The Inheritance Anomaly Revisited

Ryan J. Wisnesky

August 2006

Contents

Contents i

Preface ii

Introduction 1

1 The Inheritance Anomaly and Formal Framework 2
- 1.1 An Overview of the Inheritance Anomaly . . . 2
- 1.2 An Overview of the Framework . . . 4

2 History-based Guard Languages 7
- 2.1 History-based guard languages . . . 7
- 2.2 Two tweaks . . . 8
  - 2.2.1 The first tweak: Behavior Preservation . . . 8
  - 2.2.2 The second tweak: Patching the Proof . . . 10
- 2.3 Discussion . . . 13
- 2.4 Related Work and Future Directions . . . 13
- 2.5 Conclusion . . . 15

3 Intra-object Concurrency 16
- 3.1 Methods and Messages . . . 16
  - 3.1.1 New synchronization constructs . . . 17
  - 3.1.2 Synchronization, Methods, and Messages . . . 18
- 3.2 New Types . . . 19
  - 3.2.1 Mutual Exclusion: $Types_S$ . . . 20
  - 3.2.2 Mutual Exclusion and Containment: $Types'_S$ . . . 24
  - 3.2.3 $Types_{T,S}$ . . . 25
- 3.3 Conclusion and Related Work . . . 27

4 Specification Languages 28

5 Overall Conclusion, Future Directions, and a Note on Typing 29

6 Related Work 30
- 6.1 Core Papers . . . 30

6.2 Concurrent Formal Systems 31
6.3 Anomaly Avoidance Mechanisms 33
6.4 Anomaly Generalizations 36
6.5 Anomaly Experiences 36
6.6 Surveys 37
6.7 Aspect-Oriented Programming 37
6.8 Others 38

Bibliography **41**

Introduction

The *Inheritance Anomaly* has been a thorn in the side of the concurrent object-oriented language community for 15 years. Simply put, the anomaly is a failure of inheritance and concurrency to work well with each other, negating the usefulness of inheritance as a mechanism for code-reuse in a concurrent setting.

Over the years, many researchers have proposed language constructs to mitigate the effects of the anomaly; many new languages have been designed to try to avoid the anomaly altogether. Until about ten years, all of this research was ad-hoc; there was so unifying theory to guide researchers in discovering how effective their efforts were. In fact, some argued that the anomaly wasn't even really a problem at all. todo: cite

About ten years ago, a Ph.D. thesis was written that gives a unifying theoretical framework for examing what the anomaly is and how it may be avoided. That work is the sole framework for formally examining what the anomaly is, how languages are vulnerable to it, and how it may be minimized.

Unfortunately, there has been no formal work on the topic since. Even now, languages are designed without regard to the basic results of the formal analysis and confusion about the anomaly persists.

This thesis is an attempt to extend the formal analysis of the anomaly in both breadth and depth. It builds heavily on the original work, and it is hoped that by increasing the scope of the formal work on the anomaly, others will begin to undertand and use the results and methods developed both here and in the original analysis.

Each chapter examines a previously unanalyzed aspect of the anomaly. Analysis of new languages and domains extends the breadth of the treatment, and an analysis of more expressive notions of concurrency extends the breadth of the treatment.

A brief overview of the anomaly and framework is presented in todo cite chapter 1; however, it is extremely unlikely that a reader will be able to follow this thesis without first reading the (very accessible) Ph.D. thesis [1] of Lobel Cnogorac.

Chapter 2 extends the results of the thesis to history-based guard languages.

Chapter 3 extends the results of the thesis to account for intra-object concurrency and introduces several new notions of typing in this context; a conversion procedure for generalizing the notions from the thesis to this new framework is given.

Chapter 4 extends the results of the thesis to specification languages, where an analagous anomaly has been discovered.

Chapter 1

The Inheritance Anomaly and Formal Framework

This chapter presents a quick overview of the inheritance anomaly and the cornerstone formal framework found in Lobel's thesis [1].

Definitions that are used in more than one chapter are defined in this section. Definitions that are used in only one chapter are defined where they are used.

This chapter is included as a convenient reference for many of the concepts used in the later chapters. It is in no way a replacement for [1].

Key concepts that are defined in this chapter include *behavior preservation* (inheritance entails subtyping), *incremental inheritance* (inheritance by only adding code), $Types_M$ (normal subtyping), $Types_R$ and $Types_T$ (concurrent subtyping); the formal definition of *inheritance anomaly* (mismatch between subtyping and inheritance), subtyping in concurrent languages (obeying superclass synchronization), *Lobel's Theorem* (no inheritance mechanism is perfect), *traces* (sequences of object states) and *behaviors* (sets of traces representing objects) and the *history-sensitive anomaly* (a particular anomaly instance).

1.1 An Overview of the Inheritance Anomaly

The Inheritance Anomaly is a failure of inheritance to be a useful mechanism for code-reuse that is caused by the addition of synchronization constructs (method guards, locks, etc) to object-oriented languages. When deriving a subclass through inheritance, the presence of synchronization code often forces method overriding on a scale much larger than when synchronization constructs are absent, to the point where there is no practical benefit to using inheritance at all.

A good introduction to the anomaly may be found in [14].

Example 1.1.1. A classic example of the inheritance anomaly is the `BoundedBuffer` *History-Sensitive Anomaly* [14]. Consider the following class:

```
public class BoundedBuffer {
  protected Object[] buf;
  protected int MAX;
  protected int current = 0;
```

```
  Buffer(int max) {
    MAX = max;
    buf = new Object[MAX];
  }
  public synchronized Object get() throws Exception {
    while (current<=0) { wait(); }
    current--;
    Object ret = buf[current];
    notifyAll();
    return ret;
  }
  public synchronized void put(Object v) throws Exception {
    while (current>=MAX) { wait(); }
    buf[current] = v;
    current++;
    notifyAll();
  }
}
```

This is a classic bounded buffer. The inheritance anomaly occurs when we would like to reuse our `BoundedBuffer` implementation in deriving a new subclass, `HistoryBuffer`, that behaves exactly like `BoundedBuffer`, except that it has an additional method `gget()`. `gget()` behaves exactly like `get()` except that it cannot be called immediately after `get()`. To create the new subclass requires redefining all inherited methods, as shown below:

```
public class HistoryBuffer extends BoundedBuffer {
  boolean afterGet = false;
  public HistoryBuffer(int max) { super(max); }
  public synchronized Object gget() throws Exception {
    while ( (current <= 0) || afterGet ) {
      wait();
    }
    afterGet = false;
    return super.get();
  }
  public synchronized Object get() throws Exception {
    Object o = super.get();
    afterGet = true;
    return o;
  }
  public synchronized void put(Object v) throws Exception
    super.put(v);
    afterGet = false;
  }
```

```
}
```

The reader is encouraged to examine [14] for more examples, and general discussion, of the anomaly.

Many have speculated that the anomaly occurs because of a conflict between how synchronization is expression and how inheritance works. However, in reality, the problem is that in concurrent object-oriented languages, our notions of subtyping and inheritance taken from sequential languages lead to problems. These problems are compounded because in some languages and in many programmers' minds, inheritance *is* subtyping, and vice-versa. A clear seperation between these two concepts will help the reader a great deal in the chapters that follow.

The essence of the anomaly, from a programmers perspective, is as follows: I have a class C which implements some behavior B. I have defined a subtype of behavior B, say B', and now I must create a new class C' that implements B'. C' should be able to inherit from C to reuse the code in C. However, I am forced to redefine much of C's behavior in writing C'. This re-writing is the anomaly, as it occurs far more often when concurrency is involved than when it is not.

We do not see the anomaly in sequential languages because our notion of subtyping in sequential languages is very weak. In many languages, an object is a subtype of another if it has all the methods of the parent; in other languages, subtyping is synonymous with inheritance and subtypes (usually) *must* have all the methods of the parent. Regardless of the particulars of the language, the fact remains that to create a subtype requires simply adding new methods to the inheriting class. In other words, every subtype of a class can be created through *incremental inheritance* – that is, by only adding methods to the class definition. This way of thinking about subtypes is the notion of $Types_M$ defined in [1].

In concurrent languages, we have different expectations of what it means to be a subtype. In addition to having all the methods of the parent, we want a subtype to behave like the parent with respect to synchronization. This notion is very fuzzy – formalizing it is one of the key contributions of [1]. But suffice it to say that all the subtype behaviors we would like to be able to express are not defineable using only incremental inheritance. Thus the inheritance anomaly occurs.

The contribution of the work in [1] is many-fold. Among the primary results are

1. A formalization of what subtyping means in concurrent object-oriented languages.
2. Several useful definitions of subtyping ($Types_M$, $Types_R$, $Types_f$, $Types_R$).
3. A taxonomy of anomalies generated by those notions of subtyping.
4. A definition of incremental inheritance and behavior preservation.
5. A characterization of inheritance as a syntactic notion.
6. Lobel's Theorem, which states that no inheritance mechanism is perfect.
7. An application of the methods to over a dozen languages.

1.2 An Overview of the Framework

These notions are all defined in [1], but are presented here for quick reference.

Definition 1.2.1 (Method Sequences). A sequence of methods is an ordered sequence whose members are drawn from $Keys$, where $Keys$ is the set of method names in a given language. The concatenation of two method sequences α and β is denoted $\alpha\beta$. In the bounded buffer example, $\langle put, get\rangle$ is a method sequence, and $put, get \in Keys$ and $\langle put, get\rangle \in Keys^*$. Note that a method sequence need not correspond to any particular calling sequence on an object; that is, $\langle get, put\rangle$ cannot occur for any instance of `BoundedBuffer`, but it is still considered a sequence.

Definition 1.2.2 (Incremental Inheritance). Let P and Q be classes in some language. That is, P and Q are sequences of symbols recognized by the language as representing a class; this is a characterization of inheritance as a syntactic notion. Q *incrementally inherits* from P, written $P \dashrightarrow_I Q$, if and only if Q is derivable from P through the use of inheritance without method overriding or deletion of any kind.

Definition 1.2.3 ($Types_M$). A class Q is a subtype of P, with respect to $Types_M$, if the methods of Q are a subset of those of P. We know that if Q incrementally inherits from P, then Q is a subtype of P with respect to this notion.

Definition 1.2.4 (Behavior Preservation). An inheritance mechanism is *behavior preserving* with respect to some notion of $Types$ if and only if for all classes P and Q, $P \dashrightarrow_I Q$ implies Q is a subtype of P.

In sequential languages, an object's methods may be invoked at any time. In concurrent languages, methods may not execute at any time; there are synchronization constraints that prevent this. For instance, in a locking buffer, there must be a call to unlock in between any two calls to lock. We say that the object only accepts message sequences where this property holds. If we incrementally inherit from the locking buffer, because we have not overridden methods, only added them, we expect the way this new class responds to messages to be identical to the old class, until a new method not in the old class executes. That is, if $\langle m_1, m_2, \ldots, m_n\rangle$ was acceptable to the locking buffer, that sequence must also be acceptable to the new subclass. However, once the new class receives a message that the parent can't handle, its behavior has no relation to the parent anymore. This notion of subtyping – that the subtype behaves like its supertype until a new message is received – is called $Types_T$ [1]. This intuition leads to two related notions:

Definition 1.2.5 (Traces and Behaviors). Let P be a class in some language. We say that $bec(P)$ denotes the "behavior" of P. That is, $bec(P)$ is the set of all sequences of messages that an instance of P can accept. Note that we assume that every instance belonging to a class has the same behavior. Thus, this formalism cannot express that the behavior of say, `Buffer(4)` which might be different from `Buffer(8)`. This assumption seems limiting but it is shown in [1] that it does not actually restrict the power of the formalism. In the language of *traces* [6], $bec(P)$ is the set of all possible traces from a new instance of P.

