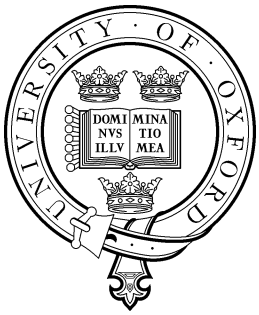
Formal Relationships in Sequential Object Systems



Eric Kerfoot

St Catherine's College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2010

Contents

Abstract

Formal Relationships in SequentialObject Systems
Eric Kerfoot, St Catherine's College
D.Phil. Thesis, Trinity 2010

Formal specifications describe the behaviour of object-oriented systems precisely, with the intent to capture all properties necessary for correctness. Relationships between objects, and in a broader sense the relationship between whole components, may not be adequately captured by specifications. One critical component of specifications having a role in relationships are invariants which define a constraint between multiple objects. If an object's invariant relies on external objects for its conditions, correct operations which abide by their specifications modifying these external objects may violate the constraint. Such an invariant defines a relationship between multiple objects which is unsound since it does not adequately describe the responsibilities which the objects in the relationship have to each other.

The root cause of this correctness loophole is the failure of specifications to capture such relationships adequately in addition to their correctness requirements. This thesis addresses this shortcoming in a number of ways, both for individual objects in a sequential environment, and between concurrent components which are defined as specialized object types. The proposed Colleague Technique [68] defines sound invariants between two object types using classical Design-by-Contract [89] methodologies. Additional invariant conditions introduced through the technique ensure that no correct operation may produce a post-state which does not satisfy all invariants satisfied by the pre-state.

Relationships between objects, as well as their correct specification and management, are the subjects of this thesis. Those relationships between objects which can be described by invariants are made sound with the Colleague Technique, or the lightweight ownership type system that accompanies it. Behavioural correctness beyond these can be addressed with specifications in a similar manner to sequential systems without concurrency, in particular with the use of runtime assertion checking [29].

Acknowledgements

Four long years seemingly have gone by rather quickly in the course of my graduate studies. It's not been easy at times of course, who I'm most indebted to for helping me through it all is my supervisor Dr. Steve M^{c}Keever, whose advice and support over the years has been invaluable. We have together collaborated on a number of papers [68, 69, 70], and he has graciously traveled halfway across Europe to give a talk on my behalf. Many other lecturers at the Software Engineering Programme whom I've worked with have been incredibly important to my studies: Dr. Jeremy Gibbons, Dr. Andrew Simpson, Dr. Alessandra Cavarra, and Dr. Ralf Hinze. At one time or another I've shared offices with some great and brilliant fellow students who have contributed no small measure to my experiences here: John Lyle, Jun Ho Huh, Aadya Shukla, and others. In particular I want to acknowledge my fellow Software Engineering teaching assistants, Clint Sieunarine and fellow Canadian Jackie Wang, for their help over the years with courses and with administering our network. A good deal of work on active objects was done in conjunction with Faraz Torshizi at the University of Toronto, who Jackie and I met while working together under Dr. Jonathan Ostroff at York University in Toronto, and with whom Steve and I collaborated with on [70]. Both Faraz and Jonathan have been great collaborators in the past, so I have much thanks for them. Finally I have to most of all thank friends and family both here in Oxford and back home in Canada, whose support and company have been essential.

Chapter 1

Introduction

Relationships between objects in object-oriented programs represent critical aspects of an object's functionality. Objects interact with each other, co-operate to complete tasks, and aggregate together to form data structures. These relationships define the means by which the modular components of systems are interlinked and made to function together. Many of the challenges relating to ensuring correct behaviour are rooted in incorrect relationships between objects. These problematic relationships can often be described in terms of the objects' contracts, such that the relationship is described formally, but with conditions and constraints which can be violated.

The Design-by-Contract [89] (DbC) approach specifies properties with invariants which in general should always hold for a running program. These are predicates stating correctness properties for the members of an object which must be true when that object is accessible to its clients. In effect they describe the form an object should have when it is free to interact with others it has relationships with, rather than strictly unvarying properties which must hold for all program states.

Method contracts describe the requirements and effects of method calls as precondition and postcondition predicates. The precondition defines the state the receiving object must be in before the method can be called, as well as requirements on arguments which the caller must satisfy. The postcondition states what effects on the method has on the state of objects, and what value it returns.

The correctness expectation of Design-by-Contract is that invariants and method conditions, if respected, ensure that objects always transition from one correct state to another, where an object's correct state is defined by the invariant. If a method is called when its receiver's invariant held and the precondition was established by the caller, then the method will perform an operation which ensures the receiver's invariant holds upon completion, as well as satisfying the postcondition. At any point that the receiver is accessible to clients while this method executes, its invariant must be re-established before access is permitted. Thus an object is always in a correct state when accessible, and can only transition between correct states through the operations of methods.

An invariant typically defines properties which an object's members must have, such that method definitions and clients rely on these properties for their correct operation. When the members are only modified by the object's methods, only the object itself is responsible for ensuring the invariant always holds. If however members may be affected by external clients or the invariant relies on the members of external objects, then those external objects must also bear the responsibility of correctness. If this were not the case then an object could be modified in ways which are correct according to its own specifications, but which result in a state not satisfying the invariant of another

object depending upon it.

From a higher level perspective, invariants define correctness properties internal to individual modules. The expectation is that the composition of one module with any other will not affect whether these invariants hold for all relevant states of running programs. Given a module whose reachable states adhere to the invariant conditions, if this is composed with another such that states not adhering to the conditions become reachable, then there is a soundness problem with those invariants.

In particular this describes a relationship problem, where one module may function correctly in isolation or in co-operation with certain others, but which then malfunctions when in the presence of other modules. One very important property of modules is that, once considered to be correct, they should remain so regardless of context. Otherwise compositionality will always be hampered by the threat of combining two modules which are composable at the language level, but whose co-operative behaviour leads to malfunction.

Returning to the object level, it is often desirable for the invariant of one object to depend on another object for its conditions. Such invariants are between members of the same module, and so are amongst those internal correctness properties. This creates an implicit relationship between the two objects which is critical to the program's correctness.

For example, a Java [51, 52] iterator will certainly require basic correctness properties of the collection it traverses to hold over its lifetime. A simple property would require that the collection be no smaller than it was when the the iterator was created. Consider an iterator class *ListIterator* whose instances traverse *List* instances:

```
class List implements Iterable{
...
  ensures size() == 0;
  public void empty();

  public int size() { ... }
  public Iterator iterator() { ... }
}

class ListIterator implements Iterator {
  private List list;
  private int position, last;

  invariant position <= last;
  invariant list.size() >= last;
...
}
```

In this example, which includes invariants stated in *invariant* annotations, the *ListIterator* instances are responsible for their own invariants. However the *List* instances they rely on must also ensure their state satisfies the given constraint. Expecting the methods of the class *ListIterator* to ensure its invariant is reasonable and modular, but expecting *List* methods to do the same without any further information is not. A *List* instance with no dependent iterators has no constraint to maintain, but this cannot be differentiated from another instance which is constrained, thus clients of either object cannot be expected to not perform operations which violate an iterator's invariant.

The responsibility for maintaining the *ListIterator*'s invariant is easily enforced only in simple cases, like the example here where the last statement will obviously break the iterator's invariant:

```
List l = new List();
... // add some elements to the list
ListIterator i = l.iterator();
l.empty(); // l.size() becomes 0, breaking i's invariant
```

Although this example includes no method call or other operation which does not abide by the given contracts, the invariant of the iterator can still be broken. This can be easily identifier since the relationship between the two objects in question is immediate and known. A given module using these types may be shown to ensure this relationship between all lists and dependent iterators, however the introduction of new types through module composition may allow for incorrect behaviour to result from ostensibly correct operations. Consider a *Set* type which uses a *List* object:

```
class Set {
  protected List list;

  ensures this.list.size()==0;
  public Set(List list){
    this.list=list;
    list.empty();
  }
  ...
}
```

In isolation this class is correct in that the contracts of method calls are respected. However if the given list is depended upon by an iterator then this will result in a state not satisfying the iterator's invariant. Whether *Set* produces an incorrect state or not depends on the relationships between objects at runtime, thus no static analysis can show that it is always correct. Testing at runtime would be required to demonstrate some measure of correctness, although this is still made difficult since the relationship between lists and iterators is implicit in the latter's invariant and no information, and certainly no correctness obligations are present in the list's specification.

This problem is termed the Indirect Invariant Effect in [89], which captures the basic problem of one object's invariant depending on another. This dependee object must abide by certain constraints to ensure the invariant of those dependent upon it, however clients may correctly modify it such that this constraint is not respected. The suggested solution is to include invariants in any object depended upon by the invariants of others which ensure it abides by these constraints. This is not in itself adequate since the relationship between depending and dependee objects must be constructed and maintained correctly for such added invariants to accomplish this goal. Furthermore, this reduces modularity by more tightly coupling two objects types, thus other approaches which impose constraint in other ways would be desirable alternatives.

1.1 Reasoning About Objects

Mutable state allows this problem to arise, where the state of multiple objects must be co-ordinated while being modified. If iterators relied on immutable lists rather than mutable ones, their invariants would always hold if they were initially established. No operation may change an immutable list such that it ceases to adhere to any invariant constraints, whereas a mutable list of course can be modified such that their state does not satisfy some invariant constraint. Therefore all state must be immutable in a similar manner to many purely functional languages [7, 64, 65, 90, 98, 100, 109, 118], or there must be some approach to ensuring mutation abides by the present constraints.

The approach which Java applies in its standard library, as well as in generally accepted practice, is to throw an exception when a data structure with dependent iterators is mutated. This certainly identifies when unwanted mutation has occurred, but it only serves as a runtime testing aid and often provides little help in identifying what objects were involved in the relationship, and how they caused the exception to occur. Such an approach also cannot be used statically at compile time to analyzed a program and detect when such errors occur before the program is even run. A reasoning-based approach, which identifies the problem in terms of a logical error rather than an exceptional event, could be used in detecting such errors at compile-time.

Global reasoning could be applied to analyze an entire program and verify that all relationship constraints are respected by all possible operations. This would require reasoning about what relationships may form, and what operations will respect all possible invariants. All uses of *List* instances, for example, would have to be shown to involve lists which have no dependent iterators or only allow operations which satisfy the iterators' invariants.

This form of reasoning, either as a runtime testing framework or in conjunction with formal theorem proving, is extremely cumbersome and impractical for large programs. Either proofs are far too large to be conducted, or else the non-determinism of certain choices implies that it cannot be known if a certain relationship may be formed or not. Invariants which state properties between objects can be recast as global invariants, which global reasoning must demonstrate are true after all operations. For example, the global invariant for lists and iterators is the following:

$$\forall\, l : List, i : ListIterator \mid i.list == l \bullet l.size() >= i.last$$

Ensuring that this is true whenever the two objects are accessible by their clients is not a simple task, requiring strict control over how and when the list and its iterators are aliased. A method for example may accept a *List* instance and perform some operations on it. The global approach must recognize when a list with dependent iterators is used as an argument in such a method call. If there is a chance the method may modify the list contrary to the global constraint, the call must be disallowed.

Even if a module can be shown in this way to be correct where global constraints are always preserved, any newly introduced modules may create relationships which do not respect this invariant. With this approach the correctness of each module would have to be re-checked when they are composed into a final complete system, otherwise it cannot be known when two objects may become related in a way which leads to an invariant not being respected. A module is therefore never considered correct in isolation, only when it is integrated into a whole program and verified.

Classical local reasoning, that is ensuring the correctness of a program by checking that the object types it is composed of abide by their contracts, is not sufficient either. What local reasoning attempts to do is ensure the correctness of the entire module, or entire program, that each type is part of by analyzing its methods. Those methods which abide by their specifications and those of any other object they interact with are expected to never break invariants other than that of their receiver object. As shown in the example, all three object types abide by their contracts, yet still errors can manifest as invariant violations. The invariant of *ListIterator* can be broken either through variable assignments or method calls which violate no contract at the point of execution. Such an invariant is considered to be unsound for this reason; sound invariants are always maintained by correct operations once completed, and are necessary for a local reasoning approach to ensure correctness.

The practical solution to this soundness problem is to apply local reasoning on individual object

types with additional restrictions that ensure only sound invariants can be formulated. An object must have a particular and well-defined relationship with another to predicate its invariant upon it, such that these relationships are more closely controlled and more easily reasoned about. This allows types to be analyzed in isolation since the relationships between objects at runtime would not affect correctness. Using a sound local reasoning approach, if two object types related through an invariant such as *List* and *ListIterator* are defined correctly, the invariant will always hold irrespective of what other correct types such as *Set* are introduced.

If it were known that the instances of *ListIterator* were the sole clients of the instance of *List* their invariants relied upon, then no operation of any other object could violate their invariants. This is to say that the iterators encapsulate their lists, such that they are protected from adverse modification by being accessible to no other clients. Ownership type systems [6, 24, 34, 60, 92] have been used to enforce this constraining property statically, and allow local reasoning only about the correctness of *ListIterator*'s methods when considering if its invariant will always hold.

Normally the iterator pattern is predicated on the list and iterator being accessed by external clients, thus encapsulation cannot be employed with this example. These two objects co-operate as a single module and so will both have many external clients to whom they provide services. They still rely on each other for their invariants, so making this relationship explicit makes the information necessary for correctness available to external clients.

Specifically this means defining the same constraint in both types, hence any operation that would break one partner's invariant will also break that of the other. This ensures that no operation of one partner which violates the other's invariant is considered correct. External clients must still perform correct operations, so modularity is retained in that modules can be composed correctly if their constituent contracts are respected. The advantage is that the relationship between iterator and list is first of all made explicit, and secondly protected by both partners having obligations to the other. This replaces the original situation where *List* is related to *ListIterator* implicitly and has no invariant constraints disallowing invalid states.

Local reasoning now involves reasoning about the two objects simultaneously, but is a reasonable approach albeit at the expense of closer coupling between the two types. The end result is a DbC approach where no operation can result in a state where invariants do not hold without a contract violation. This is the goal of the Colleague Technique whose mechanisms defined in this thesis allows this soundness between co-operating objects.

1.2 The CoJava Approach

This thesis presents the CoJava language which employs ownership and invariant techniques to ensure the relationships between objects remain correct. CoJava is a subset of the Java language containing adequate features from the full language to be a useful and manageable subject for discussing these techniques. These relationships require that the invariants defining them be sound, which is not guaranteed by default as the Indirect Invariant Effect demonstrates. If all object invariants are sound however, then any time a correct operation occurs, all those invariants that were satisfied before the operation will be also be satisfied afterwards.

CoJava uses a simplified ownership scheme based on its specialized type system to enforce encapsulation. An object owns another if it is aliased through an owned reference, thus allowing the type system to statically enforce encapsulation. This type system is very lightweight in comparison to other definitions ([24, 34, 92] to name a few) in that it relies on simpler type definitions,

fewer concepts, and less complex formal rules, although at the expense of a more rigid definition of encapsulation.

To allow two non-encapsulated objects to co-operate formally, the Colleague Technique is used to provide additional invariant support between partners. If an object b relies on a for its invariant, then a's invariant must state the same constraint from its perspective. Additional support is included to ensure that these two objects always alias one another so long as the dependency exists. The CoJava Tool automatically generates the code to do this, and also calculates what the additional invariant for a should be.

Similar to the Friends Mechanism [17] devised for the Spec# Language [16], this technique relies on explicit bidirectional aliasing by the involved objects and a means of generating additional contract support. The technique however does not require the explicit definition of ancillary invariants or conditions to ensure correctness. Another means of addressing the same problem is to employ ownership primarily but weaken its constraints in certain situations and introduce additional proof obligations [94].

These two techniques share the common themes of formalizing the way objects co-operate with one another soundly and correctly. Collectively these relationships contribute to the correct composition of modules by defining precisely how objects interact and therefore how modules are integrated together. CoJava's techniques can be extended into the full Java language, or any other object-oriented language, with few major extensions required since the applied techniques do not require specialized language constructs not present in common languages. The following chapters will define CoJava, its mechanisms, and demonstrate their correctness and application to object-oriented systems.

The CoJava Tool has been developed which implements these concepts for the Java subset language slightly larger than CoJava. The tool demonstrates that such concepts are practical solutions to the problems of formalizing relationships between objects. Through type checking and code generation, the Colleague Technique is implemented as standard Java classes. Combined with a runtime assertion checking technique based on AspectJ [72] aspects, the tool demonstrates an effective software development methodology capable of compiling, running, and checking simple CoJava examples as well as larger complex systems.

The chapters of this thesis are broken down as follows:

Chapter 2 discusses challenges of specifying object systems in terms of relationships. This will include a discussion on the application of ownership to enforce encapsulation, as well as the limitations of the type-based approach. Secondly, the challenges of specifying object relationships is delved into. The Colleague Technique is introduced at this point as the solution to the problem through examples These examples are given in Java with specifications provided as annotations in the Java Modeling Language (JML) [75, 76].

Chapter 3 will define the sequential CoJava language by providing its abstract syntax, type and operation rules. The previous chapter shall describe the problem space in terms of Java, however this language is much too large to provide a coherent proof in the scope of this thesis, so CoJava is formalized as a core subset of Java which captures much of the semantics of interest. The semantic description of CoJava will be used in later chapters to prove properties about CoJava, specifically that encapsulation is a static property of the type system as defined by the rules, and that sound invariants can be defined with ownership and the Colleague Technique. An overview of the Design-by-Contract methodology is given.

Chapter 4 will discuss the properties of the CoJava ownership type system. It will be shown

that ownership can statically enforce encapsulation as a side-effect of the type system's constraints. The implications this has for invariant soundness will also be discussed, defining a sound invariant as one which can only rely on owned objects.

Chapter 5 discusses the Colleague Technique in detail, describing how the technique is used to augment specifications. This allows sound invariants to be defined which may rely on non-owned objects in certain controlled situations.

Chapter 6 introduces the CoJava Tool briefly as a code generator/translator. This tool checks input code in a Java-like language and enforces the ownership type system as well as the restrictions for the Colleague Technique. It then generates the code to implement the technique, as well as AspectJ aspects which implement a simple form of runtime assertion checking.

Finally, Chapter 7 will conclude this discussion and elaborate on possible future work regarding these techniques. Specifically, a concurrency mechanism for CoJava based on active objects [74] has been the subject of experiment, and represents a significant avenue of future work. CoJava's ownership, Colleagues, and concurrency methodologies also have their sets of disadvantages that future extensions and refinements can address.

Chapter 2

Managing Relationships

Relationships between objects can be managed in a rigorous way such that many classes of errors can be detected more effectively compared to more common informal approaches. This chapter will discuss how ownership and specification with the Colleague Technique provides a unified methodology for constraining and describing these relationships. Ownership is used to enforce an encapsulating relationship between objects statically using the type system. The Colleague Technique addresses a serious shortcoming of object invariants and provides greater expressiveness and correctness to the specification of relationships.

This chapter will discuss these two components in terms of Java programs with the addition of JML-like specification elements. Ownership and colleagues imply a specific kind of structure for Java programs which otherwise would not be adopted, in particular they have implications for many common design patterns and other programming idioms.

Section 2.1 will first discuss specifying object-oriented specifications with the Design-by-Contract approach. Section 2.2 introduces the problem of encapsulation and correctness, and how an ownership type system can address it. Section 2.3 poses a common design pattern which cannot be captured with ownership, so the Colleague Technique is proposed as a means of ensuring correctness when ownership cannot be used.

2.1 Specifying Java

The Java Modeling Language (JML) [75, 76] describes a methodology for applying Design-by-Contract [89] specifications to Java. It includes definitions for invariants, method contracts, and annotations applied to various Java constructs such as types and methods. The Java examples used in this chapter and subsequently will use a JML subset to define specifications, and will be extended with additional constructs to describe ownership and the Colleague Technique. Briefly, the constructs used in these specifications are the following:

- **Invariants**: These predicates describe properties a type's instances must have when they are accessible to their clients, that is when they are in a visible state.
- **Method Context**: These predicates describe what a method requires to be true about its receiver and arguments (precondition), and what it guarantees to be true about the receiver, arguments, and return value (postcondition).

- **Member Annotations**: These state particular properties about methods or attributes.

Consider a simple Java class, *Counter*, with a JML specification stated in special comment blocks denoted by /*@ or //@ tags:

```
class Counter {
  protected int value,max;

  //@ invariant value>=0 && value<=max;

  //@ requires max>=0;
  //@ ensures value==0 && this.max==max;
  public Counter(int max) { value=0; this.max=max; }

  //@ requires value<max;
  //@ ensures value==\old(value+1);
  public void inc() { value=value+1; }

  //@ requires (value+n)>=0 && (value+n)<=max;
  //@ ensures value==\old(value+n);
  public void add(int n) { value=value+n; }

  //@ ensures \result==value;
  public /*@ pure @*/ int get() { return value; }
}
```

This class defines an invariant requiring *value* to be no more than *max* whenever an object can access an instance of *Counter*. The constructor requires a positive value for its argument *max*, and ensures that *value* and *max* are initialized. The method *inc()* requires that *value* be less than *max*, so that when it is incremented the invariant is not violated, and states that once a call to the method completes *value* will store the previous value it had plus one. *add()* similarly defines requirements of the argument n and the state of the receiver so that the invariant holds once calls to it complete, and states that the effect of the method is to add n to *value*. *get()* states simply that the value it returns is equal to *value*, and also is declared to be pure, thus having no visible side-effects.

Annotations of this form capture a significant amount of information about the semantics of Java classes. They can describe what requirements methods have and what operations they perform to a significant degree. JML does provide a richer set of specification elements to describe concepts such as the frame property [21, 22] which is the scope of data elements a method is allowed to modify, what exceptions might be thrown and under what conditions, and others.

This thesis will cover only invariants, pre- and postconditions in specifications as demonstrated in this example. For reasons of coherence and readability, the subsequent Java examples will omit the comment tags for specification elements, although actual JML-annotated code requires them to be parsable.

Invariants and contracts are effective at defining behaviour, but they are inadequately expressive when attempting to describe object relationships. Consider the following subclass of *Counter* which keeps track of previous counter values with a list of integers:

```
class ListCounter extends Counter {
  protected IntegerList history;

  invariant (\forall Integer i ; history.contains(i) ; i.intValue()<value);

  public ListCounter(int max) { super(max); history = new IntegerList(); }

  ensures history.contains(new Integer(\old(value)));
  ensures history.size() == \old(history.size() + 1);
  public void add(int n){ history.add(new Integer(this.value)); super.add(n); }

  ensures history.contains(new Integer(\old(value)));
  ensures history.size() == \old(history.size() + 1);
  public void inc(){ history.add(new Integer(this.value)); super.inc(); }
}
```

ListCounter introduces a new attribute *history* and the requirement that all values it stores be less than *value*. The type *IntegerList* is understood to be a list of *Integer* values which quantifier expressions may range over. The methods *add()* and *inc()* are overridden with new versions which add the current value to *history* before performing their expected operations. These methods add new postconditions but are still bound to uphold the contracts defined for the methods they override in *Counter*.

The introduced invariant defines a relationship between an instance of *ListCounter* and one of *IntegerList* beyond the fact that one aliases the other. This predicate defines some property that the counter requires of the list, such that so long as the aliasing relationship exists the list is obligated to ensure this property. Enforcing this obligation on the list requires making the relationship between the objects more explicit, such that clients of either will possess sufficient information through the specification to prevent operations which result in the invariant condition not being respected.

One way is to regard the internal representation of this class to be composed of the *history* attribute as well as those attributes inherited from *Counter*. Internal representation objects, like the one *history* aliases, define the internal structure for a containing object, and are meant to be inaccessible to that object's external clients. This is to say that representation objects are encapsulated by their enclosing object. Encapsulation is thus a particular type of relationship between objects with specific constraints as to how objects may alias one another.

However the encapsulation relationship between an instance of *ListCounter* and the object *history* aliases is not captured by this specification. If *history* is meant to be only accessible to its enclosing counter, then this encapsulation property must be stated in the specification and enforced, otherwise the following method could be defined for *ListCounter*:

```
public IntegerList getHistory() { return history; }
```

This obviously naïve method allows the internal representation to escape to the outside world. External clients could add an integer to the list which then breaks the invariant of the encapsulating counter. A more reasonable implementation would create a copy of the list, then fill it with the values:

```
public IntegerList getHistory(){
  IntegerList list=new IntegerList();
  list.addAll(history);
  return list;
}
```

This certainly ensures encapsulation by not exposing *history*, however the contracts required to state this become quite complex and must be re-stated for any method which might produce data derived from internal representation objects. Worse still, there is no way of requiring subtypes to enforce the same encapsulation properties in all of their method definitions.

Alternatively, the *IntegerList* can be defined as not being encapsulated by the *ListCounter* which created it, but is meant to be available to external clients. The relationship still exists between the two objects however, since the invariant of *ListCounter* still relies on it. There must then exist a mechanism to prevent clients of the *IntegerList* object from adding an integer to the list which does not satisfy the invariant, as the following demonstrates:

```
ListCounter lc = new ListCounter();
lc.add(5); // lc.value == 5
IntegerList il = lc.getHistory(); // il == lc.history
il.add(10); // breaks invariant of lc since 10 > lc.value
```

There exists a particular relationship between the objects *lc* and *il*, which in this simple example is easily inferred. In the general case however an instance of *IntegerList* may be associated with a *ListCounter* about which nothing is known, or associated with none at all. Either scenario implies that there exists insufficient knowledge to know if a particular integer value will break the invariant of a *ListCounter* object or not. A mechanism to first explicitly define the relationship between the two objects when it exists is needed, as well as a means of defining reciprocal obligations the two objects have to one another necessary to prevent invariant constraints from being inadvertently violated.

2.2 Encapsulation With Ownership

Ownership is a type-based approach to enforcing encapsulation or other relationship properties between objects. It relies on type system constraints to ensure these properties at runtime. The type system given here, later defined for CoJava, is a minimal one which provides the basic encapsulating property. An invariant relying on an encapsulated object is not subject to the influence of external clients, such that only through the methods of the encapsulating object can it be violated.

The three primary properties which ownership of this form provides are encapsulation, hierarchy, and invariant soundness. An object is encapsulated by its owner in that it is not available to the owner's clients, thus is contained by the owner. This containment also has a hierarchical property in that owners can also be owned and so define tree structures with owners as nodes, while also be acyclic in that owned objects cannot own their owners. The last property allows an invariant to soundly rely on owned objects for its conditions, such that only operations on these objects and their owners can produce a state which does not satisfy this invariant.

A new keyword *owned* is introduced which is applied to types, and indicates that variables of that type alias owned objects. For any existing object type T, there exists an owned version called $owned.T$. Encapsulation is enforced by disallowing certain operations from being considered well-typed, which is summarized here:

- Owned values, those stored by variables with the *owned* keyword, cannot be assigned to non-owned variables and attributes, or vice versa. This ensures that an owned object can only be aliased through an owned reference, and a non-owned object only through a regular reference.

- Methods with owned arguments can be called only when the receiver is *this*, and constructors cannot have owned arguments at all. This prevents any client from changing the relationship between owned objects, especially from breaking the hierarchical structure.
- Owned attributes can be assigned to only when the receiver is *this*.
- Methods returning owned references, and owned attributes, can only be accessed through an owned receiver. This prevents non-owners from accessing critical internal state.
- Within the constructor and method bodies of some class T, *this* has type $owned.T$. Since non-owning clients cannot call methods returning owned values, *this* is not accessible to non-owners.

This implies a number of things for the *ListCounter* example above if *history* is changed to have an owned type:

```
class ListCounter extends Counter {
  protected owned IntegerList history;
...
  public ListCounter(int max) { super(max); history = new owned IntegerList(); }
}
```

Here an instance of *ListCounter* is the owned of a *IntegerList* object, in particular it is the immediate owner since it created the object. With this change, the original version of *getHistory()* will no longer be considered correctly typed, thus this version would be required:

```
public owned IntegerList getHistory(){ return history; }
```

Since methods returning owned values can only be called when the receiver is also owned, this prevents any client which does not own the counter from accessing the list. An object must be responsible for their own invariants as well as those of objects it owns. Clients which do own the counter are considered to also own the list as transitive owners, hence they are responsible for the invariant of both objects. These owners are thus required to keep track of any relationships that exist between owned objects so that operations which violate invariant constraints are prevented.

Consider the example below:

```
ListCounter lc1=new ListCounter();
IntegerList il1 = lc1.getHistory(); // illegal, disallowed by type system

owned ListCounter lc2 = new owned ListCounter();
owned IntegerList il2 = lc2.getHistory(); // OK
il2.add(10); // can infer that lc2 relies on il2, this breaks the invariant
```

The object *lc1* is not owned, thus calling *getHistory()* with it as a receiver is not allowed. However *lc2* is owned and so the call is allowed. Both *lc2* and *il2* are owned by the same object which thus is responsible for ensuring the relationship between them remains consistent with respect to invariants. Ownership has the advantage for correctness in that it restricts what objects can access the owned representation of others, such that objects like *il2* cannot have arbitrary clients who cannot be held accountable for the invariant that depends on it. It does also allow an object to identify clearly what other objects are part of its representation, and prevent them from being accessed by any client, owner or not.

Ownership thus implies a number of properties about the programs that use this type system. These are summarized here and will be formalized later in Chapter 4:

- Objects cannot be aliased by owned and non-owned references at once (excluding *this* which is only internally accessible).
- An owner of an object a may access an object it owns called b, thus becoming b's transitive owner through a.
- If owned objects are considered nodes and aliasing relationships between them edges, owned objects form acyclic digraphs rooted at a topmost owner. Without transitive ownership the relationship would actually form a tree.
- If an object is aliased through a non-owned reference, its owned objects are inaccessible to its clients since no well-typed method call or attribute access makes them available.
- Since methods accepting owned arguments and owned attributes can only be accessed by *this*, clients cannot pass owned references to an object. This implies that no object can directly modify the object structures created by those objects it owns.
- Because ownership only restricts what expressions and statements are considered well-typed, and does not introduce any new constructs or semantics, then type safety is preserved in the presence of owned types.
- Methods which only refer to owned objects can be termed to be *local* methods and have that annotation attached to them. The side-effects of local methods are thus localized within the receiver's internal representation.

2.2.1 LinkedList Example

Ownership provides three main facilities to object-oriented programs: enforced encapsulation, enforced hierarchical structure, and invariant soundness. For rigid structures which should have encapsulation and hierarchy, ownership provides some very useful guarantees. A linked list is the simplest such structure but still an effective example when implemented with each node owning the next in the chain. This ensures that the list will have no cycles in its structure and that the nodes cannot be exposed to external clients. It does also imply that the rigidness of ownership is present and disallows certain operations.

An implementation of a linked list data structure is given here as a larger example. The *Node* type defines a node in the linked structure, with methods for querying the value it stores, finding the last node in the chain, appending and removing nodes. The *LinkedList* type defines the topmost owner in a linked list of *Node* objects. Its methods for the most part defer to those present in *Node*, such that external clients can interact with the internal components but only in a way managed by the topmost owner.

A number of features of this example illustrate aspects of ownership, both its advantages and drawbacks. This section will briefly discuss these properties of CoJava's ownership type system which have broader implications in programming in general.

Note that certain attributes are initialized to *null*. Contracts and invariants do not consider whether these are null at the appropriate times they must hold, the invariant of *LinkedList* for example is defined assuming non-null values. Guarding expressions, such as p *!= null ==>* C, ensure that predicates such as C are evaluated only when some variable p it relies on is non-null. Such guards and other conditions would normally be present in contracts but are omitted here for brevity.

Node

Figure 2.1 gives the code for the *Node* class. Each node owns the next in the chain, as defined by the *next* attribute whose type is ***owned****.Node*. Every node owns not just the next in the chain, but through transitive ownership every other node below that. This allows *remove()*, for example, to assign *next.next* to *next* and so remove a node from the chain. The *value* attribute represents the value being stored in the structure rather than a part of the internal representation.

Node has no invariant for lack of needed constraint. Often types representing nodes in linked lists or trees need a structural invariant stating the lack of cycles. Given a method *subNodes()* which returns a collection containing those nodes supposedly below the receiver in the hierarchy, an invariant such as *!subNodes().contains(this)* would accomplish this. This property in CoJava is enforced statically by the nature of ownership, thus requiring nothing of the nodes themselves or their clients.

Methods like *remove()* and *append()* function in the familiar recursive manner. Ownership prevents giving owned references to other owned objects, so *insert()* must call *insertValue()* when the current object is the appropriate place for the new item to be in the chain. *insertValue()* assigns the given value to the current object's *value* attribute, then recursively calls itself passing the old *value* to the *next* object. This has the effect of shuffling the stored values down the list until the end is reached, in which case a new node is created with the shuffled value. Consequently new nodes are only added to the end of the list since this is the only place to do so. Each existing node in the chain below where the new value was inserted then receives the value stored by the node above it, until the value of the last node is given the new one being created.

This contrasts with adding a new *Node* instance into the appropriate location. Without certain ownership constraints, the following definition of *insertValue()* would be allowed:

```
public owned Node insertValue(Object value) {
  owned Node n = new owned Node(this.value);
  n.next = next;
  next = n;
  this.value = value;

  return getTail();
}
```

CoJava ownership prevents the line *n.next = next;* from being correctly typed, demonstrating a drawback to ownership. This operation now requires iterating through the whole chain even when adding an element to the beginning.

An alternative approach is to loosen the rigid constraints of ownership when it can be shown to be safe. In *insertValue()* a new *Node* object is being created for variable *n*, therefore it definitely is not part of any existing representation. Newly created objects are termed "fresh", and remain fresh until they form relationships with non-fresh objects. By passing an external owned reference to the *n*, there is no chance of creating a cyclic ownership relationship. Allowing *n.next = next;* would not violate the encapsulation definition set out in Definition 4.1.1 as a consequence, and so should be allowed. After which, *n* ceases to be fresh now that it is linked into an existing representation, and no assumptions of freshness can any longer be made.

