, which is unsatisfactory.”

Support for overloaded methods is present internally, but the method resolution operation is not fully featured. The extra implementation details of resolving method invocations were not considered until after the extended contexts had been included, and further work would be needed to decide how to perform static method binding in the presence of overloaded methods, some featuring specific contexts and some featuring polymorphic contexts.

The runtime support system could be enhanced to check that, in a method call, the ownership structure of an actual parameter is that required by the type of the formal parameter. This is straightforward to implement – all the necessary information is to be found in the symbol table – but was left as further work. Improving garbage collection with runtime ownership information, and other further work for the runtime support, was detailed in section 6.4.

Ownership types can be thought of as augmenting ordinary variable declarations with roles; an object owned by a particular context has the “role” whose name is that context. A design methodology [RBO][RO] that models roles explicitly then maps roles to objects or classes could be altered to map roles to ownership contexts. Leveraging an established development method in this way would improve the acceptance of ownership types as a useful language feature for programmers.

For program documentation purposes, the type-checker could write out, in the ownership-free .java files, comments in the special form /**...*/. In these, it includes details of ownership contexts used in variable declarations and method and class signatures. The Sun tool javadoc can then be made to recognise directives in these comments and include output in the standard HTML-based documentation format. In the long run, better techniques than this for visualising ownership structures are, I believe, sorely needed.
A.jot

```jot
class A<,n> { 
norep B<,m> x; 
rep B<n> y; 
}
```

A.java

```java
class A { 
/** 
@modes A m,n 
@owner x norep 
@modes x m 
@owner y rep 
@modes y n 
*/ 
B x; 
B y; 
}
```

Producing javadoc-compatible documentation from JOT programs
Appendix A – Test cases

Note that the comment on the first line of each test case is always line #1 in the file. The command jot used to check a file is a UNIX alias for perl $JOTPATH/jot.pl, where $JOTPATH is an environment variable used by jot.pl to find the other scripts that it has to invoke (since the current working directory is where the .jot source files are, rather than the scripts’ own directory).

Class

T01.jot

// Contexts in field declaration must feature in the class signature
class A<m,n> {
    m Object a;
    n Object b;
    o Object c;  // Fails
}

> jot T01.jot
Processing T01.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0:  Reading from file T01.jot
T01.jot parsed successfully
T01.jot:6: Context o is unrecognised in class A
T01.jot: 1 error found

T02.jot

// Method signatures must be unique when ownership is removed
class A<m,n,o> {
    m Object f() {}  
    n Object f() {}  
    o Object f() {}  

    norep A<rep,norep,rep> g() {}  
    rep   A<m,n,o>         g() {}  
}

> jot T02.jot
Processing T02.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0:  Reading from file T02.jot
T02.jot parsed successfully
T02.jot:4: Method f@4 has the same post-ownership signature as f@5
T02.jot:4: Method f@4 has the same post-ownership signature as f@6
T02.jot:9: Method g@9 has the same post-ownership signature as g@8
T02.jot:8: Method g@8 has the same post-ownership signature as g@9
T02.jot:5: Method f@5 has the same post-ownership signature as f@4
T02.jot:5: Method f@5 has the same post-ownership signature as f@6
T02.jot:6: Method f@6 has the same post-ownership signature as f@4
T02.jot:6: Method f@6 has the same post-ownership signature as f@5
T02.jot: 8 errors found
Ownership Types Restrict Aliasing

T03.jot

// Types in fields must be structurally correct with respect to the class’ ownership scheme

class A<m,n> {
    rep A<rep> a1;     // Fails
    rep A<rep,rep> a2; // OK
    rep A<rep,norep,rep> a3; // Fails
}

> jot T03.jot
Processing T03.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T03.jot
T03.jot parsed successfully
T03.jot:4: Definition of class A has 2 context parameter(s), but 1 was declared
T03.jot:6: Definition of class A has 2 context parameter(s), but 3 were declared
T03.jot: 2 errors found

T04.jot

// Types in method forms must be structurally correct with respect to the class’ ownership scheme

class A<m,n> {
    void f(rep A<rep> a1) { } // Fails
    void g(rep A<rep,rep> a2) { } // OK
    void h(rep A<rep,norep,rep> a3) { } // Fails
}

> jot T04.jot
Processing T04.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T04.jot
T04.jot parsed successfully
T04.jot:4: Definition of class A has 2 context parameter(s), but 1 was declared
T04.jot:6: Definition of class A has 2 context parameter(s), but 3 were declared
T04.jot: 2 errors found
Method

T01.jot

// Contexts in method formals must feature in the class signature

class A<m,n> {
    void f(m Object x, n Object y, o Object z) { // Fails
    }
}

> jot T01.jot
Processing T01.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T01.jot
T01.jot parsed successfully
T01.jot:4: Context o is unrecognised in class A
T01.jot: 1 error found

T02.jot

// Contexts in method returns must feature in the class signature

class A<m,n> {
    m A<n,o> f() { // Fails
    }
}

> jot T02.jot
Processing T02.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T02.jot
T02.jot parsed successfully
T02.jot:4: Context o is unrecognised in class A
T02.jot: 1 error found

T03.jot

// Contexts in local variables must feature in the class signature

class A<m,n> {
    void f() {
        m Object a;
        n Object b;
        o Object c; // Fails
    }
}

> jot T03.jot
Processing T03.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T03.jot
T03.jot parsed successfully
T03.jot:7: Context o is unrecognised in class A
T03.jot: 1 error found
Ownership Types Restrict Aliasing

T04.jot

// Types in local variables must be structurally correct with respect to the class' ownership scheme

class A<m,n> {
  void f() {
    rep A<rep> a1; // Fails
    rep A<rep,rep> a2; // OK
    rep A<rep,norep,rep> a3; // Fails
  }
}

> jot T04.jot
Processing T04.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T04.jot
T04.jot parsed successfully
T04.jot:5: Definition of class A has 2 context parameter(s), but 1 was declared
T04.jot:7: Definition of class A has 2 context parameter(s), but 3 were declared
T04.jot: 2 errors found

T05.jot

// Types in method returns must be structurally correct with respect to the class' ownership scheme

class A<m,n> {
  rep A<rep> f() {} // Fails
  rep A<rep,rep> g() {} // OK
  rep A<rep,norep,rep> h() {} // Fails
}

> jot T05.jot
Processing T05.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T05.jot
T05.jot parsed successfully
T05.jot:4: Definition of class A has 2 context parameter(s), but 1 was declared
T05.jot:6: Definition of class A has 2 context parameter(s), but 3 were declared
T05.jot: 2 errors found
FieldAccess

T01.jot

// Can't type subexpression (b.c in b.c.o)

class A {
    void f() {
        rep B<rep> b = new B();
        System.out.println(b.c.o);  // Fails
    }
}

class B<m> {
    rep C<m> c;
}

class C<n> {
    n Object o;
}

> jot T01.jot
Processing T01.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)

JOTParser Version 1.0:  Reading from file T01.jot
T01.jot parsed successfully
T01.jot:6: Unable to type field access (Static visibility does not hold)
T01.jot:6: Unable to type field access (Unable to type field access)
T01.jot: 2 errors found