Definition 1.2.6 ($Types_T$). Let X and Y be sets of method sequences. Then $X \sqsubseteq_T Y$ denotes that X is a supertype of Y with respect to $Types_T$. Let $Super$ and Sub be classes. Then $bec(Super)$ and $bec(Sub)$ are sets of method sequences. We have that $bec(Super) \sqsubseteq_T bec(Sub)$ if and only if $bec(Super) \subseteq bec(Sub)$ and for every method sequence $u \in bec(Sub)$, $u = vz$ for some $v \in bec(Super)$ and some z, possibly empty, such that the first element of z, if it exists, never occurs in any sequence of $bec(Super)$.

To investigate the anomaly, one defines a notion of $Type$ that should be incrementally derivable from a supertype, and then examines the inheritance mechanism to see if that subtype can, in fact, be defined by incremental inheritance. If not, an inheritance anomaly has occurred. Behavior preservation under $Types_T$ holds for virtually all concurrent object-oriented languages [1].

Definition 1.2.7 ($Types_R$). $Types_R$ is a refined notion of $Types_T$ that is sufficient to define the *history-sensitivity* anomaly [1, 5].

The main result of [1] is the following theorem, which I have dubbed *Lobel's Theorem* as it is part of Lobel Crnogorac's Ph.D thesis:

Theorem 1.2.8 (Lobel). *If a language is incrementally behavior preserving with respect to $Types_T$, then it is not anomaly-free with respect to $Types_R$.*

Lobel's theorem tells us that there is no such thing as a perfect inheritance mechanism – that we either have to allow for classes that are not behavior preserving (which leads to its own problems; see [1], or live with the potential of the anomaly occuring (which, although disappointing, does not mean that the anomaly must occur in anything other than pathological cases. In fact, most attempts that claim to have solved the anomaly actually just banish it to pathological cases, which in many cases is just as good).

Chapter 2

History-based Guard Languages

In 2002, another language designed to minimize the anomaly was released: the Java-dialect JEEG.

JEEG uses a combination of method guards and temporal logic to minimize the anomaly. During execution, the JEEG runtime maintains a per-object history of method invocations, and method guards may use both temporal-logic quantifiers over the object's history and instance variables to determine if a method may be invoked. JEEG uses temporal logic for both ease of use and because a more expressive logic would add considerable runtime cost. JEEG's combination of temporal logic and method guards is too weak to solve the inheritance anomaly, so the question naturally arises as to whether allowing a stronger logic with method guards would solve the anomaly.

The cornerstone formal analysis goes a long way toward answering this question. However, the techniques in the paper cannot be directly applied to *history-based guard languages* without some tweaking. This chapter provides that tweaking.

2.1 History-based guard languages

A history-based guard language is any language that associates with each method a corresponding boolean guard. When the guard evaluates to $true$, the method may be invoked; when the guard evaluates to $false$, any attempt to invoke the method will block until the guard evaluates to true. Only one thread may be inside of an object at a time, and so methods are invoked in isolation of each other.

The guard may refer to instance variables and to the history of completed method invocations on that object. The evaluation of the guard must not change the state of the object, must be side-effect free and must be deterministic. In addition, each guard must only allow invocation of the associated method; guards may not be shared.

This definition, although intuitive, leads to a restriction of the expressive power of the language in the case where a method is invoked that in turn invokes a method with a guard. Unfortunately, this situation is not analyzed in [1], other than to say that the analysis can be extended to handle this situation. For our purposes, this means that when a method is invoked, there are two possibilities: calls from that method to other methods either 1. ignore the guard or 2. the method's guard must guarantee that other guards will also be true. That is, if method m_1 invokes m_2, then it must always be the case that if the guard for m_1 evaluates to true the the guard for m_2 evaluates to true. Alternatively, classes may be re-written to avoid this possibility. In any case, this issue is of

no relevence to the analysis.

In a history-based guard language, classes have the following structure:

```
class P {
  ivar1, ivar2, ..., ivarN
  method1, method2, ..., methodM
  method1: guard, ..., methodN: guardN
}
```

During inheritance, methods and method guards may be overridden separately. When a guard is overridden in a subclass, it is language-dependent whether or not the guard will also be evaluated in the superclass.

2.2 Two tweaks

Our goal is to prove the following theorem: *history-based guard languages are behavior preserving with respect to $Types_T$ and are therefore not anomaly-free with respect to $Types_R$.* To do so requires two tweaks:

1. We must establish that history-based guard languages are, in fact, behavior preserving with respect to $Types_T$. This result seems obvious but there is some confusion regarding *meta-level* information. That is, use of an object history feels like a meta-level construct, and some other languages employing meta-level constructs, like composition filters [3] are not behavior preserving. Therefore we establish the behavior preservation property for history-based guard languages.

2. The proof of Lobel's theorem requires a result that, according to the author, *holds for all languages that we know of.* This result holds for history-based guard languages, but the author's method for deriving it fails. We therefore present an example of how the method fails and give an alternate construction which establishes the required result, therefore allowing Lobel's theorem to apply to history-based guard languages.

These two tweaks allow Lobel's theorem to hold and thus give our main result, which tells us that no matter how strong the logic for method guards is, their use cannot solve the anomaly.

2.2.1 The first tweak: Behavior Preservation

Our first tweak is to show that history-based guard languages are, in fact, behavior preserving with respect to $Types_T$. We know that regular guard languages are behavior preserving [1], and so we must show that the addition of object histories does not alter this property. Note that the reader will definitely need to consult [1] to fully understand this section.

Theorem 2.2.1. *History-based guard based languages are incrementally behavior preserving with respect to the notion $Types_T$.*

Proof. Let *Super* and *Sub* be classes in a history-based guard language such that $Super \dashrightarrow_I Sub$. Formally, to show that history-based guard languages are incrementally behavior preserving with respect to $Types_T$, we must show that $imp_T(Sub) \subseteq imp_T(Super)$. We will not define imp_T here as we will prove an alternate property that implies this condition. However, the proof below, that proving our alternative condition is sufficient, is included for completeness.

1)	$bec(P) = \sqcap_{beh}\ imp_T(Super)$	by 4.13 in [1]
2)	$bec(Q) = \sqcap_{beh}\ imp_T(Sub)$	by 4.13 in [1]
3)	$bec(P) \sqsubseteq_T bec(Sub)$	to be proved
4)	$\sqcap_{beh}\ imp_T(Super) \sqsubseteq_T$ $\sqcap_{beh}\ imp_T(Sub)$	by 1,2,3
5)	$imp_T(Sub) \subseteq imp_T(Super)$	by 4.13 in [1]

Therefore, if we can show that $bec(Super) \sqsubseteq_T bec(Sub)$, that $bec(Super)$ is a supertype of $bec(Sub)$ with respect to $Types_T$, then the theorem is proved. To prove this requires showing two things: $bec(Super) \subseteq bec(Sub)$ and that for every method sequence $u \in bec(Sub)$, $u = vz$ for some $v \in bec(Super)$ and some z, possibly empty, such that the first element of z, if it exists, never occurs in any sequence of $bec(Super)$. We begin with the first conjunct, that $bec(Super) \subseteq bec(Sub)$:

Definition 2.2.2. Let X be an instance of some class. Then $history(X)$ denotes the sequence methods accepted by X, and $ivars(X)$ denotes the values of its instance variables. Note that we are implicitly assuming that at any point in the execution of a program, these values are always well defined.

Proposition 2.2.3. *The state of an object X in a history-based guard language is completely captured by $ivars(X)$ and $history(X)$.*

Proof. We know that in a non-history based guard language, the state of an object is completely captured by $ivars(X)$ [1]. As the only new information history-based languages add is the history of method sequences accepted by X, it follows that $history(X)$ and $ivars(X)$ determine the state of X. □

We may now begin our proof in earnest. Let P (for Parent) be a new instance of *Super* and C (for Child) a new instance of *Sub*. We will first prove two lemmas and need a definition

Definition 2.2.4 (Running). Let X be an instance of a class and u a sequence of methods. Then we may *run* u on X, and provided u is acceptable to X, we obtain a new instance X' whose state is that of X after receiving the method calls in u.

Lemma 2.2.5. *Let $u \in bec(Sub)$. Then if u contains only methods common to Super and Sub, then $u \in bec(Super)$.*

Proof. We may prove this by induction on the length of the methods in u.
Base case: Let $u = \langle\rangle$. Then $u \in bec(Super)$, vacuously. Also note that after running u, $history(C) = history(P)$ and $ivars(C) = ivars(P)$. *Induction:* Let $u = \langle m_1, m_2 \dots m_n \rangle \in bec(Sub)$, and run P and C with u. As no new methods from *Sub* are in u, we know that u must be acceptable to P. Because the code invoked is identical in P and C, we have that $history(P) = history(C)$ and $ivars(P) = ivars(C)$. We wish to show that m_{n+1} is accepted by P,

and thus that $um_{n+1} \in bec(Super)$. Note that C accepts m_{n+1} and C's guard for method m_{n+1} is identical to P's. Because $history(P) = history(C)$ and $ivars(P) = ivars(C)$, it follows that P must accept m_{n+1} by todo citeabove. As in our formalism all instances of a class behave identically, $u \in bec(Super)$. □

Lemma 2.2.6. *Let $u \in bec(Super)$. If u contains only methods common to Super and Sub, then $u \in bec(Sub)$.*

Proof. The proof is analogous to the proof of the previous lemma. □

Lemmas 2.2.5 and 2.2.6 state that until a subclass receives a method not defined by its parent, the behaviors of both the parent and child are the same. This is intuitively obvious because until a new method is received by a subclass, because no new code is executed, instances in the super and subclass are executing identical code. This result also holds for non-history based guard languages [1]. Note that this result depends on each guard only affecting the invocability of its associated method.

With the preliminaries out of the way, we may now show that $bec(Super) \subseteq bec(Sub)$. This follows immediately from 2.2.5 and 2.2.6.

Next we must prove the second conjunct, that for every method sequence $u \in bec(Sub)$, $u = vz$ for some $v \in bec(Super)$ and some z, possibly empty, such that the first element of z, if it exists, never occurs in any sequence of $bec(Super)$.

Choose $u \in bec(Sub)$. There are two possible cases:

1. u contains only methods common to *Super* and *Sub*. Then from 2.2.5 we have that $u \in bec(Super)$. Thus $u = v$ and $z = \langle\rangle$, and the definition holds.

2. u contains at least one method that *Sub* defines that is not in *Super*. In this case their must be a first method $z_0 \in u$ such that z_0 is defined in *Sub* but not *Super*. We may therefore write u in the form $u = vz_0z'$, where v contains only methods common to *Super* and *Sub*. Note that both v and z' may be empty.

It is obvious that $z \in bec(Sub)$; this is the prefix property of traces. If v is empty, $v \in bec(Super)$ by definition. Otherwise, v contains only methods common to *Super* and *Sub*, and by Lemma 2.2.5 $v \in bec(Super)$. z_0 cannot occur in any sequence of $bec(Super)$ by definition. Thus the definition holds.

We have thus shown that Definition 4.11 in [1] holds, and hence history-based guard languages are incrementally behavior preserving with respect to $Types_T$. □

2.2.2 The second tweak: Patching the Proof

The proof of Lobel's theorem rests on an a result that does not necessarily hold for history-based guard languages. To state the result we first need a definition.

Definition 2.2.7 (States). Let Z be a set of method sequences. Then $state(Z)z$ denotes the set of method sequences from Z acceptable after z is received. That is, $state(Z)z$ is the restriction of Z to only sequences that begin with z. So, in essense, $state(bec(Q))q$ is the set of method sequences that an instance of Q can accept after it receives method q. This is a more formal definition of running from above.