Static analysis can be used to show such situations are safe and allow bending the rules in such instances while ensuring encapsulation and acyclicality. Further annotations on object freshness or uniqueness may allow, within the scope of the type system itself, the operations needed to circumvent

```
class Node {
  public owned Node next;
  public Object value;

  ensures next == null;
  ensures this.value == value;
  public local Node(Object value) {
    next = null;
    this.value = value;
  }

  requires count >= 0;
  requires count > 0 ==> next != null;
  ensures count==0 ==> next==\old(next.next);
  public void remove(int count) {
    if(count==0)
      next=next.next;
    else
      next.remove(count-1);
  }

  ensures \old(next == null) ==>
      (next != null && next.value == value);
  public local void append(Object value) {
    if(next == null)
      next = new owned Node(value);
    else
      next.append(value);
  }

  requires count >= 0;
  requires count > 0 ==> next != null;
  ensures count == 0 ==> this.value == value;
  public local void
          insert(int count, Object value) {
    if(count == 0)
      insertValue(value);
    else
      next.insert(count-1,value);
  }

  ensures this.value == value;
  ensures next != null;
  ensures next.value == \old(this.value);
  protected local void
        insertValue(Object value) {
    if(next == null)
      append(this.value);
    else
      next.insertValue(this.value);

    this.value=value;
  }

  requires pos >= 0;
  requires pos > 0 ==> next != null;
  ensures pos == 0 ==> \result == value;
  ensures pos != 0 ==>
          \result == next.get(pos-1);
  public pure local Object get(int pos) {
    Object val=value;

    if(pos != 0)
      val=next.get(pos-1);

    return val;
  }

  ensures \result.next == null;
  public pure local owned Node getTail() {
    owned Node n=this;

    if(next != null)
      n = next.getTail();

    return n;
  }
}
```

Figure 2.1: The Node Class

ownership's limitations. However the balance must be struck between how complex such a type system gets, and hence the burden on the programmer to learn and correctly employ a myriad of new concepts, and the use of static analysis tools to instead demonstrate when the rules can be safely broken.

LinkedList

The class *LinkedList* (Figure 2.2) defines an abstract data type which uses a linked list of *Node* instances for its implementation. This class recalls both the head and the tail of the list. Methods are provided to add items to the end of the list, push items to the front, pop items from the back, query items from arbitrary positions in the list, and calculate the list's length. These methods make use of those found in *Node* as much as possible.

The *size()* method demonstrates the use of transitive ownership. The *LinkedList* object owns the head of the list directly, but may acquire each other node in turn and count how many of these there are. Without transitivity, this would not be possible since no object beyond *head* would be accessible, and the method would have to rely on the presence of a *size()* method in *Node*.

Defining an instance of the Iterator Pattern [50] for this type is not in itself difficult, but the simple solution will have poor performance. Every time the next item is queried, the iterator will have to call *get()* which has linear time complexity. A more complex solution involves the *LinkedList* type keeping track of what iterators exist for it and what node they would need next, thus searching from the beginning of the list would not be necessary.

A more practical approach would be an internal iterator implementation, given an interface *Action* and a method *traverse()* added to *LinkedList*:

```
interface Action { public void op(Object o); }

public void traverse(Action a) {
  owned Node n=head;

  while(n!=null){
    a.op(n.value);
    n=n.next;
  }
}
```

Internal iterators allow a data structure to traverse the objects it owns and allow external objects access to stored non-owned data. A similar approach can be taken to implement the Visitor Pattern, with the added assurance that an external visitor may traverse the owned structure without programming errors allowing it access to the actual structural objects themselves.

The type is declared with the *contained* keyword, ensuring that no external client of any sort may access the internal structure. When this annotation is applied to classes, it is required that all methods accepting owned arguments or returning owned values, and all owned attributes, be declared as private. Contained types can assume that they exclusively control the objects they own, and that transitive ownership is prevented.

Some invariants are difficult or outright impossible for an owner to re-establish once they have mutated transitively owned objects. The correctness requirement that transitive owners respect the constraint between owned objects might require assigning to owned attributes of other objects, which is prohibited. For example, if an owner of a list had access to the internal nodes and subsequently

```
contained class LinkedList {
  private owned Node head = null, tail = null;

  invariant tail == head.getTail();

  ensures size() == \old(size()+1);
  public void add(Object value) {
    if (head == null) {
      head = new owned Node(value);
      tail = head;
    } else {
      tail.append(value);
      tail = tail.getTail();
    }
  }

  requires pos >= 0 && pos < size();
  ensures size() == \old(size() + 1);
  public void insert(int pos, Object value){
    if (head == null)
      add(value);
    else {
      head.insert(pos,value);
      tail = tail.getTail();
    }
  }

  requires pos >= 0 && pos < size();
  ensures \result == head.get(pos);
  public pure Object get(int pos) { return head.get(pos); }

  requires pos >= 0 && pos < size();
  ensures size() == \old(size() - 1);
  public void remove(int pos) {
    if (pos == 0)
      head = head.next;
    else
      head.remove(pos - 1);
  }

  public pure int size() {
    int res = 0;
    owned Node n = head;
    while (n != null) {
      res = res + 1;
      n = n.next;
    }
    return res;
  }
}
```

Figure 2.2: The LinkedList Class

removed the last node in the sequence, it could not assign a value to *tail* to re-establish the list's invariant, which states the constraint *tail == head.getTail()*.

Invariants such as this, which state structural properties additional to those guaranteed by ownership, are difficult to guarantee when transitive ownership is present. Containment ensures strong encapsulation so that the type need not take transitive ownership into account in its construction. It need not provide any facilities to aid transitive owners in re-establishing the invariant.

Containment also ensures that subtypes cannot be defined which subvert the functionality of its supertype. Because any members dealing with owned types are required to be private, any class inheriting from *LinkedList* cannot access the inherited representation. This enforces stronger abstraction by completely hiding inherited internal components, and prevents subtypes from defining methods exposing these components to transitive owners.

2.2.2 Reasoning With Ownership

The previous section has discussed the structural properties of the *Node* and *LinkedList* types. Ownership ensures the hierarchical and acyclic structure of internal representations composed of owned objects. Containment enforces a stronger abstraction and encapsulation property that makes types more robust and not susceptible to incorrect modifications by transitive owners. Ownership also has implications when reasoning about invariants, state transitions, and contracts, as discussed here.

Visible States

The invariant *tail == head.getTail()* states a structural property ensuring that the *tail* attribute does always alias the tail end of the list. This allows an efficient *add()* method which does not need to iterate through the entire list. However this also implies that there is an update problem when new objects are added. If a new item is placed at the end of the list, then *tail* no longer refers to the real tail, and similarly if the last item is removed then *tail* refers to a now useless node. After such an operation and before the value of *tail* is updated, the list is in an inconsistent state.

The implementation in *LinkedList* first adds the item to the end of the list, then updates *tail*:

```
1  public void add(Object value) {
2    if (head == null) {
3      head = new owned Node(value);
4      tail = head;
5    }
6    else {
7      tail.append(value);
8      tail = tail.getTail();
9    }
10 }
```

Between lines 3 and 4, and 7 and 8, the invariant does not hold. In line 4 this is not an issue since a visible state is not reached for the current object; the *tail* attribute can be assigned without a method call. However line 8 does require a method call and so a visible state for the list may be reached.

This raises a particular issue with ownership. Although the owned objects are encapsulated within their owners, they may still acquire a reference to their owner. If an owner is not itself owned, then regular references to it exist and can be passed down to the objects it owns. Any

method call on an owned object is therefore still a visible state for the owner, unless it is certain the owner is itself owned. Classes are defined without any assumption of ownership so in general there is no method present in this type system to constrain when visible states are reached by owners.

For example, a list may store a reference itself like any other stored value:

```
LinkedList ll = new LinkedList();
ll.add(ll);
```

A more complex situation involves the list aliasing itself through other objects, such that it may modify itself by calling methods on seemingly unrelated objects. Objects stored by the list should not be modified as part of this operation, however their pure methods may be called and so they must be consistent. The *add()* must assume that any nodes it operates on may have access to the list through such an operation as this. Although it can be shown that the methods it calls do not modify the objects stored, there is no way of enforcing this in subtypes of either *LinkedList* or *Node*.

In fact line 8 does not result in a visible state for the list since *add()* is a local method called on an owned receiver. Since the nodes are owned, they do not own the list and so cannot call its methods in their local methods. The *add()* method thus cannot call a method on any object which results in a visible state for the enclosing list, thus the invariant can be correctly re-established with this statement. Simple analysis can aid the programmer in identifying when a method call may represent a visible for the calling object, and can take advantage of this combination of owned receiver and local method to correctly recognize such calls as non-visible states.

Sequential Reasoning

Locality allows sequential reasoning to make assumptions about the side-effects on other objects. Owned objects are quite often involved in local methods, thus ownership is indirectly a factor to these assumptions. Given a sequence of local method calls, properties established about objects whose methods are not called can be assumed to hold since only the local state of the receivers will be affected. Consider the Hoare triple rule for sequential composition:

$$\frac{\{P\}\ R_1\ \{Q_1\} \qquad \{Q_1\}\ R_2\ \{Q\}}{\{P\}\ R_1\ ;\ R_2\ \{Q\}}$$

If method calls to an object are considered the operations in the rule, then the pre- and post-states can be derived from the pre- and postconditions of the contracts for those methods and the object's invariant. For example, given two *Counter* instances whose methods are known to be local, a sequence of operations can be shown to establish a post-state for the objects after they have been executed:

```
Counter c1 = ... , c2 = ... ;
                  // Assume c1 != c2 && c1.get() == 5 && c2.max == 10
                  // P : c1.get() == 5 && c2.get() == 0, established by constructor
c2.add(5);        // R1: P satisfies the precondition, postcondition establishes Q1
                  // Q1: c1.get() == 5 && c2.get() == 5
c2.add(c1.get()); // R2: Q1 satisfies the precondition, postcondition establishes Q
                  // Q : c1.get() == 5 && c2.get() == 10
```

In this sequence of deductive steps, P represents the knowledge known about the system before the operations are applied. It assumes along with every other step that the objects are distinct. The second method call *c2.add()* establishes *Q1* as its postcondition, which is the necessary precondition

for the following call. *Counter.add()* requires that the argument of the call plus the current count must be no more than the counter's maximum; *Q1* states that the argument *c1.get()* satisfies this precondition.

This is a correct sequence of deductive steps due to the nature of ownership and locality. It correctly demonstrates that *c1* is not modified by the call to *add()* since this is a local method and *c1()* is not owned since it is aliased through a regular reference. If *add()* was not local, then the object *c2* could modify *c1* through this call in a way which did not correctly establish *Q1*. The locality of *add()* is maintained over inheritance, so *c2* cannot alias an object which has an overridden version of this method affecting other objects. Local methods thus ensure their side-effects are constrained to only impact their receiver's state.

2.3 The Colleague Technique

Ownership cannot capture many important object relationships. The iterator example is one of the clearest examples where an iterator cannot be encapsulated by the structure over which it traverses. It illustrates how one object created by another can have a particular relationship throughout its lifetime. Often relationships are not meant to last for as long as one of the partners exists, but is created and broken at various times.

The purpose of the Colleague Technique is to allow dynamic relationships to be soundly defined with invariants. Colleague objects place constraints on one another through invariants, such as the iterator's invariant constraining the way in which lists may be modified. The technique requires bi-directional relationships, and these in themselves can be used to describe program structure without invariants. In general however, the technique is used to define relationships which may be created and broken at runtime, whose correctness is at least partially defined by invariant conditions.

Returning to the *ListCounter* example, instead of the *IntegerList* being encapsulated by the counter, it is allowed to be freely aliased by external clients. For these clients to have adequate knowledge about the dependency relationship between the counter and the list, the relationship must be made explicit and bi-directional. A new annotation *colleague T.a* is introduced for define colleague attributes. This states that the attribute aliases colleague objects of type T which will maintain an alias back to the original object through their attribute a.

An different version of *ListCounter* and *IntegerList* is given using this annotation:

```
class ListCounter extends Counter {
  protected colleague IntegerList.counter IntegerList history;
...
}

class IntegerList {
  protected colleague Counter.history ListCounter counter;
...
}
```

The implicit invariant is that two objects which are declared to be colleagues of one another, eg. instances of these two classes, must alias one another through their respective colleague attributes. This ensures that any client of the list is aware that it is associated with a counter, and that the invariant for the counter restricts what operations are correct to perform on the list. The bi-directional aliases are not owned and so can be made available to clients who can inspect the colleague objects and determine what operations are correct.

To reflect the reciprocal nature of the relationship, a mirror invariant can be defined for *IntegerList* derived from that for *ListCounter* which states the same constraint but from the list's perspective:

```
class IntegerList {
  protected colleague Counter.history ListCounter counter;

  (\forall Integer i ; this.contains(i) ; i.intValue()<counter.value);
...
}
```

This restated constraint makes explicit what the list's obligations are and allows clients to reason about the object in the same way as those which are not colleagues. Helper methods can be defined which help clients determine what operations are valid, such as passing on the *counter.value* value, so that the identity of colleague objects may remain hidden.

Defining the list as the past values of the counter also requires that clients other than the counter cannot add or remove items. This is a harder access constraint to define, so some patterns of relationships are best described with ownership where the owner can enforce its own controls. Alternatively a means of stating constraints on the caller in a method's precondition could be introduced, requiring that the caller of methods such as *add()* must be colleagues of the list.

The following examples describe other applications of the Colleague Technique. It describes relationships which are one-to-many and not just one-to-one as the above example is. The formalization of the technique will be given in Chapter 5.

2.3.1 Subject-Observer

The Subject-Observer Pattern [50], where a subject object informs its observers that it has changed state, is a commonly-used pattern which the Colleague Technique is suited to define. Figure 2.3 defines an abstract instance of the pattern using the Colleague Technique. This simple subject has an attribute *value* representing its internal state, which must be synchronized in the observers. This is stated by the observer's invariant which requires the value to be equal when the subject is not updating itself.

Note that a number of methods are included to represent properties and operations associated with colleagues. Methods beginning with *associate* or *dissociate* are used to construct relationships, creating and removing bi-directional aliases between the colleague objects. Others such as *isAssociable* and *isAssociated* state properties about the possible relationship between the receiver object and an argument object. When the CoJava Tool generates Java code to implement the technique, these are added auxiliary methods defined in standard Java with correct JML annotations.

A number of features are important to subtle correctness properties regarding subjects and observers. The relationship between a subject and its observers must be one way. A call to *update()* causes *notify()* to be called on all observers, so if this method calls *update()* again then an infinite call sequence will be initiated. A mechanism is needed to ensure that overlapping update requests do not occur, or else a way of queuing updates is needed. Otherwise a subtype of observer could, for example, define a *notify()* method which calls *update()* on the subject:

```
class BadObserver extend Observer {
  ...
  public void notify() { subject.update(...); }
}
```

```
class Subject {
  private collegial Observer.subject Set observers;
  protected boolean isUpdating;
  protected Object value;

  public Subject(Object o){ isUpdating = false; value = o; }

  requires isUpdating;
  ensures isUpdating;
  private void notifyAll() {
    for(Observer o : this.observers)
      o.notify();
  }

  requires !isUpdating;
  ensures !isUpdating;
  public void update(Object o) {
    isUpdating = true;
    value = o;
    notifyAll();
    isUpdating = false;
  }

  ensures \result == value;
  public pure local Object get() { return value; }
}

class Observer {
  private collegial Subject.observers Subject subject;
  protected Object value;

  invariant !subject.isUpdating ==> value == subject.value;

  requires subject.isUpdating;
  public void notify() { value = subject.get(); }

  requires isAssociable(s);
  ensures isAssociated(s);
  public void setSubject(Subject s) {
    value = subject.get();
    associate_subject(s);
  }

  requires isAssociated(subject);
  ensures !isAssociated(\old(subject));
  public void clearSubject() { dissociate_subject(subject); }
}
```

Figure 2.3: The Subject-Observer Classes

This example prevents the problem by using the *isUpdating* attribute to indicate that an update is in progress. A precondition requiring this attribute to be true for the collegial subject is present in the specification for *notify()*. Therefore it can be easily inferred that the precondition for *update()* cannot be satisfied. This prevents *BadObserver* from correctly calling *update()* as shown. A more complex situation would involve an observer instance interacting with some other object which then wished to update the subject, in which case the normal burden of establishing the precondition of any call prevents overlapping updates.

The alternative approach is to define *notify()* as being local, in which case only the local state of the observer can be affected by the updating process. Such a constraint would work in the given example but an observer would reasonably be more complex than this and would require non-local operations in reaction to a subject updating itself.

The mirror invariant injected into the specification for *Subject* is the following:

(\forall Observer o ; observers.contains(o) ; !isUpdating ==> value == o.value)

The method *update()* allows the subject to be modified by setting *isUpdating* to true, and then performs the update. The implication in this invariant requires the values to be the same only when an update is not in progress, thus setting *isUpdating* to true allows making changes will may invoke invariant checks. The *notify()* will have to set *isUpdating* to false before exiting, so it must re-establish the invariant in its post-state. This invariant ensures that the method *update()* transitions the subject and its observers only from one correct state to another.

The invariant stated in this example represents a basic consistency property between subjects and observers. Other instances of the pattern can be defined stating more complex requirements between the two types, but in either case the basic advantage the Colleague Technique provides is to manage the relationship and explicitly state what the obligations are. Any instance of the pattern would have to define the constraints in a way which works with the pattern so that relationships can be correctly created and destroyed, and observers correctly notified of changes. This must be done anyway even without the technique or any formal specification at all, however with specification and colleagues the requirements are defined formally and explicitly where they would otherwise be implicit to the structure and type definitions.

2.3.2 Doubly-Linked List

Returning to the linked list example, it is clear that ownership disallows a doubly-linked structure used to define a double-ended queue (deque). The Colleague Technique on the other hand allows such a structure through a self-collegial *DequeNode* class (Figure 2.4) as discussed in Section 5.3.2. This type is defined to be collegial with itself, such that the *next* and *prev* attributes are colleague attributes of one another. If some node *a* through its attribute aliases another *b*, then certainly *b.prev* will alias *a*. Although an internal representation should be composed of owned objects, instead non-owned instances of the *DequeNode* are used to define the internal representation of *Deque*.

The Colleague Technique thus statically ensures the doubly-linked structure. What is not provided is the encapsulation or ownership, therefore the nodes in the list are not guaranteed to be structured in hierarchies nor contained by an owning object. An acyclic structure is enforced with the given invariant requiring each successive node to have a position one greater than the last, thus

```
class DequeNode {
  private collegial DequeNode.prev DequeNode next;
  private collegial DequeNode.next DequeNode prev;
  private spec_public int pos;

  public Object value;

  invariant pos > 0;
  invariant pos == next.pos - 1;
  invariant pos == prev.pos + 1;

  ensures this.value == value;
  public DequeNode(Object value) { this.value = value; }

  ensures next != null ==> \result == next.size();
  ensures next == null ==> \result == pos;
  public pure size() {
    int size = pos;

    if(next != null)
      size = next.size();

    return size;
  }

  requires next != null;
  ensures next.next == null ==> \result == next;
  ensures next.next != null ==> \result == next.getTail();
  public pure DequeNode getTail() {
    DequeNode d = next;

    if(next.next != null)
      d= next.getTail();

    return d;
  }
...
}
```

Figure 2.4: The DequeNode Class

preventing a node from linking to one previous in the chain whose *pos* would not satisfy the constraint. Encapsulation is much harder to ensure since there is no static mechanism to prevent leaking a node's reference to an arbitrary client save for careful design. However an iterator traversing the structure would be able to alias the nodes directly and so efficiently be able to move forward. The correctness properties this iterator would require are the same as those discussed in Chapter 5 and so the type would be defined as being collegial with *Deque*.

Methods performing common functions with this type require slightly different definitions. The *pos* value actually records the size of the list up to that point, thus if the node is at the end of the list then its position value is the total size. The method *size()* is defined to take advantage of this. *getTail()* has a slight problem in that *this* is owned and so cannot be returned by the method. The alternative is to look one node ahead and return *next* when that is the tail of the list. If the

list contains only one node then this method's precondition cannot be satisfied, thus the enclosing *Deque* class must account for this special case instead.

This definition also has the drawback that its methods cannot be defined as local since they must call methods on colleague objects, which are by definition not local state. Methods of the *Deque* type consequently cannot be local either if they call those of the contained nodes. A type based on collegial linked nodes thus will have a different interface from those based on owned nodes, so there is difficulty in implementing common interfaces as well as using such types in situations where locality is needed. The invariant of *Deque* will not be defined in quite the same way since the nodes comprising its structure are not owned nor collegial with *Deque*. Either the *DequeNode* type should be modified to also be collegial with *Deque*, hence introducing problems of even closer object coupling, or else structural correctness must be described in some other fashion.

2.3.3 Other Uses

Beyond simple node structures, the Colleague Technique can be used to define complex aggregate structures whose relationships are changeable. Patterns such as Visitor become much easier to implement in the familiar manner since the components of these structures are not contained within a rigid ownership structure, but which can still define internal correctness criteria in a sound manner. The Composite pattern also is more easily implemented by defining a *Composite* type which maintains one-to-many relationships with those other objects composing its structure. The Colleague Technique is flexible enough to allow one type to be collegial with its subtype, thus this is a practical way to formalize this pattern.

Ownership is applicable in other patterns where rigid structure is not a drawback. When one object wraps another, such as Proxy or Decorator, the wrapping object can often own the internal object if it's not needed to be externally accessible. Instances of Facade can employ ownership to ensure the objects which define its functionality are completely hidden from external clients.

Ownership's strength and weakness is its rigidity, which allows it to statically guarantee the encapsulation and invariant soundness properties, but at the expense of runtime flexibility. The Colleague Technique represents a very different way of defining sound object relationships. These relationships are far more dynamic and flexible than those defined with ownership.

However Colleagues requires types to anticipate these relationships with dedicated members, thus producing close coupling between object types. Invariants can soundly constrain owned and collegial objects, with ownership defining static relationships while the Colleague Technique defines dynamic ones. Different situations depending on what relationships are needed determine which of the two techniques to use, if not both in co-operation.

2.4 Conclusion

This chapter has briefly touched upon the problems of specifying relationships between objects, and presented the ownership and Colleague Technique solutions. Ownership has the immediate benefit of statically enforcing encapsulation. A simple but effective example of this involves the internal definitions of abstract data types. Beyond these specific types of classes, encapsulation promotes abstraction in any type definition, thus a guarantee of its presence even in subtypes is very useful.

Reasoning with ownership and its related concepts of containment offer some advantages in the form of aliasing assumptions about objects. Whether ownership objects will have fewer clients

and hence the possibility of side-effects invalidating any local reasoning is reduced. Local methods certainly will produce side-effects only for their receivers and the objects it owns; this assumption allows sequential reasoning where one local method call can certainly not affect properties already established about other objects.

Colleagues are useful mechanisms for specifying complex relationships between types. This has been shown here to be applicable to the definitions of abstract data types and the specification of instances of many popular design patterns. Reasoning with collegial types is the same as with classical approaches since the guarantees they provide employ standard invariants. A system using the technique can define more dynamic object structures than one which only used ownership.

The next chapter will provide a formal definition of the CoJava language. This is based on the Lightweight Java language which defines a very small core subset of Java. Although the techniques in this chapter have been stated in terms of Java, formalizing them in the following chapters correctly captures the semantics that would be expected when they are applied to the full Java language.

Chapter 3

CoJava

The CoJava language is defined in this chapter as a subset of the Java language [51, 52] which includes the features of interest. The language is defined as an extension to the Lightweight Java[1] [115, 116] language, whose type-safety has been proven using Isabelle/HOL [95] based on proof obligations generated by the Ott Tool [112].

CoJava represents a small sequential subset of Java. Functional subsets of Java lacking statements have been defined in other work [49, 66, 103] as a basis for the formalization of certain concepts. Having statements in a language however allows a discussion on ownership and specification to include program state. This chapter will first outline Lightweight Java then define the extensions which create the CoJava language. These extensions do not affect the type-safety of Lightweight Java, thus CoJava also is type-safe.

In this chapter, Section 3.1 will discuss the previous research in formalizing Java or various subsets thereof. Section 3.2 introduces the Lightweight Java language as a formally-defined object-oriented language whose semantics can be described as a subset of that of Java. The CoJava language is then defined as an extension to Lightweight Java in Section 3.3, which includes the addition of the ownership type system and specification predicates to capture a DbC specification.

3.1 Formalizing Java

The primary definition for Java is the official Java Language Specification [52]. Serving as the technical reference to the language, it describes Java in relatively informal language, although the full context free grammar is given. As a basis for proving formal properties of Java, such as type safety, this is not sufficiently precise nor useful in elegant mathematical proofs.

Proving type safety of Java quickly became an important topic soon after the language's introduction. It was found early on that that user-defined class loaders indeed broke type safety [110]. Assuming this loophole is corrected then the Java type system can be shown to be type safe for subsets of the language of varying sizes [43, 44, 96, 99, 117]. To prove type safety, these efforts have provided formal definitions for Java subsets which contain enough of Java's type system that extending the proved property to the whole language is relatively straight forward.

The definitions for ownership type systems based on Java's type system extend and modify this formal work, as seen in [3, 4, 5, 34, 92]. These definitions of subset languages aim to capture the

[1] `http://www.cl.cam.ac.uk/research/pls/javasem/lj/`

operational semantics of Java while restricting aliasing in particular ways. Necessarily these describe the same runtime behaviour but impose constrained type requirements, such that certain constructs no longer type correctly.

The introduction of generics into Java [11, 25] presented a new challenge to type safety. Again definitions for subset languages have been devised [66] for which type safety is proved. Extending this property to the full language is meant to be simple given that the subset language includes all of Java's relevant type features. This can be further extended into generic ownership [39, 40] which aims to enforce the same encapsulation properties in the presence of generic types.

These research efforts have defined Java or Java subsets using Structural Operational Semantic [102] (small-step) or Natural Semantic [67] (big-step) rules. Such rules define the relationship between types, the type of expressions and other language constructs, and the runtime semantics of the language in terms of inference rules. For example, the transitive nature of Java types, where if type D subtypes E and C subtypes D then C subtypes E, can be stated as a general rule:

$$\frac{C \preccurlyeq D \qquad D \preccurlyeq E}{C \preccurlyeq E}$$

Inference rules of this form can be used in derivations to prove properties of the language at hand. This is used to prove type safety by showing that no conclusion can be derived from the language's rules stating that a well-typed operation has allowed a program to "go wrong". For example, a value of type E cannot be assigned to a variable with type D, therefore the rule defining assignment cannot be used to derive a valid operation that allows this.

Alternatively the semantics of Java could be defined with the denotational [111] style, which describe semantics in terms of functions over state, or axiomatic [57] semantics, which describe semantics as properties that hold before and after operations. These approaches are better suited for certain purposes, however operational semantics based on inference rules is well suited to proving properties about a language's runtime behaviour. Abstract state machines [54] are suited for defining Java where the execution at runtime often diverges considerably from what the syntax of the code would indicate [20]. LJ lacks branching statements such as *break* or *continue*, allows *return* only in one place, and lacks exceptions except as definitions of terminal states arising from null pointer referencing. Thus the language's runtime behaviour more closely matches the syntax, making state machines a less attractive means of definition.

Lighweight Java is defined in this chapter using operational semantic rules. Firstly the abstract syntax is defined, followed by rules describing the type information of a LJ program, subtyping, well-formedness, and variable translation. The last set of rules define statement reductions, which describe the computation of a LJ program in terms of small-step semantics. Through the use of the Ott Tool to generate proof obligations from LJ's formal definition, the type-correctness of these rules has been proven mechanically with the Isabelle/HOL theorem prover.

3.2 Lightweight Java

The abstract syntax for Lightweight Java (LJ) is given in Figure 3.1, illustrating a core subset of Java which includes enough interesting behaviour for discussing semantics and correctness. The language itself is not practical to use nor give examples in, but the lack of features does not prevent certain programs being defined. Features of Java such as constructors, static members, inner and anonymous types, may add convenience to the language, but these and others are not essential

P	::=	$\overline{cld}$	– Program
C, cl, dcl	::=		– Class names
		`Object`	– Base class Object
	|	dcl	– Class name
cld	::=	**class** dcl **extends** cl $\{\overline{fd}$ $\overline{meth\_def}\}$	– Class definition
fd	::=	$cl\ f;$	– Attribute definition
$meth\_def$	::=	$meth\_sig\ \{meth\_body\}$	– Method definition
$meth\_sig$	::=	$cl\ meth(\overline{vd})$	– Method signature
vd	::=	$cl\ var$	– Variable definition
$meth\_body$	::=	$\overline{s}$ **return** $y;$	– Method body
s	::=		– Statements
		$\{\overline{s_k}^k\ \}$	– Block Statement
	|	$var = x;$	– Assign variable to variable
	|	$var = x.f;$	– Assign attribute to variable
	|	$x.y = y;$	– Assign variable to attribute
	|	**if** $(x == y)$ s **else** s'	– Conditional statement
	|	$var =$ **new**$_{ctx}$ $cl();$	– Object creation
	|	$var = x.meth(\overline{y});$	– Method call
$TVar, x, y$	::=		– Term variables
		var	– Regular variables
	|	**this**	– Ref. to current object

Figure 3.1: Lightweight Java Abstract Syntax

to Turing-completeness. However inheritance, as a mechanism for re-use and abstraction, is so important to much of object-oriented system design and architecture that its presence is essential.

Only object types are present in the language. In pure object-oriented languages numbers, such as *int*, are represented with object types. Integer numbers can be defined with objects through Church Encoding [30, 31], therefore primitive types with their associated arithmetic operations can be represented in LJ albeit in a cumbersome form. The literal number *3* can, for example, be thought of as representing *(new Nat()).succ().succ().succ()* in Java, where *Nat* is the natural number class with *succ()* defining the successor function.

The absence of certain features does mean that examples in the LJ language would be very cumbersome to define. The lack of loops means that recursive methods would be used instead, and the lack of local variables means that additional method arguments would have to be defined and then used as variables rather than input for the method call. Consequently, examples in this chapter will be mostly given in Java, but will include no concepts which cannot be defined in LJ. For example, the *Counter* class in LJ would be the following:

```
class Counter {
  int value;
  int max;

  Counter init(int max) {
    this.value = new int();
    this.max = max;
    return this;
  }
```

```
  Counter inc(int temp) {
    temp = this.value;
    temp = temp.succ();
    this.value = temp;
    return this;
  }

  Counter add(int n,int temp) {
    temp = this.value;
    temp = temp.add(n);
    this.value = temp;
    return this;
  }

  public int get() { return value; }
}
```

A method *init()* is introduced to serve the purpose of a constructor. Methods which would normally return *void* instead return a reference to the current object. Since LJ methods must always return a value, either an instance of a *Void* type must be constructed and returned, or else this simple expedient used. Temporary variables, in the form of arguments, must be used to call methods of the attributes (*succ()* representing the next value and *add()* representing mathematical addition) owing to the limitations of the syntax. This definition is functionally equivalent to the Java version, given a correct class type to represent *int*.

Owing to this difficulty in stating real examples in LJ, a larger subset of Java will be used in examples. Added syntax such as operators, local variables, and literals will be used for brevity, but which have semantically equivalent constructs in LJ. As shown above, the *inc()* method body of the original Java version of *Counter* consisted of one statement (*value = value + 1;*) while the equivalent LJ body must have four statements. Both versions of *inc()* and any following examples given in Java will have a clear equivalence in LJ, thus any concepts described in LJ's formalization are applicable to a subset of Java which has an LJ equivalent.

Figure 3.2 defines the syntax for lists in the above syntax and that of the formal definition to follow. A list of ι type elements in represented as $\overline{\iota}$. Such a list of a size k is given as $\overline{\iota_k}^k$. A list whose elements are composed of multiple components may be presented as $\overline{\iota_k\ \iota'_k}^k$, in which case the list $\overline{\iota'_k}^k$ is derived from this by taking the ι' component from each element. An empty list is represented as [], whereas the absence of an atom ι is represented as $\varnothing$.

$\overline{\iota}$	$::= [] \mid \iota \ldots \iota$	– List of elements of type ι
$\overline{\iota_k}^k$	$::= [] \mid \iota_1 \ldots \iota_k$	– List of k labeled elements of type ι
$\iota_0 : \overline{\iota_k}^k$	$::= \iota_0 \mid \iota_0, \iota_1 \ldots \iota_k$	– List of k elements prepended with element ι_0
ι_{opt}	$::= \varnothing \mid \iota$	– Optional element, either none or ι
$\overline{p(\iota_k)}^k$	$::= p(\iota_1) \wedge \ldots \wedge p(\iota_k)$	– List universal quantification
$\overline{v(\iota_k, \iota'_k)}^k$	$::= \overline{\iota'_k}^k = \{\iota : \overline{\iota_k}^k \mid v(\iota, \iota') \bullet \iota'\}$	– List definition

Figure 3.2: Lightweight Java List Syntax

The overline syntax is also used to define universal quantification and define lists through predi-

cates. If a predicate p holds for all elements of some list $\overline{\iota_k}^k$, this can be stated as $\forall\, \iota : \overline{\iota_k}^k \bullet p(\iota)$ or $\overline{p(\iota_k)}^k$.

Given a predicate v which relates two atoms, $v(\iota, \iota')$, then $\overline{v(\iota_k, \iota'_k)}^k$ states that a list $\overline{\iota'_k}^k$ is composed of each ι' such that $v(\iota, \iota')$ for every ι in $\overline{\iota_k}^k$. This is equivalent to a list comprehension of the form $\{\iota : \overline{\iota_k}^k \mid v(\iota, \iota') \bullet \iota'\}$.

3.2.1 Types, Type Environment, and State

Figure 3.3 gives the abstract syntax for types, the type environment function Γ, and the state functions L and H. A type is composed of a context and a class identifier as (ctx, cld) or $ctx.cld$. The context portion is defined for now as empty but shall be used as an extension mechanism in LJ to define different categories of object types. It will be used in CoJava to define ownership types in conjunction with a modified notion of subtyping. A type τ is thus a context/identifier pair, no type at all ($\varnothing$), or a lookup in Γ or H.

The following definitions describe Γ, L, and H:

Definition 3.2.1 (Environment Function Γ)
The function Γ recalls the static type of variables. It maps variables to types: $TVar \to \tau$.
$\Gamma[x \mapsto \tau']$ states the function override associating the variable x with type τ'.

Definition 3.2.2 (Variable State Function L)
The function L recalls the state of variables, hence it relates variable names to values: $TVar \to Val$.
$L[x \mapsto v]$ associates the variable x with the value v.

Definition 3.2.3 (Heap Function H)
The function H represents the heap of a running program. It maps object references to a pair, the first element of which is the object's runtime type, and the second is a map from attribute names to values: $oid \to (\tau, f \to Val)$.