T02.jot

// Can't type subexpression (b.c in b.c.o)

class A {
    void f() {
        rep B<rep> b = new B();
        System.out.println(b.c.o);  // Fails
    }
}

class B<m> {
    m C<rep> c;
}

class C<n> {
    n Object o;
}

> jot T02.jot
Processing T02.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)

JOTParser Version 1.0:  Reading from file T02.jot
T02.jot parsed successfully
T02.jot:6: Unable to type field access (Static visibility does not hold)
T02.jot:6: Unable to type field access (Unable to type field access)
T02.jot: 2 errors found
T03.jot

// Can type subexpression, but non-existent field accessed

class A {
    void f() {
        rep B<rep> b = new B();
        System.out.println(b.d);      // Fails
    }
}

class B<m> {
    m C<m> c;
}

class C<n> {
    n Object o;
}

> jot T03.jot
Processing T03.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTP Parser Version 1.0:  Reading from file T03.jot
T03.jot parsed successfully
T03.jot:6: Unable to type field access (Field d does not exist in class B)
T03.jot: 1 error found

T04.jot

// Field access is successful

class A<n> {
    void f() {
        rep B<n> b;
        System.out.println(b.o);      // OK
    }
}

class B<m> {
    m Object o;
}

> jot T04.jot
Processing T04.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTP Parser Version 1.0:  Reading from file T04.jot
T04.jot parsed successfully
T04.jot typed successfully
Rewriting T04.jot without ownership annotations, into T04.java
T05.jot

// Field access is successful

class A<n> {
    void f() {
        rep B<n> b;
        System.out.println(b.o); // OK
    }
}

class B<m> {
    norep Object o;
}

> jot T05.jot
Processing T05.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTPParser Version 1.0: Reading from file T05.jot
T05.jot parsed successfully
T05.jot typed successfully
Rewriting T05.jot without ownership annotations, into T05.java

T06.jot

// Static visibility fails

class A {
    void f() {
        rep B<n> b = new B();
        System.out.println(b.c); // Fails
    }
}

class B<m> {
    rep C<m> c;
}

class C<n> {
}

> jot T06.jot
Processing T06.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTPParser Version 1.0: Reading from file T06.jot
T06.jot parsed successfully
T06.jot typed successfully
T06.jot:6: Unable to type field access (Static visibility does not hold)
T06.jot:1 error found
**T07.jot**

// Advanced context substitution

class A {
    void f() {
        rep B b = new B();
        b.c.d.o;   // OK
    }
}
class B {
    owner C<norep> c;
}
class C<m> {
    m D d;
}
class D {
    owner Object o;
}

> jot T07.jot
Processing T07.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T07.jot
T07.jot parsed successfully
T07.jot typed successfully
Rewriting T07.jot without ownership annotations, into T07.java

**T08.jot**

// Advanced context substitution

class A {
    void f() {
        rep B b = new B();
        b.c.d.o;   // Fails due to rep in b.c.d.o
    }
}
class B {
    owner C<norep> c;
}
class C<m> {
    m D d;
}
class D {
    rep Object o;
}

> jot T08.jot
Processing T08.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T08.jot
T08.jot parsed successfully
T08.jot:6: Unable to type field access (Static visibility does not hold)
T08.jot:1 error found
Ownership Types Restrict Aliasing

T09.jot

// Advanced context substitution
class A {
    void f() {
        rep B b = new B();
        b.c.d.o; // Fails due to rep in b.c
    }
}
class B {
    owner C<rep> c;
}
class C<m> {
    m D d;
}
class D {
    owner Object o;
}

> jot T09.jot
Processing T09.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T09.jot
T09.jot parsed successfully
T09.jot:6: Unable to type field access (Static visibility does not hold)
T09.jot:6: Unable to type field access (Unable to type field access)
T09.jot:6: Unable to type field access (Unable to type field access)
T09.jot: 3 errors found

T10.jot

// Static visibility fails
class A {
    void f() {
        rep B<rep> b = new B();
        System.out.println(b.c); // Fails
    }
}
class B<m> {
    norep C<rep> c;
}
class C<n> {
}

> jot T10.jot
Processing T10.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T10.jot
T10.jot parsed successfully
T10.jot:6: Unable to type field access (Static visibility does not hold)
T10.jot: 1 error found
T11.jot

// Static visibility fails
class A {
    void f() {
        rep B<rep> b = new B();
        System.out.println(b.c.o); // Fails
    }
}
class B<m> {
    m C<m> c;
}
class C<n> {
    rep Object o;
}

> jot T11.jot
Processing T11.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTPParser Version 1.0:  Reading from file T11.jot
T11.jot parsed successfully
T11.jot:6: Unable to type field access (Static visibility does not hold)
T11.jot: 1 error found

T12.jot

// Field access is successful
class A {
    void f() {
        norep B<rep> b;
        System.out.println(b.o); // OK
    }
}
class B<m> {
    norep Object o;
}

> jot T12.jot
Processing T12.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTPParser Version 1.0:  Reading from file T12.jot
T12.jot parsed successfully
T12.jot typed successfully
Rewriting T12.jot without ownership annotations, into T12.java
Ownership Types Restrict Aliasing

T13.jot

// Field access is successful
class A {
    void f() {
        norep B<rep> b;
        System.out.println(b.o); // OK
    }
}
class B<m> {
    m Object o;
}

T14.jot

// Field access is successful
class A {
    void f() {
        rep B<rep> b = new B();
        System.out.println(b.c.o); // OK
    }
}
class B<m> {
    m C<m> c;
}
class C<n> {
    n Object o;
}

T13.jot parsed successfully
T13.jot typed successfully
Rewriting T13.jot without ownership annotations, into T13.java

T14.jot parsed successfully
T14.jot typed successfully
Rewriting T14.jot without ownership annotations, into T14.java
T15.jot

// Distinguish between a rep-moded reference in "this", and a rep-moded reference in another object
class A {
    void f() {
        rep Object o;
        g(o); // OK
    }
    void g(rep Object x) { }
}
class B {
    void f() {
        norep A a = new A();
        rep Object o;
        a.g(o); // Fails
    }
}

> jot T15.jot
Processing T15.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTPaser Version 1.0: Reading from file T15.jot
T15.jot parsed successfully
T15.jot:16: Unable to type method call (Static visibility does not hold)
T15.jot: 1 error found

T16.jot

// Using "this" is optional
class A {
    void f() {
        System.out.println(o); // Fails
        System.out.println(this.o); // Fails
    }
}

> jot T16.jot
Processing T16.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTPaser Version 1.0: Reading from file T16.jot
T16.jot parsed successfully
T16.jot:5: Undeclared variable o
T16.jot:6: Unable to type field access (Field o does not exist in class A)
T16.jot: 2 errors found
T17.jot

// Using "this" is optional
class A {
    norep Object o;
    void f() {
        System.out.println(o); // OK
        System.out.println(this.o); // OK
    }
}

> jot T17.jot
Processing T17.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T17.jot
T17.jot parsed successfully
T17.jot typed successfully
Rewriting T17.jot without ownership annotations, into T17.java