The proof of Lobel's theorem given in [1] rests on the following result that does hold for history-based guard languages even though the given construction fails:

Cornerstone assumption: Let P and Q be classes. Whenever $P \dashrightarrow_I Q$ and $z \in bec(Q)$ then it is possible to construct new classes P', Q' such that

$$\begin{array}{ll} P' \dashrightarrow_I Q' & \text{and} \\ bec(Q') = state(bec(Q))z & \text{and} \\ bec(P') = state(bec(P))w & \text{for some } w \in bec(P) \end{array}$$

In the context of most languages this means that P' and Q' differ from P and Q only in the values of their instance variables. This is because the state of an instance is determined by the values of its instance variables. But in other languages like JEEG, the state of an instance also depends on the history of method invocations in addition to instance variables. Importantly, this history can not be modified by changing instance variables. (Note that the requirement for some $w \in bec(P)$ seems artificial; it requires that whenever we have an instance of Q that has been mutated, we must be able to mutate an instance of P using only method calls such that we can change the instance variables of the instances to obtain the new inheritance relationship between P' and Q'. It is not obvious that this property is true; we take it here on faith. It is possible that if this property were false, the entire proof would collapse, and so elucidating why this property is true would make an excellent addition for any future work.)

The method for obtaining P' and Q' is to change the instance variables in P and Q. This is not sufficient in history-based languages. However, we can still construct P' and Q' in a different way. The classes P' and Q' will not have as much as a resemblance to P and Q as they would have had in the original construction, but they still do exist. To see how his method fails, consider these classes:

```
class P {
  method p() {}
  guard p when lastEvent == p || null
}

class Q extends P {
  method q() {}
  guard q when lastEvent == q || null
}
```

Choose $z = \langle q \rangle$. Assume we can create classes $P' \dashrightarrow_I Q'$ such that $bec(Q') = state(bec(Q))q$ and $bec(P') = state(bec(P))w$ for some $w \in bec(P)$. Class P's behavior is that it accepts any number of calls to $p()$, and so for all $z \in bec(P)$, $bec(P') = state(bec(P))z = \{p\}^*$. For class Q', note that $bec(Q')= state(bec(Q))\langle q \rangle = \{q\}^*$.

According to [1] we should be able to change the state of instance variables in P and Q to obtain these new classes. But since P and Q have no instance variables, we must have that $P = P'$ and $Q = Q'$. Note that $\langle p \rangle \in bec(Q)$ but $\langle p \rangle \notin bec(Q')$. Therefore, $bec(Q) \neq bec(Q')$ and $Q = P$, a contradiction, and so the method fails.

The failure occurs because in non-history based languages, the guards may only refer to instance variables. In history-based languages such as JEEG, the guards are also able to quantify over previous method invocations. Because this history is not captured in the object state in instance variables, the construction fails.

We can, however, construct new classes P' and Q', we just have to use more extreme measures. Basically, we have to rely on the fact that every class with history-based guards has the same behavior as another class with only normal guards and an explicit history instance variable. In other words, we are relying on the fact that the non-history based fragment of a history-based language is really just as powerful as the whole language.

We need to figure out how to construct, from a class P, a new class $\hat{P}$ such that $bec(P) = bec(\hat{P})$ and $\hat{P}$ has only guards that refer to instance variables. Given this new class, we may apply the old construction and the proof procedes correctly. So, our approach is to duplicate P, but add "logging." That is, for each instance variable in P, we must add a list of values that the variable has been through. Likewise, we must add a list to store when every method is invoked. Then we must add code to P such that whenever a method is invoked, the list is updated, and whenever an instance variable is changed, the associated list is also updated. For instance:

```
class P {
 ... ivarX = ?...
 ... method1(...) { ... ivarX = 2; ...; }
 ... guard1 ...
}
class P^ {
  ... ivarX = ?...
  ... method1(...) { ...;
    ivarX = 2; ivarXList.add(2); methodList.add("method1");    }

  ... guard1 newGuard1Body ...
       //see below

  List methodList;
  List ivarXlist;
}
```

We know that however we are going to re-write the guards, we will need access to the same information stored in the object history. Therefore we must add a List for method invocations and for instance variable histories, if the history-mechanism allows quantification over those.

From here we must simply re-write the guard bodies. We can take advantage of the fact that we can essentially use the same decision procedure that was already in place; because we have stored all the information that the runtime had access to, we should be able also use whatever technique was in use before, but refer to the new instance variables. For instance,

```
class P {
  method p() {}
```

```
  guard p when lastEvent == p || null
}

class P^ {
  List methodList;
  method p() { methodList.add("p"); }
  guard p when methodList.get(methodList.size()-1) == "p" || null
}
```

In this example, the language runtime would implicitly be executing something equivalent to

```
methodList.get(methodList.size()-1) == "p" || null}
```

when it evaluates the guard for p in class P.

It is easy to see that this procedure produces new classes equivalent to the old ones. Also note that an explicit appeal to the already existing decision procedure for method guards by the runtime is not needed. Because an object's history is finite, it is easy to write a decision procedure that is guaranteed to halt.

By showing that classes P' and Q' exist, via their construction from $\hat{P}$ and $\hat{Q}$, the proof of Lobel's theorem may continue. The proof resumes by applying this result to specially constructed classes P and Q to demonstrate a case of non-behavior preservation with respect to $Types_T$, thus establishing Lobel's theorem by contradiction.

2.3 Discussion

Both tweaks are incremental improvements and extensions to the work in [1]. The first tweak is instructive in that even in [1], there is not an explicit proof of behavior preservation for a specific language. The proof is based on the assumption that the state of an object is captured completely by its instance variables and method invocation history; therefore, if a language is history-based with guards but violates this assumption (see related work), then this proof will fail. Fortunately, most history-based guard languages conform to this assumption.

The second tweak relies on the non-history based subset of a history-based language to create classes that allow the proof in [1] to resume. In a sense, this result is unsurprising because a history-based guard language is really a superset of a guard-based language, and because the non history-based subset is already known to allow the assumption to hold, it follows that the whole language will allow the assumption to hold because adding a history isn't really adding information a class can't maintain for itself. So, a history really isn't a meta-level construct that would allow for non-behavior preservation. Note, however, that is is relatively easy to allow guards to become truly meta-level constructions; this is discussed in the related work.

2.4 Related Work and Future Directions

A separate formal treatment of the *state partitioning anomaly* with guards todo this may be found in [13].

The method in [1] does not depend on a specific semantics for inheritance. However, there is current work on concurrent, object-oriented formal systems with specific semantics that has run into the anomaly [12].

Many proposals for avoiding the anomaly have been proposed over the years. Typically, the language is presented as solving many examples of the inheritance anomaly but no formal proof is given. In general, these proposals tend to be overly optimistic. A few proposals may be found in [7] and [8], and a good overview in [5]

Just as this work is an extension of the work on guard based languages in [1], there are many extensions of history-based guard languages that would be interesting to analyze. Two languages are discussed below.

An extension of a history-based guard language with "access guards" is presented in [10]. In this language, each method has a corresponding guard that may refer to instance variables and the object's invocation history. However, the guard may also contain an access modifier:

- *Observe:* In this mode, other processes may also observe the attributes of the object, however, no process can change the attributes.
- *Weak exclusion:* Any access to the object is done as an atomic transaction.
- *Strong exclusion* No other access to the object is allowed until either the current guard is evaluated to false or the current method execution is complete.

In short, this language allows for multiple threads to be executing inside of an object at one time, and the access guards control how this concurrency works. In our analysis, only one thread may be allowed inside of an object at one time, in a way somewhat equivalent to their *strong exclusion*. To analyze this language, one would need to extend the work on intra-object concurrency found in [1]. This is partially done in chapter 2.

Another language, proposed in [11], reverses the relationship between methods and guards to allow for a form of incremental inheritance. That is, rather than associating each method with a guard, the language associates a guard with some number of methods. In this language, synchronization has the form:

```
property P : enables method
property Q : disables method
property Z : enables all-except method
```

It seems like most of this language could be translated into a regular history-based guard language by taking, for each method m, the conjunction of the properties that enable that method and the negation of the properties that disable it and using this as the method guard for m. However, the `all-except` method is problematic; it affects derived classes. If a superclass declares a property that enables `all-except m`, then methods in subclasses are also enabled by this property. So in a very real sense, this is a truly meta-level mechanism that is not behavior preserving [1]. In addition, different ways of resolving conflicts between guards, i.e. if one property enables a guard and another disables it, may lead to languages with different properties.

Behavior preservation may not necessarily be bad; in some cases it may be advantageous to drop behavior preservation in exchange for more expressive synchronization mechanisms, and so an analysis beyond the extremely minimal treatment in [1] of this language would be quite useful.

2.5 Conclusion

This chapter presents two modifications to the work in [1] that allow the mechanisms therein to be applied to history-based guard languages. The first tweak establishes that history-based guard languages are, in fact, behavior preserving with respect to $Types_T$. The second tweak fixes a small gap in the proof that languages cannot be both behavior preserving with respect to $Types_T$ and anomaly-free with respect to $Types_R$, thereby allowing the main result of [1] to be applied: *history-based guard languages are behavior preserving with respect to $Types_T$ and are therefore not anomaly free with respect to $Types_R$.*

Chapter 3

Intra-object Concurrency

In the cornerstone formal analysis, every object was assumed to only allow one thread to be executing inside it at any time. For many languages, including most industrial-strength languages like Java and C++, objects may have multiple threads executing in them at once and have method invocations overlap in time. This chapter takes the suggestion in the cornerstone work and develops a notion of intra-object concurrency.

3.1 Methods and Messages

We examine internal concurrency by changing the domain of message sequences, as suggested in [1]. Before, message sequences consisted of methods; that is, for message sequence u, $u \in Keys^*$. Now, we must create a new set $Keys_{intra}$ so that for every $key \in Keys$, we now have $key_s, key_e \in Keys_{intra}$. That is, the domain of $Keys_{intra}$ is no longer methods, but messages denoting the start (key_s) and end (key_e) of method invocations. From this point forward, *message* and *method* will have distinct meanings; *methods* are found in classes and method sequences, and *messages* are found in message sequences.

Also note that we may now apply our old notions of typing to these new message sequences, as shown in [1]. That is, when A and B are sets of message sequences, a subtyping relation $\sqsubseteq$ is defined exactly as before, but using $Keys_{intra}$ instead of $Keys$. The full details are described in [1], but the mechanism is simple: simply draw keys from $Keys_{intra}$ rather than $Keys$. Note that we are also using the assumption that message sequences must have some correspondence with reality: in any message sequence u, for any $m \in Keys$, there can not be an $m_e \in u$ without a corresponding $m_s \in u$ earlier in the sequence. This assumption precludes sequences that cannot be the trace of any possible execution of an object, for instance, with the `Buffer` class, $\langle get_e, get_s \rangle$ is clearly non-sensical and is prohibited. This assumption is used occasionally in the definitions that follow.

Definition 3.1.1 (Methods from messages). Let u be a sequence of messages, and m a method. Then $m \in methods(u)$ if and only if $m_s \in u$ or $m_e \in u$. Thus, $methods(u)$ denotes all the methods that are mentioned in u.

Let S be a set of message sequences. Then $Methods(S) = \bigcup_{s \in S} methods(s)$. Note that we differentiate $Methods$ from *methods* only by capitalization. As $Methods$ is a straightforward generalization of *methods*, this nomenclature seems reasonable.

Because of the previous definition of $Methods$ for a set of sequences of methods in [1], the function $Methods$ is effectively overloaded – given a set of either message or method sequences, it will return the methods mentioned in the sequence. Context will always make clear which version of $Methods$ we are using.

With this distinction between messages and methods made, we may now begin to define more expressive synchronization constructs.

methods:	$Keys^*_{intra} \rightarrow \mathcal{P}(Keys)$
$Methods$ (new):	$\mathcal{P}(Keys^*_{intra}) \rightarrow \mathcal{P}(Keys)$
$Methods$ (old):	$\mathcal{P}(Keys^*) \rightarrow \mathcal{P}(Keys)$

Figure 3.1: Functions for Methods.