- *The function override $H[oid \mapsto (\tau, f_1 \mapsto v_1, \ldots, f_k \mapsto v_k)]$ introduces a new object referred to by oid with type τ and attributes f_1 to f_k.*
- *$H[(oid, f) \mapsto v]$ replaces the value of the attribute f of the object oid with the value v but otherwise does not change the state of the heap.*
- *The application $H(oid, f)$ yields the value of the attribute f of the object oid. This is shorthand for $((H\ oid).2)\ f$, where .2 represents the second element of the pair.*
- *The application $H(oid)$ yields the runtime type of the object oid, hence is is shorthand for $(H\ oid).1$.*

---

ctx	::=		– Context (none for now)
$ctxcld$	::=	(ctx, cld)	– Class definition in context
$ctxmeth\_def$	::=	$(ctx, meth\_def)$	– Method definition in context
$Type, \tau$	::=		– Type
		$ctx.\texttt{Object}$	– Supertype of all types
	|	$ctx.dcl$	– Class identifier
τ_{opt}	::=		– Result of type lookup
		$\varnothing$	– None
	|	τ	– Some
	|	$\Gamma(x)$	– Static type lookup
	|	$H(oid)$	– Dynamic type lookup
$\tau_{opt}^{\perp}$	::=		– Type lookup that can abort
		τ_{opt}	– Result
	|	$\perp$	– No type found
π	::=	$\overline{\tau} \to \tau$	– Method type definition
Γ	::=		– Type environment
		$\Gamma[x \mapsto \tau]$	– Γ with $x \mapsto \tau$
	|	$[x_1 \mapsto \tau_1 \ldots x_k \mapsto \tau_k]$	– Type mappings
Val, v, w	::=		– Value
		$\texttt{null}$	– Null value
	|	oid	– Object identifier
v_{opt}	::=		– Value lookup result
		v	– Value
	|	$L(x)$	– Lookup value of local variable
	|	$H(oid, f)$	– Lookup value of field
L	::=	$L[x \mapsto v]$	– Variable state L with $x \mapsto v$
H	::=		– Heap
		$H[oid \mapsto (\tau, f_1 \mapsto v_1, \ldots, f_k \mapsto v_k)]$	– H with new oid of type τ
	|	$H[(oid, f) \mapsto v]$	– H with $(oid, f) \mapsto v$
$config$	::=		– Configuration
		$(P, L, H, \overline{s_k}^k)$	– Normal config
	|	$(P, L, H, Exception)$	– Exception occurred
$Exception$	::=	NPE	– Exceptions (Null pointer exception only)

Figure 3.3: Lightweight Java Formal Definition Elements

3.2.2 Configurations

A program configuration is the state of a running program, including the heap and variable state as well as the program statements to be executed. A normal configuration will have a sequence of statements representing the computation of the program to follow, whereas an exceptional configuration will state what exception was thrown. An exceptional configuration represents a terminal state of a program, since no progress in LJ is possible after an exception is encountered.

Definition 3.2.4 (Configurations $(P, L, H, \overline{s})$)
A program configuration is a tuple $(P, L, H, \overline{s})$ representing the state of a LJ program. P is the program being executed, L the state of variables in the program, H the heap of objects, and $\overline{s}$ the statement sequence representing the computation to follow.

Any configuration of the form $(P, L, H, Exception)$ is an exceptional terminal configuration indicating the program has encountered an error and is unable to progress.

3.2.3 Type Information

The following predicate and function definitions are used to reason about the type information in a program P. They are used to discuss the notions of well-formedness and subtyping. The first of these simply represent the information about class members, such as **class_name**(cld) which represents the name for the class definition cld. Later functions represent the collected information about all the members inherited by a given class definition, such as collecting together all the attributes a class inherits as well as newly defined. The full definition of these functions is found in Appendix A.

These functions recall the constituent parts of a class definition:

class_name(**class** dcl **extends** cl $\{\overline{fd}\ \overline{meth\_def}\}) = dcl$
superclass_name(**class** dcl **extends** cl $\{\overline{fd}\ \overline{meth\_def}\}) = cl$
class_fields(**class** dcl **extends** cl $\{\overline{fd}\ \overline{meth\_def}\}) = \overline{fd}$
class_methods(**class** dcl **extends** cl $\{\overline{fd}\ \overline{meth\_def}\}) = \overline{meth\_def}$
method_name(cl $meth(\overline{vd})\{meth\_body\}) = meth$

Other functions found in the appendix and used in the following definitions are summarized here, given a program P:

distinct(X)	– The list X of constructs contains distinct elements
distinct_names(P)	– The names of classes in P are distinct
find_path$(P, \tau) = \overline{(ctx, cld)}$	– The list of classes which τ inherits from (including τ itself)
ftype$(P, \tau, f) = \tau'$	– The type of the attribute f of type τ is τ'
mtype$(P, \tau, meth) = \overline{\tau_k}^k \to \tau'$	– The type of method $meth$ of type τ is $\overline{\tau_k}^k \to \tau'$
find_type$(P, ctx, cl) = \tau$	– The type representing the class cl with context ctx is τ

3.2.4 Subtyping

The subtyping relation $\prec$ is defined as a partial order on types. Given a class named dcl which inherits from another cl, the relation between the types representing these classes is $ctx.dcl \prec ctx.cl$ for some context ctx. Subtyping is transitive, so for any type $ctx.cl'$ such that $ctx.cl \prec ctx.cl'$, then $ctx.dcl \prec ctx.cl'$ is also true.

All types are subtypes of ctx.`Object`, where ctx is the same context as that for τ[2]:

find_path$(P, \tau) = \overline{ctxcld}$
$\tau = ctx.cld$

$P \vdash \tau \prec ctx$.`Object`

[2]This is an addition to the original LJ rule since `Object` was not originally associated with a context

Given a type τ and a list of types representing the inheritance path from τ to $ctx.\texttt{Object}$, τ is a subtype of each type in that list. In this way $\prec$ is defined as a transitive relation, since if $\tau \prec \tau'$ and $\tau' \prec \tau''$ then τ' and τ'' will be in the list of types τ inherits from, hence $\tau \prec \tau''$ is also true. Since τ is also in the path, then this implies that $\tau \prec \tau$ is also true.

$$\frac{\begin{array}{l}\textbf{find\_path}(P, \tau) = \overline{(ctx_k, cld_k)}^k \\ \overline{\textbf{class\_name}(cld_k) = dcl_k}^k \\ (ctx', dcl') \in \overline{(ctx_k, cld_k)}^k\end{array}}{P \vdash \tau \prec ctx'.dcl'}$$

Given two lists of types of equal length where each type in one list subtypes another in the same position of the second list, then the first list is defined as a subtype of the second:

$$\frac{\begin{array}{l}\overline{\tau} = \overline{\tau_k}^k \\ \overline{\tau}' = \overline{\tau_k'}^k \\ \overline{P \vdash \tau_k \prec \tau_k'}^k\end{array}}{P \vdash \overline{\tau} \prec \overline{\tau}'}$$

If two optional types are in fact types, and not $\varnothing$, which are related through subtyping, then the optional variable is related through subtyping as well:

$$\frac{\begin{array}{l}\tau_{opt} = \tau \\ {\tau_{opt}}' = \tau' \\ P \vdash \tau \prec \tau'\end{array}}{P \vdash \tau_{opt} \prec {\tau_{opt}}'}$$

null is a subtype of all types:

$$\frac{\tau_{opt} = \tau}{P, H \vdash \textbf{null} \prec \tau_{opt}}$$

If the heap records that an *oid* has a particular runtime type which is a subtype of τ_{opt}, then the *oid* itself also is defined as subtype of τ_{opt}:

$$\frac{P \vdash H(oid) \prec \tau_{opt}}{P, H \vdash oid \prec \tau_{opt}}$$

3.2.5 Well-formedness

Well-formedness covers the criteria for well-formed sets of types, correct type hierarchy, and well-typedness. A well-formed program, heap, or variable store must meet certain conditions relating

to correct types and configuration. It is assumed that a program can only be executed if it is well-formed, and no permissible operation will render anything ill-formed.

A variable store L is well-formed if it is finite and stores values whose type correspond to the type of the variable:

$$\frac{\begin{array}{l}\textbf{finite}(\mathrm{dom}(L)) \\ \forall\, x \in \mathrm{dom}(\Gamma) \bullet P, H \vdash L(x) \prec \Gamma(x)\end{array}}{P, \Gamma, H \vdash L} \quad \text{WF\_VarStore}$$

A heap H is well-formed if it is finite and every attribute stores a value which is a subtype of the attribute's type:

$$\frac{\begin{array}{l}\textbf{finite}(\mathrm{dom}(H)) \\ \forall\, oid \in \mathrm{dom}(H) \bullet \\ \quad \exists\, \tau \mid H(oid) = \tau \bullet \\ \quad\quad \forall f : \textbf{fields}(P, \tau) \bullet P, H \vdash H(oid, f) \prec \textbf{ftype}(P, \tau, f)\end{array}}{P \vdash H} \quad \text{WF\_Heap}$$

An exceptional configuration is well-formed if the program, heap, and variable store are all well-formed:

$$\frac{\begin{array}{l}\vdash P \\ P \vdash H \\ P, \Gamma, H \vdash L\end{array}}{\Gamma \vdash (P, L, H, \mathit{Exception})} \quad \text{WF\_ExConfig}$$

An normal configuration is well-formed if the program, heap, variable store, and statements to reduce are all well-formed:

$$\frac{\begin{array}{l}\vdash P \\ P \vdash H \\ P, \Gamma, H \vdash L \\ \overline{P, \Gamma \vdash s_k}^{k}\end{array}}{\Gamma \vdash (P, L, H, \overline{s_k}^{k})} \quad \text{WF\_NormalConfig}$$

A statement block is well-formed if each statement is well-formed:

$$\frac{\overline{P, \Gamma \vdash s_k}^{k}}{P, \Gamma \vdash \{\overline{s_k}^{k}\}} \quad \text{WF\_StmtList}$$

A variable assignment is well-formed if the type of the value to assign is a subtype of that of the variable:

$$\frac{P \vdash \Gamma(x) \prec \Gamma(var)}{P, \Gamma \vdash var = x;} \quad \text{WF\_AssignVar}$$

A variable assignment of an attribute is well-formed if the type of the attribute is a subtype of that of the variable:

$$\frac{\begin{array}{l}\Gamma(x) = \tau \\ \textbf{ftype}(P, \tau, f) = \tau' \\ P \vdash \tau' \prec \Gamma(var)\end{array}}{P, \Gamma \vdash var = x.f;} \quad \text{WF\_AssignAttr}$$

An attribute assignment is well-formed if the type of the variable is a subtype of that of the attribute:

$$\frac{\begin{array}{l}\Gamma(x) = \tau \\ \textbf{ftype}(P, \tau, f) = \tau' \\ P \vdash \Gamma(y) \prec \tau'\end{array}}{P, \Gamma \vdash x.f = y;} \quad \text{WF\_AttrAssign}$$

A conditional statement is well-formed if the values x and y to be compared are related through subtyping, and that the statements are well-formed:

$$\frac{\begin{array}{l}P \vdash \Gamma(x) \prec \Gamma(y) \vee P \vdash \Gamma(y) \prec \Gamma(x) \\ P, \Gamma \vdash s_1 \\ P, \Gamma \vdash s_2\end{array}}{P, \Gamma \vdash \textbf{if } (x == y)\ s_1 \textbf{ else } s_2} \quad \text{WF\_If}$$

An object creation statement is well-formed if the class name cl corresponds to an existing type, which is also a subtype of that of the variable:

$$\frac{\begin{array}{l}\textbf{find\_type}(P, ctx, cl) = \tau \\ P \vdash \tau \prec \Gamma(var)\end{array}}{P, \Gamma \vdash var = \textbf{new}_{ctx}\ cl();} \quad \text{WF\_New}$$

A method call is well-formed if the type of each argument is a subtype of that of the argument variable they are being substituted for, and if the return type of the method is a subtype of that of the variable:

$$\frac{\begin{array}{l}\overline{y} = \overline{y_k}^{\,k} \\ \Gamma(x) = \tau \\ \textbf{mtype}(P, \tau, meth) = \overline{\tau_k}^{\,k} \to \tau' \\ \overline{P \vdash \Gamma(y_k) \prec \tau_k}^{\,k} \\ P \vdash \tau' \prec \Gamma(var)\end{array}}{P, \Gamma \vdash var = x.meth(\overline{y});} \quad \text{WF\_MethCall}$$

A method is well-formed as defined for the type $ctx.dcl$ if its arguments have unique names and existing types, each statement is well-formed, its declared return type exists, and the type of the return value is a subtype of the declared return type:

$$\frac{\begin{array}{l}\textbf{distinct}(\overline{var_k}^{\,k}) \\ \overline{\textbf{find\_type}(P, ctx', cl_k) = \tau_k}^{\,k} \\ \Gamma = [\overline{var_k \mapsto \tau_k}^{\,k}][\textbf{this} \mapsto ctx.dcl] \\ \overline{P, \Gamma \vdash s_l}^{\,l} \\ \textbf{find\_type}(P, ctx'', cl) = \tau \\ P \vdash \Gamma(y) \prec \tau\end{array}}{P \vdash_{ctx.dcl} cl\ meth(\overline{cl_k\ var_k}^{\,k})\{\overline{s_l}^{\,l}\ \textbf{return}\ y;\ \}} \quad \text{WF\_Meth}$$

A well-formed type definition in the context ctx must inherit from an existing type τ, have unique attribute names which are disjoint from all those inherited through τ, have attributes with defined types, and have well-formed methods with unique names such that any method with the same name as one inherited from τ (ie. a method override) also has the same type:

$$\frac{\begin{array}{l}\textbf{find\_type}(P, ctx, cl) = \tau \\ ctx.dcl \neq \tau \\ \textbf{distinct}(\overline{f_j}^{\,j}) \\ \textbf{fields}(P, \tau) = \overline{f} \\ \overline{f_j}^{\,j} \perp \overline{f} \\ \overline{\textbf{find\_type}(P, ctx, cl_j) = \tau_j}^{\,j} \\ \overline{P \vdash_{ctx.dcl} meth\_def_k}^{\,k} \\ \overline{\textbf{method\_name}(meth\_def_k) = meth_k}^{\,k} \\ \textbf{distinct}(\overline{meth_k}^{\,k}) \\ \textbf{methods}(P, \tau) = \overline{meth'_l}^{\,l} \\ \overline{\textbf{mtype}(P, ctx.dcl, meth'_l) = \pi_l}^{\,l} \\ \overline{\textbf{mtype}(P, \tau, meth'_l) = \pi'_l}^{\,l} \\ \overline{meth'_l \in \overline{meth_k}^{\,k} \Rightarrow \pi_l = \pi'_l}^{\,l}\end{array}}{P \vdash_{ctx} (dcl, cl, \overline{cl_j\ f_j;}^{\,j}, \overline{meth\_def_k}^{\,k})} \quad \text{WF\_Type}$$

A well-formed class in a program P must be a member of that program's class list and have a well-formed type definition:

$$\frac{\begin{array}{l}P = \overline{cld} \\ \textbf{class}\ dcl\ \textbf{extends}\ cl\ \{\overline{fd}\ \overline{meth\_def}\} \in \overline{cld} \\ P \vdash (dcl, cl, \overline{fd}, \overline{meth\_def})\end{array}}{P \vdash \textbf{class}\ dcl\ \textbf{extends}\ cl\ \{\overline{fd}\ \overline{meth\_def}\}} \quad \text{WF\_Class}$$

A well-formed program must define distinct class names, contain only well-formed classes whose type hierarchy is acyclic, and have a well-formed initial statement s:

$$\frac{\begin{array}{l}P = \overline{cld_k}^{\,k} \\ \textbf{distinct\_names}(P) \\ \overline{P \vdash cld_k}^{\,k} \\ \textbf{acyclic\_clds}(P)\end{array}}{\vdash P} \quad \text{WF\_Program}$$

3.2.6 Variable Translation

The following variable translation rules are used to ensure that variable names remain distinct during reduction operations. θ is a function mapping old variable names to new ones. The judgment $\theta \vdash s \rightsquigarrow s'$ states that s' is the translated version of an initial statement s with old variable names replaced with new as θ dictates.

Definition 3.2.5 (Variable Translation Function θ)
θ is a function from variables to variables: $x \to y$. An override of the function is given as $\theta[x \mapsto y]$ in which the variable x is mapped with y in addition to any other mappings present in θ.

$$\frac{\overline{\theta \vdash s_k \rightsquigarrow {s'}_k}^{k}}{\theta \vdash \{\overline{s_k}^{k}\} \rightsquigarrow \{\overline{s'_k}^{k}\}}$$

$$\frac{\theta(var) = var' \qquad \theta(x) = x'}{\theta \vdash var = x; \rightsquigarrow var' = x';}$$

$$\frac{\theta(var) = var' \qquad \theta(x) = x'}{\theta \vdash var = x.f; \rightsquigarrow var' = x'.f;}$$

$$\frac{\theta(x) = x' \qquad \theta(y) = y'}{\theta \vdash x.f = y; \rightsquigarrow x'.f = y';}$$

$$\frac{\theta(x) = x' \qquad \theta(y) = y' \qquad \theta \vdash s_1 \rightsquigarrow {s'}_1 \qquad \theta \vdash s_2 \rightsquigarrow {s'}_2}{\theta \vdash \textbf{if } (x == y)\ s_1 \textbf{ else } s_2 \rightsquigarrow \textbf{if } (x' == y')\ {s'}_1 \textbf{ else } {s'}_2}$$

$$\frac{\theta(var) = var'}{\theta \vdash var = \textbf{new}_{ctx}\ cl(); \rightsquigarrow var' = \textbf{new}_{ctx}\ cl();}$$

$$\frac{\theta(var) = var' \qquad \theta(x) = x' \qquad \overline{\theta(y_k) = {y'}_k}^{k}}{\theta \vdash var = x.meth(\overline{y_k}^{k}); \rightsquigarrow var' = x'.meth(\overline{y'_k}^{k});}$$

3.2.7 Statement Reductions

Statement reductions represent the computation of a LJ program. Each rule defines how a statement is eliminated from the list of statements to reduce, and how this affects the heap and variable store. The result is a transition from a normal configuration to a normal or exceptional configuration. When erroneous runtime conditions are encountered, the result is a exceptional configuration, otherwise the result is a normal configuration.

$$\frac{}{(P, L, H, \{\overline{s_k}^k\}\overline{s'_l}^l) \longrightarrow (P, L, H, \overline{s_k}^k\overline{s'_l}^l)} \quad \text{SR\_Block}$$

$$\frac{L(x) = v}{(P, L, H, var = x;\ \ \overline{s_l}^l) \longrightarrow (P, L[var \mapsto v], H, \overline{s_l}^l)} \quad \text{SR\_AssignVar}$$

$$\frac{L(x) = \mathbf{null}}{(P, L, H, var = x.f;\ \ \overline{s_l}^l) \longrightarrow (P, L, H, \mathbf{NPE})} \quad \text{SR\_AssignAttrEx}$$

$$\frac{\begin{array}{l}L(x) = oid\\ H(oid, f) = v\end{array}}{(P, L, H, var = x.f;\ \ \overline{s_l}^l) \longrightarrow (P, L[var \mapsto v], H, \overline{s_l}^l)} \quad \text{SR\_AssignAttr}$$

$$\frac{L(x) = \mathbf{null}}{(P, L, H, x.f = y;\ \ \overline{s_l}^l) \longrightarrow (P, L, H, \mathbf{NPE})} \quad \text{SR\_AttrAssignEx}$$

$$\frac{\begin{array}{l}L(x) = oid\\ L(y) = v\end{array}}{(P, L, H, x.f = y;\ \ \overline{s_l}^l) \longrightarrow (P, L, H[(oid, f) \mapsto v], \overline{s_l}^l)} \quad \text{SR\_AttrAssign}$$

$$\frac{L(x) = L(y)}{(P, L, H, \mathbf{if}\ (x == y)\ s_1\ \mathbf{else}\ s_2\ \overline{s'_l}^l) \longrightarrow (P, L, H, s_1\ \overline{s'_l}^l)} \quad \text{SR\_IfTrue}$$

$$\frac{L(x) \neq L(y)}{(P, L, H, \mathbf{if}\ (x == y)\ s_1\ \mathbf{else}\ s_2\ \overline{s'_l}^l) \longrightarrow (P, L, H, s_2\ \overline{s'_l}^l)} \quad \text{SR\_IfFalse}$$

$$\frac{\begin{array}{l}\mathbf{find\_type}(P, ctx, cl) = \tau\\ \mathbf{fields}(P, \tau) = \overline{f_k}^k\\ oid \notin \mathrm{dom}(H)\\ H' = H[oid \mapsto (\tau, \overline{f_k \mapsto \mathbf{null}}^k)]\end{array}}{(P, L, H, var = \mathbf{new}_{ctx}\ cl();\ \ \overline{s_l}^l) \longrightarrow (P, L[var \mapsto oid], H', \overline{s_l}^l)} \quad \text{SR\_New}$$

$$\frac{L(x) = \textbf{null}}{(P, L, H, var = x.meth(\overline{y_k}^k);\ \ \overline{s_l}^l) \longrightarrow (P, L, H, \textbf{NPE})} \quad \text{SR\_MethEx}$$

$$\frac{\begin{array}{l} L(x) = oid \\ H(oid) = \tau \\ \textbf{find\_meth\_def}(P, \tau, meth) = (ctx, cl\ meth(\overline{cl_k\ var_k}^k)\{\overline{s'_l}^l\ \textbf{return}\ y;\ \}) \\ \overline{var'_k}^k \perp \mathrm{dom}(L) \\ \textbf{distinct}(\overline{var'_k}^k) \\ x' \notin \mathrm{dom}(L) \\ x' \notin \overline{var'_k}^k \\ \overline{L(y_k) = v_k}^k \\ L' = L[\overline{var'_k \mapsto v_k}^k][x' \mapsto oid] \\ \theta = [\overline{var_k \mapsto var'_k}^k][\textbf{this} \mapsto x'] \\ \overline{\theta \vdash s'_l \rightsquigarrow s''_l}^l \\ \theta(y) = y' \end{array}}{(P, L, H, var = x.meth(\overline{y_k}^k);\ \ \overline{s_l}^l) \longrightarrow (P, L', H, \overline{s''_j}^j\ \ var = y';\ \overline{s_l}^l)} \quad \text{SR\_Meth}$$

3.3 CoJava Extensions

CoJava is based on the above definition of Lightweight Java with a number of extensions:

- Ownership is introduced through the extension of the context mechanism and slight changes to the well-formedness rules. The context mechanism must also be extended to apply to attributes and argument variables for this to be possible.
- Design-by-Contract specification is introduced as predicates describing class invariants and method conditions. This requires a new syntax and semantics for predicate expressions which these specification elements need in order to describe logical properties. Additionally, a *Boolean* type with *True* and *False* subtypes is introduced to capture conditions in terms of true or false.

3.3.1 Ownership

Ownership enforces encapsulation through the type system. By constraining what operations involving owned types are allowed, and the relationship between owned and non-owned types, there exists greater control over how an object allows its clients to access its internal structure. The set of owned objects forms a series of rooted acyclic digraphs when considering the aliasing relationship as edges and objects as nodes. An owned object can only be accessed by objects higher up in the digraph, which are its owners. The assumption of hierarchy is important to structural assumptions about the owned objects and about the meaning of encapsulation.

P	::=	$\overline{cld}$	– Program
C, cl, dcl	::=		– Class names
		`Object`	– Base class Object
	|	dcl	– Class name
ctx	::=		– Context
			– Regular types with no context
	|	**owned**	– Owned types
cld	::=	**class** dcl **extends** cl $\{\overline{fd}$ $\overline{meth\_def}\}$	– Class definition
fd	::=	ctx cl f	– Attribute definition
$meth\_def$	::=	$meth\_sig$ $\{meth\_body\}$	– Method definition
$meth\_sig$	::=	ctx cl $meth(\overline{vd})$	– Method signature
vd	::=	ctx cl var	– Variable definition
$meth\_body$	::=	$\overline{s}$ **return** y;	– Method body
s	::=		– Statements
		$\{\overline{s_k}^k\}$	– Block Statement
	|	$var = x$;	– Assign variable to variable
	|	$var = x.f$;	– Assign attribute to variable
	|	$x.y = y$;	– Assign variable to attribute
	|	**if** $(x == y)$ s **else** s'	– Conditional statement
	|	$var =$ **new**$_{ctx}$ $cl()$;	– Object creation
	|	$var = x.met(\overline{y})$;	– Method call
$TVar, x, y$	::=		– Term variables
		var	– Regular variables
	|	**this**	– Ref. to current object

Figure 3.4: CoJava Abstract Syntax

The restrictions which impose these properties as described in Section 2.2 are enumerated here:

1. Owned values cannot be assigned to non-owned variables and attributes, or vice versa.
2. Methods with owned arguments can be called only when the receiver is *this*.
3. Methods returning owned references can only be accessed through an owned receiver.
4. Owned attributes can only be assigned to when the receiver is *this*.
5. Owned attributes can only be accessed when the receiver is also owned.
6. Within the method of a class dcl, **this** is typed as the owned version of dcl.

The context construct in LJ is intended to be used for extensions to the language. Ownership can be represented with this by introducing a context **owned** which indicates owned references. The syntax of LJ is also modified to include context with attribute, argument and method definitions. Figure 3.4 gives the syntax for CoJava with these added context values.

With the changes to the definition of attributes (fd), arguments (vd), and method signatures ($meth\_sig$), a few of the type information functions and rules must also be altered. The original type functions are defined for LJ in Appendix A.

The functions defining lists of attribute and path names must be amended to incorporate the extra ctx component:

$$\frac{\textbf{class\_fields}(cld) = \overline{ctx_j\ \ cl_j\ \ f_j\ ;}^{\,j} \quad \textbf{fields\_in\_path}(ctxcld_2 \ldots ctxcld_k) = \overline{f} \quad \overline{f}' = \overline{f_j}^{\,j} \frown \overline{f}}{\textbf{fields\_in\_path}((ctx, cld)\ \ ctxcld_2 \ldots ctxcld_k) = \overline{f}'}$$

$$\frac{\textbf{class\_methods}(cld) = \overline{method\_def_l}^{\,l} \quad \overline{method\_def_l = ctx_l\ \ cl_l\ \ method_l(\overline{vd}_l)\{meth\_body_l\}}^{\,l} \quad \textbf{methods\_in\_path}(cld_2 \ldots cld_k) = \overline{meth}' \quad \overline{meth} = \overline{meth_l}^{\,l} \frown \overline{meth}'}{\textbf{methods\_in\_path}(cld\ \ cld_2 \ldots cld_k) = \overline{meth}}$$

ftype_in_fds$(P, ctx, \overline{f}, f)$ finds the type of f in the given list of attributes. The definition must be changed to ensure that the given context *ctx* matches that of the attribute with name f as well as to take into account the added context part of attribute definitions.

$$\frac{\textbf{find\_type}(P, ctx, cl) = \varnothing}{\textbf{ftype\_in\_fds}(P, ctx, ctx'\ \ cl\ f \frown fd_2 \ldots fd_k, f) = \bot}$$

$$\frac{\textbf{find\_type}(P, ctx, cl) = \tau \quad ctx = ctx'}{\textbf{ftype\_in\_fds}(P, ctx, ctx'\ \ cl\ f \frown fd_2 \ldots fd_k, f) = \tau}$$

$$\frac{f \neq f' \quad \textbf{ftype\_in\_fds}(P, ctx, fd_2 \ldots fd_k, f') = \tau^{\bot}_{opt}}{\textbf{ftype\_in\_fds}(P, ctx, ctx'\ \ cl\ f \frown fd_2 \ldots fd_k, f') = \tau^{\bot}_{opt}}$$

The second two deduction rules for **find_meth_def_in_list** must reflect the added *ctx* component:

$$\frac{meth\_def = ctx\ \ cl\ \ meth(\overline{vd})\{meth\_body\}}{\textbf{find\_meth\_def\_in\_list}(meth\_def\ \ meth\_def_2 \ldots meth\_def_k, meth) = meth\_def}$$

$$\frac{meth\_def = ctx\ \ cl\ \ meth'(\overline{vd})\{meth\_body\} \quad meth \neq meth' \quad \textbf{find\_meth\_def\_in\_list}(meth\_def_2 \ldots meth\_def_k, meth) = meth\_def_{opt}}{\textbf{find\_meth\_def\_in\_list}(meth\_def\ \ meth\_def_2 \ldots meth\_def_k, meth) = meth\_def_{opt}}$$

Lastly, **mtype** is changed to reflect the changes to method return and argument types:

$$\frac{\begin{array}{l}\textbf{find\_meth\_def}(P, \tau, meth) = (ctx', meth\_def) \\ meth\_def = ctx\ \ cl\ \ meth(\overline{ctx_k\ \ cl_k\ \ var_k}^{k})\{meth\_body\} \\ \textbf{find\_type}(P, ctx, cl) = \tau' \\ \overline{\textbf{find\_type}(P, ctx_k, cl_k) = \tau_k}^{k} \\ \pi = \overline{\tau_k}^{k} \rightarrow \tau'\end{array}}{\textbf{mtype}(P, \tau, meth) = \pi}$$

The definition of subtyping need not be changed since a critical property of contexts is preserved. A type $ctx.cld$ is a subtype of $ctx'.cld'$ only if $ctx = ctx'$ and the class cld inherits from cld' (or are the same class). This implies that an owned type is not a subtype or a regular type, thus for example $\textbf{owned}.cld \prec cld'$ is never true. This ensures that owned objects are always only aliased through owned references since the rules for assignment prevent an owned value from being assigned to a non-owned variable.

For example, WF_AssignVar defining the well-formedness of the statement '$var = x;$ ' requires that the type of x be a subtype of that of var. If var is declared as owned but x is not, then the statement is not well-formed and a non-owned variable cannot refer to an owned object. This same restriction prevents the other assignment statements from creating owned and non-owned references to the same object, thus satisfying the first restriction given above.

Modifications must be made to the well-formedness rules to prevent owned attributes from being accessed, and methods with owned return type from being called, when the receiver is non-owned. The rules must also prevent owned attributes of objects from being assigned to, or methods with owned arguments from being called, when the receiver is not **this**.

Two new predicates are first introduced which state whether a given type is owned or non-owned:

$$\frac{\begin{array}{l}\tau = (ctx, cld) \\ ctx = \textbf{owned}\end{array}}{\textbf{owned}(\tau)} \qquad \frac{\begin{array}{l}\tau = (ctx, cld) \\ ctx \neq \textbf{owned}\end{array}}{\textbf{reg}(\tau)}$$

The following rules replace those previously defined in this chapter to take into the account the stated restrictions necessary to enforce ownership-based encapsulation:

$$\frac{\begin{array}{l}\Gamma(x) = \tau \\ \textbf{ftype}(P, \tau, f) = \tau' \\ P \vdash \tau' \prec \Gamma(var) \\ \textbf{owned}(\tau') \Rightarrow \textbf{owned}(\tau)\end{array}}{P, \Gamma \vdash var = x.f;} \qquad \text{WF\_AssignAttr}$$

The added constraint $\textbf{owned}(\tau') \Rightarrow \textbf{owned}(\tau)$ requires that the type of x must be owned if that of f is owned, satisfying the fifth restriction. If x were not owned but f was, then x's internal representation would be exposed once the statement was executed.

$$\frac{\begin{array}{l}\Gamma(x) = \tau \\ \textbf{ftype}(P, \tau, f) = \tau' \\ P \vdash \Gamma(y) \prec \tau' \\ x \neq \textbf{this} \Rightarrow \textbf{reg}(\tau')\end{array}}{P, \Gamma \vdash x.f = y;} \qquad \text{WF\_AttrAssign}$$

The added constraint $x \neq \textbf{this} \Rightarrow \textbf{reg}(\tau')$ prevents assigning to f if it is owned and x is not **this**, which satisfies the fourth restriction. Disallowing such operations prevents non-owning clients of x from creating cycles in the ownership hierarchy.

$$\frac{\begin{array}{l}\overline{y} = \overline{y_k}^k \\ \Gamma(x) = \tau \\ \textbf{mtype}(P, \tau, meth) = \overline{\tau_k}^k \to \tau' \\ \overline{P \vdash \Gamma(y_k) \prec \tau_k}^k \\ P \vdash \tau' \prec \Gamma(var) \\ \overline{x \neq \textbf{this} \Rightarrow \textbf{reg}(\tau_k)}^k \\ \textbf{owned}(\tau') \Rightarrow \textbf{owned}(\tau)\end{array}}{P, \Gamma \vdash var = x.meth(\overline{y});} \quad \text{WF\_MethCall}$$

The same constraint introduced in WF_AssignAttr above is present for method calls, requiring that the receiver x have an owned type if the method returns an owned value, and so satisfies the third restriction. In an identical way this prevents non-owning clients from accessing objects owned by x. Additionally, if the receiver is not the **this** value then no arguments may have owned types, thus satisfying the second restriction. This is stated as $\overline{x \neq \textbf{this} \Rightarrow \textbf{reg}(\tau_k)}^k$, meaning that each argument type τ_k must be regular type if x is not **this**.

Next the well-formedness definition for methods themselves must be changed to reflect the requirement that **this** must always have an owned type and so satisfy the sixth restriction:

$$\frac{\begin{array}{l}\textbf{distinct}(\overline{var_k}^k) \\ \overline{\textbf{find\_type}(P, ctx_k, cl_k) = \tau_k}^k \\ \Gamma = [\overline{var_k \mapsto \tau_k}^k][\textbf{this} \mapsto \textbf{owned}.dcl] \\ \overline{P, \Gamma \vdash s_l}^l \\ \textbf{find\_type}(P, ctx, cl) = \tau \\ P \vdash \Gamma(y) \prec \tau\end{array}}{P \vdash_{ctx'.dcl} ctx\ cl\ meth(\overline{ctx_k\ cl_k\ var_k}^k)\{\overline{s_l}^l\ \textbf{return}\ y;\ \}} \quad \text{WF\_Meth}$$

The definition of Γ is altered such that it maps **this** to the type **owned**, *dcl*, which is the owned type defined by the class *dcl* in which this method is considered to be well-formed. This redefinition takes into account the added *ctx* component to the return type and argument types. The judgment on well-formedness is also in terms of the class *dcl* which the method is defined for, but in a context ctx' different from those used in the rule.

3.3.2 DbC Specification

Specifying Java programs with JML [75, 76] or other Design-by-Contract [12, 16, 19, 61, 89] approaches involves defining contracts for classes and their members. Besides invariant or condition predicates, more complex properties such as pure methods are defined with added keywords. For example, the *Counter* Java class given in Figure 3.3.2 is annotated with JML to define its class invariant and the contracts on its methods.

This example demonstrates the definition of a class invariant, pre- and postconditions for methods, and modifiers such as */*@ pure @*/* which define properties not described by contracts.

```
class Counter {
  protected /*@ spec_public @*/  int value,max;

  //@ invariant  value >= 0 && value <= max;

  //@ requires max >= 0;
  //@ ensures value == 0 && this.max == max;
  public Counter(int max)  { value = 0; this.max = max; }

  //@ requires  value < max;
  //@ ensures value == \old(value + 1);
  public void inc() { value = value + 1; }

  //@ requires (value + n) >= 0;
  //@ requires (value + n) <= max;
  //@ ensures value == \old(value + n);
  public void add(int n){ value = value + n;}

  //@ ensures \result == value;
  public /*@ pure @*/ int get()  { return value; }
}
```

Figure 3.5: Counter Java+JML Example

Invariants and conditions may be defined as predicates which must be satisfied at certain stages of execution for a class to be considered correct, but purity and other properties imply more complex concepts. The annotation */*@ spec_public @*/* allows a non-public member to be mentioned in a public specification; since access keywords are not present in CoJava, this will not be necessary when specifying CoJava.