T18.jot

// Using "this" is optional
class A {
    void f() {
        norep Object o;
        System.out.println(o); // OK
        System.out.println(this.o); // Fails
    }
}

> jot T18.jot
Processing T18.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T18.jot
T18.jot parsed successfully
T18.jot:8: Unable to type field access (Field o does not exist in class A)
T18.jot: 1 error found

T19.jot

// Using "this" is optional
class A {
    void f(norep Object o) {
        System.out.println(o); // OK
        System.out.println(this.o); // Fails
    }
}

> jot T19.jot
Processing T19.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T19.jot
T19.jot parsed successfully
T19.jot:6: Unable to type field access (Field o does not exist in class A)
T19.jot: 1 error found
FieldUpdate

T01.jot

// rep-owned (by the same object) objects are assignment-compatible

class A {
    rep Object a;
    rep Object b;
}

> jot T01.jot
Processing T01.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T01.jot
T01.jot parsed successfully
T01.jot typed successfully
Rewriting T01.jot without ownership annotations, into T01.java

T02.jot

// norep-owned (by the system) objects are assignment-compatible

class A {
    norep Object a;
    norep Object b;
}

> jot T02.jot
Processing T02.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T02.jot
T02.jot parsed successfully
T02.jot typed successfully
Rewriting T02.jot without ownership annotations, into T02.java

T03.jot

// Objects in the same context are assignment-compatible

class A<m> {
    m Object a;
    m Object b;
}

> jot T03.jot
Processing T03.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T03.jot
T03.jot parsed successfully
T03.jot typed successfully
Rewriting T03.jot without ownership annotations, into T03.java
Ownership Types Restrict Aliasing

T04.jot

// Objects in the same context are assignment-compatible

class A<m> {
    owner Object a;
    owner Object b;

    void f() {
        a = b; // Ok
    }
}

> jot T04.jot
JOTParser Version 1.0: Reading from file T04.jot
T04.jot parsed successfully
T04.jot typed successfully
Rewriting T04.jot without ownership annotations, into T04.java

T05.jot

// Objects in different contexts are not assignment-compatible

class A<m> {
    rep Object a;
    m   Object b;

    void f() {
        a = b; // Fails
    }
}

> jot T05.jot
JOTParser Version 1.0: Reading from file T05.jot
T05.jot parsed successfully
T05.jot:8: Declared type Object<rep>|> does not match assigned type Object<m|>
T05.jot: 1 error found

T06.jot

// Objects in different contexts are not assignment-compatible

class A<m,n> {
    m Object a;
    n Object b;

    void f() {
        a = b; // Fails
    }
}

> jot T06.jot
JOTParser Version 1.0: Reading from file T06.jot
T06.jot parsed successfully
T06.jot:8: Declared type Object<m|> does not match assigned type Object<n|>
T06.jot: 1 error found
Ownership Types Restrict Aliasing

T07.jot

// Objects in different contexts are not assignment-compatible

class A {
    rep Object a;
    norep Object b;

    void f() {
        a = b; // Fails
    }
}

> jot T07.jot
Processing T07.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T07.jot
T07.jot parsed successfully
T07.jot:8: Declared type Object<rep|> does not match assigned type Object<norep|>
T07.jot: 1 error found

T08.jot

// Objects with different contexts are not assignment-compatible

class A<m> {
    rep A<rep> a;
    rep A<norep> b;

    void f() {
        a = b; // Fails
    }
}

> jot T08.jot
Processing T08.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T08.jot
T08.jot parsed successfully
T08.jot:8: Declared type A<rep<rep> does not match assigned type A<rep<norep>
T08.jot: 1 error found
T09.jot

// Compatible after context substitution
class A {
    rep B<rep> b;
    rep C c;

    void f() {
        b.f(c); // OK
    }
}
class B<n> {
    n C myC;

    void f(n C someC) {
        myC = someC; // Both are role n
    }
}
class C { }

> jot T09.jot
Processing T09.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T09.jot
T09.jot parsed successfully
T09.jot typed successfully
Rewriting T09.jot without ownership annotations, into T09.java

T10.jot

// Objects with different numbers of contexts are not assignment-compatible
class A {
    rep B<norep> a;
    rep B<norep,norep> b;

    void f() {
        a = b; // Fails
    }
}

> jot T10.jot
Processing T10.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T10.jot
T10.jot parsed successfully
T10.jot:8: Declared type B<rep,norep> does not match assigned type B<rep|norep,norep>
T10.jot: 1 error found
LocalUpdate

T01.jot

// rep-owned (by the same object) objects are assignment-compatible
class A {
    void f() {
        rep Object a;
        rep Object b;
        a = b; // OK
    }
}

> jot T01.jot
Processing T01.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0:  Reading from file T01.jot
T01.jot parsed successfully
T01.jot typed successfully
Rewriting T01.jot without ownership annotations, into T01.java

T02.jot

// norep-owned (by the system) objects are assignment-compatible
class A {
    void f() {
        norep Object a;
        norep Object b;
        a = b; // OK
    }
}

> jot T02.jot
Processing T02.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0:  Reading from file T02.jot
T02.jot parsed successfully
T02.jot typed successfully
Rewriting T02.jot without ownership annotations, into T02.java

T03.jot

// Objects in the same context are assignment-compatible
class A<m> {
    void f() {
        m Object a;
        m Object b;
        a = b; // OK
    }
}

> jot T03.jot
Processing T03.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0:  Reading from file T03.jot
T03.jot parsed successfully
T03.jot typed successfully
Rewriting T03.jot without ownership annotations, into T03.java
// Objects in the same context are assignment-compatible

class A<m> {
    void f() {
        owner Object a;
        owner Object b;
        a = b; // Ok
    }
}

> jot T04.jot
Processing T04.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTPParser Version 1.0:  Reading from file T04.jot
T04.jot parsed successfully
T04.jot typed successfully
Rewriting T04.jot without ownership annotations, into T04.java

// Objects in different contexts are not assignment-compatible

class A<m> {
    void f() {
        rep Object a;
        m Object b;
        a = b; // Fails
    }
}

> jot T05.jot
Processing T05.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTPParser Version 1.0:  Reading from file T05.jot
T05.jot parsed successfully
T05.jot:8: Declared type Object<rep|> does not match assigned type Object<m|>
T05.jot: 1 error found

// Objects in different contexts are not assignment-compatible

class A<m,n> {
    void f() {
        m Object a;
        n Object b;
        a = b; // Fails
    }
}

> jot T06.jot
Processing T06.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTPParser Version 1.0:  Reading from file T06.jot
T06.jot parsed successfully
T06.jot:8: Declared type Object<m|> does not match assigned type Object<n|>
T06.jot: 1 error found
**T07.jot**

// Objects in different contexts are not assignment-compatible
class A {
    void f() {
        rep Object a;
        norep Object b;
        a = b; // Fails
    }
}

> jot T07.jot
Processing T07.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTPParser Version 1.0: Reading from file T07.jot
T07.jot parsed successfully
T07.jot:8: Declared type Object<rep|> does not match assigned type Object<norep|>
T07.jot: 1 error found