3.1.1 New synchronization constructs

Because intra-object concurrency allows for more expressive synchronization constraints, we would like to examine intra-object concurrency constraints in detail in order to discover how they generate inheritance anomalies. (To see that intra-object concurrency allows for more expressive constructs, note that interleaving two methods line-by-line is impossible without intra-object concurrency, whereas it is relatively easy to force atomic method execution with intra-object concurrent constructs. In fact, it is usually the case that the concurrent constructs available for intra-object concurrency are a superset of atomic method constructs. This section begins our analysis of these new synchronization constructs.

Definition 3.1.2 (Subsequence). Let u be a sequence of messages. Then $u[n, m)$ denotes the subsequence of u between indices n, inclusive, and m, exclusive. For instance, if $u = \langle m_1, m_2, m_3, m_4 \rangle$, then $u[0, 2) = \langle m_1, m_2 \rangle$. Note that index 0 represents the first element of a sequence.

The notion of a subsequence is straightforward. We use it to define how often a method has occurred:

Definition 3.1.3 (Method count). Let u be a sequence of messages. Then the count of a message m at index n, denoted $count(m, u, n)$ is the number of times m occurs in $u[0, n+1)$. Note that this count includes the index n.

The method count of a sequence is also straightforward. It is needed primarily to define *active methods*.

Definition 3.1.4 (Active methods). Let u be a sequence of messages. A method m is *active* with degree k at index n, denoted $k = active(m, u, n)$, is defined as $active(m, u, n) = count(m_s, u, n) - count(m_e, u, n)$. The *degree* of activity is the difference between the count of m_s and the count of m_e.

Intuitively, a method m is active ($degree > 0$) at index n if there is still an invocation of m that has not yet completed. Note that by our sanity condition on message sequences the degree of a method can never be less than zero. We may use this definition to define when methods are mutually exclusive.

Definition 3.1.5 (Mutual Exclusion). Let p and q be methods, and u a sequence of messages. Then methods p and q are *mutually exclusive in* u, written $mutex(p, q, u)$ if and only if $\forall n \leq |u|$ $(p \neq q \wedge active(p, u, n) + active(q, u, n) \leq 1) \vee (p = q \wedge active(p, u, n) \leq 1)$. (A method may be mutually exclusive with itself if only one invocation of it is active at a time.) For example, in $\langle m_s, m_e, n_s, n_e \rangle$, methods m and n are mutually exclusive, and in $\langle m_s, m_e, m_s, m_e$, m is mutually exclusive with itself. In the sequence $\langle m_s, n_s, m_e, n_e$, no methods are mutually exclusive.

Let S be a set of message sequences, and $p, q \in Methods(S)$. Then p and q are *totally mutually exclusive in* S, denoted $Mutex(p, q, S)$ if and only if $\forall u \in S, mutex(p, q, u)$. For example, if $S = \{\langle m_s, m_e, n_s, n_e \rangle, \langle m_s, m_e, m_s, m_e \rangle\}$, we have $Mutex(m, m, S)$ and $Mutex(m, n, S$.

For a sequence of messages, two methods are mutually exclusive if there is no point in the sequence where they are both active – both methods being active would indicate that both methods are executing simultaneously. In the case of a set of message sequences, two methods are mutually exclusive if they are mutually exclusive in each message sequence.

subsequence :	$Keys^*_{intra} \times \mathbb{N} \times \mathbb{N} \rightarrow Keys^*_{intra}$
active:	$Keys \times Keys^*_{intra} \times \mathbb{N} \rightarrow \mathbb{N}$
$mutex$:	$Keys \times Keys \times Keys^*_{intra} \rightarrow \mathcal{B}$
$Mutex$:	$Keys \times Keys \times \mathcal{P}(Keys^*_{intra}) \rightarrow \mathcal{B}$

Figure 3.2: Newly defined functions for intra-object concurrency.

3.1.2 Synchronization, Methods, and Messages

We would like to see how our definitions of methods and messages are related. In [1], methods always ran to completion before another method could begin. Therefore we can represent a sequence of methods in the old system ($Keys^*$) as an equivalent sequence of messages in the new system ($Keys^*_{intra}$) such that every method sequence is a message sequence with *maximal* mutual exclusion. This section defines this concept.

Definition 3.1.6 (Projection). Let u be a sequence of methods. That is, $u \in Keys^*$. Then the *projection* function that maps u into the domain of message sequences, denoted $\mu'(u)$, is defined as follows. Let m be a method and let α and β be method sequences. (Note that $\alpha\beta$ is the concatenation of α and β.) The definition of μ' is given by:

$\mu'(\langle\rangle) = \langle\rangle$ and
$\mu'(\langle m \rangle) = \langle m_s, m_e \rangle$ and
$\mu'(\alpha\beta) = \mu'(\alpha)\mu'(\beta)$

Furthermore, let S be a set of method sequences. Then the projection function μ of S into the domain of sets of message sequences is defined as $\mu(S) = \{\mu'(u) \mid u \in S\}$. Note that $|\mu(S)| = |S|$ (i.e. the sizes of S and $\mu(S)$ are the same), and that there is straightforward a bijection between $\mu(S)$ and S, namely $u \Leftrightarrow \mu'(u)$.

Intuitively, the projection function μ' maps sequences of methods into sequences of messages such that every method in u is totally mutually exclusive to all other methods in the projected sequence. For example, $\mu'(\langle \alpha, \beta, \gamma \rangle) = \langle \alpha_s, \alpha_e, \beta_s, \beta_e, \gamma_s, \gamma_e \rangle$.

The projection function μ extends this definition to sets of method sequences in a straightforward way. For instance, if $S = \{\langle\alpha\rangle, \langle\beta, \gamma\rangle\}$, then $\mu(S) = \{\langle\alpha_s\alpha_e\rangle, \langle\beta_s, \beta_e, \gamma_s, \gamma_e\rangle\}$. We may now formalize some properties of projection.

Proposition 3.1.7 (Bijection). *Let u be a sequence of methods and S a set of method sequences. Then $u \in S$ if and only if $\mu'(u) \in \mu(S)$.*

Proof. This property stems from the bijective nature of μ.

First, let $u \in S$. By construction, $\mu'(u) \in \mu(S)$.

Secondly, let $\mu'(u) \in \mu(S)$. Then by construction there must be some $u \in S$ such that $\mu'(u) \in \mu(S)$ (there is simply no other way for $\mu'(u)$ to appear in $\mu(S)$); therefore $u \in S$. □

Also note that it is possible to define μ^{-1} and μ'^{-1}; however, these are necessarily partial functions. That is, they are undefined on message sequences and sets of message sequences that cannot be constructed via projection. Therefore, in the proofs that follow we will write "there must exist a sequence in the pre-image of $\mu(S)$" rather than using either μ^{-1} or μ'^{-1} directly.

Proposition 3.1.8 (Equivalence). *Let S be a set of method sequences. Then $Methods(S) = Methods(\mu(S))$. (Note that we are using both definitions of Methods here.)*

Proof. Let $m \in Methods(S)$. Therefore there exists a $u \in S$ such that $u = \langle\ldots m \ldots\rangle$. Therefore $\mu'(u) = \langle\ldots m_s, m_e \ldots\rangle$. By proposition 3.1.7, $\mu'(u) \in \mu(S)$ and hence $m \in Methods(\mu(S))$. The other direction follows similarly. □

We are now in a position to define maximal mutual exclusion and show how this property interacts with projection: because of the way in which method sequences are projected as message sequences, the new message sequence always has maximal mutual exclusion.

Proposition 3.1.9 (Maximal Mutual Exclusion). *Let S be a set of method sequences. Then $\forall p, q \in Methods(\mu(S))$, $mutex(p, q, \mu(S))$.*

Proof. Let $u \in \mu(S)$. By construction of μ', whenever a m_s appears in u it is immediately followed by a m_e. Therefore no distinct methods are active at the same time, and only one invocation of m may be active at one time. □

$$\begin{array}{|ll|}\hline \mu': & Keys^* \rightarrow Keys^*_{intra} \\ \mu: & \mathcal{P}(Keys^*) \rightarrow \mathcal{P}(Keys^*_{intra}) \\ \hline\end{array}$$

Figure 3.3: Projection Functions.

3.2 New Types

Our goal is to use this newly developed machinery to define various new notions of typing that will be useful when we examine the inheritance anomaly for intra-object concurrency. Note that it is possible to define new, useful types without the specific mechanisms we developed – this path is

simply one option out of many, although this path is not entirely arbitrary. The motivations for these choices of typing are explained in each subsection.

The general approach suggested by this section is to take a commonly used synchronization mechanism and formalize how the mechanism behaves by examining the traces of objects that use this mechanism. For instance, in the first section below that defines $Types_S$, the notion of method guards is formalized as mutual exclusion between methods. Other synchronization constructs, for instance semaphores, can be formalized in the same way, although it has been this author's experience that the properties on traces required to define more complex synchronization constructs quickly become extremely complex. In short, what is needed is a general theory of synchronization based on object-traces in the spirit of CSP.

Also note that it is possible to "work backwards" and "distill" interesting notions of subtyping by examining arbitrary sets of message sequences. However, these notions need not correspond to anything in reality since they are generated by arbitrary traces, and arbitrary traces may not be generated by real-world synchronization constructs. (Of course, these distilled notions may be interesting in their own right.) One way to implement this approach is to ask the following question: given a set of message sequences S, what is the set of predicates P such that $\forall p \in P, s \in Sp(s)$? With P known, any $p \in P$ may correspond to a synchronization property. The difficult part is to determine which p's should be further examined. We do not take this more unorthodox approach here.

3.2.1 Mutual Exclusion: $Types_S$

Our first notion of subtyping is based purely on synchronization expressed through mutual exclusion.

Definition 3.2.1 ($Types_S$). Let $Super$ and Sub be sets of message sequences. Then $Super \sqsubseteq_S Sub$ if and only if $\forall p, q \in Methods(Super)$, $Mutex(p, q, Super) \rightarrow Mutex(p, q, Sub)$.

Intuitively, we have that $Super \sqsubseteq_s Sub$ if and only if whenever two methods are totally mutually exclusive in $Super$, they are totally mutually exclusive in Sub.

However, before we may analyze $Types_S$, we must first establish that $\sqsubseteq_S$ is, in fact, a preordering [1]. To do so we must show that it is reflexive and transitive.

Proposition 3.2.2. *$\sqsubseteq_S$ is a preordering.*

Proof. Reflexivity is obvious from the definition.

To prove transitivity, let $A \sqsubseteq_S B$ and $B \sqsubseteq_S C$. We must show that $A \sqsubseteq_S C$. That is, that $\forall p, q \in Methods(A)$, $Mutex(p, q, A) \rightarrow Mutex(p, q, C)$. Let $p, q \in Methods(A)$, and assume that p and q are totally mutually exlusive in A. Because $A \sqsubseteq_S B$, they are totally mutually exclusive in B. And since p and q are totally mutually exclusive in B, because we have $B \sqsubseteq_S C$, we have that p and q are totally mutually exclusive in C. Therefore we have established that $Mutex(p, q, A) \rightarrow Mutex(p, q, C)$. □

Of course, there are legitimate situations where we might want a subclass to disregard the mutual exclusion properties of its parent, but this possibility is not explored here.

Finally, note that there is no possibility of "accidentally" identifying methods as mutually exclusive. That is, when we examine the anomaly we use sets of message sequences as behavior sets for objects, and behavior sets for objects contain every possible trace of that object, not just traces that happen during any particular execution. So, if it is possible for an object to run two methods concurrently, then there will be a message sequence in the behavior set of that object where that happens, and therefore our mutual exclusion property, which must hold for every trace in an object, will fail for that object. Hence, no methods will be identified as mutually exclusive that are not, in fact, mutually exclusive. Note, however, that two methods being mutually exclusive in a set of traces does not necessarily imply that any class with that behavior will have synchronization code between those two methods. Thus, our definition of mutually exclusive is solely based on traces, and not on synchronization code. In this sense, methods may be "accidentally" identified as mutually exclusive without there being synchronization code between them if the mutual exclusion in the traces comes from a different property of the class's code.