CoJava extends LJ with predicates to describe these contracts as well as a function-based approach to declaring specifications for types and methods. Specifications in CoJava will be composed of three types of elements:

- **Invariants**: These are predicates describing a property which must be true for an object when it is accessible by its clients. This property need not always hold, thus isn't strictly unvarying, but must describe the state of an object when clients are allowed to interact with it. To external objects, the property is unvarying even if the object's methods temporarily produce a state for which it does not hold. An object must be able to satisfy its invariant when it is first created.

- **Method Contracts**: A precondition for a method is a predicate stating what properties the receiver object and arguments must satisfy before the call proceeds. The caller is responsible for ensuring that the arguments and receiver's state satisfies this predicate.

  A postcondition for a method is a predicate stating what effects the call will have on the state of the receiver and any other reachable object, as well as properties about the value the method returns. Postconditions may refer to relationships between the states before and after the call was executed. The *old*() expressions evaluate to the value an attribute or argument had before the call began, thus the pre- and post-state of these values can be compared in predicates. The receiver is responsible for ensuring the postcondition is satisfied.

- **pure**: This is an annotation applied to a method definition indicating that the operations of the methods will not change the state of existing objects. A pure method may create new objects and mutate their state, but objects existing before the call began will not be changed.

This requires new syntax in Figure 3.6 for predicate expressions, since LJ lacks expressions entirely. Predicates in CoJava may include method calls and can be evaluated, thus they represent a form of computation which always results in a truth value. Instead of embedding these predicates in the text of CoJava classes where appropriate, a set of functions relates specification predicates to the appropriate types and methods.

pr	::=	– Predicates
	pr && pr	– Conjunction
	pr || pr	– Disjunction
	pr ==> pr	– Implication
	$!pr$	– Negation
	$forall(ctx\ cl\ var;\ pr;\ pr)$	– Universal Quantification
	$exists(ctx\ cl\ var;\ pr;\ pr)$	– Existential Quantification
	$\mathbf{this}.meth(\overline{prv})$	– Method of current object
	$\mathbf{this}.x.meth(\overline{prv})$	– Method of attribute x
	$w.meth(\overline{prv})$	– Method of argument/quantifier variable w
prv	::=	– Predicate Values
	$\mathbf{this}.z$	– Attribute z of current object
	w	– Argument variable w
	$old(\mathbf{this}.z)$	– Attribute z of current object in pre-state
	$old(w)$	– Argument variable w in pre-state
	result	– Result value

Figure 3.6: CoJava Predicate Syntax

JML associates specifications with types by embedding contracts within the code in special comments. Eiffel and others have explicit language constructs to express the specification. In either language the specification is a myriad of elements defined across large areas of the program text. The CoJava approach instead relates types and methods to predicates through specification functions, thus defining the specification separately from the program text in a concise and coherent manner:

- $\mathbf{inv}(\tau) = pr$ relates a type τ to the predicate defining its invariant.
- $\mathbf{pre}(\tau, meth) = pr$ and $\mathbf{post}(ctx, cld, meth) = pr$ relates a type τ and the named method $meth$ defined by that type to that method's pre- and postconditions, respectively.
- $\mathbf{pure}(\tau) = \overline{meth}$ equals the list of methods in τ which are defined as pure, that is having no side-effects.

These functions decouple the text of a CoJava program from its specification. The *Counter* example above defines its invariant in JML tags within its body, whereas the equivalent specification would be recorded with **inv** by stating the application of *Counter* to the function yields the correct predicate (given here in Java): $\mathbf{inv}(Counter) = \mathbf{this}.value >= 0 \;\&\&\; \mathbf{this}.value <= \mathbf{this}.max$.

It should be noted at this point that specifications will usually be given in Java rather than CoJava predicates owing to the brevity afforded by Java operators. The invariant for *Counter*, if stated in CoJava, would be something like **this**.*isPositive*(**this**.*value*) && **this**.*ltEq*(**this**.*value*, **this**.*max*), which has an obvious equivalence given the presence of helper predicate methods but is much less readable. Since primitive values are objects, their nullity would normally need to be checked before any predicate about them can be asserted. When stating predicates in Java, guard implication predicates of the form $X! = \textbf{null} \Rightarrow P$ which check the null state of X before asserting P, are considered to be implicit.

The full specification for *Counter* is given here in Java:

inv(*ctx*.*Counter*) = **this**.*value* >= 0 && **this**.*value* <= **this**.*max*
post(*ctx*.*Counter*, *init*) = **this**.*max* == *max* && **this**.*value* == 0
pre(*ctx*.*Counter*, *inc*) = **this**.*value* < **this**.*max*
post(*ctx*.*Counter*, *inc*) = **this**.*value* == **this**.*value* + 1
pre(*ctx*.*Counter*, *add*) = (**this**.*value* + *n*) >= 0 && (**this**.*value* + *n*) <= **this**.*max*
post(*ctx*.*Counter*, *add*) = **this**.*value* == *old*(**this**.*value* + *n*)
post(*ctx*.*Counter*, *get*) = *result* == **this**.*value*
pure(*ctx*.*Counter*) = [*get*]

Predicate Well-Formedness

Predicates are expressions with boolean values. The type *Boolean* is introduced with two subtypes, *True* and *False*:

Type, τ ::=		– Type
	ctx.`Object`	– Supertype of all types
	ctx.`Boolean`	– Boolean type
	ctx.`True`	– Type representing true
	ctx.`False`	– Type representing false
	ctx.*dcl*	– Class identifier

A new rule defines that `True` and `False` are subtypes of `Boolean`:

$P \vdash ctx.\texttt{Boolean} \prec ctx.\texttt{Object}$
$P \vdash ctx.\texttt{Boolean} \prec ctx.\texttt{Boolean}$
$P \vdash ctx.\texttt{True} \prec ctx.\texttt{Boolean}$
$P \vdash ctx.\texttt{False} \prec ctx.\texttt{Boolean}$

An invariant predicate is well-formed in the context of a class, since it refers to the members of that class. A contract predicate is well-formed in the context of a method defined for a class, since it refers to that method's arguments as well as the class's members. Well-formedness must thus always be stated in terms of a class τ and method *meth*, denoted by the judgment $\vdash_{\tau,meth}$. An invariant predicate is not stated in terms of a method, so $\vdash_{\tau,\varnothing}$ is a valid judgment for invariant validity. Given this, the following rules define well-formedness for predicates:

$$\frac{P, \Gamma \vdash_{\tau,meth} pr \qquad P, \Gamma \vdash_{\tau,meth} pr'}{P, \Gamma \vdash_{\tau,meth} pr \;\&\&\; pr'} \quad \text{WF\_And}$$

$$\frac{P, \Gamma \vdash_{\tau, meth} pr \qquad P, \Gamma \vdash_{\tau, meth} pr'}{P, \Gamma \vdash_{\tau, meth} pr \ || \ pr'} \quad \text{WF\_Or}$$

$$\frac{P, \Gamma \vdash_{\tau, meth} pr \qquad P, \Gamma \vdash_{\tau, meth} pr'}{P, \Gamma \vdash_{\tau, meth} pr \ ==> \ pr'} \quad \text{WF\_Impl}$$

$$\frac{P, \Gamma \vdash_{\tau, meth} pr}{P, \Gamma \vdash_{\tau, meth} !pr} \quad \text{WF\_Neg}$$

$$\frac{\begin{array}{l}\textbf{find\_type}(P, ctx, cl) = \tau \\ \Gamma = \Gamma'[var \mapsto \tau] \\ P, \Gamma \vdash_{\tau, meth} pr \\ P, \Gamma \vdash_{\tau, meth} pr'\end{array}}{P, \Gamma' \vdash_{\tau, meth} forall(ctx \ cl \ var; \ pr; \ pr')} \quad \text{WF\_Forall}$$

$$\frac{\begin{array}{l}\textbf{find\_type}(P, ctx, cl) = \tau \\ \Gamma = \Gamma'[var \mapsto \tau] \\ P, \Gamma \vdash_{\tau, meth} pr \\ P, \Gamma \vdash_{\tau, meth} pr'\end{array}}{P, \Gamma' \vdash_{\tau, meth} exists(ctx \ cl \ var; \ pr; \ pr')} \quad \text{WF\_Exists}$$

Only methods which do not change the state of the program can be called in predicates, that is they must be pure. The predicate $\textbf{pure}(ctx.cld)$ gives the list of method names for methods in the class $ctx.cld$ which meet this criteria. Method calls in predicates use this predicate to enforce the purity requirement:

$$\frac{\begin{array}{l}meth' \in \textbf{pure}(ctx, cld) \\ \textbf{mtype}(P, \tau, meth') = \overline{\tau_j}^{\,j} \rightarrow ctx'.\texttt{Boolean} \\ \overline{P, \Gamma \vdash_{\tau, meth, \tau_k} prv_k}^{\,k} \\ P \vdash \overline{\tau_k}^{\,k} \prec \overline{\tau_j}^{\,j}\end{array}}{P, \Gamma \vdash_{\tau, meth} \textbf{this}.meth'(\overline{prv_k}^{\,k})} \quad \text{WF\_ThisMeth}$$

Predicates of the form $w.meth(\overline{y})$ may represent calls to an argument w or a quantifier variable w. Two separate definitions for well-formedness must be provided:

$$\frac{\begin{array}{l}\textbf{find\_meth\_def}(P, \tau, meth) = (ctx', meth\_def) \\ meth\_def = cl \ meth(\overline{ctx_l \ cl_l \ var_l}^{\,l})\{meth\_body\} \\ ctx_j \ cl_j \ w \in \overline{ctx_l \ cl_l \ var_l}^{\,l} \\ meth' \in \textbf{pure}(ctx_j, cl_j) \\ \textbf{mtype}(P, ctx_j.cl_j, meth') = \overline{\tau_j}^{\,j} \rightarrow ctx'.\texttt{Boolean} \\ \overline{P, \Gamma \vdash_{\tau, meth, \tau_k} prv_k}^{\,k} \\ P \vdash \overline{\tau_k}^{\,k} \prec \overline{\tau_j}^{\,j}\end{array}}{P, \Gamma \vdash_{\tau, meth} w.meth'(\overline{prv_k}^{\,k})} \quad \text{WF\_ArgMeth}$$

$$\frac{\begin{array}{l}\Gamma(w) = \tau' \\ meth' \in \mathbf{pure}(\tau') \\ \mathbf{mtype}(P, \tau', meth') = \overline{\tau_j}^{\,j} \to ctx'.\texttt{Boolean} \\ \overline{P, \Gamma \vdash_{\tau, meth', \tau_k} prv_k}^{\,k} \\ P \vdash \overline{\tau_k}^{\,k} \prec \overline{\tau_j}^{\,j}\end{array}}{P, \Gamma \vdash_{\tau, meth} w.meth'(\overline{prv_k}^{\,k})} \quad \text{WF\_VarMeth}$$

Predicates of the form $\mathbf{this}.x.meth'(\overline{prv_k}^{\,k})$ represent a call to a method of an attribute x of the current object:

$$\frac{\begin{array}{l}\mathbf{ftype}(P, \tau, x) = \tau' \\ meth' \in \mathbf{pure}(\tau') \\ \mathbf{mtype}(P, \tau', meth') = \overline{\tau_j}^{\,j} \to ctx'.\texttt{Boolean} \\ \overline{P, \Gamma \vdash_{\tau, meth', \tau_k} prv_k}^{\,k} \\ P \vdash \overline{\tau_k}^{\,k} \prec \overline{\tau_j}^{\,j}\end{array}}{P, \Gamma \vdash_{\tau, meth} \mathbf{this}.x.meth'(\overline{prv_k}^{\,k})} \quad \text{WF\_AttrMeth}$$

Judgments about predicate values are also in terms of their type τ':

$$\frac{\mathbf{ftype}(P, \tau, z) = \tau'}{P, \Gamma \vdash_{\tau, meth, \tau'} \mathbf{this}.z} \quad \text{WF\_Attr}$$

$$\frac{\begin{array}{l}\mathbf{find\_meth\_def}(P, \tau, meth) = (ctx', meth\_def) \\ meth\_def = ctx\ \ cl\ \ meth(\overline{ctx_k\ \ cl_k\ \ var_k}^{\,k})\{meth\_body\} \\ ctx_j\ \ cl_j\ \ w \in \overline{ctx_k\ \ cl_k\ \ var_k}^{\,k} \\ \mathbf{find\_type}(P, ctx_j, cl_j) = \tau'\end{array}}{P, \Gamma \vdash_{\tau, meth, \tau'} w} \quad \text{WF\_Arg}$$

$$\frac{P, \Gamma \vdash_{\tau, meth, \tau'} \mathbf{this}.z}{P, \Gamma \vdash_{\tau, meth, \tau'} old(\mathbf{this}.z)} \quad \text{WF\_OldAttr}$$

$$\frac{P, \Gamma \vdash_{\tau, meth, \tau'} w}{P, \Gamma \vdash_{\tau, meth, \tau'} old(w)} \quad \text{WF\_OldArg}$$

$$\frac{\mathbf{result} \mapsto \tau' \in \Gamma}{P, \Gamma \vdash_{\tau, meth, \tau'} \mathbf{result}} \quad \text{WF\_Result}$$

Predicate Evaluation

Evaluating a predicate to determine it's truth value does not change the state of a program. Predicates are evaluated in terms of two heaps, one representing the state before a method is executed and one after, and a variable store representing the arguments of the method associated with contracts. Both of these heaps are the same when evaluating invariants and preconditions, but postconditions require two distinct heaps since they describe the relationship between the states. Predicates are normally evaluated in terms of the second heap, except those within **old** predicates which are evaluated in terms of the first.

The judgment $[\![pr]\!]_{H,H',L}$ states that, given the heaps H and H' and the variable store L, the predicate pr evaluates to a value with type *True*, meaning that the predicate describes a property which is true for the given states. Truth is stated in terms of any value whose type is *True* since there exists in CoJava no globally defined values, thus there is no *true* value like in Java which a true predicate would evaluate to. A predicate can evaluate to an instance of *False* or any other instance of *Boolean* or its subtypes, but such a predicate would be considered to be false in the context of the given state.

For values, $[\![prv]\!]_{H,H',L} = v$ states that the attribute or argument prv evaluates to the value v. Note that the rule E_ArgVarMeth is used for evaluating method calls with quantifier variables or argument variables as receivers.

The following rules define the conditions required for a predicate to be evaluated to true for given states:

$$\frac{[\![pr]\!]_{H,H',L} \quad [\![pr']\!]_{H,H',L}}{[\![pr \;\&\&\; pr']\!]_{H,H',L}} \qquad \text{E\_And}$$

$$\frac{[\![pr]\!]_{H,H',L} \vee [\![pr']\!]_{H,H',L}}{[\![pr \;\; || \;\; pr']\!]_{H,H',L}} \qquad \text{E\_Or}$$

$$\frac{[\![pr]\!]_{H,H',L} \Rightarrow [\![pr']\!]_{H,H',L}}{[\![pr \;\; ==> \;\; pr']\!]_{H,H',L}} \qquad \text{E\_Impl}$$

$$\frac{\neg\, [\![pr]\!]_{H,H',L}}{[\![!pr]\!]_{H,H',L}} \qquad \text{E\_Neg}$$

$$\frac{\forall\, oid : \mathrm{dom}\, H \cup \mathrm{dom}\, H' \mid [\![pr]\!]_{H,H',L[var \mapsto oid]} \bullet [\![pr']\!]_{H,H',L[var \mapsto oid]}}{[\![forall(ctx\ cl\ var;\ pr;\ pr')]\!]_{H,H',L}} \qquad \text{E\_Forall}$$

$$\frac{\exists\, oid : \mathrm{dom}\, H \cup \mathrm{dom}\, H' \mid [\![pr]\!]_{H,H',L[var \mapsto oid]} \bullet [\![pr']\!]_{H,H',L[var \mapsto oid]}}{[\![exists(ctx\ cl\ var;\ pr;\ pr')]\!]_{H,H',L}} \qquad \text{E\_Exists}$$

$$\frac{\begin{array}{l} L(\mathbf{this}) = oid \\ x \notin \operatorname{dom} L \\ var \notin \operatorname{dom} L \\ \overline{y_k \notin \operatorname{dom} L}^{k} \\ \overline{[\![prv_k]\!]_{H,H',L} = v_k}^{k} \\ L' = L[x \mapsto oid][var \mapsto \mathbf{null}][\overline{y_k \mapsto v_k}^{k}] \\ (P, L', H', var = x.meth(\overline{y_k}^{k});\ \overline{s_l}^{l}) \longrightarrow (P, L'', H'', \overline{s_l}^{l}) \\ L''(var) = oid' \\ H''(oid') = \mathbf{True} \end{array}}{[\![\mathbf{this}.meth(\overline{prv_k}^{k})]\!]_{H,H',L}} \quad \text{E\_ThisMeth}$$

$$\frac{\begin{array}{l} L(w) = oid \\ x \notin \operatorname{dom} L \\ var \notin \operatorname{dom} L \\ \overline{y_k \notin \operatorname{dom} L}^{k} \\ \overline{[\![prv_k]\!]_{H,H',L} = v_k}^{k} \\ L' = L[x \mapsto oid][var \mapsto \mathbf{null}][\overline{y_k \mapsto v_k}^{k}] \\ (P, L', H', var = x.meth(\overline{y_k}^{k});\ \overline{s_l}^{l}) \longrightarrow (P, L'', H'', \overline{s_l}^{l}) \\ L''(var) = oid' \\ H''(oid') = \mathbf{True} \end{array}}{[\![w.meth(\overline{prv})]\!]_{H,H',L}} \quad \text{E\_ArgVarMeth}$$

$$\frac{\begin{array}{l} L(\mathbf{this}) = oid' \\ H(oid', w) = oid \\ x \notin \operatorname{dom} L \\ var \notin \operatorname{dom} L \\ \overline{y_k \notin \operatorname{dom} L}^{k} \\ \overline{[\![prv_k]\!]_{H,H',L} = v_k}^{k} \\ L' = L[x \mapsto oid][var \mapsto \mathbf{null}][\overline{y_k \mapsto v_k}^{k}] \\ (P, L', H', var = x.meth(\overline{y_k}^{k});\ \overline{s_l}^{l}) \longrightarrow (P, L'', H'', \overline{s_l}^{l}) \\ L''(var) = oid'' \\ H''(oid'') = \mathbf{True} \end{array}}{[\![\mathbf{this}.x.meth(\overline{prv})]\!]_{H,H',L}} \quad \text{E\_AttrMeth}$$

$$\frac{\begin{array}{l} L(\mathbf{this}) = oid \\ H'(oid, x) = v \end{array}}{[\![\mathbf{this}.x]\!]_{H,H',L} = v} \quad \text{E\_Attr}$$

$$\frac{L(w) = v}{[\![w]\!]_{H,H',L} = v} \quad \text{E\_Arg}$$

$$\frac{[\![\mathbf{this}.x]\!]_{H,H,L} = v}{[\![old(\mathbf{this}.x)]\!]_{H,H',L} = v} \qquad \text{E\_OldAttr}$$

$$\frac{[\![w]\!]_{H,H,L} = v}{[\![old(w)]\!]_{H,H',L} = v} \qquad \text{E\_OldArg}$$

$$\frac{L(\mathbf{result}) = v}{[\![\mathbf{result}]\!]_{H,H',L} = v} \qquad \text{E\_Result}$$

Pure Methods

Predicates are intended to represent a property about a program in terms of that program's constructs (attributes and methods). Through expressions, values, and method calls they evaluate to a truth value representing a logical statement about the program. The objective is to state a logical property about a program in terms of its own constructs, rather than employing a separate language with which to make assertions. They therefore do not represent computation, so their evaluation does not represent a transition between states.

If a method which does represent a state transition were part of a predicate, then the meaning of the predicate becomes more than a logical assertion. It now involves ambiguity as to whether the stated property holds for the initial or final state. Checking contracts at runtime as a debugging technique [18, 29] would also introduce undesirable side-effects.

The evaluation of a predicate should thus be a pure operation having no observable side-effects. If all methods used in a predicate do not represent observable state transitions, then the predicate as a whole is pure since no other component may affect state.

Methods satisfying this criteria are termed pure and have no ostensible side-effects on a program, although they may change state. This restriction prevents a method from modifying any object existing when the call begins, but allows it to create new objects and assign to argument variables. Since no pre-existing object observes change, the method appears to do nothing but represent a value or a property, although a new object may be created and returned. Pure methods thus can be employed as predicates and checked at runtime without affecting the semantics of the program.

Given a method *meth* defined for some class τ, for all invocations of *meth* which transition a program's state from (L, H) to (L', H'), *meth* is pure if L is a subset of L' and H is a subset of H'. This implies that all state in H must be present in the final heap H', that is all objects present in H must be present unmodified in H'. Similarly, this implies that all variable mappings in L must be present in L' and store the same value. Purity is formalized as the following definition which any method mentioned by **pure** must satisfy:

$$\frac{\begin{array}{l} L(x) = \tau \\ P, \Gamma \vdash var = x.meth(\overline{y_k}^k); \\ \forall\, H, H', L, L' \mid (P, L, H, var = x.meth(\overline{y_k}^k);\ \overline{s_l}^l) \longrightarrow (P, L', H', \overline{s_l}^l) \bullet H \subseteq H' \wedge L \subseteq L' \end{array}}{P \vdash meth \in \mathbf{pure}(\tau)}$$

Inheritance and Substitutability

The Substitutability Principle [80, 81] describes the formal relationship between super- and subtypes. A subtype is substitutable with its supertype if it meets a certain set of criteria. Instances of substitutable types can be used to replace instances of their supertypes, where objects of that type were expected in a program, without affecting the ostensible behaviour of that program.

This is a necessary property that ensures polymorphism does not affect the behaviour of objects at runtime vis-a-vis their specifications. Given an object type C and it's subtype D (ie. $D \prec C$), the following must be true for D to be substitutable with C:

- The invariant of D must imply the invariant of C: $I_D \Rightarrow I_C$.
- For any method $meth_D$ in D that overrides a method $meth_C$ inherited from C:
  - The precondition of $meth_C$ must imply the precondition of $meth_D$: $P_{meth_C} \Rightarrow P_{meth_D}$.
  - The postcondition of $meth_D$ must imply the postcondition of $meth_C$: $Q_{meth_D} \Rightarrow Q_{meth_C}$.

Since the invariant of D is generally understood to include all inherited invariants, the first requirement is implicitly met. The relationship between $meth_D$ and $meth_C$ is more complex. It requires that the specification for $meth_D$ include a precondition that is no stronger than that in $meth_C$, and must include a postcondition that is no weaker than that in $meth_C$. This means that $meth_D$ may require more liberal constraints on arguments and members, and may have side-effects in addition to those specified for $meth_C$.

Why this is important in object-oriented systems is due to the nature of polymorphic types. If an instance of D were referenced through a variable of type C, then only the specification for C would be known. Indeed, it should be possible to compile a correct program knowing only about C, to which the type D is only subsequently introduced without compromising correctness. As far as this program would know, any object accessible through a reference of type C is in fact an instance of this type, therefore it would expect the object to behave as the specification of C indicates. Were D not substitutable with C, then the behaviour of its methods would not conform to the known specification.

For example, a non-substitutable subtype of *Counter* can override methods with implementations whose behaviour is entirely unexpected. If such a subtype called *BadCounter* defined a new *inc()* method which performed no operation at all, that is having a postcondition $this.value == old(this.value)$, then any client relying on the counter being incremented may not behave correctly.

The definition for *BadCounter* is given here in Java with its specification in JML:

```
class BadCounter extends Counter {
  ...
  //@ ensures value == \old(value);
  public void inc() { }
}
```

The *inc()* override method performs no operation whatsoever, and so the state of *value* remains unchanged. A client of a *BadCounter* instance which references it through a variable of type *Counter* would expect *value* to be incremented, and will malfunction if correctness is dependent on this behaviour:

```
ensures c.get() == \old(c.get())+1;
public void incCounter(int n, Counter c) { c.inc(); }
```

```
...
Counter counter = new BadCounter(10);
incCounter(5,counter);
```

The postcondition for the *incCounter()* call will not be satisfied if an instance of *BadCounter* is passed as an argument, but would be satisfied given an instance of *Counter* or any substitutable subtype thereof. Substitutability would require that the *inc()* method perform at least the same operations as those of the method it overrides. Since there is no elegant or desirable way of distinguishing in a method's specification what the actual type of an argument is, substitutability is necessary when contracts rely on the behaviour of overridden methods.

In essence substitutability ensures objects always behave at least as their specifications indicate. It takes into account the fact that the known specification for an object may be that defined for a different type than the object's actual runtime type. The definition of a substitutable method is formalized as the following, where the judgment $P, \mathbf{pre}, \mathbf{post} \vdash_{meth} \tau \prec \tau'$ states that the method named *meth* in τ is a substitutable override for that in τ'. It must be shown first that for all initial states (H, L) and final states (H', L') for any call to an overridden method *meth*, the relationship between pre and postconditions is maintained:

$$\frac{\begin{array}{l}\tau \prec \tau' \\ \mathbf{find\_meth\_def}(P, \tau, meth) = (ctx, meth\_def) \\ \mathbf{find\_meth\_def}(P, \tau', meth) = (ctx, meth\_def') \\ meth\_def = ctx'\ cl\ meth(\overline{ctx_k\ cl_k\ var_k}^{k})\{meth\_body\} \\ \forall H, H', L, L' \mid (P, L, H, var = x.meth(\overline{y});\ \ \overline{s_l}^{l}) \longrightarrow (P, L', H', \overline{s_l}^{l}) \wedge \\ \qquad L'' = L[\mathbf{this} \mapsto L(x)][\overline{var_k \mapsto L(y_k)}^{k}] \wedge L''' = L'[\mathbf{this} \mapsto L(x)][\overline{var_k \mapsto L(y_k)}^{k}] \bullet \\ \quad (\llbracket \mathbf{pre}(\tau', meth) \rrbracket_{H,H,L''} \Rightarrow \llbracket \mathbf{pre}(\tau, meth) \rrbracket_{H,H,L''}) \wedge \\ \quad (\llbracket \mathbf{post}(\tau, meth) \rrbracket_{H,H',L'''} \Rightarrow \llbracket \mathbf{post}(\tau', meth) \rrbracket_{H,H',L'''})\end{array}}{P, \mathbf{pre}, \mathbf{post} \vdash_{meth} \tau \prec \tau'}$$

The definition of a substitutable type can be formalized as the following, where $P, \mathbf{inv}, \mathbf{pre}, \mathbf{post} \vdash \tau \prec \tau'$ is the judgment that τ is a substitutable subtype for τ' given the specification as defined by **inv**, **pre**, and **post**. It must be shown first that for all states (H, L), the correct relationship between invariants exist, and all overriding methods are substitutable:

$$\frac{\begin{array}{l}\tau \prec \tau' \\ \forall H, L \bullet \llbracket \mathbf{inv}(\tau) \rrbracket_{H,H,L} \Rightarrow \llbracket \mathbf{inv}(\tau') \rrbracket_{H,H,L} \\ \mathbf{methods}(P, \tau) = \overline{meth} \\ \mathbf{methods}(P, \tau') = \overline{meth'} \\ \forall m : (\overline{meth} \cap \overline{meth'}) \bullet P, \mathbf{pre}, \mathbf{post} \vdash_{m} \tau \prec \tau'\end{array}}{P, \mathbf{inv}, \mathbf{pre}, \mathbf{post} \vdash \tau \prec \tau'}$$

Correctness

Invariants and method contracts are used to describe properties about a program, in particular what states are valid for an object and what operations are correct for a method. With these contracts a definition of correctness can be formulated which is intended to ensure the correct operation of a program if these contracts are respected.

The DbC approach bases correctness on the idea that invariants define what a correct program state is, and contracts determine what correct state transitions are. If the contracts are always respected for all operations, then the program can only transition from one correct state to another. In particular an object will satisfy its invariant at any point it may be accessible to its clients, which allows its own methods to temporarily violate the condition before re-establishing it.

The basis for defining method behaviour in terms of requirements and effects is derived from the Hoare triple [57] axiomatic approach to semantics, which has the following basic form:

$$\{P\}\ R\ \{Q\}$$

This defines some operation R which requires that a predicate P to hold before it can be performed. Once it has been performed then the predicate Q will hold. Both P and Q describe the state of the program in which R is executed, P being the precondition describing the initial state, and Q being the postcondition describing the final state.

This notion was first applied to the specification of the Eiffel language [88] and described in [87] and later [89], where R is a method call with P as its precondition and Q its postcondition. Hoare triples are used in this way since they succinctly describe the transition from one state to another in terms of the condition predicates, and allow properties known about triples to be applied when reasoning about method calls.

If one operation R_1 has a postcondition satisfying the precondition of R_2, then they may be composed sequentially where the final state will be a correct postcondition for R_2:

$$\frac{\{P\}\ R_1\ \{Q_1\} \qquad \{Q_1\}\ R_2\ \{Q\}}{\{P\}\ R_1\ ;\ R_2\ \{Q\}}$$

The objective is to show that a program, consisting of a series of operations $R_1 \ldots R_n$, may be composed in this way starting from a correct initial state and terminating in a correct final state:

$$\frac{\{P\}\ R_1\ \{Q_1\} \qquad \forall\, i : 2..(n-1) \bullet \{Q_{i-1}\}\ R_i\ \{Q_i\} \qquad \{Q_{n-1}\}\ R_n\ \{Q\}}{\{P\}\ R_1\ ;\ \ldots\ ;\ R_n\ \{Q\}}$$

Demonstrating this correctness for every instance of $R_1 \ldots R_n$ as a whole is quite difficult. Demonstrating instead that each individual R is executed correctly allows the correctness proof to be broken down into smaller, modular proof problems. In terms of a CoJava program and its specification, this means demonstrating that the specification is respected, in that invariants hold at the correct times and method calls abide by their associated contracts.

When R is a method call, the pre and postconditions are composed of the receiver's invariant I as well as the stated contracts for that method. Given a method R, invariant I, and conditions P and Q, the Hoare triple for a method call in full defined as such:

$$\{I \wedge P\}\ R\ \{I \wedge Q\}$$

An invariant is more than simply a convenient shorthand for inserting the predicate I into every condition since it is inherited. If R is defined in a subtype that inherits some of the predicates that compose I, then these also must be respected in addition to those parts of the invariant defined in

the same object. This triple thus defines the definition of a method which transitions the receiver object from one correct state, that is one satisfying I, to another.

The importance of substitutability becomes evident here, since the definition of P and Q are derived from the static type of the receiver of the method call. If the actual method being called was not substitutable with the one from which P and Q are derived, then the precondition may not be met, and partially as a consequence the postcondition may not be what is expected. Since the next operation in the sequence relies on the fact that Q describes the state of the program, if it was not in fact established then the composition of statements will not be correct. For example, if R was *x.inc();* where x has static type *Counter* but aliases an instance of *BadCounter*, the next statement in a sequence may result in an error if it expected the value stored by x to have actually been incremented as described $\mathbf{post}(ctx.Counter, inc)$.

When R is an assignment to an attribute, the value being assigned must be one which satisfies the object's invariant, given that the invariant held beforehand:

$$\{I\}\ R\ \{I\}$$

If this were not the case then the assignment represents a state transition in which the final state is not correct in terms of the invariant, and the object is accessible to its clients in such an incorrect state. An exception exists if the receiver is the current object whose method is being executed, thus allowing an object to temporarily transition to a state not satisfying the invariant.

Method calls and assignment are the two operations in CoJava which can modify an object's state. If both are always executed in a way which satisfies these conditions, then the receiver object will always transition from one correct state satisfying its invariant to another. The points in program execution before and after either operation are the visible states [55, 89] for the receiver object, corresponding to when the object is accessible to its clients. While a method call is in progress, the receiver's invariant need not always hold, but must be established if the receiver reaches a visible state before the call completes.

The definition of state transitions for method calls and attribute assignment can thus be augmented with these new requirements. A correct method call must establish the precondition beforehand, assuming that the invariant of the receiver already held. The precondition is the obligation of the caller and while the invariant and postcondition are those of the receiver, therefore this added constraint should be sufficient for correctness:

$$\frac{\begin{array}{l} L(x) = oid \\ H(oid) = \tau \\ \textbf{find\_meth\_def}(P, \tau, meth) = (ctx, meth\_def) \\ meth\_def = ctx'\ cl\ meth(\overline{ctx_k\ cl_k\ var_k}^{k})\{\overline{s'_l}^{l}\ \textbf{return}\ y;\ \} \\ \overline{var'_k}^{k} \perp \mathrm{dom}(L) \\ \textbf{distinct}(\overline{var'_k}^{k}) \\ x' \notin \mathrm{dom}(L) \\ x' \notin \overline{var'_k}^{k} \\ \overline{L(y_k) = v_k}^{k} \\ L' = L[\overline{var'_k \mapsto v_k}^{k}][x' \mapsto oid] \\ \theta = [\overline{var_k \mapsto var'_k}^{k}][\textbf{this} \mapsto x'] \\ \theta(y) = y' \\ \overline{\theta \vdash s'_l \rightsquigarrow s''_l}^{l} \\ [\![\textbf{pre}(\tau, meth)]\!]_{H,H,L[\textbf{this}\mapsto oid][\overline{var_k \mapsto L(y_k)}^{k}]} \end{array}}{(P, L, H, var = x.meth(\overline{y_k}^{k});\ \ \overline{s_l}^{l}) \longrightarrow (P, L', H, \overline{s''_j}^{j}\ \ var = y';\ \overline{s_l}^{l})}\ \text{SR\_Meth}$$

A correct state transition involving the assignment of a value to an attribute must ensure that the invariant of the receiver holds in the post-state. In this case the invariant in the post-state is the obligation of the assigner and not the receiver, since the receiving object has no mechanism to ensure that the value will satisfy the invariant's conditions:

$$\frac{\begin{array}{l} L(x) = oid \\ L(y) = v \\ H' = H[(oid, f) \mapsto v] \\ [\![\textbf{inv}(H(x))]\!]_{H',H',L[\textbf{this}\mapsto L(x)]} \end{array}}{(P, L, H, x.f = y;\ \ \overline{s_l}^{l}) \longrightarrow (P, L, H', \overline{s_l}^{l})}\ \text{SR\_AttrAssign}$$

A method is implemented correctly if every invocation thereof completes in a state satisfying the postcondition and invariant. When a correct method is called with a receiver whose invariant holds and arguments satisfying the precondition, the state of the program after the method completes will satisfy the receiver's invariant and the method's postcondition. This is necessary since the obligation for callers as given previously is only to establish the precondition, and not ensure that the postcondition is met.