**T08.jot**

// Objects with different contexts are not assignment-compatible
class A<m> {
    void f() {
        rep A<rep> a;
        rep A<norep> b;
        a = b; // Fails
    }
}

> jot T08.jot
Processing T08.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTPParser Version 1.0: Reading from file T08.jot
T08.jot parsed successfully
T08.jot:8: Declared type A<rep|rep> does not match assigned type A<rep|norep>
T08.jot: 1 error found
**T09.jot**

// Compatible after context substitution

class A {
    void f() {
        rep B<rep> b;
        rep C c;
        b.f(c); // OK
    }
}

class B<n> {
    void f(n C someC) {
        n C myC = someC; // Both are role n
    }
}

class C {
}

> jot T09.jot
Processing T09.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T09.jot
T09.jot parsed successfully
T09.jot typed successfully
Rewriting T09.jot without ownership annotations, into T09.java

**T10.jot**

// Objects with different numbers of contexts are not assignment-compatible

class A {
    void f() {
        rep B<norep> a;
        rep B<norep,norep> b;
        a = b; // Fails
    }
}

> jot T10.jot
Processing T10.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T10.jot
T10.jot parsed successfully
T10.jot:8: Declared type B<rep|norep> does not match assigned type B<rep|norep,norep>
T10.jot: 1 error found
MethodCall

T01.jot

// Can't type subexpression

class A {
    void f() {
        rep B<rep> b = new B();
        b1.f();    // Fails
    }
}

class B<m> { }

> jot T01.jot
Processing T01.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTP parser Version 1.0: Reading from file T01.jot
T01.jot parsed successfully
T01.jot:6: Undeclared variable b1
T01.jot:6: Unable to type method call (Unable to type object whose method is being invoked)
T01.jot: 2 errors found

T02.jot

// Can't type subexpression

class A {
    void f() {
        rep B<rep> b = new B();
        b.f().g();    // Fails
    }
}

class B<m> {
    rep C f() { }
}

class C {
    norep Object g() { }
}

> jot T02.jot
Processing T02.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTP parser Version 1.0: Reading from file T02.jot
T02.jot parsed successfully
T02.jot:6: Unable to type method call (Static visibility does not hold)
T02.jot:6: Unable to type method call (Unable to type object whose method is being invoked)
T02.jot: 2 errors found
T03.jot

// Can type subexpression, but non-existent method invoked
class A {
    void f() {
        rep B<rep> b = new B();
        b.f1(); // Fails
    }
}
class B<m> {
    rep C f() {} // Fails
}

> jot T03.jot
Processing T03.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTP: Reading from file T03.jot
T03.jot parsed successfully
T03.jot:6: Unable to type method call (Unable to find method f1 in class B)
T03.jot:1 error found

T04.jot

// Actual parameters have correct type after context substitution
class A {
    norep Object x;
    norep Hashtable y;
    void f() {
        rep Vector v;
        int z;
        g(x, y.z()); // Fails
        g(x, v); // Fails
        g(x, z); // Fails
    }
    void g(norep Object x, norep Vector y) {} // Fails
}

> jot T04.jot
Processing T04.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTP: Reading from file T04.jot
T04.jot parsed successfully
T04.jot:11: Unable to type method call (Method signature requires Vector<norep|> for parameter #2, but UNKNOWN<norep|> is passed)
T04.jot:12: Unable to type method call (Method signature requires Vector<norep|> for parameter #2, but Vector<rep|> is passed (Trying to bind norep to rep, when norep already binds to norep))
T04.jot:13: Unable to type method call (Method signature requires Vector<norep|> for parameter #2, but int is passed)
T04.jot: 3 errors found
T05.jot

// Actual parameters have correct type after context substitution

class A<m,n> {
    rep Object a;
    m Object b;
    n Object c;

    void f() {
        g(a, c); // Fails
        g(b, c); // OK
    }

    void g(m Object y, n Object z) {
    }
}

> jot T05.jot
Processing T05.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T05.jot
T05.jot parsed successfully
T05.jot:9: Unable to type method call (Method signature requires Object<m|>
for parameter #1, but Object<rep|> is passed (Cannot map from context m to
rep))
T05.jot: 1 error found

T06.jot

// Unification prevents context overloading

class A<m,n> {
    m Object b;
    n Object c;

    void test() {
        int x;
        f(b, b); // OK
        f(b, c); // Fails
        f(c, c); // Fails
        f(x, x); // Fails
    }

    void f(m Object y, m Object z) {
    }
}

> jot T06.jot
Processing T06.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T06.jot
T06.jot parsed successfully
T06.jot:11: Unable to type method call (Method signature requires Object<m|>
for parameter #2, but Object<n|> is passed (Trying to bind m to n, when m
already binds to m))
T06.jot:12: Unable to type method call (Method signature requires Object<m|>
for parameter #1, but Object<n|> is passed (Cannot map from context m to n))
T06.jot:13: Unable to type method call (Method signature requires Object<m|>
for parameter #1, but int is passed)
T06.jot: 3 errors found
Ownership Types Restrict Aliasing

T07.jot

    // Static visibility of return type
    class A {
        void f() {
            rep A a1 = new A();
            rep Object o1 = a1.g(); // Fails
            rep Object o2 = this.g(); // OK
            rep Object o3 = g(); // OK
        }
    }
    rep Object g() { }

    > jot T07.jot
    Processing T07.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
    JOTParser Version 1.0:  Reading from file T07.jot
    T07.jot parsed successfully
    T07.jot:7: Unable to type method call (Static visibility does not hold)
    T07.jot:7: Unable to type right-hand side of assignment
    T07.jot: 2 errors found

T08.jot

    // Static visibility of method formals
    class A {
        void f() {
            rep A a1 = new A();
            rep Object o = new Object();
            a1.g(o); // Fails
            this.g(o); // OK
            g(o); // OK
        }
        void g(rep Object o) { }
    }

    > jot T08.jot
    Processing T08.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
    JOTParser Version 1.0:  Reading from file T08.jot
    T08.jot parsed successfully
    T08.jot:8: Unable to type method call (Static visibility does not hold)
    T08.jot: 1 error found
T09.jot

// Static visibility of method formals

class A<m> {
  void f() {
    rep A<rep> a1 = new A();
    m B<rep> b = new B();
    a1.g(b); // Fails
    this.g(b); // OK
    g(b); // OK
  }

  void g(m B<rep> o) { }
}

class B<n> {
  n Object o;
}

> jot T09.jot
Processing T09.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T09.jot
T09.jot parsed successfully
T09.jot: 8: Unable to type method call (Method signature requires B<rep|rep>
for parameter #1, but B<m|rep> is passed (Trying to bind rep to rep, when rep
already binds to m))
T09.jot: 1 error found

T10.jot

// Objects with different contexts, including returned objects, are not
assignment-compatible

class A {
  void f() {
    rep A a = g(); // OK
  }

  rep A g() { } // OK
}

> jot T10.jot
Processing T10.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T10.jot
T10.jot parsed successfully
T10.jot typed successfully
Rewriting T10.jot without ownership annotations, into T10.java
T11.jot