Remark. Our notion of $Types_S$ is, in a way, a formalization of the following maxim: a subclass should only further restrict the synchronization properties of its parent. We have only the machinery to do this for mutual exclusion, but machinery for other synchronization constructs could be developed. In a sense, restricting synchronization in subclasses "feels right." So much so, in fact, that this notion has been independently developed here and in other papers, most notably [15]. In this paper, the authors argue that subtypes must only restrict the synchronization behavior of their parents through the use of additional negative guards. Negative guards have the form "disable method m when p", and by allowing subclasses to only add negative guards, it can be guaranteed that subclasses only restrict synchronization behavior. That is, if a method is disabled in a superclass during a given run, it should also be disabled in a subclass during the same run. One potential benefit of this approach is that common synchronization code can then be factored up the inheritance hierarchy. Interestingly for us, there is also a discussion in this paper as to what extent languages using this proviso generate subtypes via inheritance; the paper's authors say that their approach entails inheritance not leading to subtyping, but they are not working within the same subtyping framework as we are. In our terms, their constructs would guarantee subtyping, except that they allow a "disable all methods except x" construct which essentially forces inheritance to become not behavior preserving. As repeated discovery of a concept usually implies there is some substance to it, we take this as an indication that this path toward understanding intra-object concurrency can be a fruitful one. Unfortunately, the paper itself did not clear up the issues regarding the anomaly way [1] did, although it seems like pushed further, the paper could have.

The Inheritance Anomaly in $Types_S$

Before we may examine the inheritance anomalies in $Types_S$, there is an important issue created by intra-object concurrent that needs addressing. In our earlier notion of concurrency, where methods were always atomic, a method was not considered to be executing until the synchronization constructs guarding the method were true. That is, evaluation of method guards was not part of the execution of the method. Therefore, the following two code snippets would generate identical sets of traces:

```
class Buffer {
```

```
  List list;
  void push when list.size() < MAX { ... }
  Object pop when list.size() > 0 { ... }
}
```

and

```
class Buffer {
  List list;
  synchronized void push() {
    while (list.size() == MAX) {
      wait();
    }
    ...
  }
  synchronized Object pop() {
   while (list.size() == 0) {
     wait();
   }
  }
```

In the second code snippet, the use of the `while-wait` construct is essentially the use of a method guard. In analyzing code in Java-like languages, for all intents and purposes, a method is not considered to be executing until the `while-wait` construct falls through. This convention is adopted because semantically, these explicit method guards and the use of `while-wait` are identical. Treating these two cases identically leads to no problems and is quite handy.

However, in dealing with intra-object concurrency, treating these two cases the similarly must be adopted by fiat rather than based on an argument like above. This is because it is clear that a method guard evaluating does not lead to the start of a method because method guards may be evaluated many times before becoming true and a method may only start once. Therefore, traces from snippet one look something like this:

$$\langle \text{ guard evaluates to false } N \text{ times and then true once } m_s, m_e \rangle$$

However, when using `while/wait`, it seems reasonable that traces from the second snippet could look like this:

$$\langle m_s, \text{ while evaluates to false } N \text{ times and then falls through } m_e \rangle$$

. In other words, in the second snippet, we can view the `while-wait` as part of the method body. This view is justified because, after all, the `while-wait` code is, in fact, inside the method.

Thus we are faced with a choice: to consider these `while-wait` constructs as method guards proper (the *identical approach*), and by convention treat snippets one and two the same even though control has reached into the body during evaluation, or two differentiate the two cases (the *differentiating approach*). A priori there is no real reason to consider one approach superior to the other; differentiating the cases leads to an enhanced ability to distinguish between guard-based

and Java-like languages and treating them the same leads to a unification of semantic concepts that allows us to treat more languages uniformly. For the purposes of illustrating the inheritance anomaly below, we will treat `while-wait` constructs as method guards – we will take the identical approach – because it makes it easier to see occurrences of the anomaly. This is because we end up with "extra" traces when take the differentiating approach. That is, for every trace of the form $\langle ..., pop_s, pop_e ... \rangle$ in the identical approach, there are many other traces in the differentiating approach, where the pop_s is "slid" to left different amounts. That is, in the identical approach, pop_s occurs as far to the right as possible; that is, when the guard finally evaluates to true. But in the differentiating approach, pop_s occurs whenever the guard is first evaluated, which is by definition a larger set of positions than when the guard evaluates to true. So, if we take the differentiating approach, then we will have extra traces that correspond to the same semantic behavior. These extra traces make it more difficult to illustrate the anomaly, as it becomes impossible to reason about where a m_s must occur.

Unfortunately, the two different approaches lead to different sets of traces for object behaviors in Java-like languages, and therefore lead to different instances of the inheritance anomaly. This is simply the price we must pay for analyzing more expressive systems of concurrency. Studying the relationship between the two approaches in terms of the anomalies found in only one or in both is an interesting area for further study.

With that unpleasantness out of the way, we may proceed with an example of the anomaly in $Types_S$, and because $Types_S$ is a relatively broad definition it is relatively easy to come up with examples. I have dubbed one class of anomalies that comes up immediately *Enabling Anomalies*.

Example 3.2.3. Consider a `Buffer` class, written in a Java-like language:

```
class Buffer {
  List list = new List();
  synchronized Object pop() {
    while (list.size() == 0) {
      wait();
    }
    return list.remove(0);
  }
  synchronized void push(Object o) {
    list.add(o);
    notifyAll();
  }
}
```

Note that, as described above, we treat the `wait` statements are method guards; that is, reaching the point of waiting does not count as beginning the invocation of the method.

It is clear that *pop* and *push* are mutually exclusive because of the `synchronized` keyword. In an anomaly-free behavior-preserving language we expect that for any subtype of `Buffer` we would be able to create, through incremental inheritance, a class that implements that subtype and vice versa. Moreover, any subtype of this class, with respect to $Types_S$, must have this mutual exclusion property of `push` and `pop`. One such subtype is the type generated by the subclass `EnablingBuffer`:

```
class EnablingBuffer extends Buffer {
  boolean enabled = false;
  synchronized Object pop() {
    while (!enabled) {
      wait();
    }
    return super.pop();
  }
  synchronized void push(Object o) {
    while (!enabled) {
      wait();
    }
    super.push(o);
  }
  synchronized void enable() {
    enabled = true;
    notifyAll();
  }
}
```

It is clear that *push* and *pop* are still mutually exclusive in `EnablingBuffer` because of the `synchronized` keyword. Therefore, $bec(EnablingBuffer) \sqsubseteq_S bec(Buffer)$. However, the type implemented by `EnablingBuffer` is not incrementally derivable from `Buffer`, because at the very least, the synchronization behavior of `put` and `get` must be changed from their behavior in the superclass (a new variable tracking if the buffer is enabled must be added), resulting in an instance of the inheritance anomaly.

Note that this is just one example of an enabling anomaly; it is very likely that there are other kinds of anomaly induced by $Types_S$. Also note that it is likely that there are some languages that prevent this class of anomaly.

Our next notion of subtyping strengthens $Types_S$ to prevent anomalies like above.

3.2.2 Mutual Exclusion and Containment: $Types'_S$

Definition 3.2.4 ($Types'_S$). Let *Super* and *Sub* be sets of message sequences. Then $Super \sqsubseteq'_S Sub$ if and only if $Super \sqsubseteq_S Sub$ and $Super \subseteq Sub$.

As always, we must check that $\sqsubseteq'_S$ is a preordering:

Proposition 3.2.5. *$\sqsubseteq'_S$ is a preordering.*

Proof. Reflexivity of $\sqsubseteq'_S$ follows from the reflexivity of $\subseteq$ and $\sqsubseteq_S$. Likewise, transitivity also follows from the transitivity of $\subseteq$ and $\sqsubseteq_S$. □

Notice that this definition of subtyping prevents the `EnablingBuffer` anomaly. This is because we do not have that $bec(Buffer) \subseteq bec(EnablingBuffer)$. For example, let $u = \langle put_s, put_e, get_s, get_e \rangle \in$

$bec(Buffer)$. Because every sequence in $bec(EnablingBuffer)$ must begin with $\langle enable_s, enable_e \rangle$, we have $u \notin bec(EnablingBuffer)$.

However, $Types'_S$ is vulnerable to its own class of anomalies. Some of these anomalies stem from being able to modify the semantics of the subtype.

Example. Consider another subclass of `Buffer`:

```
class GenerousBuffer extends Buffer {
  synchronized void put(Object o) {}
  synchronized Object get() {return null;}
}
```

It is clear that *put* and *get* are both still mutually exclusive in this subtype because of the use of the `synchronized` keyword. Moreover, because `GenerousBuffer` accepts any sequence of `get` and `put`, it is obvious that $bec(Buffer) \subset bec(GenerousBuffer)$. Therefore, we have that $bec(Buffer) \sqsubseteq'_S bec(GenerousBuffer)$, but we cannot incrementally derive from `Buffer` to get a class that behaves like `GenerousBuffer` because to do so requires modifying the synchronization constraints of `get` and `put`. Of course, some languages may be immune to this type of anomaly. (One example would be a language that would allow us to initialize the buffer to contain an infinite sequence of nulls. In that case, any sequence of calls to `get` and `put` would be allowed.)

This notion of subtyping is anathema to our usual intuition of subtype. That is, this subclass is too different from the superclass for us to really allow this as a genuine instance of the inheritance anomaly. Therefore, rather than using $Types'_S$, we will take our "usual" notion, $Types_T$, and restrict it by adding this synchronization property of $Types_{S'}$.

3.2.3 $Types_{T,S}$

Definition 3.2.6 ($Types_{T,S}$). Let *Super* and *Sub* be sets of message sequences. Then $Super \sqsubseteq_{T,S} Sub$ if and only if $Super \sqsubseteq_T Sub$ and $Super \sqsubseteq_S Sub$. Note that we are using $\sqsubseteq_S$ rather than $\sqsubseteq_{S'}$ because $\sqsubseteq_T$ entails containment, making the use of $\sqsubseteq_{S'}$ redundant.

Intuitively, we have $Super \sqsubseteq_{T,S} Sub$ when we have two separate properties:

1. $Super \sqsubseteq_T Sub$, so that the *Sub* object behaves exactly like a *Super* object until a message not in *Super* is received, and

2. Whenever two methods are mutually exclusive in *Super*, they are mutually exclusive in *Sub*. That is, *Sub* does not violate the mutual exclusion properties of *Super*, even after a new message is received.

Intuitively, this definition of type captures many behaviors we would like to model. For instance, if a `HardDisk` class enforces mutual exclusion between methods `read` and `write`, then we would want a subclass to also not to allow `read`s during `write`s, and vice-versa, at any time – including after using functionality not found in `HardDisk`. Note that $Types_{T,S}$ captures this notion much more so than $Types_S$ does. As always, however, there may be some situations where this definition of subtyping is inadequate.