The judgment $P, \textbf{inv}, \textbf{pre}, \textbf{post}, \textbf{pure} \vdash_\tau meth$ states that the method $meth$ defined in type τ fulfills this definition of correctness given the program P, the type environment Γ, and its specification:

$$\frac{\begin{array}{l}\textbf{find\_meth\_def}(P, \tau, meth) = (ctx, meth\_def) \\ meth\_def = ctx'\ cl\ meth(\overline{ctx_k\ cl_k\ var_k}^{k})\{meth\_body\} \\ \Gamma = \{x \mapsto \tau, var \mapsto ctx'.cl, \textbf{this} \mapsto \tau, \overline{y_k \mapsto ctx_k.cl_k}^{k}\} \\ P, \Gamma \vdash var = x.meth(\overline{y_k}^{k}); \\ \Gamma' = \Gamma[\overline{var_k \mapsto ctx_k.cl_k}^{k}] \\ P, \Gamma \vdash_{\tau, meth} \textbf{inv}(\tau) \\ P, \Gamma' \vdash_{\tau, meth} \textbf{pre}(\tau, meth) \\ P, \Gamma'[\textbf{result} \mapsto ctx'.cl] \vdash_{\tau, meth} \textbf{post}(\tau, meth) \\ \forall\, H, H', L, L' \mid (P, L, H, var = x.meth(\overline{y});\ \ \overline{s_l}^{l}) \longrightarrow (P, L', H', \overline{s_l}^{l}) \bullet \\ \quad (\llbracket \textbf{inv}(\tau) \rrbracket_{H,H,L[\textbf{this} \mapsto L(x)]} \wedge \llbracket \textbf{pre}(\tau, meth) \rrbracket_{H,H,L[\textbf{this} \mapsto L(x)][\overline{var_k \mapsto L(y_k)}^{k}]}) \Rightarrow \\ \quad (\llbracket \textbf{inv}(\tau) \rrbracket_{H',H',L[\textbf{this} \mapsto L(x)]} \wedge \llbracket \textbf{post}(\tau, meth) \rrbracket_{H,H',L'[\textbf{this} \mapsto L(x)][\overline{var_k \mapsto L'(y_k)}^{k}][\textbf{result} \mapsto L'(var)]})\end{array}}{P, \textbf{inv}, \textbf{pre}, \textbf{post}, \textbf{pure} \vdash_{\tau} meth}$$

A class is correct if all of its methods satisfy this definition as well as being substitutable with its supertype:

$$\frac{\begin{array}{l}P = \overline{cld} \\ cld \in \overline{cld} \\ cld = \textbf{class}\ dcl\ \textbf{extends}\ cl\ \{\overline{fd}\ \overline{meth\_def_k}^{k}\} \\ \overline{\textbf{method\_name}(meth\_def_k) = meth_k}^{k} \\ \overline{P, \textbf{inv}, \textbf{pre}, \textbf{post}, \textbf{pure} \vdash_{dcl} meth_k}^{k} \\ \textbf{find\_type}(P, ctx, dcl) = \tau \\ \textbf{find\_type}(P, ctx, cl) = \tau' \\ P, \textbf{inv}, \textbf{pre}, \textbf{post} \vdash \tau \prec \tau'\end{array}}{P, \textbf{inv}, \textbf{pre}, \textbf{post}, \textbf{pure} \vdash cld}$$

A program P is correct if all classes are correct:

$$\frac{\begin{array}{l}P = \overline{cld_k}^{k} \\ \overline{P, \textbf{inv}, \textbf{pre}, \textbf{post}, \textbf{pure} \vdash cld_k}^{k}\end{array}}{\textbf{inv}, \textbf{pre}, \textbf{post}, \textbf{pure} \vdash P}$$

A complete definition of correctness can thus be stated which uses these definitions and outlines all the requirements:

Definition 3.3.1 (Design-by-Contract Correctness)
A CoJava program is correct with respect to its contracts if it satisfies the following:

- *Methods are only called when the arguments and state of the receiver satisfies the precondition and the invariant.*
- *When such calls are made, the postcondition and the invariant always holds once the call completes.*
- *When a value is assigned to an attribute of an object, the new value satisfies that object's invariant unless it is the same object as the assigner.*

- *Objects must satisfy their invariants when first instantiated.*
- *Whenever a method modifies its receiver such that the invariant does not hold, it must be re-established before the method completes and before other clients can access the receiver.*
- *The substitutability constraint must be maintained between a type and all its subtypes.*

Given these properties of a correct program, all objects can only transition from one state satisfying their invariants to other such states. The program will thus always remain correct in terms of these invariants.

The given definition for correctness relies on a particular assumption about invariants. If an object's invariant relies on state which only that object's methods can modify, then only the operations of these methods can transition the object to an incorrect state. However if the state an invariant relies on can be modified by external clients, all contracts may be respected but still result in a state in which the invariant does not hold.

Consider the *ListCounter* class in Section 2.1 as an example. If the *history* object is directly accessed by a client other than the list, it can be modified in a way that abides by its specifications but results in a state which does not satisfy the invariant. Soundness requires a means of ensuring that an invariant for an object can only rely on state which the methods of that object alone may modify. This ensures that the burden of correctness resides with the object for which the invariant is defined.

The following chapters will discuss the formalization of ownership and the Colleague Technique to do just that. Without such mechanisms, DbC correctness possesses a significant soundness loophole in the way in which invariants work.

3.4 Conclusion

This chapter has introduced the Lightweight Java language as a formalized subset of the full Java language. Although very small, it describes enough interesting semantics of Java to be a basis for the following discussion of ownership and specification. CoJava was subsequently defined as an extension of LJ with the addition of ownership and DbC specification. The next chapter shall discuss ownership in depth and provide proofs for the properties of CoJava programs it statically ensures. The chapter following this shall describe a central problem with invariants and the DbC specification approach as defined in this chapter, and how the Colleague Technique provides the solution.

Chapter 4

Ownership

"The big lie of object-oriented programming is that objects provide encapsulation."

-John Hogg [60]

Encapsulation is an important property in the design of software systems. It ensures that the internal mechanisms of components remain hidden, thus shielding them from modification which may interfere with that component's operation. It also enforces abstraction, where the internal operations of the component are hidden from the outside world.

For object-oriented systems, encapsulation is the property that an object's clients cannot access its internal representation, which is composed of other objects referred to as representation objects. External clients cannot alias the representation objects, hence they cannot apply modifications or rely on particular properties they may have. Representation object thus should contribute to the internal representation of one object only, and be controlled solely by it.

Enforcing this property isn't always straight-forward. Keeping attributes and certain methods private does provide significant encapsulation, but aliases to internal objects may be created whose association with representation objects is lost and are allowed to escape the containing object. There is in general no guarantee that an object meant to be encapsulated by another will remain so, only that the mechanisms and structure of object orientation promote encapsulation [60].

A representation object occupies a particular category of objects in a running program since they are depended upon by others for structural and behavioural correctness. By explicitly stating such objects are special and restricting how they might be aliased, methodologies based on types can be used to statically enforce encapsulation.

Ownership [6, 16, 24, 34, 60, 92, 97] represents this type-based enforcement approach. Internal representation objects are given specialized types which have restrictions on what operations are permissible.This is the basis for the ownership type system employed with CoJava. In comparison to other ownership schemes, CoJava's is quite simple and straight-forward, as well as being compatible with existing Java code that does not use ownership. This simplicity comes at the price of inflexibility in certain instances, which however can be addressed with careful design and implementation.

Given a precise definition of encapsulation, this chapter will prove that the CoJava type system does enforce this desired property. The previous two chapters have first introduced ownership and then defined its semantics as extensions to Lightweight Java. It will be shown here that ownership of this form prevents internal objects from being aliased externally, ensures the ownership structure remains hierarchical and acyclic, and constrains which objects may manipulate this structure.

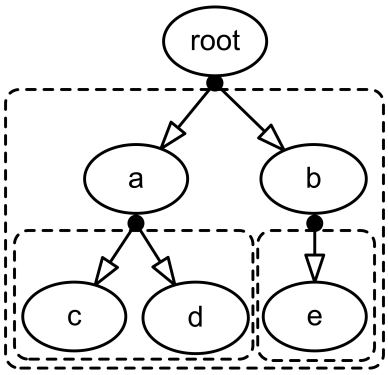


Figure 4.1: Tree Data Structure

Encapsulation is first defined in Section 4.1 in terms of ownership, and discusses some properties and consequences of using encapsulated values. Section 4.2 proves that CoJava's ownership type system does in fact statically enforce this encapsulation property. Section 4.3 defines what a sound invariant is, why this is important for correctness, and what a sound invariant in the context of CoJava is. Lastly, Section 4.4 contrasts the CoJava ownership system against other definitions of ownership by discussing advantages and disadvantages.

4.1 Encapsulation

Beyond hiding internal objects from unauthorized clients, the concept of encapsulation is meant to be one of containment, where each object has its own container (or ownership context) in which all its owned objects are placed. An object can only belong to one container, that is aliased only by their owner, or to an outer container in which the first is found, that is aliased by transitive owners.

This is illustrated with Figure 4.1 where the object *root* has a container with objects a and b in it. The object *root* owns these two directly, but also owns c, d, and e transitively since they are in containers in its container. This hierarchy is important for reasoning about the effects of method calls. If both a and b were allowed to own e then method calls to one which modified e would appear to affect the state of the other. In this case e is effectively internal to neither object, in that they have exposed their representations to another. An object's internal representation defines its structure which is meant to be unique for each object, hence an acyclic ownership structure must be defined.

Ownership always imposes hierarchy on owned objects, such that ownership structures form trees rooted at one topmost owner (discounting transitive ownership). This structure is enforced by the restrictions on when methods accepting owned arguments may be called, and when assigning to owned attributes is allowed. Assuming that a pre-existing tree structure exists, this prevents an operation that would result in a non-tree structure, other than transitive ownership creating "shortcuts", which still preserve the acyclic nature of the owned relationship.

A definition of encapsulation thus should describe both containment and hierarchy:

Definition 4.1.1 (Encapsulation)
Encapsulation is the property where internal representations (owned objects) stay contained within their owners. This requires that owned objects be:

1. *Accessible (aliased) only by their owners and not by non-owning clients*
2. *Not owned by two objects themselves unrelated through ownership*
3. *Modified directly only by their owners*

Section 2.1 discussed the *ListCounter* class (reproduced below) as an example of an owner encapsulating an owned object. This type does fulfill the given definition in that the *history* object is only available to the list that creates it or any object owning the list (which is permitted to call *getHistory()*). Hierarchy is maintained by disallowing an object which is not an owner from accessing a lists' *history* object. Since owners alone can alias owned objects, which is a consequence of disallowing an owned reference being converted to a non-owned one, only the owners can call methods on owned objects and hence all modifications originate from higher up in the hierarchy.

```
class ListCounter extends Counter {
  protected owned IntegerList history;

  invariant (\forall Integer i ; history.contains(i) ; i.intValue()<value);

  public ListCounter(int max) { super(max); history = new owned IntegerList(); }

  ensures history.contains(new Integer(\old(value)));
  ensures history.size() == \old(history.size() + 1);
  public void add(int n){ history.add(new Integer(this.value)); super.add(n); }

  ensures history.contains(new Integer(\old(value)));
  ensures history.size() == \old(history.size() + 1);
  public void inc(){ history.add(new Integer(this.value)); super.inc(); }

  ensures \result == history;
  public owned IntegerList getHistory(){ return history; }
}
```

4.1.1 Strong Containment

CoJava ownership is transitive, that is if a owns b and b owns c, then a is permitted to acquire the reference to c, thus becoming one of its owners. Transitive owners have greater responsibilities towards those objects they own than normal. If b relies on c for its invariant then a must always be aware of this relationship and ensure that any modification it performs on c satisfies b's invariant before it reaches a visible state.

There are situations when this transitive property should be disallowed. Abstract data types [82] must ensure their internal representations are completely hidden. This is the abstract property, where "irrelevant details" are hidden. In particularly the details of what constitutes correct should be hidden, so that clients cannot be held responsible for ensuring internal correctness as defined by invariants constraining members that no external clients can directly manipulate.

In Java using the ownership type system, this means that external clients, whether owners or not, and instances of subtypes, must all not have access to owned objects. Transitive ownership is a useful property that allows an owner to traverse a hierarchy of owned objects, but in many cases an owner must enforce stronger encapsulation since it must assume sole access to internal state.

The simple way to do this is to declare all owned attributes as private, as well as any method returning owned values or accepting owned arguments. Classes satisfying this definition are termed contained classes which may be designated as such with an additional annotation *contained* preceding their *class* keyword. These properties can be trivially checked by the CoJava Tool, and guarantee that any instance of a contained class is the topmost owner of the digraph of objects it owns. For example, *ListCounter* would require slight modification to pass the tool's checks:

```
contained class ListCounter extends Counter {
  private owned IntegerList history;
...
  ensures \result == history;
  private owned IntegerList getHistory(){ return history; }
}
```

There is no inheritance obligation with containment of this sort. A contained class can inherit from a non-contained class, and can be in turn extended by a non-contained class. What containment does mean is that the particular "slice" of data and code that a contained class represents does not allow external clients of any sort access to the portion of the representation it defines. In this example the *ListCounter* class defines a contained slice added to the non-contained slice which *Counter* has provided.

CoJava does not consider member visibility and so other approaches involving the slight adjustment of well-formedness rules must be taken. Containment is first defined in a more general way:

Definition 4.1.2 (Containment)

A contained object does not allow the internal representation it newly defines to be accessible to external clients. This does not imply any responsibilities to the representation inherited from supertypes, nor impose any obligation on subtypes.

For a CoJava class to meet this definition, it must not override methods which return owned values. If a contained class *dcl* overrode a method in its superclass *cl* which did return an owned value, this value may reference an object belonging to the slice which *dcl* defines. All CoJava methods must also be prevented from accessing owned attributes, or methods returning owned objects, defined in contained classes.

4.1.2 Local Methods

A local method relies only on the local state of its receiver, which is the owned objects it has access to as well as primitive values in Java. This implies that it may not access the members of non-owned objects, but may otherwise reference such objects. It may also only call other local methods of owned objects. Consequently a local method is meant to rely solely on the state encapsulated by its receiver object for its behaviour. The references to non-owned objects are considered local state

themselves since they are simple values, but local methods may not rely on the actual state of the objects they reference.

Such methods can be identified in Java with annotations by introducing a **local** modified which may be applied, like **pure**, to method signatures before the type. This would be represented of course with JML in Java code as */*@ local @*/*. A method overriding a local method must still be local and have this annotation, conversely an override of a non-local method cannot be made local.

In terms of CoJava, a local method must only permit attribute access, attribute assignment, and method call statements when the receiver is owned. It furthermore must only allow local methods to be called, and may only construct owned objects. The set of names of all methods satisfying this constraint defined for type τ is represented by $\mathbf{local}(\tau)$. The judgment $P, \Gamma \vdash_{local} s$ states that s is a local statement satisfying these properties:

$$\frac{\overline{P, \Gamma \vdash_{local} s_k}^{k}}{P, \Gamma \vdash_{local} \{\overline{s_k}^{k}\}} \qquad \frac{}{P, \Gamma \vdash_{local} var = x;}$$

$$\frac{\Gamma(x) = \mathbf{owned}.dcl}{P, \Gamma \vdash_{local} var = x.f;} \qquad \frac{\Gamma(x) = \mathbf{owned}.dcl}{P, \Gamma \vdash_{local} x.f = y;}$$

$$\frac{P, \Gamma \vdash_{local} s_1 \quad P, \Gamma \vdash_{local} s_2}{P, \Gamma \vdash_{local} \mathbf{if}\ (x == y)\ s_1\ \mathbf{else}\ s_2} \qquad \frac{\Gamma(x) = \mathbf{owned}.dcl \quad meth \in \mathbf{local}(\mathbf{owned}.dcl)}{P, \Gamma \vdash_{local} var = x.meth(\overline{y});}$$

$$\frac{}{P, \Gamma \vdash_{local} var = \mathbf{new_{owned}}\ cl();}$$

The membership of **local** can thus be defined by the following:

$$\frac{\begin{array}{l}\tau = ctx.cld \\ \mathbf{find\_meth\_def}(P, \tau, meth) = (ctx, meth\_def) \\ meth\_def = ctx'\ cl\ meth(\overline{ctx\ cl\ var})\{\overline{s_k}^{k}\ \mathbf{return}\ y;\ \} \\ \overline{P, \Gamma \vdash_{local} s_k}^{k}\end{array}}{P, \Gamma \vdash meth \in \mathbf{local}(\tau)}$$

The value of a local method is the ability to assume its behaviour is not affected by external objects. Whatever state a local method depends on is contained within its receiver, thus the side-effects of the method are more easily understood and controllable. A local method's behaviour also will not vary so long as the receiver is not changed, thus a method which is both pure and local can be defined to always return the same value.

A contract predicate using such a method can rely on this consistency and on the behaviour depending solely on the receiver. Predicates meant to describe a property exclusively concerned with the call's receiver can now be defined without having to be concerned with external dependencies.

Locality only requires that non-owned mutable objects accessible outside the current scope of the method be ignored. A object newly created during a method call can be termed a "fresh" object,

and will remain fresh so long as it only interacts with other fresh objects. Such objects can be interacted with in any way without involving non-local state, having no clients beyond their creator and other fresh objects.

Ensuring freshness can leverage specialized types, such as being interpreted as a form of owned objects, yet still be assignable to regular attributes of the current object or otherwise be compatible with regular references. Static data flow analysis [35, 38, 47, 48, 56, 73] can also be employed to determine which objects are fresh and do not interact with non-fresh non-owned mutable objects. This is not covered however in CoJava, the strict interpretation of local is used since its enforcement involves much simpler checks.

4.1.3 Limitations

The hierarchy imposed by ownership and the restrictions on operations have their drawbacks in certain situations. As an owned reference cannot be given to another object, only acquired from owned objects lower down in the hierarchy, certain operations cannot be performed without breaking the ownership rules.

Tree balancing, for example, involves assigning new children to nodes that are being moved lower or higher in the structure. A node may acquire a new child by querying it from lower down in the hierarchy, but balancing may require that a node from another context in the other side of the tree be given to an object as a new child. This is prohibited since the new child originated from somewhere else other than lower down in the ownership hierarchy, thus the copying of nodes would be necessary.

The **this** value is typed as ***owned***.*dcl* within the methods of class *dcl*. This prevents an object from producing non-owned references to itself which, if the object were owned, would allow it to be aliased through owned and non-owned references in contravention of the type system. Consequently, an object cannot give a reference to itself to another object that it has created. For instance when a data structure creates an iterator to traverse its elements, the value ***this*** cannot be passed to the iterator.

Ownership is also a fixed property of the object once created. Statically it cannot be known if an owned object is no longer part of a representation and so can be aliased through regular references, or when a regular object can join a representation without being externally aliased.

Collections of owned objects are not possible in the general sense since a data structure accepting owned objects cannot have methods callable by external clients. Since a method with owned arguments can only be called when *this* is the receiver, a data structure cannot be given an owned item to store. Owned objects are thus more useful as a means of building internal structures that have particular shapes, such as linked lists and explicit tree data structures.

4.2 Proof of Encapsulation

To demonstrate that CoJava's typing and operational rules uphold the given definition of encapsulation, a set of conditions must be proved:

1. **Owner Aliasing:** The owners of an object are those which access it through an owned reference. The owner constructed the object in the first place or acquired the reference from another owned object. If an owned object is aliased through a regular reference then it can become accessible to non-owners. It must be shown that the type system ensures an object

created as an owned object is aliased only through owned references. Static type rules must ensure that operational rules do not produce a state where two known variables may alias the same object but have types with differing ownership states.

2. **Hierarchy:** Hierarchy implies that two objects may only share owned object references if one owns the other. If two objects unrelated through ownership were to share owned objects, then essentially their representations were exposed to one another, an undesirable situation even if they had common owners. By ensuring that this is never the case, which also implies that two objects cannot own one another transitively or otherwise, the type system ensures that owned objects are organized into acyclic digraphs rooted at one topmost owner.

3. **Self Modifies:** An object may only call methods with owned arguments, or assign to owned attributes, when it is the receiver. Methods returning owned objects, and owned attributes, can only be accessed on owned receivers. These restrictions ensure that an object alone may change its own internal owned structure, and so only its methods are responsible for constructing correct structures. It also prevents an owner from allowing two owned objects to share representations, thus breaking the hierarchical requirement. Since only owners may mutate an owned object, they are responsible for exclusively performing valid operations which respect the owned object's invariant. External clients are thus not directly responsible for such invariants.

If these properties are ensured by the type system, then no owned object can be exposed to any client save transitive owners, either through regular or owned references. It will also prevent non-owners from modifying owned objects directly, thus the methods of owners must be used to mutate internal representation objects. This places the burden of correctness on the owners alone and not on external clients, as well as enforces abstraction since external clients are only aware of the owners' public interfaces.

4.2.1 Owner Aliasing

In the previous chapter it was stated that CoJava is type-safe since it extends another type-safe language, Lightweight Java, without introducing type concepts which might affect safety. This ensures that the type of a value will always be related to the initial type, thus an instance of type τ will always be typed as such or as a supertype of τ. It will never be typed with a subtype of τ or any type unrelated to τ, which would allow a program to "go wrong" if members not present in the definition of τ were accessed.

This is enforced by the subtype requirements of the statement well-formedness rules along with the definition of the $\prec$ operator. A variable with type τ can only be assigned a value whose type is τ or a subtype thereof. Similarly the types of arguments used in a method call must each be a subtype of their respective formal argument declared for the method. Variable assignment and method arguments are the only means by which a value might be accessible with a type differing from its actual runtime type.

These conditions and the definition for $\prec$ ensure the same for owned reference values, which represent a restriction on what statements are considered well-formed rather than an introduction of new concepts. If a reference value initially having type **owned**.*dcl* were instead typed as *dcl*, the program would still not go wrong since both types are defined by the class *dcl*. However this would imply that this value now makes the object it references accessible as an owned and non-owned

object. To prevent this, no relationship is defined between owned and regular references, such that no predicate $\mathbf{owned}.dcl \prec dcl'$ or $dcl \prec \mathbf{owned}.dcl'$ is ever true regardless of what types dcl and dcl' are.

This absence of a relationship implies that the well-formedness definitions of the assignment statements prevent an owned value from being assigned to a non-owned variable, and vice versa. Consider the well-formedness definitions for each of the five assignment statements:

- WF_AssignVar ($var = z;$)
  This requires that the type of x be a subtype of var, thus if one is owned and the other is not the statement is not well-formed.

- WF_AssignAttr ($var = z.f;$)
  Similarly this requires the type of f to be a subtype of var, and so imposes the same restriction.

- WF_AttrAssign ($z.f = var;$)
  This rule defines the inverse operation but again requires var to be a subtype of f.

- WF_New ($var = \mathbf{new}_{ctx}\, cl()$)
  When creating a new object whose reference is assigned to var, it is required that the type $ctx.cl$ be a subtype of var, which takes into account when ctx is nothing or **owned**.

- WF_MethCall ($var = z.meth(\overline{y});$)
  Lastly, the return type of $meth$ must be a subtype of that of var. It also requires that the type of each actual argument in $\overline{y}$ be a subtype of the formal argument declared for $meth$.

In each of these definitions of well-formedness, a value (either a variable or method return value) may not be assigned to a variable if the ownership status of the two does not match. For example, a statement $var = z;$ will not be considered well-formed if the type of var is owned but that of z is not. If it were, then the object z aliased would be accessed as an owned object through var and a regular object through z. The case for arguments is similar, in that if z were used as an argument whose type should be owned, then the object it aliases appears to be owned within the body of that method call.

These simple type restrictions, arising from the interpretation of ownership as defining two exclusive classes of types, ensures that an object constructed as owned will never have non-owned references. If an object is constructed as a non-owned object, the **this** reference is still owned, but will not be accessible to external clients. To acquire **this**, a method returning an owned object would have to be called; this is not possible for a non-owned object since such methods can only be called through owned receivers.

4.2.2 Hierarchy

The hierarchical nature of ownership is important, since it disallows two objects which do not own one another from sharing owned objects. Given some object a which owns c, allowing another object b to access c essentially allows representation exposure between a and b. Figure 4.2 illustrates this, where a is considered the "legitimate" owner of c, in that it created c or acquired the reference to c from some other owned object. In this situation b is not meant to own c, therefore the type system must prevent such exposure from occurring, even if a and b had a common owner and hence existed in the same container (ownership context).

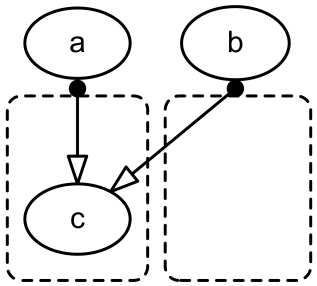


Figure 4.2: Representation Exposure

This is equivalent in general to saying that owned objects can only be referenced within nested ownership contexts. Nested sets impose their own tree structure, therefore a relation which maps objects from one set to those from sets contained in the same set will always form a tree when omitting transitive shortcuts. A relation does not respect the tree property when it maps objects from two disjoint contexts, which may have the same containing context or none at all. The given figure represents such a case where the context of a and b are disjoint.

An object acquires an owned reference in a number of ways: creating a new object, calling an object's method, accessing an object's attribute, being the receiver of a method call, or being the receiver of an attribute assignment. Assuming that a hierarchical structure is in place in a given point of execution, any operation which would break this structure does not type correctly according to CoJava's well-formedness rules. Each operation which may allow an object to acquire a new owned reference is analyzed here:

- **Creating a new owned object**: A new object of course does not exist in any other context, so it now exists in its creator's context only. This is defined by WF_New.
- **Accessing an owned attribute of another object**: This is only allowed when the receiver z is owned, as the rule WF_AssignAttr defines. If z is owned, then it exists in the caller's context only since the hierarchical structure is assumed. Accessing the owned object f from z creates a shortcut in the tree where the caller now owns it transitively. Since f is exclusively in the context of z and z is in the context of the caller, f is still only referenced within nested ownership contexts.
- **Calling a method of another object which returns an owned reference**: This is only allowed when the receiver z is owned, as rule WF_MethCall defines. Any owned object which such methods can return must either be owned objects newly created by the method, those stored by attributes of z, or those acquired from other objects owned by z. In any situation, these objects are within the context of z, and hence now become transitively owned by the caller. In such a case they remain within nested contexts as required.
- **Having another object assign to its owned attributes**: This is disallowed except when the receiver is *this*, as the rule WF_AttrAssign defines. By not allowing such assignment, an object cannot be given an owned reference by an external client of any sort. The object b in Figure 4.2, for example, cannot acquire a reference to c as a result of a client assigning to any of its attributes.

- **Having another object call one of its methods accepting owned arguments**: Only when the receiver is *this* can a method with owned arguments be called; the rule WF_MethCall disallows such calls when any other receiver is used. Again an object cannot be given owned references by clients, so object b once more cannot be given a reference to c by any client.

These restrictions ensure that owned references can be acquired from the object creation statement, or from objects lower down in the object hierarchy. If only attribute access and method return values can be used to acquire owned references, then only shortcuts which represent transitive ownership can be made in the owned object tree. This ensures that two objects a and b which are not part of the same hierarchy, or are siblings residing in the same context, cannot share owned objects and hence expose their internal representations to one another. If transitive ownership is discounted, then every owner represents the root of its own sub-tree of owned objects which exclusively comprise its internal representation.

As a consequence the internal structure of objects stays internal and cannot be directly modified by clients. Since an object's owned attributes cannot be assigned to, nor its methods accepting owned arguments called, the relationship it has with the objects it owns cannot be changed by any external client. To change these relationships, an object's methods which represent internal operations must be called. This allows a class to be defined with the knowledge that owners of its instances might have access to its internal structure but cannot modify it. Although this disallows certain operations which are correct, the responsibility for structural correctness remains encapsulated by owning objects.

4.3 Invariant Soundness

This section will discuss the definition of a sound invariant and what constraints ensure an invariant satisfies the definition. Ownership allows an invariant to soundly rely on owned objects, but also represents added responsibilities to owners with respect to their owned state. If an invariant for an object is sound then that object will remain consistent with it if the methods of all objects are correct. Without soundness, an invariant may define a condition depending on objects which may not satisfy its constraint after being modified by correct operations.

First a definition of soundness is given:

Definition 4.3.1 (Invariant Soundness)
The invariant I *for object* a *is sound if no correct operations of other objects produce a visible state for* a *in which* I *does not describe its state.*

This implies that a is only transitioned to a state not satisfying I by its own methods, but is then transitioned back to a state satisfying I before a reaches a visible state or the call completes. A lemma can thus be derived regarding visible states of execution:

Lemma 4.3.2
If all the invariants within a system are sound, then the invariants of all objects will hold when they reach their visible states.

Not all invariants are sound by this definition. The specific challenge to soundness which this thesis addresses is when an invariant relies on mutable objects for its condition. For example, consider a client of a *Counter* instance:

```
class CountHolder {
  private int count;
  private Counter counter;

  invariant counter.get() == count;

  public CountHolder() { counter = new Counter(10); count = 0; }

  requires count < 10;
  ensures count == \old(count + 1);
  public void inc() { counter.inc(); count=count+1; }

  ensures \result == counter;
  public pure Counter getCounter() { return counter; }
}
```

The classical definition of the Design-by-Contract approach holds that the methods of an object alone should be responsible for ensuring the invariant, such that they must ensure it holds when they finish execution [89]. It also requires that an object's invariant can only be broken by its methods. In this way an object can only transition from one valid state to another, since state transitions are only performed by method calls. Assignment to attributes by external clients should therefore be disallowed, or else clients must be obligated to only assign values which satisfy the invariant.

However the class *CounterHolder* demonstrates an invariant which can be broken by the methods of another object. The *Counter* instance which the invariant depends on can be accessed by external clients, who might then modify it such that it does not satisfy the invariant. Such an operation may still be correct, in that contracts involved in the operation are respected and the invariant of the receiver holds upon completion, yet still leave other objects in inconsistent states. Consider the following, where the *inc()* invocation is an example of just such an operation:

```
CountHolder holder = new CountHolder();
Counter counter = holder.getCounter();
counter.inc();
```

The call to *inc()* respects the method's contract and so would be expected in the classical reasoning approach to result in a valid state for all objects. However the invariant of *holder* now no longer holds for its state. Classically a correct operation is expected to transition the system from one correct state to another, yet an incorrect final state may be reached if unsound invariants are present, such as that defined for *CountHolder*.

4.3.1 Defining Sound Invariants

The source of this invariant unsoundness is the dependence on mutable objects. If an invariant relies on numeric values or immutable objects, classical reasoning approaches such as [2, 89] guarantee soundness. However such simple invariants are often not sufficient to specify anything beyond trivial object types.

The solution is to constrain object relationships to a degree. If the set of possible clients of *counter* from the above example is controlled, then only those which respect the relationship between it and

holder would be allowed. The constraining methodology for doing this is ownership, such that the clients of *counter* are always its owners, *holder* being its immediate owner. Other owners might own both objects and hence would be aware of the relationship. These objects can therefore be reasonably expected to ensure the invariant is maintained.

Ownership can thus be used to reduce the complexity of object relationships. When a method call breaks the invariant of *holder*, the original caller will always be one of its owners (including itself). This reduces the sources of error-producing operations to a known set of objects which can be analyzed or checked in greater depth. In particular, such analysis would be used to ensure owners uphold the invariant conditions between two owned objects.

If *CountHolder* is a contained type, then *counter* will always have exactly one owner, namely *holder*. In this situation, the methods of *holder* alone can violate that portion of the invariant depending on *counter*, thus satisfying the classical definition of the relationship between method and invariant. Strong containment thus implies simpler correctness obligations, it therefore must factor into the design of a class if allowing transitive access to internal representation objects warrants the added complexity of demonstrating correctness.

4.3.2 CoJava Invariants

An invariant can thus rely soundly on a subset of values. If an invariant is predicated on an immutable value, being either a reference to an immutable object or a primitive value in Java, then no operation can transition such a value to a state not satisfying the condition. If a value references an owned object, then only the owners may modify it in a way which breaks the invariant. Since there is significant control over who these owners are, they can be reasonable held accountable for re-establishing the invariant before allowing any object involved to reach a visible state.

Only methods which rely directly or indirectly on such values may be called in an invariant. This prevents invariants from relying second-hand on values which do not meet these criteria. One exception is that an object attribute of the current object can be soundly compared to other reference values using the equality and inequality operators in Java, or in the conditional statements of CoJava methods. This is because the local arithmetic value of these references, rather than the state of the objects they refer to, is being used by the invariant.

Therefore, CoJava invariants must be pure-local predicates, relying only on the local state which the current object encapsulates, as well as being pure. They may use only methods which are both pure and local, and may not rely on the state of non-owned mutable objects. Quantifier variables declared in quantifier predicates must similarly have an owned type, and the sub-predicates within quantifiers must still be pure and local. Defining immutable types is not possible in CoJava since it requires members to be declared private, however in Java types like *String* can be defined which never changed state and so may be soundly depended upon by invariants.

A sound CoJava invariant is defined in terms of pure-local predicates:

Definition 4.3.3 (Invariant Soundness)
A sound invariant may only rely on owned objects or reference values themselves. Invariant predicates are therefore pure and local, with the added requirement that quantifier variables have an owned type as well as quantifier sub-predicates being pure-local.

This definition would require the *counter* attribute of *CountHolder* to be owned if the invariant is to depend upon it. This prevents external non-owning clients from modifying it in any way, while allowing owning clients to do so but with the additional responsibility to ensure dependent invariants are satisfied.Transitive ownership must then factor into the definition of the visible state of an object a, at which point its invariant must hold:

- When any method with a as a receiver begins or completes execution.
- Whenever a's attributes have been assigned to, and then a becomes accessible to any client other than the assigner, such as when the assigner's method completes.
- Whenever an owner of a directly modifies an object b owned by a, and then a or b becomes accessible to any client other than the modifying owner, such as when the owner's method completes.

This allows an owner to violate the invariant of the objects it owns through correct operations. The owner is however responsible for ensuring the invariants of the owned objects are satisfied before they are allowed to be accessible by any other client.

4.4 Other Ownership Schemes

The concepts of alias control and containment which ownership provides have evolved from the observation that object-oriented programs often have few encapsulation problems. Aliasing between objects is usually kept local, and so programmers in general can conceptualize what local relationships exist. Introducing references from external sources does not often introduce incorrectness since these are implicitly treated by code as having an external origin. These properties are maintained only by good practice, and many program errors are rooted in confusion over the intended purpose of an object reference (internal versus external), thus a means of automatically enforcing and checking this practice will help identify such errors.

This observation is stated as part of the formulation behind Islands [60], where one topmost object controls access to those objects that exist within its island. This captures explicitly the local aliasing behaviour present in object-oriented programs, such that instances where the distinction between external and internal objects has been misapplied will be caught as static errors. Encapsulation within islands is enforced by type requirements on classes, such that the instances of a class are bridges between their islands and outside world if the methods of the class do not expose the objects in their islands.