// Objects with different contexts, including returned objects, are not assignment-compatible

class A {
    void f() {
        rep A a = g(); // Fails
    }

    norep A g() { }
}

> jot T11.jot
Processing T11.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T11.jot
T11.jot parsed successfully
T11.jot:5: Declared type A<rep|> does not match assigned type A<norep|>
T11.jot: 1 error found
Return statements

**T01.jot**

```java
// Returned objects must match method return type

class A {
    void f() {
        rep A a = new A();
        return a; // Fails
    }
}
```

> jot T01.jot
Processing T01.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T01.jot
T01.jot parsed successfully
T01.jot:6: A<rep|> does not match declared return type void for method f@4
T01.jot: 1 error found

**T02.jot**

```java
// Returned objects must match method return type

class A {
    rep Object f() {
        rep Object o = new Object();
        return o; // OK
    }
}
```

> jot T02.jot
Processing T02.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T02.jot
T02.jot parsed successfully
T02.jot typed successfully
Rewriting T02.jot without ownership annotations, into T02.java

**T03.jot**

```java
// Returned objects must match method return type

class A {
    rep Object f() {
        return (new Object()); // Fails
    }
}
```

> jot T03.jot
Processing T03.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T03.jot
T03.jot parsed successfully
T03.jot:5: Object<norep|> does not match declared return type Object<rep|> for method f@4
T03.jot: 1 error found
T04.jot

// Returned objects must match method return type
class A {
    norep Object f() {
        return (new Object()); // OK
    }
}

> jot T04.jot
Processing T04.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T04.jot
T04.jot parsed successfully
Rewriting T04.jot without ownership annotations, into T04.java

T05.jot

// Returned objects must match method return type
class A<m,n> {
    rep A<m,m> f() {
        rep A<m,n> a = new A();
        return a; // Fails
    }
}

> jot T05.jot
Processing T05.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T05.jot
T05.jot parsed successfully
T05.jot:6: A<rep|m,n> does not match declared return type A<rep|m,m> for
method f@4
T05.jot: 1 error found
Arrays

T01.jot

// Distinguish between array types and reference types
class A {
    B<norep> a; // Defaults to norep
    void f() {
        norep B<norep>[] b = new B[10];
        a = b; // Fails
        b = a; // Fails
    }
}

> jot T01.jot
Processing T01.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T01.jot
T01.jot parsed successfully
T01.jot:9: Declared type B<norep|norep> does not match assigned type B<norep|norep>[]
T01.jot:10: Declared type B<norep|norep>[] does not match assigned type B<norep|norep>
T01.jot: 2 errors found

T02.jot

// Distinguish between array types and reference types
class A {
    B<norep>[] a; // Defaults to norep
    void f() {
        norep B<norep>[] b = new B[10];
        a = b; // OK
        b = a; // OK
    }
}

> jot T02.jot
Processing T02.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T02.jot
T02.jot parsed successfully
T02.jot typed successfully
Rewriting T02.jot without ownership annotations, into T02.java
Ownership Types Restrict Aliasing

T03.jot

// Instantiate an element in an array as a non-array ownership type
class A {
    void f() {
        norep B<norep>[] b = new B[10];
        b[0] = new B(); // OK
    }
}

jot T03.jot
Processing T03.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T03.jot
T03.jot parsed successfully
T03.jot typed successfully
Rewriting T03.jot without ownership annotations, into T03.java

T04.jot

// Can’t invoke methods on an array type
class A {
    void f() {
        norep B<norep>[] b = new B[10];
        b.f();
    }
}
class B<m> {
    void f() { }
}

jot T04.jot
Processing T04.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T04.jot
T04.jot parsed successfully
T04.jot:7: Unable to type method call (Attempt to invoke method f on an array reference of type B<norep|norep>[])
T04.jot: 1 error found

T05.jot

// Pass array in method call
class A {
    void f() {
        norep B<norep>[] b = new B[10];
        g(b[0]); // OK
        g(b); // Fails
    }
}

jot T05.jot
Processing T05.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T05.jot
T05.jot parsed successfully
T05.jot:8: Unable to type method call (Method signature requires B<norep|norep>[] for parameter #1, but B<norep|norep>[] is passed)
T05.jot: 1 error found
T06.jot

// Pass array in method call

class A {
    void f() {
        norep B<norep>[] b = new B[10];
        b = h(b[1]); // Fails
        b[0] = h(b[1]); // OK
        for (int i = 0; i < 10; i++) {
            b[i] = h(b[i % 2]); // OK
        }
    }
    B<norep> h(B<norep> x) { }
}

T06.jot:7: Declared type B<norep|norep>[] does not match assigned type B<norep|norep>
T06.jot: 1 error found

T07.jot

// Pass array in method call

class A {
    void f() {
        norep B<norep>[] b = new B[10];
        int[] i = new int[10];
        int j;
        g(b[0], i); // Fails
        g(b, j); // Fails
        g(b[0], j); // OK
    }
    void g(B<norep> x, int y) { }
}

T07.jot:9: Unable to type method call (Method signature requires int for parameter #2, but int[] is passed)
T07.jot:10: Unable to type method call (Method signature requires B<norep|norep>[] for parameter #1, but B<norep|norep>[] is passed)
T07.jot: 2 errors found
T08.jot

// Distinguish between primitives and arrays of primitives
class A {
    void f() {
        int[] i = new int[10];
        int j;
        g(i);       // Fails
        g(j);       // OK
        for (int k = 0; k < 10; k++) {
            g(i[k]);  // OK
        }
    }
    void g(int x) { }
}

> jot T08.jot
Processing T08.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T08.jot
T08.jot parsed successfully
T08.jot:8: Unable to type method call (Method signature requires int for parameter #1, but int[] is passed)
T08.jot: 1 error found
Constructors & initializers

**T01.jot**

// Contexts in method formals must feature in the class signature

class A<m,n> {
    A(m Object x, n Object y, o Object z) { // Fails
    }
}

> jot T01.jot
Processing T01.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0:  Reading from file T01.jot
T01.jot parsed successfully
T01.jot:4: Context o is unrecognised in class A
T01.jot: 1 error found

**T02.jot**

// Contexts in local variables must feature in the class signature

class A<m,n> {
    A() {
        m Object a; // OK
        n Object b; // OK
        o Object c; // Fails
    }
    {
        m Object a; // OK
        n Object b; // OK
        o Object c; // Fails
    }
}

> jot T02.jot
Processing T02.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0:  Reading from file T02.jot
T02.jot parsed successfully
T02.jot:7: Context o is unrecognised in class A
T02.jot:13: Context o is unrecognised in class A
T02.jot: 2 errors found
Ownership Types Restrict Aliasing

T03.jot

// Types in local variables must be structurally correct with respect to the class' ownership scheme

class A<m,n> {
A() {
    rep A<rep> a1; // Fails
    rep A<rep,rep> a2; // OK
    rep A<rep,norep,rep> a3; // Fails
}
}

rep A<rep> a1; // Fails
rep A<rep,rep> a2; // OK
rep A<rep,norep,rep> a3; // Fails