However, before we can analyze $Types_{T,S}$, we must first establish that is is a notion of subtyping by showing that it is a preordering:

Proposition 3.2.7. *$\sqsubseteq_{T,S}$ is a preordering.*

Proof. Reflexivity is obvious from the definition of $\sqsubseteq_{T,S}$. To prove transitivity, let $A \sqsubseteq_{T,S} B$ and $B \sqsubseteq_{T,S} C$. $A \sqsubseteq_T C$ follows from transitivity of $\sqsubseteq_T$ [1], and $A \sqsubseteq_S C$ follows from the transitivity of $\sqsubseteq_S$. □

At first glance it is not immediately obvious that $Types_T \neq Types_S$ – that is, it is not obvious that $Types_T$ must be different from $Types_{T,S}$. Consider a new set of message sequences $BrokenBuffer$ such that $Buffer \subset BrokenBuffer$ and with the follwing property: for every $\alpha \in Buffer$ and $\beta \in \{get_s, put_s\}^*$, there is a $z \in BrokenBuffer$ such that $z = \langle \alpha, break_s, break_e, \beta \rangle$. (Technical note: α and β must be finite, so this concatenation is allowed.) That is, $BrokenBuffer$ behaves like $Buffer$ until message *break* is received, at which point it will accept any sequence of get_s and put_s. Clearly, because of this, `get` and `put` are not synchronized in $BrokenBuffer$, and so $BrokenBuffer$ is not a subtype under $Types_S$. However, $BrokenBuffer$ a valid subtype under $Types_T$. Hence, $Types_S \neq Types_T$. (Also note that for a more realistic example, rather than drawing β from $\{get_s, put_s\}^*$, we could instead draw β from sane, in the sense of section 1, sequences from $\{get_s, put_s, get_e, put_e\}^*$ and obtain the same result.)

There is an interesting relationship between $Types_{T,S}$ and our projection functions. The rest of this section explores this relationship.

Proposition 3.2.8 (Methods to Messages Homomorphism). *μ is a homomorphism with respect to $\sqsubseteq_{T,S}$ and $\sqsubseteq_T$ when moving from the methods domain to the messages domain.*

Proof. Let $Super$ and Sub be sets of method sequences. Suppose $Super \sqsubseteq_T Sub$. We must show that $\mu(Super) \sqsubseteq_{T,S} \mu(Sub)$. This requires showing that $\mu(Super) \sqsubseteq_T \mu(Sub)$ and $\mu(Super) \sqsubseteq_S \mu(Sub)$.

We now prove part 1. We know that $Super \sqsubseteq_T Sub$, and we wish to show that $\mu(Super) \sqsubseteq_T \mu(Sub)$. We must show that $\mu(Super) \subseteq \mu(Sub)$ and for every message sequence $u \in \mu(Sub)$, $u = vz$ for some $v \in \mu(Super)$ and some z, possibly empty, such that the first element of z, if it exists, never occurs in any sequence of $\mu(Super)$.

First we establish that $\mu(Super) \subseteq \mu(Sub)$. Let $u \in \mu(Super)$. We wish to show that $u \in \mu(Sub)$. By construction of $\mu(Super)$, there is some sequence $u' \in Super$ such that $\mu'(u') = u$. Because $Super \sqsubseteq_T Sub$, we know that $Super \subseteq Sub$, and thus that $u' \in Sub$. Therefore $\mu'(u') \in \mu(Sub)$ (by Proposition 3.1.7) and as $u = \mu'(u')$, we have $u \in \mu(Super)$ and hence $\mu(Super) \subseteq \mu(Sub)$.

Lastly we must establish that for every message sequence $u \in \mu(Sub)$, $u = vz$ for some $v \in \mu(Super)$ and some z, possibly empty, such that the first element of z, if it exists, never occurs in any sequence of $\mu(Super)$. Let $u \in \mu(Sub)$. There are two cases:

1. $u \in \mu(Super)$. Then $z = \langle\rangle$ and $v = u$.

2. $u \notin \mu(Super)$. By construction of $\mu(Sub)$, there is some $u' \in Sub$ such that $\mu'(u') = u$. Note that $u' \notin Super$ (if it were, we'd have $u = \mu'(u') \in \mu(Super)$, violating our assumption). So, because $u' \notin Super$, since $Super \sqsubseteq_T Sub$ we know that $u' = v'z'$ for some v' in $Super$ and some z', such that the first element of z' never occurs in any sequence of $Super$. By applying μ' to $u' = v'z'$ we obtain $u = \mu'(u') = \mu(v'z') = \mu(v')\mu(z')$. That is, $u = \mu(v')\mu(z')$

where $v' \in Super$ and z' starts with a method not in $Super$. By proposition 3.1.7, Because $v' \in Super$, $\mu(v') \in \mu(Super)$, and because $z' \notin Super$, we have, $\mu'(z') \notin \mu(Super)$.

This proves part 1.

We now prove part 2, that $\forall p, q \in Methods(\mu(Super)), Mutex(p, q, \mu(Super)) \rightarrow Mutex(p, q, \mu(Sub))$. Let $p, q \in Methods(\mu(Super))$ and suppose that $Mutex(p, q, \mu(Super))$. We must show that $Mutex(p, q, \mu(Sub))$. Because we have $Super \sqsubseteq_T Sub$, we know that $Methods(Super) \subseteq Methods(Sub)$. It follows from proposition 3.1.7 that $Methods(\mu(Super)) \subseteq Methods(\mu(Sub))$. Thus, $p, q \in Methods(Sub)$. By proposition 3.1.9 we have maximal mutual exclusion and hence we know that $\forall a, b \in Methods(\mu(Sub)), Mutex(a, b, \mu(Sub))$. Therefore, we have that $Mutex(p, q, \mu(Sub))$ holds, and we have proven part 2. □

Essentially, $Types_{T,S}$ is the first new notion of subtyping that feels correct. $Types_{T,S}$ is a more restrictive notion that $Types_T$, and so we analyze it to generate new anomaly types.

The Inheritance Anomaly in $Types_{T,S}$

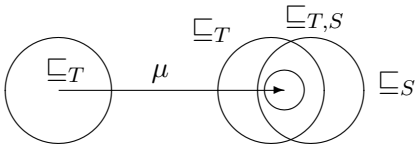

Figure 3.4: Relationship between Methods and Messages. The domain on the left is pairs of sets of method sequences; on the right, pairs of sets of message sequences.

3.3 Conclusion and Related Work

Placeholder.

Chapter 4

Conclusion

Stepping back and viewing this work in a broader scope has led me to the following conclusions:

- *The relationship between inheritance and subtyping is still not fully understood.* Mainly, the problem of *open recursion* still leads to debate in the community as to whether or not it is a great idea or a terrible one. Open recursion is the term for allowing subclasses to override methods, which in effect means that code written in a supertype is now dependent on code in a subtype. To fans of well defined control structures, this is anathema. But it does seem to capture some common scenarios in OO languages miraculously well. I believe that a better, more structured but still powerful method is needed as part of the next step for the evolution of OO languages away from arbitrary open recursion. You could say that the OO community needs a "goto considered harmful" type of paper to address open recursion.

- *It is extremely difficult to tell if further classification of the anomalies is useful.* Lobel's original classificaton of the anomalies was quite useful because it categorized all of the examples present in the literature. The classification appears natural, and it seems like there is little to be gained from increasing the precision of the taxonomy. For intra-object concurrency, the situation is reversed. Here, there are no examples from the literature driving a classification system; any classification system must essentially start from scratch, and the goal must be to discover useful properties about the anomalies independently of how the anomalies are encountered in practice. Thus, this work is open ended, and its usefulness may therefore be questioned.

- *State-aware type systems are becoming ever more important.* A state-aware type system seems almost like a contradiction in terms. But if there is one thing that the inheritance anomaly shows us, it is that most people think of type systems, in a concurrent setting, in terms of state; that is, a concurrent object's subclasses should behave like the superclass, which is a property on state. I therefore believe that the single most useful research direction suggested by this work is to determine the limits to which this notion of behavioral, concurrent subtyping can be expressed in a type system. Especially as concurrent programs become more common, programmers will demand better type systems to help avoid problems like the anomaly.

Chapter 6

Related Work

This chapter provides short summaries of various papers that have had even the slightest impact in this work. For instance, there are a number of papers on concurrent formal systems; these works are included because it is believed to be possible to characterize this anomaly in terms of a formal system. The papers are broadly categorized. The bibliography has references where some of these papers (and every paper cited) may be found.

6.1 Core Papers

- *Jeeg: A Programming Language for Concurrent Objects Synchronization.* G. Milicia and V. Sassone. This paper introduces the java dialect Jeeg, which provided the motivation for examining history-based guard languages. It contains a brief description of the inheritance anomaly and a classification for some of the anomaly types. In a certain sense, Jeeg's use of sync blocks is reminiscent of aspect-oriented solutions to the anomaly. Most of the paper is spent describing the temporal logic used in method guards and showing that this technique is powerful enough to describe all star-free regular states of a given object. Their belief is that because these star-free states are definable using method guards, it is impossible for the anomaly to occur in classes where the states of the objects of that class can only be star-free. In practice, this simply means that there is a separation between concurrency constructs and other language constructs; in [1], this is not considered to be a solution to the anomaly. An interesting extension of this work would be to further characterize how objects with star-free states behave.

- *Classifying Inheritance Mechanisms in Object-Oriented Programming.* L. Crnogorac et al. This paper is one of a series of papers that culminate with Lobel's thesis. This paper is interesting in that it spends more time examining the nature of typing from an informal viewpoint itself than in further works. Most of the notions used in the thesis and in this work can be traced back to this paper.

- *Inheritance Anomaly: A Formal Treatment.* L. Crnogorac et al. This is another paper in the series of anomaly papers that culminate with Lobel's thesis. This is by far the most formal of the papers, examining typing in a very formal way.

- *Formal Analysis of Inheritance and Specialization* L. Crnogorac. This is the Ph.D. thesis that is the culmination of the series of papers investigating the anomaly. Its first section is the clearest introduction to the formal framework; its analysis of typing is not as in-depth as some of the other papers, indicating a refinement of the idea over time. The second half of the thesis is an investigation, via this framework, into the properties of a language developed for the AI field that minimizes the anomaly. In fact, it was the occurrance of the anomaly in heavily inheritance-based and concurrent agent systems that was the motivation behind this approach to understanding the anomaly.

- *Synchronization Constrains With Inheritance: What is not Possible–so what is?* A. Yonezawa et al. This paper is one of the few other formal treatments of the anomaly aside from Crnogorac's. The authors of this paper originally identified the inheritance anomaly; this paper is also one of the very first *formal* analysis of the anomaly. In many ways, Lobel's first paper can be considered an extension of this one. Many of his formalisms, for instance *beh*, are drawn from this paper, as are the roots of state-based semantics. Lobel's development of a general cause for the anomaly is in many ways a generalization of this paper's treatment of the anomaly for accept sets. A proof that the anomaly exists in accept-set languages is, along with the formalization, the main contribution of this paper. Lobel's treatment stands very much independently, however.

- *A Behavioral Notion of Subtyping.* B. Liskov et al. This classic paper introduces the "subtype requirement": Let $\phi(X)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S when S is a subtype of T. A related notion has often been dubbed the "Liskov substitution principle." Liskov is interested mostly in safety properties such as invariants or history properties; this definition of subtyping is the foundation on which Lobel's notion of subtyping is based, which is to say that Liskov presents two notions of subtyping that obey this property, and Lobel's various notions of subtyping also obey this property.

6.2 Concurrent Formal Systems

- *Inheritance in Concurrent Objects.* C. Laneve. This paper introduces a concurrent process calculi that models inheritance and that is strong enough to model languages using enabled sets and method guards. A variety of interesting synchronization and record constructs are discussed. An example of the inheritance anomaly translated into the process calculus is presented, and an in-depth formal treatment of how the anomaly manifests itself is presented. The papers also gives various philosophical arguments for the position that the anomaly is not something that should be expected (i.e. the anomaly is a problem). In addition, there are intuitive connections between this work and other works that have found that neither method guards nor enabled sets are strong enough to solve the anomaly.

- *Inheritance in the Join Calculus.* C. Laneve. This paper presents the join calculus and a way in which inheritance may be modeled in it. In addition, it demonstrates different kinds of inheritance anomaly (partitioning of states and history sensitive) inside the calculus.