Balloon types [6] divide type definitions into two categories: non-balloon types representing uncontrolled types present in languages by default, and balloon types whose instances cannot be aliased by multiples instance variables and which do not expose any reachable state to external clients. The one reference to an instance of a balloon type which is held by an object external to the balloon instance, and no object internal to the balloon instance is accessible to any external objects. Local variables may alias balloon objects, but instance variables with balloon types need to be assigned new objects. Therefore the object storing the one reference to a balloon type is in effect its exclusive owner, and those objects the balloon object aliases are owned by it.

In both of these approaches it is the static properties of the object type definitions that ensure that instances thereof exhibit alias control properties. Bridge classes in Islands and Balloon classes

in Balloon Types must meet defined criteria to be considered correct. This enforces the desired encapsulation property at the expense of flexibility and the need for complex criteria and checking.

Encapsulation within Java packages ensures that certain types are not accessible to types in other packages. Confined types [119, 122] is a method of statically enforcing this property. In effect the instances of a confined type are owned by the package, such that instances of types from other packages cannot access them. Confinement thus enforces an encapsulation property on types rather than on objects as is the case with ownership. The approach is applied to ensuring confinement in Enterprise JavaBeans [32]. Generic type parameters have also been applied as a means of providing confinement without additional language constructs or checks [103].

CoJava ownership in contrast relies on the designation of certain objects as being owned and hence are treated differently. Rather than restrict the form types can have or what types are accessible in external packages, the restriction is on what statements are deemed to be well-formed. The form class types and packages can have is not restricted as much and not in the same way, and so there are fewer structural constraints while still providing the same encapsulation. The means of enforcing this ownership is simpler as well, requiring only checks on statements to determine if they meet the well-formedness criteria.

CoJava's approach of attaching meaning to object reference is shared with later ownership formulations. Flexible Alias Protection [34, 97] augments types with modifiers which describe their roles. For example, a type *rep* T is the type of objects that compose the internal representation of another. The class T still defines their type, however type rules prevent their references from being exported to clients external to the object whose representation they compose. References with type *arg* T may only be used to perform non-mutating operations, while a *free* T reference is the unique reference for a particular object. Value types are represented as *val* T, whereas *var* T is the default reference type.

AliasJava [5] extends Flexible Alias Protection and introduces further type annotations. The *owned* T type is very similar to that in CoJava. Objects aliased through variables with this type cannot be exposed to external clients of the enclosing object. This is a stronger encapsulation property than the one provided by CoJava since transitive ownership is not present. This is extended with ownership domains [3, 4] that allows owned objects to be categorized into related sets with definable access rules.

Flexible Alias Protection and AliasJava ensure encapsulation through a very similar approach to CoJava. They introduce different type annotations and domains, which provide features beyond simple containment which CoJava lacks. Essentially CoJava is a minimal ownership system used to enforce encapsulation and provides nothing more. This limits its applicability in many situations which the added features in these approaches attempt to address.

Spec# [16, 108] uses an ownership scheme integrated with it's notion of invariant consistency. Special fields are defined for each object recording whether the invariant currently holds, who the owner is, and if the object is open for mutation. When an object is mutated, it must be "unpacked" beforehand, which then also unpacks any owned objects for mutation as well. When mutation is complete, a "pack" operation is applied that asserts the invariants of the owned objects and then the owner. Attributes and variables storing references to owned objects are designated with the "rep" modifier.

The owner of an object can also be changed through the "transfer" operation, which is possible since the special fields recalling the owner can be modified. This methodology is designed for runtime checking as well as static checking with the Boogie [13] verifier, which is used to check that these

operations are correct and preserve the desired ownership properties. Ownership is thus not a static property of an object which cannot be changed, but a more dynamic notion of relationships allowing an object to change owners or be owned by no object at all. CoJava's approach is based on the type which never changed, and so lacks this kind of flexibility. It also must disallow certain operations as not well-formed, which might be safe in that encapsulation is not lost, but can only be identified as safe and so allowed with a verification approach.

Universes [41, 93] defines a more comprehensive type system where the type modifier depends on the client's perspective. An object is the sole owner of another when it aliases it through an owned reference. Two objects that have the same owner are defined as peers and can alias each other through peer references. Both reference types can be cast to a read-only reference through which only side-effect free operations may be performed.

The types of references depends on the relationship between supplier and client. If an object accesses an owned value of a peer, the reference type will be viewed as read-only, thus preventing it from modifying the internal representation. Similarly, if an owner accesses a peer reference of an owned object, that reference will be owned as well. This definition of viewpoint-dependent ownership has also been extended with generics [39, 40], allowing the definition of generic collections that can store sets of owned objects.

Although ownership is a static type property, the actual property itself changes on the perspective a reference is seen from. With CoJava a regular or owned reference always appears the same to all accessing clients. This precludes non-owning clients from accessing an owned object through a read-only reference, for example. The trade-off that Universes makes is between flexibility and complexity, both in the type system and in how much a programmer must learn to be able to use it. Understanding the concept behind an owned reference becoming a read-only one in certain situations is much more daunting to someone used to standard reference types, so it becomes a question of whether the simplicity and constraint of one approach is more suitable for a given situation over a more elaborate but permissive approach.

The acyclic property of ownership and the concept that holding a lock on an owner implies locks on those objects it owns is used with Safe Concurrent Java [23, 40] as well as with Universes [36]. These techniques leverage the idea that controlling the owner gives control over what it owns. They prevent multiple threads from accessing and mutating data simultaneously, or locking indefinitely while waiting for each other to act. Race conditions and data-based deadlock are thus prevented statically by the type system.

This thesis does not address the relationship between concurrency and ownership in CoJava. Previous work [69, 70] however has, but from the perspective of a lock-free concurrent environment. Ownership is instead used to organize active objects [33, 74, 114, 53, 9] representing units of concurrency into hierarchies to prevent deadlock. The premise is that an active object will not induce deadlock if it waits indefinitely for a response from an object it owns whose methods are executed asynchronously. The criteria for what objects active objects can share amongst themselves prevents data races, and so ownership is not used for the purpose of organizing data locks since they are not present.

4.5 Conclusion

This chapter has proven the encapsulation properties of the CoJava ownership type system. Compared to other types systems, in particular Universes, ownership in CoJava is simple and suffers from problems of flexibility. The restrictions on calling methods with owned parameters disallows the use of data structures to store owned references, and operations such as adding nodes to linked lists of owned nodes is also not possible. In practical use this method of ownership is most applicable for defining the important structural elements of object's internal representations.

This type system however provides a means for invariants to soundly rely on objects. An object's invariant can correctly rely on the state which it encapsulates, since external non-owning clients cannot directly mutate this state, let alone in a way which violates the stated constraints. Transitive owners are an exception, being granted access to internal owned state which invariants rely on. Since the relationship between owner and owned is better controlled and defined, owners can reasonably be held responsible for ensuring the invariant constraints between owned objects.

Not all programming idioms, patterns, and designs can be implemented with the rigid hierarchy ownership requires. An iterator for example cannot be owned by the data structure it traverses, yet there is still a correctness constraint between the two which an invariant should capture. The next chapter shall formalize the Colleague Technique which extends the notion of invariant soundness defined here. This methodology allows invariants to soundly depend on non-encapsulated objects, although at the expense of greater coupling between the types in question. Patterns such as iterators and subject-observer can be specified in greater detail with ownership and Colleagues as the next chapter will discuss.

Chapter 5

The Colleague Technique

This chapter will define and discuss the Colleague Technique whose application ensures the soundness of invariants in the classical DbC approach. An invariant which relies on objects freely aliased and mutated by arbitrary clients is not sound, as discussed in the previous chapter. The Colleague Technique defines additional invariant and structural obligations which ensures soundness in such cases.

Ownership guarantees the encapsulation of one object within another, such that an owned object can only be accessed through the interface of its owner. Consequently, an invariant dependent on owned objects can only be invalidated by the operations of the object they are defined for, except in the case of transitive ownership. Not all patterns of object relationships can be described in terms of ownership. One object may wish to depend on another for its invariant condition, yet still not encapsulate it.

The Colleague Technique provides a specific mechanism for this particular situation which involves the generation of additional invariant conditions, and ensuring that two co-dependent objects mutually alias one another so long as the relationship exists. The technique allows classical reasoning to be soundly applied to the augmented invariant conditions, thus existing reasoning techniques and tools can still be used.

A clear instance of this problem can be found in the iterator pattern. An iterator will reference the data structure it traverses and would commonly have certain correctness criteria. One desirable behavioural property is that an iterator would produce as many items as the data structure contained when it was created. When performing some iterative operation dependent upon the expected number of items, the presence of such a basic correctness property is critical. This can be stated as an invariant requiring the data structure to have that many items during the iterator's lifetime. A stronger property, although very hard to enforce or even check, would require that the items which the iterator would traverse in the structure cannot be changed during that iterator's lifetime.

Figure 5.1 gives the Java code for an iterator implementation which includes specification annotations but does not use ownership. The correctness property defining the relationship between an iterator and its data structure is recorded by the second invariant predicate of *ListIterator*. When an iterator is instantiated, *last* is set to the size the data structure at that point, therefore so long as the iterator exists it requires the *list* to be no smaller than that. The *List* instance must pass a reference to itself to the iterator it creates. This isn't possible with CoJava's type system where *this* has type ***owned***.*List*, since owned values are not valid constructor arguments. CoJava's strict ownership type system makes certain relationships very difficult to construct, a topic addressed here.

```
class List implements Iterable {
  private ArrayList items;

  public List() { items = new ArrayList(); }

  ensures items.contains(o);
  public void add(Object o) { items.add(o); }

  requires i >= 0 && i < size();
  ensures \result == items.get(i);
  public pure Object get(int i) { return items.get(i); }

  requires i >= 0 && i < size();
  ensures !items.contains(\old(items.get(i)));
  public void remove(int i) { items.remove(i); }

  ensures \result == items.contains(o);
  public pure local boolean contains(Object o) { return items.contains(o); }

  ensures \result == items.size();
  public pure local int size() { return items.size(); }

  public pure ListIterator iterator() { return new ListIterator(this); }
}

class ListIterator implements Iterator {
  private List list;
  private int position, last;

  invariant position <= last;
  invariant list.size() >= last;

  public ListIterator(List list) {
    this.list = list;
    last = list.size();
  }

  requires list!=null;
  ensures \result == (position < last);
  public pure local boolean hasNext() { return position < last; }

  requires hasNext();
  ensures position == \old(position) + 1
  ensures \result == list.get(\old(position));
  public Object next() {
    position = position + 1;
    return list.get(position - 1);
  }

  public void remove() {}
}
```

Figure 5.1: An Iterator Implementation in Java

The invariant of *ListIterator*, which relies on the list instance, is unsound according to the definition given in the previous chapter. An iterator may be created, and then the list it relies on mutated in later execution stages. This mutation may be correct in that it respects all relevant specification conditions, but may transition the list to a state which does not satisfy the iterator's invariant without ever calling a method of the iterator itself. There is no way to check at runtime that this has occurred until a method of the iterator is called and the invariant is discovered to not hold. The connection between the operation which produced the error and the operation that uncovers it is broken, that is to say the assignment of blame is not made correctly.

The problem should be considered from the perspective of the *List* type. The instances of this type have a responsibility to any iterators that rely on them, however there is nothing in the specification for *List* to indicate this nor prevent operations which result in incorrect state. The expectation in classical DbC reasoning is that any correct operation results in a correct program state. What is lacking is a feature of the specification ensuring that clients of lists cannot perform operations which respect the specification but also invalidate any iterator's invariant.

If formal proof is applied instead of runtime checking, it cannot necessarily be shown that the list does not have a client somewhere that will remove too many items, and so invalidate the invariant. Consider a class *ItemConsumer* which removes an item from a list it aliases:

```
class ItemConsumer {
  private List list;

  public ItemConsumer(List list) { this.list = list; }

  requires list.size() > 0;
  ensures list.size() == \old(list.size()) - 1;
  public void remove() { list.remove(0); }
}
```

It cannot be proven, by runtime testing or formal proof, that this type is always correct, that is the correct invocation of its methods will never result in contract violation. Any list which an instance of *ItemConsumer* references may be a dependee of an existing *ListIterator* instance, and there is no code or information available to prevent an operation which would invalidate its invariant.

To call *remove()*, it would have to be proved statically that the *List* which the instance of *ItemConsumer* references is not depended upon by any *ListIterator* instance whose invariant would be violated. This requires statically proving properties about dynamic runtime relationships between objects. In the general case this cannot be done at all, hence ownership and other type system properties have been employed to enforce certain relationship properties.

Global reasoning of this form, where all possible relationships must be analyzed, also precludes modularity. The introduction of a type would require that the entire program would have to be re-analyzed. This is necessary to ensure no error-producing relationships can be formed with instances of the new type. For example, a program that has only *List* and *ListIterator* types might be shown to be correct, but introducing *ItemConsumer* requires demonstrating that no instance of this type can be used to violate invariants. What this means in particular is that *ItemConsumer* is only sometimes incorrect, specifically when associated with certain lists and used in a certain way.

Reasoning in this way then is not a viable solution. What is needed is a way to employ classical Design-by-Contract reasoning about object types, which provides a modular correctness criteria for types which is practical to check statically or at runtime. In this way a type which is shown to abide by its specification and those of its suppliers will always be correct, regardless of what other

types exist and what relationships are formed. Relationships are the key challenge here which are not addressed by the classical DbC invariant approach.

This chapter begins with Section 5.1 introducing the technique and Section 5.2 discussing how relationships in the technique are formed and managed. Section 5.3 is the central part of this chapter where the mirror invariant and its construction is described. Section 5.4 demonstrates the technique with a list and iterator, with the subsequent proof in Section 5.5 demonstrating that the invariant for these types is in fact sound. Section 5.6 outlines related work, followed by a concluding section.

5.1 The Colleague Technique

Classical DbC reasoning can be made sound again when the methods of an object cannot be called correctly if they break the invariant of other objects. In the above example, this means that a call to a method of *List* can only break the invariant of a *ListIterator* instance if its contract was not respected. If the contract is respected, that is the invariant holds in pre- and post-states and the conditions were met, then no object's invariant can be broken.

The Colleague Technique is a formalized invariant methodology that does this. Two object types are colleagues of one another, that is they are collegial, if each has specially designated attributes which alias the instances of the other. An object can predicate its invariant on its colleague objects which these attributes alias. This is sound since this invariant is used to generate a mirror predicate which is added to the specification of the colleague type.

In the case of the *List* and *ListIterator* types, the invariant of the iterator can be violated when a method of the list is called which removes too many items. A mirror invariant predicate must be added to *List* such that its invariant states the same constraint as that of *ListIterator*. Given this invariant, any method of *List* whose post-state violates the invariant of any dependent iterators will also violate the invariant of its receiver. If references to all iterators traversing a given list are stored in an attribute called *iterators*, then this additional invariant states the necessary correctness requirement:

$$\forall\, i : \mathit{ListIterator}\ i \mid \mathit{iterators}.\mathit{contains}(i) \bullet \mathit{size}() \geq i.\mathit{last}$$

This is a restatement of the invariant from *ListIterator* which defines the constraint between it and *List*, namely $\mathit{list}.\mathit{size}() >= \mathit{last}$. In the mirror invariant, this is stated from the perspective of the current object and in terms of a variable quantified over the set of iterators, thus $\mathit{size}() >= i.\mathit{last}$ is derived. If this is added to the specification of *List*, then any method call on an instance of this type cannot violate the invariant of an iterator without also breaking this invariant. Therefore correct method calls are prevented from indirectly invalidating the invariants of other objects.

Because both types now have invariants creating dependency between their instances, the partners in a collegial relationship must always alias one another. If a *ListIterator* instance, for example, was not aliased by the *List* it traversed, then the protection the mirror invariant added to *List* would not be applied. The Colleague Technique is used in conjunction with helper methods which construct this bi-directional aliasing, and type checks are used to ensure that collegial attributes are not accessed or mutated by other methods directly. These methods are generated by the CoJava Tool when generating Java code, however in the formalization with CoJava in this chapter they will be represented by abstract predicates or statements.

The Colleague Technique is thus composed of two major components: the means by which mirror invariants are determined, and the helper methods generated by the tool which manage

```
class List {
  private collegial ListIterator.list  Set iterators;
...
}

class ListIterator {
  private collegial List.iterators List list;

  invariant list.size() >= last;
...
}
```

Figure 5.2: The List and ListIterator Example With Colleague Annotations

relationships. The next sections will define these components and restate the *ListIterator* example using Colleagues.

5.2 Constructing Relationships

Objects entering into collegial relationships must alias each other at all times. This ensures that both partners and their clients are aware that the relationship exists, and it will be shown this is critical to invariant soundness. To ensure that these relationships are explicit and maintained, the Colleague Technique uses annotations to designate what attributes are used to alias colleagues, in addition to generated helper methods to establish the association. This ensures that the relationship is always correct and explicit in specifications.

In Java, the annotation ***collegial*** $T.f$ is used before the type of an attribute which should alias a colleague of type T, which will alias the attribute's receiver through its attribute f. The attribute $T.f$ must then have the corresponding annotation stating that it is related to this attribute. Figure 5.2 demonstrates this annotation in user for the list and iterator example.

To state this relationship between CoJava types, the specification predicate **collegial** is used instead. The statement **collegial**$(\tau.f, \tau'.f')$ asserts that the type τ and τ' are colleague types. An instance a of type τ will alias one or more instances of type τ' through its attribute f, and each of these instance of τ' will alias a through their f' attributes.

5.2.1 Relationship Forms

The list and iterator example demonstrates a one-to-many relationship, where a list may have many iterators but an iterator only one list. The type *Set* is used to store a list of colleague references. In Java this type is the standard definition present in the library, but in the CoJava formalization it is treated as a defined type which may be traversed by the quantifier predicates in specifications and has methods for querying membership.

Collegial relationships can be one-to-many, such as the list-iterator example, one-to-one, and many-to-many. In Java the basic form for two collegial types A and B is one of the following:

- One-to-One:

```
class A { private collegial B.a B b; ... }
class B { private collegial A.b A a; ... }
```

- One-to-Many:

```
class A { private collegial B.a Set b; ... }
class B { private collegial A.b A a; ... }
```

- Many-to-Many:

```
class A { private collegial B.a Set b; ... }
class B { private collegial A.b Set a; ... }
```

A type may be related to more than one other through applications of the Colleague Technique. A type may also be related to itself, ie. if A and B were the same type, and where one or more attributes are collegial. If related through one attribute then this is paired with itself obviously, but if a type is collegial with itself through multiple attributes then each will be paired with the one other such attribute. Since collegial attributes are paired, there must be an even number of such attributes if there is more than one. This allows the doubly-linked list definition in Section 2.3.2, where nodes are related to each other through *next* and *prev* attributes.

Constraining aliasing is still important when ownership and colleagues are used together. Transitive owners must be responsible for ensuring the constraints between the objects it owns, otherwise invariant unsoundness is introduced. Similarly if a and b were colleagues of one another, the owner of a must be careful to which clients it allows b to be exposed.

5.2.2 Formalization in CoJava

The specification function $\mathbf{colleague}(\tau, f)$ maps a type-attribute pair (τ, f) to the type-attribute it is declared to be collegial with, or $\varnothing$ if f is not collegial. This function takes the place in CoJava of the annotation ***colleague*** *T.f* which has been shown in the previous Java examples.

The **collegial** predicate is defined for two types dcl and dcl' which are not owned, and attributes which either store a reference to their respective collegial types or reference a regular *Set* instance:

$$\frac{\begin{array}{l}\mathbf{ftype}(P, dcl, f) = dcl' \vee \mathbf{ftype}(P, dcl, f) = Set \\ \mathbf{ftype}(P, dcl', g) = dcl \vee \mathbf{ftype}(P, dcl', g) = Set \\ \mathbf{colleague}(dcl, f) = (dcl', g) \\ \mathbf{colleague}(dcl', g) = (dcl, f)\end{array}}{P, \mathbf{colleague} \vdash \mathbf{collegial}(dcl.f, dcl'.g)}$$

A statement **associate** is defined to create the relationship. With the form $\mathbf{associate}(x.f, y.g);$, it will ensure that objects x and y alias one another through their attributes f and g. It naturally requires that $\mathbf{collegial}(\tau.f, \tau'.g)$ where x has type τ and y type τ':

$$\frac{\begin{array}{l}\Gamma(x) = \tau \\ \Gamma(y) = \tau' \\ P, \mathbf{colleague} \vdash \mathbf{collegial}(\tau.f, \tau'.g)\end{array}}{P, \mathbf{colleague} \vdash \mathbf{associate}(x.f, y.g);} \quad \text{WF\_Associate}$$

Similarly a **dissociate** statement is defined with the same well-formedness criteria:

$$\frac{\begin{array}{l}\Gamma(x) = \tau \\ \Gamma(y) = \tau' \\ P, \textbf{colleague} \vdash \textbf{collegial}(\tau.f, \tau'.g)\end{array}}{P, \textbf{colleague} \vdash \textbf{dissociate}(x.f, y.g);} \quad \text{WF\_Dissociate}$$

Two pseudo-statements, only used when reducing **associate** and **dissociate**, add or remove a reference to a collegial attribute. Each has two definitions, one for when the attribute is a singleton and the second for when its type is *Set*:

$$\frac{\begin{array}{l}L(x) = oid \\ L(y) = v \\ H(oid) = \tau \\ \textbf{ftype}(P, \tau, f) \neq Set\end{array}}{(P, L, H, \textbf{add}(y, x.f);\ \overline{s'_l}^{\,l}) \longrightarrow (P, L, H[(oid, f) \mapsto v], \overline{s'_l}^{\,l})} \quad \text{SR\_Add1}$$

$$\frac{\begin{array}{l}L(x) = oid \\ L(y) = v \\ H(oid) = \tau \\ \textbf{ftype}(P, \tau, f) = Set \\ x' \neq \mathrm{dom}(L)\end{array}}{(P, L, H, \textbf{add}(y, x.f);\ \overline{s'_l}^{\,l}) \longrightarrow (P, L[x' \mapsto H(oid, f)], H, x'.add(y);\ \overline{s'_l}^{\,l})} \quad \text{SR\_Add2}$$

$$\frac{\begin{array}{l}L(x) = oid \\ L(y) = v \\ H(oid) = \tau \\ \textbf{ftype}(P, \tau, f) \neq Set\end{array}}{(P, L, H, \textbf{rem}(y, x.f);\ \overline{s'_l}^{\,l}) \longrightarrow (P, L, H[(oid, f) \mapsto \textbf{null}], \overline{s'_l}^{\,l})} \quad \text{SR\_Rem1}$$

$$\frac{\begin{array}{l}L(x) = oid \\ L(y) = v \\ H(oid) = \tau \\ \textbf{ftype}(P, \tau, f) = Set \\ x' \neq \mathrm{dom}(L)\end{array}}{(P, L, H, \textbf{rem}(y, x.f);\ \overline{s'_l}^{\,l}) \longrightarrow (P, L[x' \mapsto H(oid, f)], H, x'.remove(y);\ \overline{s'_l}^{\,l})} \quad \text{SR\_Rem2}$$

The **associate** reduces to two **add** statements, and similarly the **dissociate** reduces to two **rem** statements:

$$\frac{}{(P, L, H, \textbf{associate}(x.f, y.g);\ \overline{s'_l}^{\,l}) \longrightarrow (P, L, H, \textbf{add}(y, x.f);\ \textbf{add}(x, y.g);\ \overline{s'_l}^{\,l})} \quad \text{SR\_Associate}$$

SR_Dissociate

$$(P, L, H, \textbf{dissociate}(x.f, y.g);\ \overline{s'}^{l}) \longrightarrow (P, L, H, \textbf{rem}(y, x.f);\ \textbf{rem}(x, y.g);\ \overline{s'}^{l})$$

The specification predicates **isAssociated**$(x.f, y.g)$ and **isAssociable**$(x.f, y.g)$ test whether the objects x and y are associated through their respective attributes, or if they can become associated. They are used in specifications only to ensure relationships exist or can be formed at the appropriate times. The well-formedness criteria for both are identical to that for **associate** and **dissociate**:

WF_isAssociated

$$\frac{\Gamma(x) = \tau \quad \Gamma(y) = \tau' \quad P, \textbf{colleague} \vdash \textbf{collegial}(\tau.f, \tau'.g)}{P, \textbf{colleague} \vdash \textbf{isAssociated}(x.f, y.g)}$$

WF_isAssociable

$$\frac{\Gamma(x) = \tau \quad \Gamma(y) = \tau' \quad P, \textbf{colleague} \vdash \textbf{collegial}(\tau.f, \tau'.g)}{P, \textbf{colleague} \vdash \textbf{isAssociable}(x.f, y.g)}$$

The evaluation of **isAssociated** checks if y is the value stored by $x.f$ when it is a singleton variable, or if y is contained by $x.f$ when it is a set variable:

E_isAssociated1

$$\frac{L(x) = oid \quad L(y) = oid' \quad H(oid) = \tau \quad \textbf{ftype}(P, \tau, f) \neq \mathit{Set} \quad H(oid, f) = oid'}{[\![\textbf{isAssociated}(x.f, y.g)]\!]_{H,H',L}}$$

E_isAssociated2

$$\frac{L(x) = oid \quad L(y) = oid' \quad H(oid) = \tau \quad \textbf{ftype}(P, \tau, f) = \mathit{Set} \quad x' \notin \mathrm{dom}(L) \quad [\![x'.\mathit{contains}(y)]\!]_{H,H',L[x' \mapsto H(oid,f)]}}{[\![\textbf{isAssociated}(x.f, y.g)]\!]_{H,H',L}}$$

The evaluation of **isAssociable** checks whether singleton variables are null only, since set variables can always have more items added:

E_isAssociable1

$$\frac{L(x) = oid \quad L(y) = oid' \quad H(oid) = \tau \quad \textbf{ftype}(P, \tau, f) \neq \mathit{Set} \quad \textbf{ftype}(P, \tau', g) \neq \mathit{Set} \quad H(oid, f) = \textbf{null} \quad H(oid', g) = \textbf{null}}{[\![\textbf{isAssociable}(x.f, y.g)]\!]_{H,H',L}}$$

$$\frac{\begin{array}{l} L(x) = oid \\ L(y) = oid' \\ H(oid) = \tau \\ \mathbf{ftype}(P, \tau, f) \neq Set \\ \mathbf{ftype}(P, \tau', g) = Set \\ H(oid, f) = \mathbf{null} \end{array}}{[\![\mathbf{isAssociable}(x.f, y.g)]\!]_{H,H',L}} \text{E\_isAssociable2}$$

$$\frac{\begin{array}{l} L(x) = oid \\ L(y) = oid' \\ H(oid) = \tau \\ \mathbf{ftype}(P, \tau, f) = Set \\ \mathbf{ftype}(P, \tau', g) \neq Set \\ H(oid', g) = \mathbf{null} \end{array}}{[\![\mathbf{isAssociable}(x.f, y.g)]\!]_{H,H',L}} \text{E\_isAssociable3}$$

$$\frac{\begin{array}{l} L(x) = oid \\ L(y) = oid' \\ H(oid) = \tau \\ \mathbf{ftype}(P, \tau, f) = Set \\ \mathbf{ftype}(P, \tau', g) = Set \end{array}}{[\![\mathbf{isAssociable}(x.f, y.g)]\!]_{H,H',L}} \text{E\_isAssociable4}$$

5.3 Mirror Invariants

Given two colleague types A and B, a mirror invariant is derived from the original invariant of A and is integrated into the specification for B. This mirror invariant states the same property relating the two types together but from the perspective of B rather than A. Perspective in this sense means which type definition the predicate is defined in, and so what the type of *this* is. The mirror invariant will use the identifier *this* where variables referring to instances of B were used in the original, and use variables referring to instances of A where *this* was originally used. By swapping around which object in the relationship is *this*, the predicate retains essentially the same form and the same logical implication.

To generate a mirror invariant from an original invariant, a predicate stating only the logical conditions that relate one type to another is first defined. This is derived from the whole or a part of the original invariant and restated in terms of free variables representing instances of the types, rather than *this* and an attribute. This can then be recast again as the mirror invariant, replacing these free variables with *this* and attribute or quantifier variable names.

Only those conjuncts of the invariant defining some property of a colleague object are considered. Other conjuncts might state properties of local state, such as owned objects, which do not contribute to the definition of relationships between objects.

The process of deriving a mirror invariant will be defined here abstractly in terms of colleague types A and B. For each conjunct considered from the invariant of A, a predicate I is extracted

which states how the current object relates to instances of B. A generalized predicate P is derived from this which states this constraint in terms of instances of A and B, instead of in terms of *this* and an attribute. This is then restated again as the invariant of B, where *this* is used to access members of B rather than those of A, and attribute or variable names used to refer to the colleague instances of A.

The predicate I is a local predicate, since it is part of an invariant predicate which must be local. It may not depend on objects aliased through regular references other than colleague objects. Additionally it may only relate one colleague object to the current object, and so cannot define a relationship between three or more colleague types. These are important restrictions which ensure that I relates the internal state of exactly two objects.

The conjuncts of the invariant of A under consideration will have a particular form depending on the type of the attribute b, upon which the definition of I depends. It is assumed that the conjuncts that comprise CoJava invariants will be in one of these forms:

- When b is a singleton then the invariant is I itself.
- When b is a set, then the invariant is $\forall\, i : B \mid b.contains(i) \bullet I$ or $\exists\, i : B \mid b.contains(i) \bullet I$, where b has no free occurrences in I.

Note that the same predicate I is derived from both existential and universal quantifiers. A universal quantifier defines a relationship which must be true for all colleagues, but an existential quantifier defines one which need be true for at least one. Since it's not reasonable for the colleague object to know if it is the one which must fulfill the relationship, I is derived in a way which implies the property must hold for all colleagues.

Given I derived from these predicates, the definition of P substitutes *this* for a free variable representing an instance of the type it was defined in, and the attribute or quantifier variable substituted for a instance of the collegial type. This introduces two variables named *aa* and *bb* which must be fresh in terms of I. Although the variables used to represent the two objects are changed, the logical meaning of the predicate remains the same and well-typed:

$$P(aa : A, bb : B) = I[aa/this, bb/this.b, bb/i]$$

For example, the P predicate derived from *ListIterator* would be the following:

$$P(i : ListIterator, l : List) = l.size() >= i.last$$

The predicate P captures the relationship between two objects, irrespective of what variables through which the two relevant objects are referred. In particular P isolates the constraint between the instances of the two types. Given two collegial objects a and b, $P[this/aa, this.b/bb]$ states the same constraint between a and b as $P[this.a/aa, this/bb]$, since whichever object *this* refers to does not change the described property.

Consider an invariant of a with a sub-expression *this.m()* which will become *aa.m()* in P. The substitution $P[this.a/aa, this/bb]$ in the context of b produces $this.a.m()$. Since *this* in the context of a refers to the same object as *this.a* in the context of b, *this.m()* states the same condition as *this.a.m()*, therefore they are logically equivalent. This follows also for all forms of method calls and attribute accesses, leading to a generalized rule that $P[this/aa, this.b/bb] \Leftrightarrow P[this.a/aa, this/bb]$ given colleague objects a and b.

Using the predicate P, the mirror invariant can be derived for type B. The attribute aliasing the colleague of b, $b.a$, will be either be a single reference or a set of references. In either case a mirror invariant for B can now be derived:

- If a is a single reference: $P[this.a/aa, this/bb]$
- If a is a set of references: $\forall\, i : A \mid this.a.contains(i) \bullet P[i/aa, this/bb]$

This results in the following invariant for the *List* type given the definition above for P:

$$\forall\, i : ListIterator \mid iterators.contains(i) \bullet this.size() >= i.last$$

This mirror invariant is then integrated into the specification of B and thus becomes an obligation its members must adhere to. The process can be applied to the mirror invariant itself and will yield the original invariant as a result. This shows that the mirror invariant defines the same semantic property as the original except from the perspective of the other colleague type.

The added specification predicate for this invariant would be defined in CoJava as the following:

```
forall( ListIterator i ; iterators.contains(i) ; this.size() >= i.last)
```

5.3.1 Relationship With Global Invariants

In particular applying the method to either the original invariant conjunct or the mirror invariant produces the same P predicate. P is independent of type definition and states a constraint between all the instances of A and B. Instead of defining the constraint between A and B in terms of type invariants, a global invariant can be defined which asserts P for all relevant object pairs:

- If a and b are singleton attributes:

$$\forall\, ag : A, bg : B \mid ag.b = bg \wedge bg.a = ag \bullet P[ag/aa, bg/bb]$$

- If a is a singleton attribute and b a set attribute (or vice versa):

$$\forall\, ag : A, bg : B \mid ag.b.contains(bg) \wedge bg.a = ag \bullet P[ag/aa, bg/bb]$$

- If a and b are set attributes:

$$\forall\, ag : A, bg : B \mid ag.b.contains(bg) \wedge bg.a.contains(ag) \bullet P[ag/aa, bg/bb]$$

For example, the following global invariant restates the correctness property a list and its iterators must ensure:

$$\forall\, i : ListIterator, l : List \mid i.list = l \wedge l.iterators.contains(i) \bullet l.size() >= i.last$$

A global invariant of this form is not practical to check at runtime or statically prove to hold when given code is executed. It does however state the meaning of collegial invariants which define a constraint for all relevant pairs of objects at runtime. Mirror invariants are described here as being constructed from an original type invariant, but this has the same effect as deriving two type invariants from one global invariant of this form.

Through either approach, the effect is to produce type invariants that ensure this global property is maintained in any correct system. Global invariants often are ambiguous about which objects are responsible for ensuring their constraints always hold. The Colleague Technique restricts what properties an invariant can state, hence limiting what kind of global properties they imply. Specifically the technique requires predicates to only constrain two objects at once, that is the two colleagues, and then assign responsibility to those objects alone.

5.3.2 Self Colleagues

A type can be a colleague of itself and through the same attribute. This allows recursive types to define sound invariants, although care must be taken to define symmetric properties. For example, a *Person* type may require that spouses share a surname:

```
class Person {
  private collegial Person.spouse Person spouse;
  private String surname;

  invariant eqSurname(spouse.getSurname());

  public Person() { super(); }

  public pure local String getSurname() { return surname; }
  public pure local boolean eqSurname(String s) { return surname.equals(s); }
}
```

The given invariant yields the mirror invariant *spouse.eqSurname*(*getSurname*()). Since this invariant will be integrated into the specification of the original type, both the original and mirror invariants are now present in the same specification. By definition the mirror invariant states the same property, thus this inclusion is redundant and self colleagues do not need mirror invariants generated for them to ensure soundness.

It is in fact the case that invariants of such self-colleagues must define symmetric properties. For a given colleague variable *i*, the predicate *P(this,i)* will be part of the original invariant, and so *P(i,this)* will be part of the mirror invariant. The final invariant of the type will assert both, hence the property must be symmetric if a contradiction is not to result.