> jot T03.jot
Processing T03.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0:  Reading from file T03.jot
T03.jot parsed successfully
T03.jot:5: Definition of class A has 2 context parameter(s), but 1 was declared
T03.jot:7: Definition of class A has 2 context parameter(s), but 3 were declared
T03.jot:11: Definition of class A has 2 context parameter(s), but 1 was declared
T03.jot:13: Definition of class A has 2 context parameter(s), but 3 were declared
T03.jot: 4 errors found
Static methods & variables

T01.jot

// Static vars may feature only norep

class A<m> {
    static norep Object a; // OK
    static rep Object b;  // Fails
    static m Object c;   // Fails

    static norep A<norep> d; // OK
    static norep A<rep> e; // Fails
    static norep A<m> f;   // Fails
}

T02.jot

// Static vars may feature only norep

class A<m> {
    static {
        norep Object a; // OK
        rep Object b;  // Fails
        m Object c;   // Fails

        norep A<norep> d; // OK
        norep A<rep> e; // Fails
        norep A<m> f;   // Fails
    }
}

// Output

> jot T01.jot
Processing T01.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0:  Reading from file T01.jot
T01.jot parsed successfully
T01.jot:5: Context rep is impermissible in static type declaration
static_Object<rep|>
T01.jot:6: Context m is impermissible in static type declaration
static_Object<m|>
T01.jot:9: Context rep is impermissible in static type declaration
T01.jot:10: Context m is impermissible in static type declaration
T01.jot: 4 errors found

> jot T02.jot
Processing T02.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0:  Reading from file T02.jot
T02.jot parsed successfully
T02.jot:6: Context rep is impermissible in static type declaration
static_Object<rep|>
T02.jot:7: Context m is impermissible in static type declaration
static_Object<m|>
T02.jot:9: Context rep is impermissible in static type declaration
T02.jot:10: Context m is impermissible in static type declaration
T02.jot: 4 errors found


T03.jot

// Static vars may feature only norep

class A<m> {
    static norep Object a; // OK
    static rep   Object b; // Fails

    static {
        norep Object c;      // OK
        a = c;    // OK
        b = c;    // Fails
    }
}

jot T03.jot
Processing T03.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0:  Reading from file T03.jot
T03.jot parsed successfully
T03.jot:5: Context rep is impermissible in static type declaration
static_Object<rep|>
T03.jot:11: Type static_Object<rep|> on left-hand side is an invalid static
declaration
T03.jot: 2 errors found
Anonymous contexts

T01.jot

// Can't instantiate fields at anonymous contexts

class A {
    _ Object a;       // Fails
    void f(_ Object x) { }
}

> jot T01.jot
Processing T01.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T01.jot
Encountered "_" at line 4, column 9.
Was expecting one of:
  "}" ...
  "static" ...
  "{" ...
  "abstract" ...
  "final" ...
  ...
JOTParser Version 1.0: Encountered errors during parse.

T02.jot

// Can't instantiate local vars at polymorphic contexts

class A {
    void f(_ Object x) {
        _ Object a = new Object();    // Fails
    }
}

> jot T02.jot
Processing T02.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T02.jot
Encountered "_" at line 5, column 17.
Was expecting one of:
  "boolean" ...
  "break" ...
  ...
  "rep" ...
  "norep" ...
  "owner" ...
  <POLYMORPHIC_CONTEXT> ...
JOTParser Version 1.0: Encountered errors during parse.
Ownership Types Restrict Aliasing

T03.jot

// Basic anonymous context binding
class A<m, n> {
  m Object b;
  n Object c;

  void test() {
    f(b, c); // OK
    f(b, b); // OK
  }

  void f(_, Object x, _ Object y) { }
}

T04.jot

// Anonymously-owned objects are inaccessible
class A<a, b, c> {
  a Object x;
  b Object y;
  c Object z;
}
class B<m, n> {
  void doWork(m A<_, n, _> x) {
    System.out.println(x.x); // Fails
    System.out.println(x.y); // OK
    System.out.println(x.z); // Fails
  }
}

> jot T03.jot
Processing T03.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T03.jot
T03.jot parsed successfully
T03.jot typed successfully
Rewriting T03.jot without ownership annotations, into T03.java

> jot T04.jot
Processing T04.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T04.jot
T04.jot parsed successfully
T04.jot typed successfully
T04.jot: 4: Unable to type field access (Anonymous ownership found in type Object<_|>)
T04.jot: 6: Unable to type field access (Anonymous ownership found in type Object<_|>)
T04.jot: 2 errors found


**T05.jot**

// Anonymous contexts are assignment-incompatible in all situations

class A<m> {
    m Object x, y;
    void f(m A<_> a1, m A<_> a2) {
        a1.y = a2.x; // Fails
    }
    void g(_ Object a1, _ Object a2) {
        a1 = a2; // Fails
    }
}

> jot T05.jot
Processing T05.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0:  Reading from file T05.jot
T05.jot parsed successfully
T05.jot:7: Unable to type field access (Anonymous ownership found in type Object<_>)
T05.jot:7: Unable to type field access (Anonymous ownership found in type Object<_>)
T05.jot:7: Unable to type either side of assignment statement
T05.jot:11: Anonymous context found in types on LHS (Object<_>) and RHS (Object<_>) of assignment
T05.jot: 4 errors found

**T06.jot**

// Can’t bind anonymous context to specific context

class A<m> {
    void f(m B<_, _, _> b) {
        g(b); // Fails
        h(b); // Fails
        i(b); // OK
    }
    void g(m B<m, m, m> b2) { }
    void h(_ B<m, m, m> b2) { }
    void i(_ B<_, _, _> b2) { }
}

> jot T06.jot
Processing T06.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0:  Reading from file T06.jot
T06.jot parsed successfully
T06.jot:5: Unable to type method call (Method signature requires B<m|m,m,m> for parameter #1, but B<m|_,_,_> is passed (Trying to bind m to _, when m already binds to m))
T06.jot:6: Unable to type method call (Method signature requires B<_,m,m,m> for parameter #1, but B<_,_,_,_> is passed (Cannot map from context m to _))
T06.jot: 2 errors found
T07.jot

// Anonymous contexts are maintained across method calls

class A<m,n> {
    m Object b;
    n Object c;

    void f(_ Object x, _ Object y) {
        g(x, y); // OK
        h(x, y); // Fails
    }

    void g(_ Object x2, _ Object y2) { }
    void h(m Object x3, n Object y3) { }
}

> jot T07.jot
Processing T07.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T07.jot
T07.jot parsed successfully

T07.jot:9: Unable to type method call (Method signature requires Object<m|>
for parameter #1, but Object<_|> is passed (Cannot map from context m to _))
T07.jot: 1 error found
Polymorphic contexts

T01.jot

// Can instantiate local vars at polymorphic contexts

class A {
    void f(Object x) {
        Object a = new Object(); // OK
        Object b = new Object(); // "" is unknown
    }
}

> jot T01.jot
Processing T01.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T01.jot
T01.jot parsed successfully
T01.jot:6: Unknown polymorphic context "P found
T01.jot: 1 error found