- *Inheritance is not Subtyping.* W. Cook et al. This paper draws clear the distinction between inheritance and sybtyping, arguing that equating them leads to either restrictions on inheritance or inheritance relations that may not generate subtypes. They present a notion of inheritance formulated inside a lambda-calculus that is not behavior preserving. Many terms found in Lobel's work are defined here; for instance, incremental inheritance. In addition, the paper discusses the use of F-bounded inheritance, which has ramifications to more modern languages like Java. The paper also discusses the denotational semantics of inheritance and argues why several contemporary process calculi are not sufficient.

- *A Denotational Semantics of Inheritance and its Correctness.* W. Cook et al. This paper gives a denotational semantics for inheritance and a collection of related results. It is related to Cook's other paper on inheritance and subtyping.

- *The Reflexive CHAM and the Join-Calculus.* C. Fournet et al. This paper introduces the notion of a Chemical Machine as a notion of concurrency on par with Actors. The authors present a join-calculus with functional and object-oriented features, along with various synchronization primitives.

- *A Calculus for Concurrent Objects.* K. Fisher et al. This paper presents a concurrent object calculus. The system is unique in its use of typing, and because it is λ based. The inheritance anomaly is not discussed in this paper, nor is inheritance. These would be two logical directions to take this work.

- *A Concurrent Lambda Calculus with Futures.* Smolka et al. This paper presents a basic typed lambda calculus with futures. In is unrelated to object calculus but serves as a useful contrast.

- *Typed Concurrent Objects.* V. Vasconcelos. This paper presents a process calculi for typed concurrent objects that allows for a kind of polymorphism. The inheritance anomaly is not discussed; the paper serves as an interesting bridge between Fisher's work and Laneve's work (or rather, the respective "campus" of formal systems.

- *Quiet and Bouncing Objects: Two Migration Abstractions in a Simple Distributed Blue Calculus.* S. Dal-Zilio. This paper presents the concept of migration control abstractions being a part of a process calculi itself. As such, it might have applications to how different (say, non-behavior preserving) synchronization can be modeled in process calculi.

- *A Concurrent Object Calculus: Reduction and Typing.* A. Gordon et al. This paper presents Cardelli's object calculus extended with synchronization primitives for mutexes. It is an interesting example of how process calculi may be modified to suit other needs, and is also a in-depth treatment of the calculus itself.

- *Conformance and Refinement of Behavior in pi-calculus.* C. Canal et al. This paper presents a process algebra along with a mechanism for inheritance among processes. This algebra is used to tell if a process is a refinement of another or if a process is compatible with another. In a certain sense, traces may be viewed as processes, and so an approach like this could be

used to more formally define the properties examined in this thesis's chapter on intra-object concurrency.

6.3 Anomaly Avoidance Mechanisms

- *Integrating Concurrency and Object-Orientation using Boolean, Access and Path Guards.* This paper is a good example of how researchers have thought that they were solving the inheritance anomaly even though they were only achieving separation of inherited code. Unlike other papers, however, this paper presents a potentially non-behavior preserving inheritance mechanism. The logic that it uses in its guards is very similar to the logic used by Jeeg.

- *Virtual Synchronization: Uncoupling Synchronization Annotations from Synchronization Code.* S. Reitzner. This paper presents an archetypal notion of separation of concurrent code from sequential code and a classic demonstration of how inheritance that is split into concurrent and other parts narrows the scope of the changes that need to take place during inheritance. The paper is interesting in that an event model is used to couple the synchronization code from the sequential code.

- *A Concurrent Abstraction Model for Avoiding Inheritance Anomaly in Object-Oriented Programs.* P. Agrawal et al. This paper presents the notion of separation of concurrency concerns with others, and presents a rich language in which concurrency can be defined. This language includes semaphores as well as other, potentially not behavior preserving constructs. In addition, the paper shows how different types of the anomaly may be avoided in this scheme.

- *Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming.* J. Meseguer. This is an un-orthodox paper that takes a view opposite to Lobel: the anomaly is caused by synchronization code, and therefore, synchronization code should be eliminated. The paper describes how to use a special kind of re-write logic to describe concurrency in object-oriented systems. In addition, inheritance is broken into two different mechanisms of inheritance. This mechanism is so unusual that it is unclear to what extent the normal methods may even be used to examine it. todo: maude in chart? if so, analysis done

- *Integration of Concurrency Control in a Language with Subtyping and Subclassing.* C. Baquero et al. This paper describes the language BALLOON, a language which separates inheritance from subtyping and subtyping and inheritance. This language is briefly examined in Lobel's thesis.

- *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Lanaguages.* A. Yonezawa et al. This paper, although just a draft, is one of the best informal analysis of the problems with conventional techniques for overcoming the anomly. The paper goes through about a dozen various proposals for minimizing the anomaly and groups them according to their mechanisms (and thus provides a taxonomy); the paper also is filled with examples of how these proposals fail. The paper also discusses two proposals that are more unorthodox and potentially more powerful: use of reflection and meta-level constructs (which are almost

certainly not-behavior preserving when coupled with inheritance) and the syntactic elimination of synchronization code (i.e. use transaction). I consider the paper to be the capstone of the early work on the anomaly.

- *Abstracting Object Interactions Using Composition Filters.* A. Yonezawa et al. This paper presents the notion of "composition filters", which are an extremely general mechanism for specifying how functions may be composed during message passing. For example, an object may send a message m to another object; this message would be intercepted by various filters and be processed before being passed on. This notion of filtering can be used to create abstract communication types, where filters allow object to be dropped in to a synchronization scheme. Viewed in a certain way, composition filters form a synchronization mechanism which can be studied using the formal framework, although a number of adaptations must be made because composition filters are such an unusual and general synchronization mechanism.

- *The Universe Model: An Approach for Improving the Modularity and Reliability of Concurrent Programs.* R. Stirewalt et al. This paper can be interpreted as a sort of "proto-Jeeg," in the sense that it shows how propositional logic may be used to constraint and examine concurrent behavior of objects. The approach taken is to characterize various objects in the program as part of "universes", and various processes have control over different universes at different times. The effect of this kind of synchronization method on inheritance is not examined, although there is a short paragraph noting that because state changes inside objects do not play as large a part in concurrency control as in other languages, it is possible that the anomaly would be minimized.

- *Abstracting Process-to-Function Relations in Concurrent Object-Oriented Applications.* C. Lopes et al. This paper examines abstracting away concurrent code from sequential code, but with an eye toward freeing the programmer from having to deal with both OO-concerns and concurrent concerns at the same time; the thinking is that OO-programming will eventually be stymied in a concurrent environment because programmers will be spending their type on non-OO related issues. As such, the issue of the inheritance anomaly is not raised, although the authors argue that such an approach reduces maintanince requirements, and the anomaly certainly contributes to maintanence costs. In short, this paper presents a general model for abstracting concrrency and other concerns away from sequential code in a pre-aspect oriented manner.

- *(Objects + Concurrency) and Reusability - A Proposal to Circumvent the Inheritance Anomaly.* C. Lengauer et al. This paper shows how the language Maude may be used to overcome the anomaly; in addition, the paper shows how Maude overcomes related failures of inheritance, for instance during their new concept of "sub-configuration." This paper is an interesting exploration in that it uses an unusual language and views the anomaly in an unusual light.

- *Guarded Methods vs. Inheritance Anomaly.* S. Ferenczi. This paper argues that nested method guards may be used to defeat the anomaly. This paper is a good example of the necessity of a formal analysis of the anomaly, as the paper simple solves the usual examples of the anomaly and does not prove that the mechanism solves all instances (in fact, Lobel's theorem tells us that if the author's claim is true, then the inheritance mechanism used in

the underlying language is not behavior-preserving. But the underlying language is so typical that we can safely assume it is behavior-preserving, and so we know that method guards used in this way do not solve all instances of the anomaly, in direct contrast to the title). The guards may be inherited, leading to a separation of concurrent from sequential code that helps but does not eliminate the anomaly.

- *Concurrent Object-Oriented Languages and the Inheritance Anomaly.* D. Kafura et al. This is a good foundational paper that surveys existing concurrent OO languages and provides an interesting taxonomy of these languages. Most importantly, the paper also presents the notion of accept-sets and discusses how accept sets are used to overcome various instances of the anomaly. The paper also has one of the best descriptions of the context surrounding the anomaly that I've come across: "Can inheritance be used to organize and specialize synchronization policies in the same way that inheritance is used to organize and specialize an object's functionality?" The treatment of accept sets is formal, in contrast to most of the other papers examining accept sets.

- *Inheritance and Synchronization with Enabled-Sets.* C. Tomlinson et al. This paper presents the notion of enabled sets, which are quite similar in spirit to accept sets. The paper also provides connections with Actors, which is somewhat unusual and enlightening. This concurrency mechanism, when used with inheritance, is very likely to be non-behavior preserving because of the way derived classes inherit enabled-sets of parents.

- *Behavior Equation as Solution of Inheritance Anomaly in Concurrent Object-Oriented Programming Languages.* B. Leung et al. This paper argues that separating concurrent and sequential code increases resistance to the anomaly. In addition, the notion of a behavior equation is introduced: a class's behavior equation describes how states may evolve and how concurrency interacts with the object. In a sense, the equation is not unlike a hybrid of accept sets and explicit state management. The paper also treats multiple inheritance, which is unusual for a paper about the anomaly.

- *Solving Inheritance Anomaly Problem by State Abstraction-Based Synchronization.* Y. Kuno. This paper is interesting for several reasons. First, the concept of abstracting state so that the internal state of a data type is available from the outside in a controlled fashion is an interesting concept related to OOP. Secondly, because of this, in their language/scheme the type system reflects an objects synchronization behavior, which leads to an interesting entanglement with inheritance and substitutability. Finally, this unusual scheme is pitted against the three classic types of anomaly and shown to prevail, although a formal proof is not given.

- *Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages.* S. Frolund. This paper takes the insighful position that inherited synchronization constraints should be at least as restrictive in derived classes. Such a notion is compatible with many of this thesis's notion of subtyping (for instance, the property that when two methods are mutually exclusive in a superclass they are mutually exclusive in a subclass). As such, this paper can be viewed as having the seeds for the idea that these types of constraints should play some part in the inheritance process to make sure the anomaly does not occur.

Interestingly, it appears that solving the anomaly was not the primary motivation for the paper, but this type of restricted inheritance would substantially eliminate the anomaly (at the cost of expressiveness). The paper's treatment of the anomaly is minimal, but the paper does indicate that the integration of behavior notions of subtyping and inheritance were being thought of in other contexts besides work on the anomaly. In addition, incremental inheritance as a concept is examined here, and the use of guards based on patterns is somewhat related to Jeeg and other guard languages.

6.4 Anomaly Generalizations

- *Real-Time Specification Inheritance Anomalies and Real-Time Filters.* M. Askit et al. This paper is an interesting synthesis of a variety of different ideas. The paper presents instances of the inheritance anomaly in a real-time object oriented setting. The three types of anomaly presented (mixing real-time specifications with application code, non-polymorphic real-time specification, and orthogonally restricting specifications) are completely different from the usual three (partitioning of states, etc). In a certain sense, real-time specifications are simply concurrency constraints of a very specific kind. The solution to these anomalies is to use real-time filters; this is examined in the real-time anomalies section of this thesis.

- *Reuse Anomaly in Object-Oriented Concurrent Programming.* D. Suciu. This paper argues that the conflict between inheritance and concurrency is the same as the conflict between delegation and concurrent or association and concurrency. They propose a different taxonomy of anomaly types and demonstrate how the conventional classification is incomplete. This is an interesting direction that is completely orthogonal to the approach taken in this thesis.

- *On Composing Separated Concerns: Composability and Composition Anomalies.* L. Bergmans et al. This paper presents an extension of the anomaly to a compositional setting. The paper is short on details – it is only four pages – but the idea is simple: in a compositional setting, we may wish to obtain a behavior/process C_3 that is logically the composition of two other behaviors/processes C_2, C_1. However, this can fail if the way we are doing composition (analog: if the way we are doing inheritance) cannot combine C_2 and C_1 to yield C_3 (analog: cannot inherit from C_1 and/or C_2 to obtain C_3. The analog of being forced to re-write methods in a subclass is being forced to write "glue code" to make desired compositions achievable. The logical place to look for examples of such anomalies would be in the context of "composition filters," which can be used as a powerful anomaly avoidance mechanism.