For an example of a contradictory invariant, the *Person* class is modified to record age and the requirement that every person's spouse be older than themselves:

```
class Person {
  private collegial Person.spouse Person spouse;
  private int age;

  invariant this.age < spouse.age;
...
}
```

This results in a mirror invariant *spouse.age* $<$ *this.age* which in conjunction with the original produces a contradiction. Self colleague types thus may only define relationships between objects which are symmetric, even if a type is defined to be collegial through multiple attributes. For example, a node in a doubly-linked list which records its position in the list may state an invariant requiring its position to be one more than the previous and one less than the next:

```
class DequeNode {
  private collegial DequeNode.prev DequeNode next;
  private collegial DequeNode.next DequeNode prev;
  private int pos;

  invariant pos > 0;
  invariant pos == next.pos - 1;
  invariant pos == prev.pos + 1;
...
}
```

The mirror invariant derived from the first invariant predicate is $prev.pos = pos - 1$, the same property asserted by the second invariant predicate. The mirror invariant of the second predicate also yields the first, hence there is no need to modify this specification to ensure a sound invariant.

5.4 List and ListIterator Example

The CoJava version of the *List* type using the Colleague Technique is given in Figure 5.3 with the *ListIterator* type given in Figure 5.4. This version also incorporates ownership into the specification which can be used soundly in conjunction with Colleagues.

The attribute *iterators* stores the set of references to the iterators traversing the a list instance. Since the relationship between lists and iterators is one-to-many, this attribute will have type *Set*. The CoJava-specific annotation, *collegial*, declares that *iterators* is a collegial attribute linked to the *list* attribute of the type *ListIterator*. This states that all the iterators aliased in *iterators* will reference the current list instance in their *list* attributes. It implies that *iterators* is not meant to alias an instance of *Set* external to the current object, but is a special attribute which code cannot assign to nor use its reference value. The tool will generate code to instantiate it in the constructor. The *list* attribute in *ListIterator* must similarly have a *collegial* annotation stating the other half of the bi-directional property.

The method *iterator()* demonstrates how to create a collegial relationship. Since the *List* instance cannot pass the *this* reference to the iterator, it must create it first and then associate with it. The *setLast()* method is used to set the *last* value, which should be done after the association is created, since *last* must be initialized to 0.

Although this special mechanism is used to create and manage relationships between objects, the instances of these two types will of course still behave like regular objects. Although interfaces are not part of CoJava, *ListIterator* could reasonably implement *Iterator* in the full Java language, with *List* implementing *Iterable*. Enhanced for-loops could thus make use of these types:

```
List l = new List();
...
for(Object o : l)
  println(o);
```

The approach taken with the standard library in Java is to disallow any modification to a list such (as *java.util.ArrayList*) which is being traversed. CoJava's approach with colleagues is more liberal in that items can be added to and removed from the list so long as a minimum size is retained. If an *Iterator* instance is used explicitly, it's *remove()* method is the only way to remove items from the list without an exception being thrown. Java's specification indicates that this method will remove an item if possible, throw an exception, or possibly do nothing at all.

```
contained class List implements Iterable{
  private owned ArrayList items;
  private collegial ListIterator.list Set iterators;

  public List() { items = new owned ArrayList(); }

  ensures items.contains(o);
  public void add(Object o) { items.add(o); }

  requires i >= 0 && i < size();
  ensures \result == items.get(i);
  public pure Object get(int i) { return items.get(i); }

  requires i >= 0 && i < size();
  ensures !items.contains(\old(items.get(i)));
  public void remove(int i) { items.remove(i); }

  ensures \result == items.contains(o);
  public pure local boolean contains(Object o) { return items.contains(o); }

  ensures \result == items.size();
  public pure local int size() { return items.size(); }

  ensures isAssociated(this.iterators,\result.list);
  public pure ListIterator iterator() {
    ListIterator li = new ListIterator();

    associate(this.iterators,li.list);
    li.setLast(size());
    return li;
  }
}
```

Figure 5.3: The List Example Type Using Colleagues

With the Colleague Technique there is an explicit connection in the specification between lists and iterators, and query methods can be defined which clients may call to determine when certain operations are allowable. *ListIterator* could thus supply a *remove()* method which decrements *last* and removes an item from the list when possible, and so correctly producing the behaviour expected of *Iterator* objects. For example, a *isRemovable()* method added to *List* will indicate when the list has more items than any iterator expects, and thus when *remove()* can actually remove items rather than do nothing:

```
ensures \result ==
        (\forall ListIterator li ; iterators.contains(li) ; size() > li.getLast());
public pure boolean isRemovable() {
  boolean isLarger = true;
  for(ListIterator li : iterators)
    isLarger = isLarger && size() > li.getLast();
  return isLarger;
}
```

This method also reduces the need for clients to keep track of what iterators are current traversing certain lists. It provides the minimal information needed to know that items from the end of the list

```
contained class ListIterator implements Iterator{
  private collegial List.iterators List list;
  private int position, last;

  invariant position <= last;
  invariant list.size() >= last;

  ensures this.last == last;
  public void setLast(int last) { this.last = last; }

  ensures \result == last;
  public pure getLast() { return last; }

  requires list!=null;
  ensures \result == (position<last);
  public pure local boolean hasNext() { return position < last; }

  requires hasNext();
  ensures position == \old(position) + 1;
  ensures \result == list.get(\old(position));
  public Object next() {
    position = position + 1;
    return list.get(position -1);
  }

  public void remove() {}
}
```

Figure 5.4: The ListIterator Example Type Using Colleagues

can be removed, and other methods can be added to indicate other properties of the relationships between lists and iterators.

5.5 Proof of Invariant Soundness

Definition 4.3.1 for invariant soundness allows invariant predicates to rely on owned objects, or additionally immutable objects in Java. A new definition permits the use of colleague objects in invariants while still maintaining soundness:

Definition 5.5.1 (Colleague Invariant Soundness)
A sound invariant may only rely on owned objects or reference values themselves. Invariant predicates are therefore pure and local, with the added requirement that quantifiers have variables with immutable or owned types as well as pure-local sub-predicates.

In addition to this, colleague references may be used as receivers for pure-local method calls, but no for other purpose. The quantifier variables which range over colleague set attributes may have regular object type, but they also may only be used as receivers for pure-local method calls.

This redefinition of sound invariants does permit non-local predicates since colleague objects,

which are not owned, may be used as receivers for methods. However these methods are always pure and local, thus the predicate does not rely on non-owned objects other than the receiver colleagues. Sound invariants therefore still cannot rely on the state of Non-owned and non-collegial objects. This also prevents an invariant from surreptitiously constraining multiple colleagues together since pure-local methods may not rely on a third colleague for their result value.

To prove that this defines sound invariants, it must be shown that no correct operation on an object b produces a state which does not satisfy the invariant of another object a if it was satisfied before the operation. If b is owned, it must be shown that any invariant relying on it is defined for its owners, of which a is one. If b is a colleague of a, then the invariant of b prevent any operation on it which breaks the invariant of a from being considered correct. These two cases are considered here.

5.5.1 Owned

The object a can depend on b for its invariant constraints if it owns b. It has been shown in the previous chapter that if b is aliased through an owned reference, then no non-owned references exist to it. Consequently, only a and the owners of a can alias b.

If only a aliases b, then b is modified exclusively by the methods of a. Since these methods are required to re-establish a's invariant once they complete, they must as part of this requirement ensure that b is in a state satisfying the invariant.

If owners of a acquire a reference to b, then they become transitive owners of b. These objects can only access b through a and so the relationship between the two would be known. Although b may now be modified directly and not through a method call to a, the obligation to ensure that these modifications satisfies a's invariant can be reasonably placed on these transitive owners.

Although the owners of a did not instantiate b, it is considered part of their internal representation since they do own it. Both a and b are part of their owner's state and therefore any correctness requirements they have are also correctness requirements of any object owning both. For example, an owner of a *ListCounter* class which defines its internal list as an owned object is required to ensure the counter's invariant is satisfied at all visible states:

```
class CounterOwner {
  private owned ListCounter counter;
  private owned IntegerList list;

  public CounterOwner(){
    counter = new owned ListCounter(10);
    list = counter.getHistory();
  }
}
```

Here the relationship between *list* and *counter* objects is known through construction, although stating so explicitly in an invariant predicate *list == counter.getHistory()* may be done. This would aid the analysis of the methods of *CounterOwner* when verifying that they establish the invariant of *counter* in their postconditions.

The correctness of an object depends on the correctness of its internal representation, so the invariants of any owned object become implicit in that of their owners. An invariant condition, *(\forall Integer i ; list.contains(i) ; i.intValue() <counter.get())*, may be added to *CounterOwner* so that the correctness requirements of its representation are made explicit.

Invariants relying on owned objects thus are sound because of two crucial properties: owned objects are only ever aliased and thus directly modified by owners, and the correctness of owned objects becomes part of the total correctness criteria for their owners. No object which does not have a correctness concern with an owned object may modify it directly. Thus only objects whose correctness and implicit invariants depend on an owned object may modify it directly, and hence they must be responsible for that object's correctness as much as their own.

5.5.2 Colleague

When considering that a and b are colleagues of one another, and neither is owned, then their clients do not have a correctness concern. No invariant depends on them, and their relationship is not explicit through the type system. Instead of relying on their clients being concerned for the invariant of a, which relies on b, the objects must rely on each other's correctness to ensure the relationship is correct.

The state of b will not satisfy the invariant of a if it is modified in a certain way. To prevent this, the correct states of b as defined by its invariant must satisfy the invariant of a. For all program states H, L which are visible states for a or$b$, the following must be shown to be true if the type of a is A and the type of b is B:

$$\forall\, H, L \mid L(a) = oid \wedge L(b) = oid' \bullet \mathbf{isAssociated}(a.f, b.g) \Rightarrow \\ (H, H, L[\texttt{this} \mapsto oid] \vdash \mathbf{inv}(A) \Leftrightarrow H, H, L[\texttt{this} \mapsto oid'] \vdash \mathbf{inv}(B))$$

Given that I represents the portion of $\mathbf{inv}(A)$ which relies on b through the attribute f, the predicate P is derived. This in turn is used to generate the mirror invariant I_m, which then becomes part of the invariant of b. In the context of their respective objects, I and I_m imply one another:

$$\forall\, H, L \mid L(a) = oid \wedge L(b) = oid' \bullet \mathbf{isAssociated}(a.f, b.g) \Rightarrow \\ (H, H, L[\texttt{this} \mapsto oid] \vdash I \Leftrightarrow H, H, L[\texttt{this} \mapsto oid'] \vdash I_m)$$

This is the case since I and I_m represent the same predicate which rely on the members of the same object. In the formal syntax for CoJava predicates, methods in invariants have the for $\mathbf{this}.meth(\overline{prv})$ or $\mathbf{this}.x.meth(\overline{prv})$. A method in I of a which relates it to b through the attribute f has the form $\mathbf{this}.meth(\mathbf{this}.f)$. This predicate becomes $\mathbf{this}.g.meth(\mathbf{this})$ in I_m, however due to the collegial relationship between a and b, it is known that **this** in I and $\mathbf{this}.g$ in I_m both store a reference to a, and $\mathbf{this}.f$ in I and $\mathbf{this}.g$ in I_m both store a reference to b. Consequently, the method *meth* is being called with the same receiver with the same argument in either predicate.

Since the syntax of predicates in CoJava is simplified and only includes compound predicates or method calls of this form, this exchanging of receiver and argument variables are the only translations applied to produce I_m from I, besides converting to or from quantifier statements. Since this translation represents the same property, I and I_m state the same constraint on the program, but from the perspective of the two colleague objects.

The predicate $\mathbf{inv}(A)$ is composed of I plus constraints on the encapsulated members of the type A, while $\mathbf{inv}(B)$ must be defined to be composed of I_m plus other constraints. Since it is established that I and I_m imply one another, and the added constraints are the responsibility of a and b exclusively, it is the case that if one invariant is true for a visible state of a or b then this implies the other is true also.

For example, the invariant of the *ListIterator* type can be stated as such:

$$\textbf{this}.list.size() \geq \textbf{this}.last$$

This can be reformulated in CoJava given a method *greaterThanLast*(*Listi*) representing the expression $i \geq \textbf{this}.last$:

$$\textbf{this}.greaterThanLast(\textbf{this}.list)$$

The mirror version of this, which would appear in the *List* type, is quantifier over the members of *iterators*:

$$forall(ListIterator\ i;\ this.iterators.contains(i);\ i.greaterThanLast(\textbf{this}))$$

This predicate ranges over all possible iterators for a list. Given a list *list* and iterator *iter* whose invariant was satisfied by a particular visible state, one value for i will be *iter* for the invariant of *list*, hence the predicate **this**.*greaterThanLast*(*this.list*) in the invariant for *iter* represents the same method being called with the same receiver and argument as i.*greaterThanLast*(**this**) in the invariant for *list*.

5.5.3 Conclusion

Thus an invariant which relies on owned or collegial objects is sound by the definition above. Additionally, if immutable objects can be shown to never change state over their lifetimes, any property established for such objects will continue to hold, and hence any invariant can rely on them for a constraint without the possibility of them being modified and violating that constraint. Within the formalization of CoJava it is not possible to define immutable types because there exists no access control for class members, however in Java immutable types can be defined and are in fact integral to the language. The formalization of the Colleague Technique can be extended to Java, with only slight extensions to the mechanism used to generate mirror invariants, thus this definition of invariant soundness can be applied to the full language.

5.6 Related Work

The closest parallel to the Colleague Technique is the Friends [17] approach applied to the Boogie [13, 14] object-oriented verifier used with Spec# [15]. Friends recognizes the unsoundness problem of invariants depending on non-owned objects. The proposed solution expands on Boogie's use of auxiliary variables [108] which reflect abstract properties about objects, such as whether they are consistent in regards to their invariants or not and their ownership status. These variables are not concrete necessarily but used by the verifier in its correctness proof.

Friends adds an additional set of variable to objects which have clients depending on them for their invariant conditions. This is similar to the set collegial attributes used with Colleagues. The relationship between types is defined with a "friend" clause which states what type is the friend of the type the clause appears in and what attributes it is given permission to read. New pseudo-statements are introduced which create and break the friend relationship between objects at runtime. The friend type defines a kind of history constraint [80, 81] which describes how the type a friend depends on can update itself in a safe manner. With these invariants of friend types can soundly

rely on objects they do not own, and the objects the invariants depend on have a notion of how they may be safely updated.

Friends differs from Colleagues first in that it's formulated around the Boogie approach to specification and verification, and relies on many new introduced concepts. Collegial attributes are regular concrete variables which do not necessarily need to contribute to the structure of a program, only define relationships which exist through some mechanism. This represents a halfway approach between concrete attributes and model/auxiliary variables used only in specifications, whereas Friends relies on the auxiliary variables and the Boogie concept of invariants which departs slightly from classical DbC reasoning. The Colleague Technique relies on regular attributes and classical DbC concepts and reasoning, hence is a more portable and general approach.

The way in which Friends constructs relationships, through "attach" and "detach" pseudo-statements, is also not easily applied outside of the Boogie approach. The "friends" and history "guard" clauses also add more complexity to the specification and require greater effort on the part of the programmer to correctly implement. The use of history constraints however does make it clearer how objects can correctly update themselves, in contrast to the Colleague Technique which requires this to be inferred from the generated mirror invariant. In general though the Colleague Technique represents a portable approach based on regular invariants and methods generated by the tool which can be used with existing formal technique, like static reasoning or runtime checks.

Visibility-based [94] approaches weaken the ownership properties to a degree to allow certain invariants to be sound which otherwise would not be. This particular approach allows peers within the Universes [93] ownership type system, which are objects sharing the same owner, to define invariants relying on each others fields so long as these invariants are visible to any methods which would update such fields.

Visibility-based invariants of this sort relate objects within the same layer in object structures at the cost of added proof obligations, which may become quite onerous with increasing complexity. The approach in [94] specifically describes allowing an object X to rely on the attribute $Y.f$ if f is assigned to only by methods in the context of X, that is by the methods of objects which have the same owned as X. This implies that the invariant can only rely on these attributes of non-owned objects and that these objects must have the same owner, that is be in the same layer.

The Colleague Technique by comparison introduces no special proof obligations other than the requirement to ensure invariants hold at the appropriate times. Fewer proof obligations are introduced as complexity increases, and invariants can soundly rely on attributes and methods of non-owned objects. Two colleagues also need not share an owner, such that an owned *List* object may be a colleague of a non-owned *ListIterator* object which any client can alias.

Spacially separate objects in running systems can be reasoned with using separation logic [105]. This logic introduces a disjoint connective operator $P * Q$ which guarantees that P and Q are predicates describing disjoint memory locations (disjoint heaps). This is useful for reasoning with object-oriented languages by providing modular reasoning such that what memory locations an operation affects is clearly understood [42, 99]. The relationship between distinct objects can be defined in terms of disjoint heaps and abstract predicates which represent more vaguely-defined constraints on memory locations.

However this does not provide significant advantages in defining sound invariants between objects. If a mirror invariant is necessary to ensure that the original invariant is sound, then separation logic does not necessarily help in identifying when this is or what the mirror invariant is. To specify Java programs using separation logic would require using the disjoint operator to separate predicates,

however since JML is not intended to always convey separation information it prevents such an approach from being compatible with existing annotated code and JML-based techniques.

Booster [37], based on Z [113], B [1], and Refinement Calculus [91], describes a formal object-based language which includes associations between objects, and association invariants, as core features of system construction. Relationships between objects in Booster are often bi-directional, such that a *Reader* which references a *Book* through an attribute must itself be referenced by the *Book*. It is an invariant property of the system that these relationships be maintained, and so operations which affect relationships are checked to ensure that they do so and additional operations generated when needed.

The concepts of bi-directionality and the responsibility two parties on such relationships have to one another is very closely paralleled with the Colleague Technique. Objects in Booster do not represent mutable state, and the language itself is meant for model-driven development rather than being a coding language like Java. However the importance of relationships between objects, and analyzing their operations [45] to ensure relationships are correctly maintained, is shared by both. Being a model driven approach Booster does not have a specification language for its definitions nor is it suited as a specification language itself, thus the Colleague Technique represents an effective means of adopting some of the same ideas into object-oriented specification.

5.7 Conclusion

This chapter has defined and discussed the Colleague Technique for invariants between objects. When an object can be encapsulated, ownership can be applied so that an invariant can soundly rely on it. When ownership cannot be used in this way, the Colleague Technique allows objects to soundly rely on each other for their invariant conditions. Objects so related operate together as single modules at the expense of greater object coupling, stronger invariant conditions, and the need to explicitly create and manage relationships.

The technique is comprised of a mechanism for generating additional invariant conditions to ensure soundness, as well as constructing and managing relationships between objects. The tool-generated code implementing these features uses regular Java methods and JML invariants, such that existing tools which accept JML-annotated Java as input can accept types which use the technique. This flexibility allows existing reasoning techniques to be used with the generated invariants, since these are standard JML predicates. Without the Colleague Technique, these techniques could not soundly reason about the invariants the technique does allow. In particular the approach uses concrete attributes which store concrete references as opposed to using ghost or auxiliary variables to define the relationship between colleagues. This allows reasoning and correctness techniques which cannot operate with such specialized specification concepts, such as simple runtime assertion checking, to be correctly applied to collegial types.

Chapter 6

Testing Java Programs

Testing Java programs using the CoJava Tool is discussed in this chapter. One key property of ownership and the Colleague Technique is that they can be represented using either no specification at all, or as standard invariant and contract conditions, respectively. The CoJava Tool accepts a small subset of Java annotated with custom specification elements, checks the requirements for ownership and the Colleague Technique, and produces Java code with standard JML specifications. This output code can be used with tools that accept JML-annotated Java code, hence existing techniques can be used with CoJava ownership and the Colleague Technique without adaptation.

Checking contracts at runtime, or Runtime Assertion Checking (RAC), is a relatively simple method of using contracts to directly aid the development of programs. If nothing else a specification is precise and thorough documentation for a system, but tools and techniques can use them to test and analyze programs so that correctness is more easily achieved. The grand challenge [59] is verification at the point of compilation, so that when a correct program is compiled it can be shown to be definitely correct in terms of its specification. This is still a hard challenge to meet, but less rigorous and exacting methodologies can still exploit formal specifications. Static analysis, including type checking, can be used to enforce simpler correctness properties, such as encapsulation in the case of ownership.

As a testing methodology, checking contracts at runtime effectively aids debugging programs by asserting invariants and contract predicates when the appropriate operations are performed. While running debug versions of programs as part of testing, many defects can be discovered simply by localizing when and why contracts do not hold during a particular program run. This is not a complete verification methodology since a program run with no contract violations has only shown that its particular configuration and input was correct. In practice this will still catch many errors, both in the code and the contracts themselves, and represents a more robust and thorough form of testing.

When performing runtime assertion checking in the manner of Eiffel [88, 89] or the code produced by *jmlc* [18, 29], an invariant is evaluated as an expression in the context of the current object. These are expected to be pure expressions having no ostensible side-effects on the state of the system, but otherwise they are still subject to the same semantics as expressions in method bodies. Method contracts similarly introduce their own challenges, but are also pure expressions checkable when methods begin and complete execution.

The CoJava Tool produces output code which is acceptable as input for the *jmlc* tool. The invariants generated as part of the Colleague Technique can be correctly checked at runtime to test

the properties the technique promises to uphold. However *jmlc* itself produces debug programs which are very large and slow compared to what standard Java compilers would produce. As an alternative, the CoJava Tool also generates AspectJ [72] aspects which implement a simpler form of RAC testing. The tool will generate aspects which implement invariant and contract checks when weaved with the normally generated code. Contract checking at runtime has been shown to be a cross-cutting concern [71] which validates this approach, as demonstrated by other RAC tools based on aspects [46, 84, 85, 86, 104, 121].

This chapter discusses how the tool generates code for the Colleague Technique and the aspects to test it. The aspect-based approach has been much studied but the CoJava version represents a simple and elegant implementation, and includes special features for checking active object contracts which will be discussed briefly. Section 6.1 discusses how the CoJava tool generates output code given input in the form of a Java subset slightly larger than CoJava itself. Section 6.2 describes the CoJava runtime assertion checking methodology based on aspects which the tool is responsible for generating. Section 6.3 outlines briefly an active object-based concurrency model currently a topic of experimentation and how checking contracts for active objects introduces certain problems and behaviours.

6.1 Generating Code

Ownership has an advantage as a correctness technique in that it is a purely static concept. Being entirely a type system implies that there is no need to produce support code to implement it. The Colleague Technique on the other hand requires the CoJava Tool to generate mirror invariants and helper methods to manage the relationships between objects. The aspects implementing the RAC technique used with CoJava must also be generated. The final output from the tool is Java code derived mostly from the input code but with added specifications which represent collegial invariants and helper methods, along with the RAC aspects.

The steps comprising the operations of the CoJava Tool can be summarized as follows:

1. Read input source code and auxiliary types
2. Type check input to enforce ownership and Colleague Technique requirements
3. Calculate mirror invariants
4. Generate output Java source code
5. Generate AspectJ RAC aspects
6. Compile code, optionally with aspects weaved in

The formalized CoJava language uses the predicates **isAssociated** and **isAssociable**, and the statements **associate** and **dissociate**, to manage collegial relationships. The CoJava Tool must generate correct Java code however, so methods with differing names and signatures are used instead. These are intended to be used in the input code as expressions in contracts and bodies of methods. Given the type A with a collegial attribute called b, the following methods with JML specifications are added to the type:

- *isAssociated_b*: This is used to query whether the given object is associated with the receiver through attribute *b*. If *b* is a singleton attribute, then the expression *X* is $b == i$, otherwise *X* is *b.contains(i)*.

```
private behaviour ensures \result == X ;
public final pure boolean isAssociated_b(B i){ return X ; }
```

- *isAssociable_b*: This is used to query whether an association with the given object can be made. The expression *X* is either $b == null$ or *!b.contains(i)*.

```
private behaviour ensures \result == (i != null && X);
public final pure boolean isAssociable\_b(B i){ return i != null && X ; }
```

- *associate_b*: This creates the association between the receiver and the object *i*. It will perform the correct operation on *b* and call the correct method of *i* to create the bi-directional aliasing relationship. Note that the given specification must be respected like any other method.

```
requires this.isAssociable_b(i) && i.isAssociable_a(this);
ensures this.isAssociated_b(i) && i.isAssociated_a(this);
public final void associate_b(B i) { ... }
```

- *dissociate_b*: This breaks the relationship, requiring that it existed in the first place. It will also perform the correct operations on the receiver and *i* to end the association.

```
requires  this.isAssociated_b(i) && i.isAssociated_a(this);
ensures !this.isAssociated_b(i) && !i.isAssociated_a(this);
public final void dissociate_b(B i) { ... }
```

Additionally, every colleague type will have a method *isAssociated(Object i)* generated for it which returns true if the receiver is associated with *i* through any collegial attribute. No methods are present to query the values of collegial attributes however so that the restrictions stated in the previous chapter are adhered to. Other additional methods may also be generated to implemented those given here.

The generated code for the *List* type, initially defined in Figure 5.1, is given in Figure 6.1. Methods present in the original definition are reproduced here with slight modifications to add necessary specification elements. All methods after the second definition of *iterator()* are generated by the tool to implement the Colleague Technique. These are responsible for creating, managing, and destroying the bi-directional relationships the technique relies on.

6.2 Aspect-based Runtime Assertion Checking

The CoJava tool generates aspects with inter-type methods to check invariants and around advice to check contracts. Inter-type methods are new methods added to types when the aspect code is weaved with the original code, in this instance they are used to contain the code to check invariants when needed. Around advice in this context are code blocks which replace method calls with code to check preconditions, perform the call, and then check postconditions. The tool produces an AspectJ source file with these components present in a single aspect, which, when weaved with the Java code, will produce an instrumented program. The Java code normally executed will still be present, but the contracts associated with methods will be checked by the weaved aspect code.

```
public class List implements Iterable {
  //@ public invariant (\forall ListIterator i ; iterators.contains(i) ;
size(this) >= i.last);

  private /*@ spec_public  @*/ ArrayList items;
  private /*@ spec_public  @*/ Set iterators;

  public List() { items = new ArrayList(); iterators = new HashSet(); }

  //@ public ensures items.contains(o);
  public void add(/*@ nullable @*/ Object caller, Object o) { items.add(o); }

  /*@ public requires i >= 0 && i < size(this);
    @ public ensures \result == items.get(i); @*/
  public  /*@ pure @*/ Object get(/*@ nullable @*/ Object caller, int i)
    { return items.get(i); }

  //@ public ensures \result == items.size();
  public  /*@ pure @*/ int size(/*@ nullable @*/ Object caller) { return items.size(); }

  /*@ public requires isRemovable(this) && requires i >= 0 && i < size(this);
    @ public ensures !items.contains(\old(items.get(i))); @*/
  public void remove(/*@ nullable @*/ Object caller, int i) { items.remove(i); }

  public /*@ pure @*/ ListIterator iterator(/*@ nullable @*/ Object caller) {
    ListIterator li = new ListIterator();
    associate_iterators(this, li);
    li.setLast(this, size(this));
    return li;
  }

  public ListIterator iterator(){ return iterator(null); }

  //@ public ensures \result == items.contains(o);
  public boolean  /*@ pure @*/ contains(/*@ nullable @*/ Object caller, Object o)
    { return items.contains(o); }

  public final /*@ pure @*/ Iterator get_iterators(/*@ nullable@*/ Object caller)
    { return iterators.iterator(); }

  //@ private behaviour ensures \result == iterators.contains(i);
  public final /*@ pure @*/ boolean isAssociated_iterators
    (/*@ nullable@*/ Object caller,ListIterator i){ return iterators.contains(i); }

  //@ private behaviour ensures \result == (i != null && !iterators.contains(i));
  public final /*@ pure @*/ boolean isAssociable_iterators
    (/*@ nullable@*/ Object caller,ListIterator i){ return i != null && !iterators.contains(i); }

  public final void associatePriv_iterators(ListIterator i) { iterators.add(i); }
  public final void dissociatePriv_iterators(ListIterator i) { iterators.remove(i); }

  /*@ requires isAssociable_iterators(this,i);
   @ requires (!i.isAssociated_list(this,this) ==> i.isAssociable_list(this,this));
    @ ensures isAssociated_iterators(this,i) && i.isAssociated_list(this,this); @*/
  public final void associate_iterators(/*@ nullable@*/ Object caller,ListIterator i)
     { iterators.add(i); i.associatePriv_list(this); }

  /*@ requires isAssociated_iterators(this,i);
    @ requires  (!i.isAssociable_list(this,this) ==> i.isAssociated_list(this,this));
    @ ensures !isAssociated_iterators(this,i) && !i.isAssociated_list(this,this);  @*/
  public final void dissociate_iterators(/*@ nullable@*/ Object caller,ListIterator i)
    { i.dissociatePriv_list(this); iterators.remove(i); }

  public /*@ pure @*/ boolean isAssociated(/*@ nullable@*/ Object caller,Object i)
    { return isAssociated_iterators(caller,(ListIterator)i); }
}
```

Figure 6.1: List Type Generated By The CoJava Tool

The assertion checking components use a standard generated block of code to check predicates, which the following defines in template form with P representing the predicate to check:

```
boolean __c=false;
try { __c = (P); } catch(Throwable t) { throw new ContractEvalException(...); }
if(!__c)throw new PreconditionException(...);
```

The type *PreconditionException* is thrown when P is a precondition, otherwise *PostconditionException* or *InvariantException* would be used as appropriate. *ContractEvalException* indicates an error has occurred when evaluating P. These are all subtypes of *RuntimeException* which allows them to be thrown even when not mentioned in the enclosing method's *throws* clause.

6.2.1 Checking Invariants

Given a type C, an inter-type method called *CInvariant()* is created which contains the code for checking that type's invariant predicates. A type D that subtypes C will have a method *DInvariant()* which calls *CInvariant()* before performing its own checks.

Each class will also have an inter-type method called *checkInvariant()* which calls the appropriate invariant checking method. An interface *InvariantObject* is also defined with this method which every class is declared to implement. This allows one advice block to be defined which checks the invariant before and after every method call. If the aspect being generated is called A, then this advice would be produced as such, where the pointcut *colleagueHelper()* matches helper methods used by the Colleague Technique:

```
Object around(InvariantObject obj) : this(obj) && execution(* *(..)) &&
    !cflow(call(void *Invariant(..))) && !colleagueHelper() && !within(A)
{
  obj.checkInvariant();
  Object __result=proceed(obj);
  obj.checkInvariant();
  return __result;
}
```

A pointcut defines a point in execution in which advice can be applied. The pointcut *!cflow(call(void *Invariant(..)))* is used to prevent recursion by not allowing the advice to be applied when within the flow of a method ending with *Invariant*, such as *checkInvariant()* and *CInvariant()*. The pointcut *!within(A)* prevents invariants from being checked when contract expressions are evaluated, and *execution(* *(..))* is used to match any method call.

Together these prevent recursive cases where calling methods in invariant or contract checks initiate a new invariant check, which then repeats indefinitely. They also prevent the recursive case where an invariant check performed on one object initiates a check on its colleagues, which will then call it back and initiate infinite recursion between the two objects.

After advice is also defined which calls the *CInvariant()* method after the constructor for class C completes. When *super()* is called in the body of a constructor, this is treated like an invocation of a constructor. If *checkInvariant()* were called at this juncture, the invariant for the object whose initialization code has not been executed yet would be checked. For example, if the constructor for D had a *super()* call as its first statement, it is not likely that the invariant for D has been established at this point and so should not be checked. However once *super()* completes the invariant for C should hold, and so should be checked.

```
public void List.ListInvariant()
{
  boolean __check;

  // NULL: this.items != null
  try { __check = (this.items != null);
  } catch(Throwable t) { throw new ContractEvalException("cojavatest/ListExample.cojava",
      25,46,t,"this.items != null");}
  if(!__check)throw new InvariantException("cojavatest/ListExample.cojava",
      25,46,"cojava.List","this.items != null");

  // NULL: this.iterators != null
  try { __check = (this.iterators != null);
  } catch(Throwable t) { throw new ContractEvalException("cojavatest/ListExample.cojava",
      26,51,t,"this.iterators != null");}
  if(!__check)throw new InvariantException("cojavatest/ListExample.cojava",
      26,51,"cojava.List","this.iterators != null");

  // INV: this.maxSize < 0 || this.items.size() <= this.maxSize
  try { __check = (this.maxSize < 0 || this.items.size() <= this.maxSize);
  } catch(Throwable t) { throw new ContractEvalException("cojavatest/ListExample.cojava",
      30,28,t,"this.maxSize < 0 || this.items.size() <= this.maxSize");}
  if(!__check)throw new InvariantException("cojavatest/ListExample.cojava",
      30,28,"cojava.List","this.maxSize < 0 || this.items.size() <= this.maxSize");

  // INV: (\forall ListIterator i ; this.iterators.contains(i) ; this.size() >= i.last)
  try {
    boolean quant29101 = true;
    for(Iterator __i0 = this.iterators.iterator(); __i0.hasNext();){
      ListIterator i=(ListIterator)__i0.next();
      if(this.iterators.contains(i))quant29101 = quant29101 && (this.size(this) >= i.last);
    }
    __check = (quant29101);
  } catch(Throwable t) { throw new ContractEvalException("cojavatest/ListExample.cojava<MIRROR>",
      101,29,t,"(\\forall ListIterator i ; this.iterators.contains(i) ; this.size() >= i.last)");}
  if(!__check)throw new InvariantException("cojavatest/ListExample.cojava<MIRROR>",101,29,
      "cojava.List","(\\forall ListIterator i ; this.iterators.contains(i) ; this.size() >= i.last)");

}
```

Figure 6.2: ListInvariant() Generated By The CoJava Tool

Figure 6.2 gives the generated code for *List.ListInvariant()* which would be called by the above around-advice block. Note that this method also checks that the members of the list are not null, since the default JML type system assumes values are non-null unless otherwise designated. CoJava has been discussed in the previous chapters as null by default, hence this check is added. It also includes the code for checking the mirror invariant associated with iterators using a generated for-loop which iterates over the members of the collegial set.