T02.jot

// Can’t instantiate fields at polymorphic contexts

class A {
    Object a;
    void f(Object x) { }
}

> jot T02.jot
Processing T02.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T02.jot
Encountered "" at line 4, column 9.
Was expecting one of:
"boolean" ...
...<IDENTIFIER> ...
"(" ...
"rep" ...
"norep" ...
"owner" ...
"void"
JOTParser Version 1.0: Encountered errors during parse. ...
T03.jot

// Polymorphic context in return type must be declared in method formals

```java
class A {
    Object f(Object x, Object y) { } // OK
    \ Object f(Object x, Object y) { } // OK
    " Object f(Object x, Object y) { } // Fails: "" is unknown
}
```

> jot T03.jot
Processing T03.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T03.jot
T03.jot parsed successfully
T03.jot:4: Method f@4 has the same post-ownership signature as f@5
T03.jot:4: Method f@4 has the same post-ownership signature as f@6
T03.jot:5: Method f@5 has the same post-ownership signature as f@4
T03.jot:5: Method f@5 has the same post-ownership signature as f@6
T03.jot:6: Method f@6 has the same post-ownership signature as f@4
T03.jot:6: Method f@6 has the same post-ownership signature as f@5
T03.jot:6: Unknown polymorphic context ""P found
T03.jot: 7 errors found

T04.jot

// Polymorphic contexts cannot be overloaded

```java
class A<m,n> {
    m Object b;
    n Object c;

    void f() {
        g(b, b); // OK
        g(b, c); // Fails; '-m from b:Object<m>| then '-n from c:Object<n>| is attempted also
    }

    void g(Object y, Object z) { }
}
```

> jot T04.jot
Processing T04.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T04.jot
T04.jot parsed successfully
T04.jot typed successfully
T04.jot:9: Unable to type method call (Method signature requires Object<m|> for parameter #2, but Object<n|> is passed (Trying to bind 'P to n, when 'P already binds to m))
T04.jot: 1 error found

T05.jot

// Same polymorphic contexts are assignment-compatible

```java
class A {
    void f(Object x, Object y) {
        x = y;
    }
}
```

> jot T05.jot
Processing T05.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T05.jot
T05.jot parsed successfully
T05.jot typed successfully
Rewriting T05.jot without ownership annotations, into T05.java
Ownership Types Restrict Aliasing

T06.jot

// Different polymorphic contexts in formals are not assignment-compatible

class A {
    void f(Object x, Object y) {
        x = y;
    }
}

> jot T06.jot
Processing T06.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T06.jot
T06.jot parsed successfully
T06.jot:5: Declared type Object<"P"> does not match assigned type
Object<"P">
T06.jot: 1 error found

T07.jot

// Different polymorphic contexts in local vars are not assignment-compatible

class A {
    void f(Object x, Object y) {
        Object x2 = x;  // OK
        Object y2 = y;  // OK
        x2 = y2;        // Fails
    }
}

> jot T07.jot
Processing T07.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T07.jot
T07.jot parsed successfully
T07.jot:7: Declared type Object<"P"> does not match assigned type
Object<"P">
T07.jot: 1 error found
Ownership Types Restrict Aliasing

**T08.jot**

// Polymorphic contexts are mappable to other polymorphic parameters

class A<m,n> {
  m Object b;
  n Object c;

  void f() {
    g(b, c);
  }

  void g(' Object x, '' Object y) {
    h(x, y); // OK
    i(x, y); // OK
  }

  void h(' Object x, '' Object y) {
  }

  void i('' Object x, '' Object y) {
  }
}

> jot T08.jot
Processing T08.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T08.jot
T08.jot parsed successfully
T08.jot typed successfully
Rewriting T08.jot without ownership annotations, into T08.java

**T09.jot**

// Polymorphic contexts in return statements must obey method return type

class A {
  void f(' Object x, '' Object y) {
    ' Object z = g(x, y);
  }

  ' Object g(' Object x, '' Object y) {
    return x; // OK
    return y; // Fails
  }
}

> jot T09.jot
Processing T09.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file T09.jot
T09.jot parsed successfully
T09.jot:10: Object<'P> does not match declared return type Object<'P> for method g@8
T09.jot: 1 error found
T10.jot

// Polymorphic contexts in return types are inversed to the types mapped by the parameter binding

class A<m,n> { 
    void f('Object x, ` Object y) { 
        ` Object z1 = g(x, y); // OK
        ` Object z2 = g(x, y); // Fails
        ` Object z3 = g(y, x); // OK
    }
    ` Object g(` Object x, ` Object y) { 
        return x;
    }
}

T11.jot

// Using anonymous and polymorphic contexts

class A { 
    Object f(_ Object x, ` Object y) { 
        g(x, y); // Fails
    }
    Object z = new Object();
    g(z, z); // OK
    h(z, z); // OK
    h(y, z); // OK
}

MEng. Computing, Imperial College
Aggregate data structures

**Engine.jot**

```java
class Engine {
    void start() { }
    void stop() { }
}
class Driver {
}
class Car {
    rep Engine engine;
    norep Driver driver;

    Car() {
        engine = new Engine();
        driver = null;
    }

    rep Engine getEngine() {
        return engine;
    }

    void setEngine(rep Engine e) {
        engine = e;
    }

    void go() {
        if (driver != null) engine.start();
    }
}
class Main {
    void main() {
        norep Driver bob = new Driver();
        norep Car car = new Car();

        car.driver = bob; // OK
        car.go();
        car.engine.stop(); // Fails
        car.getEngine().stop(); // Fails

        rep Engine e = new Engine();
        car.setEngine(e); // Fails - different rep for the this object and the car object
    }
}
```

> jot Engine.jot
Processing Engine.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file Engine.jot
Engine.jot parsed successfully
Engine.jot:37: Unable to type field access (Static visibility does not hold)
Engine.jot:37: Unable to type method call (Unable to type object whose method is being invoked)
Engine.jot:38: Unable to type method call (Static visibility does not hold)
Engine.jot:38: Unable to type method call (Unable to type object whose method is being invoked)
Engine.jot:41: Unable to type method call (Static visibility does not hold)
Engine.jot: 5 errors found
Ownership Types Restrict Aliasing

Pair.jot

```java
class Pair<m,n> {
    m Object fst;
    n Object snd;
}

class Intermediate {
    rep Pair<rep, norep> pair1;
    norep Pair<rep, norep> pair2;

    rep Pair<rep, norep> a() {
        return pair1;
    }

    norep Pair<rep, norep> b() {
        return pair2;
    }

    rep Object x() {
        return pair1.fst;
    }

    norep Object y() {
        return pair1.snd;
    }

    void updateX() {
        pair1.fst = new Object();
    }
}

class Main {
    norep Intermediate safe;

    void main() {
        rep Pair<rep, norep> a;
        norep Pair<rep, norep> b;
        rep Object x;
        norep Object y;

        a = safe.a(); // Fails
        b = safe.b(); // Fails
        x = safe.x(); // Fails
        y = safe.y(); // OK
        safe.updateX(); // OK
    }
}
```