6.5 Anomaly Experiences

- *Inheritance Anomaly in CORBA Multithreaded Environments.* J. Millan. This paper presents a relatively straightforward set of inheritance examples and show how the anomaly may be avoided in CORBA by using CORBAs pre and postcondition operations.

- *How to Avoid the Inheritance Anomaly in Ada.* W Nebel et al. This paper argues for the following: "if we are able to modify or extend synchronization constraints in a derived type

so that they are compatible to the parents constraints without breaking encapsulation then we have solved the inheritance anomaly." That is a fairly accurate characterization of how to avoid the anomaly in any particular case; the paper then presents a set of guidelines for how to program in Ada such that this is accomplished.

- *POOL: Design and Experience.* P. America. This paper presents a general discussion of the design and experiences of the creators of the concurrent OO language POOL. The inheritance anomaly is not mentioned in this paper; the anomaly is mentioned in other papers about POOL. This paper is a good basic discussion of POOL suitable for understanding the other papers.

6.6 Surveys

- *The Inheritance Anomaly: Ten Years After.* G. Milicia. This is a good foundational survey paper that is relatively recent. Unfortunately, it does not mention Lobel's work; that such a thorough survey could overlook this work is one indication of how specialized and ignored this important work was. The paper examines AOP and Jeeg as solutions to the anomaly, and presents examples of the anomaly in many different languages, including Java. The bibliography is also useful in the sense that it provides many related papers that form a cluster of research along one direction.

- *Concurrent Urban Legends* P. Buhr. This paper is extremely unique: it argues that the anomaly is not a major problem and it cites Lobel's work. The main argument is that in practice, inheritance hierarchies tend to be shallow and so the anomaly is not a major concern. This is likely to be true in practice, but it is my personal belief that the anomaly is worth studying simply because it casts light on the misunderstood relationship between inheritance and subtyping in the difficult domain of concurrent programming. It has also been my personal experience that when demonstrating the anomaly to students (as I did in Stanford's CS242 in the Fall of 2005), the students invariable break into two camps: those that take this position (the anomaly isn't a problem in practice and you shouldn't expect inheritance to do this anyway) and those that take out position (the anomaly is unexpected and indicates that inheritance and subtyping have become confused). The other urban legends in this paper are also interesting.

- *On the Notion of Inheritance.* A. Taivalsaari. This paper is a very in-depth discussion of inheritance. The paper covers different approaches to inheritance (implementation reuse or specialization) and different approaches to subtyping (i.e. ensuring specialization) and shows how the mechanisms relate. The paper also presents a taxonomy of inheritance and subtyping mechanisms and goes into the philosophical and semantic ideas behind inheritance. A history of this more broad thinking of inheritance is also provided.

6.7 Aspect-Oriented Programming

- *Coding Different Design Paradigms for Distributed Applications with Aspect-Oriented Programming.* G. Cugola et al. This is a short introductory paper on AOP and how it is useful

in a concurrent environment. Ironically, their running example of how to use AOP to do useful things suffers from the inheritance anomaly, which they are quick to point out. This paper does not demonstrate how the anomaly can be solved using AOP, but shows how the anomaly occurs when aspects need to be inherited (analogous to when synchronization code needs to be inherited).

- *Towards a Catalog of Aspect-Oriented Refactorings.* M. Monteiro et al. This paper presents a collection of AOP-specific refactorings and code-smells. It is listed here because code that smells in this way is almost certainly guaranteed to suffer from the anomaly (either in the sequential code or the aspect code) more so that cleanly written code.

- *Constructs for Context-Oriented Programming.* P. Constanza. This paper is essentially a synthesis of AOP, composition filters, and environmental inheritance applied to Lisp; the resulting style of programming is called "context-oriented." During execution, different "layers" of definitions are introduced into the running program depending on context; these layers affect the execution of the program to achieve desired results. It seems likely that the judicious use of these layers would improve maintainability; of course, there is no standard notion of the inheritance anomaly in Lisp, but many of the same strands of thought about how to avoid the anomaly in similar, imperative systems can be found in this paper.

- *Aspect-Oriented Programming.* G. Kiczales et al. This paper is simply a good, broad overview of AOP, presented in such a way that the way to use AOP to avoid the anomaly is abundantly clear.

- *An Experimental Aspect-Oriented Language: AspectCOOL.* M. Mernik et al. This paper presents a mechanism for separate module compilation using AOP and the experimental OO language COOL. The main reason that it is interesting is that the advanced features of COOL already open up possibilities for synchronization constructs that are not behavior preserving; this, combined with the use of AOP leads to the intriguing possibility of a language that could use AOP to prevent many instances of the anomaly but also not have behavior preservation. It is also very likely that the combination of AOP with these constructs can lead to synchronization mechanisms that are simply much different (and potentially less vulnerable to the anomaly) than the usual constructs.

6.8 Others

- *Exclusion for Composite Objects.* Noble et al. This paper presents an "algebra of exclusion" for specifying concurrency between threads in object-oriented languages. The algebra is used to specify when threads must not be allowed inside specific components of a composite object. The algebra is unique and potentially enables more interesting concurrency constructs than conventional languages.

- *Automata-Based Verification of Temporal Properties on Running Programs.* K. Havelund et al. This paper presents a mechanism for checking if programs meet linear temporal logic constraints. It is included in this list because of its potential applications to efficient evaluation of history-based guard languages such as JEEG.

- *Object-Oriented Programming with Flavors.* D. Moon. This paper presents OOP done inside Lisp. Lisp "flavors" are classes; a lisp "object" is a member of a flavor (i.e. a person flavor has components `name`, etc). Flavor are combined through the use of multiple inheritance for instance variables and then function composition for inherited methods. This would be an interesting area to apply the framework to because the concept of inheritance used is mostly conventional but because of a functional style, the types of things possible with inheritance is different than in the imperative world.

- *Declarative Object-Oriented Programming: Inheritance, Subtyping and Prototyping.* S. Alagic et al. This paper presents an OO language that is written in a declarative style. In addition, inheritance and subtyping are examined in this declarative style. Although the anomaly is not treated, it is interesting to examine how languages such as this can be examined using the formal framework.

- *Testing Linear Temporal Logic Formulae on Finite Execution Traces.* K. Havelund et al. This paper presents an algorithm to effeciently test ltl formulae on finite traces. It is included because languages like JEEG require an algorithm like this to be anywhere near practical.

- *Inheritance of Dynamic Behavior.* T. Basten et al. This paper presents an interesting use of inheritance in the context of object lifecycle management. Enough of the core concepts of inheritance are used to make it seem plausible that an analysis of how this new, somewhat modified concept of inheritance could be done using the formal framework. The new notion of inheritance is based on blocking and hiding method calls rather than overriding them. The paper also gives an example of how this concept was applied to a real-world product.

- *Monotonic Conflict Resolution Mechanisms for Inheritance.* R. Ducournau et al. This paper presents a discussion of how conflicts may be handled during multiple inheritance. It is likely that to extend the formal framework to deal with multiple-inheritance, a better characterization of what multiple-inheritance schemes have in common would be required. For single inheritance, these concepts are understood making it easy to talk about concurrent OO languages with single inheritance in general, but without such a framework any results would likely be very language specific. The paper does have a bias toward CLOS and is presented at a high enough level that significant work would be required to adapt its ideas into a framework for multiple inheritance at the dozens of languages level.

- *Incremental Inheritance Mechanism and its Message Evaluation Method.* M. Benattou et al. This paper gives a formal treatment of incremental inheritance by using the concept of wrapper classes. The paper is based on a record and lambda calculus and the concepts are also applied to multiple inheritance. This paper would likely be useful in any characterization of incremental inheritance in languages that support multiple inheritance.

- *Environmental Acquisition: A New Inheritance-Like Abstraction Mechanism.* J. Gil et al. This paper presents the notion of environmental acquisition, where an object's behavior depends not only on its class, but also on the other object to which it is connected. One example is that of a handle; presumable, when contained in a car door it would have one behavior, but when contained in a wooden door it would have another. The paper is presenting a

fundamentally new way of thinking of programming, and the method presented has strong type-checking properties and can conceivably be used in place on inheritance. This is another paper whose language would be interesting to look at with the formal machinery, to see how the machinery fares.

- *Delegation is Inheritance.* L. Stein. This paper argues that inheritance and delegation are equally powerful; the typical view is that delegation is more expressive. A formal model of the expressiveness of inheritance and delegation is presented, and "natural" translations between how a concept is expressed in delegation versus inheritance are presented. (I do not believe the translations are particularly natural, however.) Another paper where it would be interesting to see how the formal machinery would fare when adapted to the concepts within.

- *Incremental Programming with Extensions* D. Orleans. This paper examines incremental programming, programming where components are specified in terms of differences with other components. Inheritance is one example of incremental programming; AOP is also a type of incremental programming for crosscutting concerns. This paper presents a language that unifies the treatment of any incremental concern. The language can best be described as LISP with AOP. The related work section is also interesting because it discusses many cutting-edge programming paradigms. Another paper where it would be interesting to see how the framework fares.

Bibliography

[1] Crnogorac et al: *Classifying Inheritance Mechanisms in Concurrent Object Oriented Programming*. In Lecture Notes In Computer Science, Vol. 1445: Proceedings of the 12th European Conference on Object-Oriented Programming. Springer-Verlag, 1998.

[2] G. Milicia and V. Sassone. *Jeeg: A Programming Language for Concurrent Objects Synchronization* JGI 2002, November 2002.

[3] Aksit et al: *Abstracting Object Interactions Using Composition Filters*. In Proceedings of the ECOOP 1993 Workshop on Object-Based Distributed Programming, volume 791. Springer-Verlag, 1994.

[4] Pnueli et al: *The Glory of the Past*. In Lecture Notes In Computer Science, Vol. 193: Proceedings of the Conference on Logic of Programs. Springer-Verlag, 1985

[5] S. Matsuoka and A. Yonezawa. *Analysis of inheritance anomaly in object-oriented concurrent programming language.* In Research Directions in Concurrent Object-Oriented Programming. MIT Press, 1993

[6] C. Hoare. *Communicating Sequential Processes*. International Series in Computer Science, Prentice-Hall, 1985.

[7] J. Meseguer. *Solving the Inheritance anomaly in Concurrent Object-Oriented Programming*. Proceedings of the European Conference on Object-Oriented Programming 1993, Springer-Verlag 1993.

[8] D. Kafura and G. Lavender. *Concurrent Object-Oriented Languages and the Inheritance Anomaly*. In ISIPCALA, 1993.

[9] S. Ferenczi. *Guarded Methods vs. Inheritance Anomaly.*. SIGPLAN Notices, February 1995.

[10] A. Sajeev and H. Schmidt: *Integrating Concurrency and Object-Orientation using Boolean, Access and Path Guards*. In Proceedings of the 3rd Intl. Conf. on High Performance Computing, Trivandrum. IEEE Press, 1996.

[11] F. Svend: *Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages*. Lecture Notes in Computer Science vol 615. Springer-Verlag, 1992.

[12] . C. Laneve. *Inheritance in Concurrent Objects*. Formal Methods for Distributed Processing, Cambridge University Press, 2001.

[13] S Matsuoka et al. *Synchronization constraints with inheritance*. Technical Report 10, University of Tokyo, 1990

[14] G. Milicia and V. Sasson. *The Inheritance Anomaly: Ten Years Later*. todo: this

[15] S. Frolund. *Inheritance of Synchronization Constraints in Concurrent Object-Oriented Programming Languages*. ECOOP '92, LNCS 615.