> jot Pair.jot
Processing Pair.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOPParser Version 1.0: Reading from file Pair.jot
Pair.jot parsed successfully
Pair.jot:40: Unable to type method call (Static visibility does not hold)
Pair.jot:40: Unable to type right-hand side of assignment
Pair.jot:41: Unable to type method call (Static visibility does not hold)
Pair.jot:41: Unable to type right-hand side of assignment
Pair.jot:42: Unable to type method call (Static visibility does not hold)
Pair.jot:42: Unable to type right-hand side of assignment
Pair.jot: 6 errors found
Course.jot

class Course\<s>\> {  
    rep Hashtable\<s,rep> marks;

    public void enrol(s Object s) {  
        rep Object r = new Object();  
        marks.put(s, r);
    }

    public void recordMarkFor(s Object s, String unit, int mark) {  
        /* If you miss the rep out below, x defaults to being norep  
         and you get assignment incompatiility /  
         rep Object x = marks.get(s);  
         marks.get(s).recordMarkFor(unit, mark);
    }
}

class Hashtable\<k,i> {  
    rep HashtableEntry\<k,i>\>[] contents;
    private int size;

    public void put(k Object key, i Object value) { }

    public i Object get(k Object key) {  
        rep HashtableEntry\<k,i> e;  
        return e.item;
    }
}

class HashtableEntry\<k,i> {  
    k Object key;
    i Object item;
}

> jot Course.jot  
Processing Course.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)  
JOTParser Version 1.0: Reading from file Course.jot  
Course.jot parsed successfully  
Course.jot typed successfully  
Rewriting Course.jot without ownership annotations, into Course.java
XStack.jot

class Link<n> {
    owner Link<n> next;
    n Object data;

    Link(n Object inData) {
        next = null;
        data = inData;
    }
}

class XStack<m> {
    rep Link<m> top;

    XStack() {
        top = null;
    }

    void push(m Object data) {
        rep Link<m> newTop = new Link(data);
        newTop.next = top;
        top = newTop;
    }

    public m Object pop() {
        rep Link<m> oldTop = top;
        rep Link<m> top = oldTop.next;
        return top.data;
    }

    boolean isEmpty() {
        return top == null;
    }
}

> jot XStack.jot
Processing XStack.jot... (showtree=0, showdebug=0, typecheck=1, runtime=0)
JOTParser Version 1.0: Reading from file XStack.jot
XStack.jot parsed successfully
XStack.jot typed successfully
Rewriting XStack.jot without ownership annotations, into XStack.java
Bibliography

[AK] Alias Killing: Unique Variables Without Destructive Reads
John Boyland. In [IWAOS].

[BALL] Balloon Types: Controlling sharing of state in data types

[BRT] Barat - A Compiler Front-End for Java
Boris Bokowski
http://www.inf.fu-berlin.de/~bokowski/barat/index.html

[BYJ] BYACC/Java Home Page, Bob Jamison
http://www.lincom-asg.com/~rjamison/byacc/

[CTA] Compile-Time Analysis of Object Oriented Programs
J. Vitek, R. N. Horspool, J. Uhl.
In International Workshop on Compiler Construction: CC’92.

[CS] CoffeeStrainer – Statically-Checked Constraints on the Definition and Use of Types in Java
http://www.inf.fu-berlin.de/~bokowski/CoffeeStrainer/index.html

[CT] Confined Types
Boris Bokowski, Jan Vitek. In [IWAOS].
http://www.inf.fu-berlin.de/~bokowski/ConfinedTypes/index.html

[CUP] CUP – LALR Parser Generator for Java
Scott Hudson, Frank Flannery, C. Scott Ananian
http://www.cs.princeton.edu/~appel/modern/java/CUP/

[DAP] Dynamic Alias Protection in Prototype-Based Languages

[DC] In e-mail discussion with David Clarke.

[DP] Design Patterns : Elements of Reusable Object-Oriented Software.
Addison-Wesley, 1994.
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Grady Booch.

[FAP] Flexible Alias Protection

John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, Richard Holt
In OOPS Messenger, April 1992.

[IAPP] Interprocedural Aliasing in the Presence of Pointers
William Landi, PhD thesis, Rutgers University. 1992

[IBM] In discussion with Andre Parrie, IBM Object Technology Practice / IBM Alphaworks
“On the Criteria to Be Used in Decomposing Systems into Modules,”

The Ins and Outs of Objects
John Potter, James Noble, David Clarke

Implementing “Object Ownership to Order”
Boris Bokowski. In [IWAOOS].

Islands: Aliasing protection in object-oriented languages

Intercontinental Workshop on Aliasing in Object-Oriented Systems, Lisbon, June 1999
In association with ECOOP99.
http://cuiwww.unige.ch/~ecoopws/iwaoos/

Java 2 Collections Framework
Sun Microsystems

Java 2 Platform Software Development Kit
Sun Microsystems

JAC – Java with Transitive Readonly Access Control
Gunter Kniesel, Dirk Theisen. In [IWAOOS].

JavaCC – The Java Parser Generator
Sun Microsystems
http://www.suntest.com/JavaCC/

The Java Language Specification

David Evans, University of Virginia
http://lclint.cs.virginia.edu/

A Mode System for Flexible Alias Protection

The Objects of Aliasing.
James Noble. In [IWAOOS]

Martin Abadi, Luca Cardelli.

Ownership Types for Flexible Alias Protection

Object Ownership for Dynamic Alias Protection
Clarke, Noble, Potter. TOOLS Pacific 99.
<table>
<thead>
<tr>
<th>Reference Code</th>
<th>Title and Details</th>
</tr>
</thead>
</table>
| [OOTO]         | Object Ownership To Order  
                 David Clarke, Ryan Shelswell, John Potter, James Noble.  
                 Microsoft Research Institute, 1999. |
| [PDPA]         | Program Decomposition for Pointer-induced Aliasing Analysis  
                 http://www.cs.rutgers.edu/pub/technical-reports/ |
| [PQ]           | Java examples from “Data Structures and Algorithms” course  
                 Arthur Crummer, University of Florida, Spring 1998  
| [PZ]           | Pizza – A substantial companion to Java  
                 Martin Odersky, Philip Wadler.  
                 http://www.cs.bell-labs.com/~wadler/pizza/ |
| [RBO]          | Roles Before Objects  
                 Doug Lea  
                 http://g.oswego.edu/dl/rp/roles.html |
| [RO]           | The “Role Object” Design Pattern  
                 Dirk Bäumer, Dirk Riehle, Wolf Siberski, Martina Wulf.  
| [SAB]          | Implementing A Cyclic Distributed Garbage Collector for a Heterogeneous System with Space Failures  
                 Sanjay Bhudia, MEng. Individual Project 1999 |
| [SF]           | Sun Forté for Java Community Edition  
                 http://www.netbeans.com/apis.html |
| [SL]           | Skip Lists: A Probabilistic Alternative to Balanced Trees  
                 http://www.cs.umd.edu/~pugh/ |
| [VLA]          | Vanilla, the open source language toolkit  
                 Distributed Systems Group, Department of Computer Science, Trinity College, Dublin  
                 http://www.vanilla.ie/ |
| [WA]           | Who's Afraid of Ownership Types?  
                 David G. Clarke, James Noble, John M. Potter. 1999 |