6.2.2 Checking Contracts

Method contracts are checked in around advice blocks, rather than before advice for preconditions and after advice for postconditions as in [10, 85, 86, 104]. Having one piece of advice rather than two reduces the complexity of the generated aspects, and makes the storage of *\old()* values easier. When an expression of the form *\old(E)* is present in a postcondition, a local variable is created in the advice which is assigned *E*. This is then substituted for the original expression in the postcondition predicates.

```
Object around(List obj, Object caller, int i) : this(obj) && args(caller, i) &&
      execution(Object List.get(Object, int))
{
  boolean __check;

  // PRE: i < this.size()
  try { __check = (i < obj.size(obj));
  } catch(Throwable t) { throw new ContractEvalException("cojavatest/ListExample.cojava",
      45,27,t,"i < this.size()");}
  if(!__check)throw new PreconditionException("cojavatest/ListExample.cojava",
      45,27,"cojava.List.get","i < this.size()");

  // PRE: i >= 0
  try { __check = (i >= 0);
  } catch(Throwable t) { throw new ContractEvalException("cojavatest/ListExample.cojava",
      45,17,t,"i >= 0");}
  if(!__check)throw new PreconditionException("cojavatest/ListExample.cojava",
      45,17,"cojava.List.get","i >= 0");

  Object __result = proceed(obj,caller, i);

  // POST: \result == this.items.get(i)
  try { __check = (__result == obj.items.get(i));
  } catch(Throwable t) { throw new ContractEvalException("cojavatest/ListExample.cojava",
      46,22,t,"\\result == this.items.get(i)");}
  if(!__check)throw new PostconditionException("cojavatest/ListExample.cojava",
      46,22,"cojava.List.get","\\result == this.items.get(i)");

  return __result;
}
```

Figure 6.3: Aspect Code For get() Generated By The CoJava Tool

The generated advice to check the pre- and postconditions for the method *Counter.inc()* is given below. Evaluation blocks described above are used to evaluate the pre- and postconditions and throw the appropriate exceptions when necessary. This method returns *void*, however the advice blocks for methods returning values would have a return value derived from *proceed()*. A longer example is given for *List.get()* in Figure 6.3.

```
void around(Counter obj) : this(obj) && execution(void Counter.inc(Object)) {
  boolean __check;

  try { __check = (obj.value < obj.max); }
  catch(Throwable t) { throw new ContractEvalException(...); }
  if(!__check) throw new PreconditionException(...);

  int oldvar0 = obj.value + 1;
  proceed(obj);

  try { __check = (obj.value == oldvar0); }
  catch(Throwable t) { throw new ContractEvalException(...); }
  if(!__check) throw new PostconditionException(...);
}
```

The variable *oldvar0* is used to store the value from an *\old()* expression. Where in the contracts these expressions occur, this variable will instead be used. Similarly a variable is created to store result values from non-void methods. This variable is used in place of *\result* in postcondition expressions. A more complex method of temporary variables and loops is used to evaluate quantifiers.

6.3 Checking Concurrent Contracts

Aspects provide a lightweight runtime assertion checking methodology, which was one of the main goals behind the development of the CoJava Tool. The secondary goal was to develop a RAC approach which was compatible with checking contracts between active objects in a concurrent setting. The standard JML tool *jmlc* produces code which results in deadlock when two colleagues are both active objects and hence exist in their own threads of control. Specific mechanisms are built into the CoJava Tool's generated aspects to tackle this problem.

An instance of the active object design pattern [9, 33, 53, 74, 114], called threaded objects [70], is used to introduce concurrency in CoJava. Threaded objects exist in their own threads of control, allowing them to execute operations asynchronously. A method call to an threaded receiver is represented as a message placed on that receiver's message queue, which the receiver will eventually read and perform the corresponding call. This occurs in a separate thread or process from the caller, who must wait for a response if there is one. Method invocation and execution are thus decoupled, while deadlock and race conditions are tackled through type-based approaches and ownership as described in [70].

A threaded object is created as a special instance of a normal object type, such as *Counter*:

```
threaded Counter c = new threaded Counter(); // create threaded instance of Counter
c.inc(); // call is executed in separate thread
c.add(5); // arguments are sent to receiver with call message
Result r = c.get(); // r will hold response once it is received
```

In this example the keyword **threaded** is used to indicate that *c* exists in its own concurrent context but is defined by the type *Counter*. Method calls to *c* place a message on *c*'s message queue indicating what method was called along with the argument values, rather than calling the method directly. When a method returns a value, an instance of *Result* is created which will receive the value once the receiver produces it, but will in the meantime allow the caller to check to see when this has occurred. If the value does not come within a stated period of time, a timeout exception occurs.

Checking the contracts of threaded objects introduces a few significant problems. Any contract check which mentions a threaded object must query values from that object. For attribute queries this is done through accessor methods which represent sent messages in the same way as normal method calls.

Thus the possibility of timeout has been introduced in the contract check. What timeout means in terms of contracts, i.e. has the contract been satisfied or not when these occur, is a complex issue to address within this simplified notion of concurrency. If the code that checks contracts was to wait indefinitely for a response from a method which never executes, then deadlock will have occurred.

The simple solution is to interpret timeouts as an evaluation exception separate from the notion of contract satisfaction, or to ensure that a contract check can safely wait indefinitely. Ownership is used in CoJava to organize threaded objects into hierarchies, such that an object can only wait indefinitely for a response from a threaded object that is also owned. This can be used in contracts to wait indefinitely for queries from owned threaded objects, but otherwise a more comprehensive analysis of deadlock with threaded objects is needed and would be a topic of future research.

Postconditions also have a different meaning for threaded objects. When the method *add()* is invoked, since it returns no value, the caller does not wait for it to complete before continuing. Its postcondition has thus not necessarily been established yet, so must be considered to be a property

that will be eventually established. Since messages are executed sequentially, a threaded receiver is effectively not accessible until the method call in progress completes.

In the above example, this means that the message for *c.get()* can be sent even if the previous call *c.add(5)* has not yet completed. Since *c* cannot respond to this later call until the first completes, it cannot be accessed in a state in which the postcondition has not yet been established, hence this temporal property of postconditions does not affect correctness. What it does affect is the order in which exceptions may occur, such that a precondition exception for a subsequent call might come before a postcondition exception, that is in the reverse order of the two calls.

Invariants may safely wait indefinitely for responses from owned objects when being checked at runtime, and cannot call methods of non-owned objects at all except colleague objects. The invariants of colleague objects require each to call methods of the other or query its attributes. This implies that one object must communicate with its colleague even while it is checking its own invariant and hence is not responsive.

Without special support, checking collegial invariants at runtime will thus always lead to timeout conditions, or always to deadlock if the two colleagues are allowed to wait indefinitely for responses from one another. This also arises whenever an objects calls a method of its colleague, since that colleague must check its invariant before executing the method to which the calling object cannot respond.

The solution is to identify which messages are part of colleague invariant checks and create separate threads of control to process them while suspending the current operation of the receiver. Figure 6.4 presents the aspects used to identify which messages these are, represented by the type *Message*, and set a flag called *isContractCheck* to true. Threaded objects are implemented with proxies which have a method called *sendMessage()* used to process incoming messages. This is then overridden to create new threads to handle flagged messages, as illustrated in Figure 6.5 which overrides the method for the type *Threaded_ListCounter*, the proxy for the threaded version of *ListCounter*.

When an object calls a method of its colleague, it must be in a visible state and indicate this with an internal boolean variable called *isVisible*. It must then proceed with the call and wait for a response. This implies that, while it waits, no other operations are performed by the caller in anticipation of other messages being concurrently processed as part of an invariant check. This is implemented by an around advice block given in Figure 6.6 for calls from *ListCounter* objects to *IntegerList* colleagues.

6.4 Conclusion

This chapter has discussed checking contracts for Java programs at runtime using aspects generated by the CoJava Tool. Runtime assertion checking is neither sound nor complete but aids the developer in discovering many program defects as part of a testing methodology. A simple approach to implement RAC has been described here using aspects, which also cover specific semantic issues related to active object concurrency. The CoJava Tool is the unified tool which performs type checks to enforce the ownership type system, the requirements for the Colleague Technique, and generate the output code with the augmented specifications and aspect RAC code.

Runtime assertion checking in CoJava involves checking invariant and contract predicates at appropriate points in execution. Since these predicates are defined in terms of Java expressions, they can be evaluated at these points to determine if the program state satisfies them. If they

```
public boolean Message.isContractCheck=false;

public void ThreadedObjectBase.processInvariant(Message m) {
  processMessage(m);
}

void around(ThreadedObjectBase t,Message m) : this(t) && args(m)
    && execution(void sendMessage(Message)) && cflow( call(void *Invariant()) ||
            call(void ThreadedObjectBase.processInvariant(Message)) )
{
  m.isContractCheck=true;
  proceed(t,m);
}

void around(ThreadedObjectBase t,Message m) : this(t) && args(m)
    && if(m.isContractCheck) && !cflow(call(void *.processInvariant(..)))
    && execution(void processMessage(Message))
{
  t.processInvariant(m);
}
```

Figure 6.4: Identifying Contract-checking Messages

```
public synchronized void Threaded_ListCounter.sendMessage(Message m) {

  if(!m.isContractCheck) { // continue normally
    super.sendMessage(m);
    return;
  }

  StringQueue d=(StringQueue)delegate;

  if(isActive && isVisible && m.sender instanceof Threaded_StringIterator &&
      d.isAssociated(null,(Threaded_StringIterator)m.sender))
    threads.activateSingleMessage(this,m); // process concurrent message
  else
    super.sendMessage(m); // continue normally
}
```

Figure 6.5: Executing Concurrent Contract Messages

```
Object around(ListCounter caller, Threaded_IntegerList rec):
      this(caller) && target(rec) && call(* *(..)) &&
      !cflow(call(void *Invariant())) && if(caller.isAssociated(caller,rec))
{
  caller.checkInvariant();
  caller.__thread.isVisible=true;

  Object r=proceed(caller,rec);

  if(r instanceof Result)
    ((Result)r).waitForResult(1000);

  caller.__thread.isVisible=false;

  return r;
}
```

Figure 6.6: Calling a Colleague's Method

are not satisfied, that is they do not evaluate to *true*, then an exception is thrown to indicate an erroneous program state has been reached. This testing methodology can expose many programming errors as contract violations, aiding programmers in identifying where these bugs occur, what objects were involved, and what the bug actually is in terms of contracts.

The subset of Java which the CoJava Tool accepts is larger than CoJava itself, including both class and interface definitions, more complex statements, expressions, and constructors. These added elements can be described in terms of CoJava, thus the formalized techniques described in this thesis are applicable to the Java code the tool accepts and produces. The output from the tool is correct Java which can be compiled with a standard Java compiler, *jmlc* or with the AspectJ compiler. Depending on what type of build the developer wants to produce, this implies that the tool can generate production or debug versions of the same input code.

The next chapter will provide a summation for the topics described in this thesis. It will also give an overview of the possible future research directions with CoJava. Concurrency is a particular topic of future work involving threaded objects, since ownership provides certain advantages in systems using active objects, and a simple type-based approach can be used to prevent data races and many cases of deadlock.

Chapter 7

Conclusion and Further Work

Relationships between objects are very complex but also very interesting aspects of object-oriented systems. It is the co-operation between objects which often defines the most significant part of a program's functions. This thesis has explored a number of relationships between objects and the correctness requirements they impose. A set of techniques applied to object systems help tackle the challenge of specifying and testing these relationships, from enforcing encapsulation through a simple type system to checking invariants between active objects at runtime.

A central focus of this thesis was the soundness of invariants. These predicates capture critical correctness information, and are important to the definition of relationships between objects. The very particular problem addressed was how two objects which may form arbitrary relationships with any others can soundly rely on one another for their invariant conditions. The conceptualization is that these two objects are acting in unison as one module which must provide services to any client, and must do so while these clients respect the present contractual obligations defined in the module's specification. The objective is a methodology where any operation which is correct vis-a-vis a given specification will not produce a post-state for which some invariant ceases to be true.

This final chapter will discuss what ownership and the Colleague Technique provide in terms of an effective Design-by-Contract development methodology. Section 7.1 summarizes the key contributions in this thesis, while Section **??** outlines possible future work with CoJava. In particular this section will discuss the active object-based concurrency model and its advantages, specifically it allows the relationships between threads to be defined by specifications, in a similar manner to how the relationships between objects are defined.

7.1 What CoJava Accomplishes

Ownership and the Colleague Technique describe and involve a number of different object relationship forms. Often one object requires near-exclusive control over another, and so ownership can be used to describe this in sequential and concurrent settings. When multiple objects must co-operate but not with one dominating the other, then the Colleague Technique can be applied.

What CoJava accomplishes with these relationships is to apply stronger correctness criteria where appropriate. Encapsulation is enforced at compile time through these techniques, thus ensuring correctness properties otherwise requiring testing or analysis to guarantee. This in conjunction with the Colleague Technique allow invariants to soundly rely on objects for their conditions, which can then be checked at runtime as part of a thorough and robust testing regime. Ideally, such invariants

and the rest of the specifications which they are part of would be used to formally verify CoJava programs whereby correctness can be statically assured.

The relationships and the methodology used to address their correctness concerns are summarized as such:

- **Object and Internal Representations**:

  An object's internal representation is the structure of objects which define its internal components. For a linked list type this would be the nodes in the chain rather than the values they store. It is important that these representation objects remain contained within the object whose structure they comprise, that is encapsulation must be enforced. Invariants should also be correctly applied to these objects to ensure the structure remains correct.

  Ownership is the type based approached used to enforce this property. Chapter 3 defines the type system used with CoJava which enforces encapsulation as well as defining an owner-owned relationship allowing owners to rely on owned objects for their invariant conditions. A slight restriction to ownership allows objects to be owned transitively. Additional correctness obligations are applied to transitive owners to ensure the relationship between multiple owned objects remains correct.

- **Co-Operative Objects**:

  Two objects co-operate when they are part of the same module or otherwise have been linked together to achieve some common task. They may need to predicate invariant conditions on one another, thus necessitating some technique to ensure this is done soundly. Without such a technique, one partner may not be aware of the constraint placed upon it by the other object's invariant.

  The Colleague Technique allows two objects of this sort to co-operate correctly. Lists and iterators traversing them are the clear examples of this situation discussed thoroughly in this thesis. The relationship between the two types is made explicit and bi-directional. An additional invariant states the constraint between the instances of the types in the specification of the list type. This allows lists to be aware of when iterators require certain basic correctness properties of it, thus operations which would break the necessary constraint will no longer be considered correct in terms of the specification.

- **Instances of Super- and Subtypes**:

  The relationship between instances of a type, those of its subtypes, and all other objects is also addressed. It is a relationship primarily between types, encompassing how substitutability, inherited owned members, and the Colleague Technique interact. Since a subtype must be substitutable to be correct, then the relationship its instances have with other objects should be very similar to that which instance of the original type have.

  Ownership enforces a property which persists through inheritance. A subtype cannot expose an object to external clients through an inherited owned member, thus internal representation objects remain internal regardless of what object they are found in. Colleagues also define a relationship between types that subtypes must respect, since substitutability requires they respect inherited invariants.

The final result of this work is a definition of a sound invariant, one which may be predicated only on owned and collegial objects. With a sound invariant, the relationship between objects can

be correctly described in significant detail, and correctly maintained if contracts are respected. If an invariant is not sound, then the way in which it describes a relationship may not be adequate to ensure the relationship is maintained, even if all operations are correct in terms of the specification.

7.2 Future Work

Many drawbacks are evident in the present approach with CoJava. The ultimate aim is to extend the techniques to the full Java language, so the constraints of the syntax and formal definition of CoJava are not relevant. Many features of Java absent in this thesis can be used to produce more generic and usable versions of ownership and the Colleague Technique. Ensuring correctness has been discussed here only so far as runtime assertion checking, however many techniques and technologies can be leveraged to prove stronger correctness properties, both for passive and threaded objects. This section represents work not accomplished in the time allotted for a thesis of this sort, as well as aspirations for the future of the formal techniques described herein.

7.2.1 Ownership

CoJava's ownership type system is very simple but consequently very limited. It is probably the most minimal set of constraints necessary to enforce encapsulation statically, consequently it imposes a very rigid structure on owned objects. It also lacks features other type systems have to provide flexibility, such as the conversion between owned references and read-only references, separate and definable ownership contexts with differing access rights, and other features. The approach other type systems use is to provide a variety of type annotations which serve differing purposes without violating encapsulation [5, 41, 97]. The Spec# [16, 108] approach describes ownership in a more dynamic way which isn't rigidly attached to type.

As discussed briefly, static analysis can be used to recognize when the CoJava ownership rules can be safely broken. A series of operations which individually may violate encapsulation properties can be considered together and shown through data flow analysis to ensure ownership properties are maintained. Other type annotations similar to those in the work cited here could be introduced to describe freshly created objects, argument objects lent to a method which may not be shared, objects with unique references, and other properties. These annotations can be integrated into a static analysis approach which guarantees that objects referenced through owned variables certainly are owned.

7.2.2 Abstract Specifications

CoJava's specification approach has been kept simple for the sake of brevity and to focus on the subjects of ownership, Colleagues, and threaded objects. Many useful specification features present in JML, such as frame property [21, 22, 78] specification and model variables [77], are omitted here but which can be used to produce more abstract specifications using Colleagues and ownership.

In particular there is an inability in CoJava to abstractly specify types using the Colleague Technique, in particular the absence of interfaces does not allow collegial types to be defined as modular abstract components. With the language described in this thesis, a pair of collegial classes must be extended to achieve reuse, whereas specifying the collegial relationship in interfaces would be a much more modular solution. This would require defining model variables representing the necessary colleague attributes, as well as analysis to guarantee these model variables are directly

```
interface Subject {
  protected model collegial Observer.subject Set<Observer> observers;

  requires !isUpdating();
  ensures !isUpdating();
  public void update();

  public pure local boolean isUpdating();
}

interface Observer {
  protected model collegial Subject.observers Set<Subject> subjects;

  requires subjects.contains(\caller);
  requires ((Subject)\caller).isUpdating();
  public void notify();

  requires isAssociable(s);
  ensures isAssociated(s);
  public void setSubject(Subject s);

  requires isAssociated(subject);
  ensures !isAssociated(\old(subject));
  public void clearSubject();
}
```

Figure 7.1: The Subject-Observer Interface Definition

implemented by attributes of the same type to ensure the technique's consistency requirements. The generated helper methods would also need this information so that they can interact with the correct concrete attributes in the implementation class.

For example, an abstract definition of the Subject-Observer example using interfaces and model variables would look something like that given in Figure 7.1. Note that this example also uses generics in the collegial set attributes, as well as the *\caller* special value which aliases a method's caller. The keyword *model* is used to designate model variables which are abstract and considered not to be real members of the interface, in the same sense as JML which would declare these in specification comments. These interfaces allow classes to implement them and thus to implement the subject and observer behaviour, with the generated helper methods of the Colleague Technique providing the infrastructure to correctly manage the relationship between them. Further work to extend the technique to interfaces and model variables is thus needed to be able to define such abstract specifications of co-operative types.

Additionally the *\caller* value has been the subject of some experiment with the CoJava Tool, however it requires more work in defining its semantics and applicability. It has promise as a means of defining the relationship between objects and the methods they may call, as the *Observer* example here demonstrates, and is a practical extension when performing runtime assertion checking.

7.2.3 Generics and Admissibility

One the most signification features CoJava lacks is Java generics. Without generics it is difficult to specify type-safe data structures, but more crucially it is very difficult to specify general purpose

classes which can be used to instantiate passive and threaded instances. For a method to be admissible, its arguments and return value must have admissible types, which are types whose instances can be safely shared by threaded objects without incurring data races. Any class must then define admissible methods if they are to be accessible in threaded instances, however a general purpose data structure which stores *Object* instances cannot do this since *Object* is not admissible.

Generics would allow proxies to be generated for instances of generic types with admissibility determined by the type's generic parameters. For example, the methods of *List<T>* which involve the type T would be admissible in instances where T was replaced by an admissible type. Thus *threaded List<String>* would have admissible methods allowing *String* instances to be added and queried, but *threaded List<Object>* would not.

This introduces a particular problem with relationships between such data structures and their iterators. The method *List.iterator()* would not normally be admissible since the iterator it produces would not have admissible type. A number of solutions are possible, such as defining *iteratorThreaded()* methods which would produce threaded iterators, or requiring iterators to be processed by the same thread as the data structure so that data races do not occur. Iterators would still be required to produce admissible values, but this latter approach ensures correctness at the cost of greater complexity in terms of the underlying implementation.

A generic data structure which can hold regular or threaded objects would be necessary to avoid implementing separate container types for both. Additionally, a data structure which can store owned objects of either sort would be desirable, so allowing method calls with owned arguments must somehow be shown to be correct. An *inert* reference type is possible, which may be stored but cannot be used to access an object's members. A data structure storing inert references will not interact with these objects, thus it will not violate encapsulation nor introduce races and deadlock.

Inert references would be applied to generic type parameters only, such that *List<inert T>* means any reference stored in the list of type T is inert. Non-inert references being passed into the list become inert but become non-inert again (with type T) when accessed. For example, *list.add(c);* converts c into an inert value the list stores, while *Object o = list.get();* queries a value from the list but converts it to a regular reference. The inert property is more than read-only by allowing no member access at all. It consequently permits data structures to store either passive or threaded objects with introducing concurrency errors.

7.2.4 Active Objects

The threaded objects model has been briefly described in the previous chapter. In its essential form it represents an instance of the active object design pattern with added features:

- Data races are prevented by disallowing the sharing of mutable data. Method calls may only be made if the arguments are primitive, immutable, serializable, or threaded values themselves. This constrains what objects are useful as threaded objects, as well as being incompatible with generic threaded objects.
- Deadlock is prevented by the use of the *Result* type, an instance of promise objects [79]. When a method is called, a *Result* instance is produced which will store the result once it is sent back to the caller, but also gives the caller information about whether the call has completed or not. When the result value is queried, the method call blocks for a specified finite period of time waiting for the value if it hasn't arrived yet. Once this period has elapsed then **null** is returned or an exception is thrown.

- Ownership is used to organize threaded objects into hierarchies. This prevents circular call chains, so an owner can call a method on an owned threaded receiver and wait indefinitely for a response without risking deadlock.

This simple concurrency methodology provides these stated advantages, but is a work in progress requiring further refinement. As a relational mechanism, it characterizes the relationship between threads as one between objects, and applies Design-by-Contract as a means of defining and testing this relationship.

7.2.5 Deadlock-free Communication

The use of *Result* represents a compromise approach in CoJava necessary due to the lack of exceptions and a means of specifying timeout values. A more ideal implementation of threaded objects would use futures [9, 27, 28, 83, 120] implicitly with separate static methods to state a desired timeout value. This would allow methods of threaded receivers to return the actual object type they were declared to return, or numeric types such as *Integer* in place of primitive types.

Timeout events and error results can be described as exceptions if they are included in the concurrency model. If a method returns an *int* value then its invocation on a threaded receiver would produce a subtype of *Integer* acting as a future object. This object would receive any responses and throw exceptions as appropriate. When the value's members are queried in any way, this would initiate waiting for the response in much the same way as *objectResult()*. As a subtype of various unrelated objects, it would have to employ object delegation to include the infrastructure to implement this.

Static methods would be used to query or set properties normally done through *Result*. *Threaded.hasCompleted(Object)* would accept these futures as arguments and returns true if they indicate their respective calls have completed. Another method *Threaded.waitFor(Object)* would be needed in order to pause until the given future receives a result, as well as *Threaded.setTimeout(int)* to specify a timeout value for any threaded calls initiated by the current thread.

For example, with these additions the *Counter* example would be recast as the following:

```
threaded Counter c = new threaded Counter(10);
Void v = c.add(5);
Threaded.waitFor(v);

try {
  Threaded.setTimeout(100);
  System.out.println("Counter: "+c.get());
} catch(ThreadedExeption e1) { ... }
catch(TimeoutExeption e2) { ... }
```

Instead of using timeout values and exceptions to indicate such events, formal approaches can be applied to demonstrate that a particular CoJava program is deadlock-free even in the presence of indefinite waiting. CSP [58] represents an obvious choice to model threaded objects due to the conceptualization of such objects communicating through channels [8, 26, 101], and the correlation between method calls and state transitions. This formal definition of concurrent systems can be used in conjunction with model checking [106, 107]

Applying this formal method to object-oriented systems is no small task and no work has been undertaken with CoJava to do so. Other model checking approaches like SPIN [62, 63] may be more

suited to threaded objects, but it remains a significant piece of further research not covered in the scope of this thesis.

7.2.6 Distributed Objects

Since threaded objects do not share mutable state, in theory they can exist in different processes on different physical machines and still communicate. This strategy is used by various frameworks [53, 9] to implement distributed computing.

The basic approach is to connect to a remote host and ask for a named object. What is actually instantiated on the requesting end is a proxy with the same interface as a local threaded object. The behaviour of this proxy is much the same, with the necessities of admissibility and timeout events still present. Consider a remote *Counter* example:

```
threaded Counter c = Counter.getRemoteObject(url,port,objectname);
System.out.println(c.get()); ...
```

This requires generating two proxies for every possible type, a local and remote threaded wrapper with the correct functionality. A means of organizing providers of objects into directories of some sort is also needed so that clients can find what objects are available from where. If a client shares a local threaded object with a remote one, then a remote proxy of some sort must be created so that the client can become a provider of objects as well. When connections are lost or transfer rates become too slow, a mechanism to cleanly indicate an object is no longer functional (ie. not connected to the real object anymore) is needed, as well as a protocol to communicate data between client and provider. Implementing this cleanly and seamlessly with the existing threaded object structure would provide significant advancement over existing techniques which require explicit programming.

7.3 Conclusion

This thesis has described progress towards formally specifying the relationships between objects and between components. Ownership and the Colleague Technique describe those relationships relating to encapsulation and behaviour. In this chapter, a brief overview of what has been accomplished and what lies ahead has been given. CoJava is not a practical tool for real-world systems development, but the techniques it uses are powerful and applicable if supported with the right tools and methodologies.

The critical conclusion to be reached is that the relationships between objects have significant implications often beyond the obvious as defined in standard contracts or invariants. They require special support to ensure their soundness, and to disallow those relationships which lead to erroneous states. With the techniques discussed in this thesis, advancements can be made toward more practical and effective formal software engineering methodologies for developing Java and other object-oriented systems.

Appendix A

Lightweight Java Type Information Definitions

This appendix defines predicates and functions representing type information extracted from a Lightweight Java program. The rules of well-formedness, subtyping, and other formal definitions use them to represent information about types and whole programs.

Given a program P, the predicate **distinct_names** states that all the classes it defines have distinct names:

$$\frac{P = \overline{cld_k}^k \quad \overline{\textbf{class\_name}(cld_k) = dcl_k}^k \quad \textbf{distinct}(\overline{dcl_k}^k)}{\textbf{distinct\_names}(P)}$$

Given a program, a context ctx, and a class name dcl, **find_cld** represents the class definition in the program's class list with that name, or no element ($\varnothing$) if no class with the name exists:

$$\frac{}{\textbf{find\_cld}([], ctx, dcl) = \varnothing} \qquad \frac{P = cld : \overline{cld} \quad cld = \textbf{class } dcl \textbf{ extends } cl\ \{\overline{fd}\ \overline{meth\_def}\}}{\textbf{find\_cld}(P, ctx, dcl) = (ctx, cld)}$$

$$\frac{P = cld : \overline{cld} \quad cld = \textbf{class } dcl' \textbf{ extends } cl\ \{\overline{fd}\ \overline{meth\_def}\} \quad dcl \neq dcl' \quad \textbf{find\_cld}(\overline{cld}, ctx, dcl) = ctxcld_{opt}}{\textbf{find\_cld}(P, ctx, dcl) = ctxcld_{opt}}$$

find_type defines the $ctx.cld$ type definition, as well the type of the class hierarchy root, `Object`:

$$\frac{}{\textbf{find\_type}(P, ctx, \texttt{Object}) = ctx.\texttt{Object}}$$

$$\frac{\textbf{find\_cld}(P, ctx, dcl) = \varnothing}{\textbf{find\_type}(P, ctx, dcl) = \varnothing} \qquad \frac{\textbf{find\_cld}(P, ctx, dcl) = (ctx', cld)}{\textbf{find\_type}(P, ctx, dcl) = ctx'.dcl}$$

path_length defines the path length in the inheritance tree from a type to `Object`:

$$\frac{}{(P, ctx, \texttt{Object}, 0) \in \textbf{path\_length}} \qquad \frac{\begin{array}{l}\textbf{find\_cld}(P, ctx, dcl) = (ctx', cld) \\ \textbf{superclass\_name}(cld) = cl \\ (P, ctx', cl, nn) \in \textbf{path\_length}\end{array}}{(P, ctx', dcl, nn + 1) \in \textbf{path\_length}}$$

The predicate $\textbf{acyclic\_clds}(P)$ states that the inheritance relation between classes defined in P is acyclic, such that no class inherits in anyway from itself:

$$\frac{\forall\, ctx\ \ dcl \bullet \textbf{find\_cld}(P, ctx, dcl) \neq \varnothing \Rightarrow (\exists\, nn \bullet (P, ctx, dcl, nn) \in \textbf{path\_length})}{\textbf{acyclic\_clds}(P)}$$

$\textbf{find\_path\_rec}(P, ctx, cl, [])$ represents a list of *ctxcld* pairs corresponding to the class definition for *cl* and every supertype up to `Object`:

$$\frac{}{\textbf{find\_path\_rec}(P, ctx, \texttt{Object}, \overline{ctxcld}) = \overline{ctxcld}}$$

$$\frac{(\neg\, \textbf{acyclic\_clds}(P)) \lor \textbf{find\_cld}(P, ctx, dcl) = \varnothing}{\textbf{find\_path\_rec}(P, ctx, dcl, \overline{ctxcld}) = \varnothing}$$

$$\frac{\begin{array}{l}\textbf{acyclic\_clds}(P) \\ \textbf{find\_cld}(P, ctx, dcl) = (ctx', cld) \\ \textbf{superclass\_name}(cld) = cl \\ \textbf{find\_path\_rec}(P, ctx, cl, \overline{ctxcld} \frown [(ctx', cld)]) = \overline{ctxcld}_{opt}\end{array}}{\textbf{find\_path\_rec}(P, ctx, dcl, \overline{ctxcld}) = \overline{ctxcld}_{opt}}$$

$\textbf{find\_path}(P, ctx, cl)$ is simply shorthand for $\textbf{find\_path\_rec}(P, ctx, cl, [])$:

$$\frac{\textbf{find\_path\_rec}(P, ctx, cl, []) = \overline{ctxcld}_{opt}}{\textbf{find\_path}(P, ctx, cl) = \overline{ctxcld}_{opt}}$$

A second definition relates P and a type to list of class definitions:

$$\frac{}{\textbf{find\_path}(P, ctx.\texttt{Object}) = []} \qquad \frac{\textbf{find\_path}(P, ctx, dcl) = \overline{ctxcld}_{opt}}{\textbf{find\_path}(P, ctx.dcl) = \overline{ctxcld}_{opt}}$$

Given a list of class definitions, **fields_in_path** represents the collection of the attribute names of all the class definitions:

$$\frac{}{\textbf{fields\_in\_path}([]) = []}$$

$$\frac{\begin{array}{l}\textbf{class\_fields}(cld) = \overline{cl_j\ f_j\ ;}^{\,j} \\ \textbf{fields\_in\_path}(\overline{ctxcld_k}^{\,k}) = \overline{f} \\ \overline{f}' = \overline{f_j}^{\,j} \frown \overline{f}\end{array}}{\textbf{fields\_in\_path}((ctx, cld) : \overline{ctxcld_k}^{\,k}) = \overline{f}'}$$

fields is the collection of all defined and inherited attributes for a given type τ in program P:

$$\frac{\textbf{find\_path}(P, \tau) = \varnothing}{\textbf{fields}(P, \tau) = \varnothing}$$

$$\frac{\begin{array}{l}\textbf{find\_path}(P, \tau) = \overline{ctxcld} \\ \textbf{fields\_in\_path}(\overline{ctxcld}) = \overline{f}\end{array}}{\textbf{fields}(P, \tau) = \overline{f}}$$

methods_in_path represents the collected method definitions derived from the class definitions in the given list:

$$\frac{}{\textbf{methods\_in\_path}([]) = []}$$

$$\frac{\begin{array}{l}\textbf{class\_methods}(cld) = \overline{method\_def_l}^{\,l} \\ \overline{method\_def_l = cl_l\ method_l(\overline{vd}_l)\{meth\_body_l\}}^{\,l} \\ \textbf{methods\_in\_path}(\overline{cld_k}^{\,k}) = \overline{meth}' \\ \overline{meth} = \overline{meth_l}^{\,l} \frown \overline{meth}'\end{array}}{\textbf{methods\_in\_path}(cld : \overline{cld_k}^{\,k}) = \overline{meth}}$$

methods produces all the inherited and defined methods in the type τ as defined by program P:

$$\frac{\begin{array}{l}\textbf{find\_path}(P, \tau) = \overline{(ctx_k, cld_k)}^{\,k} \\ \textbf{methods\_in\_path}(\overline{cld_k}^{\,k}) = \overline{meth}\end{array}}{\textbf{methods}(P, \tau) = \overline{meth}}$$

Given a list of attribute declarations and a attribute name f, **ftype_in_fds** attempts to determine the type of f as indicated by its entry in the list. If f is not in the list then **ftype_in_fds** results in $\varnothing$, if the type f is supposed to have does not exist in P then the result is $\bot$:

$$\frac{}{\textbf{ftype\_in\_fds}(P, ctx, [], f) = \varnothing}$$

$$\frac{\textbf{find\_type}(P, ctx, cl) = \varnothing}{\textbf{ftype\_in\_fds}(P, ctx, cl\ f : \overline{fd_k}^{\,k}, f) = \bot}$$

$$\frac{\textbf{find\_type}(P, ctx, cl) = \tau}{\textbf{ftype\_in\_fds}(P, ctx, cl\ f : \overline{fd_k}^{\,k}, f) = \tau}$$

$$\frac{\begin{array}{l}f \neq f' \\ \textbf{ftype\_in\_fds}(P, ctx, fd_2 \ldots fd_k, f') = \tau_{opt}^{\bot}\end{array}}{\textbf{ftype\_in\_fds}(P, ctx, cl\ f : \overline{fd_k}^{\,k}, f') = \tau_{opt}^{\bot}}$$

Given a list of types representing the inheritance hierarchy from the first to `Object` and an attribute name f, **ftype_in_path** represents the type of f as it is declared in one of the class definitions, or $\varnothing$ if no type can be found.

$$\frac{}{\textbf{ftype\_in\_path}(P, [], f) = \varnothing}$$

$$\frac{\textbf{class\_fields}(cld) = \overline{fd} \quad \textbf{ftype\_in\_fds}(P, ctx, \overline{fd}, f) = \bot}{\textbf{ftype\_in\_path}(P, (ctx, cld) : \overline{ctxcld_k}^k, f) = \varnothing}$$

$$\frac{\textbf{class\_fields}(cld) = \overline{fd} \quad \textbf{ftype\_in\_fds}(P, ctx, \overline{fd}, f) = \tau}{\textbf{ftype\_in\_path}(P, (ctx, cld) : \overline{ctxcld_k}^k, f) = \tau}$$

$$\frac{\textbf{class\_fields}(cld) = \overline{fd} \quad \textbf{ftype\_in\_fds}(P, ctx, \overline{fd}, f) = \varnothing \quad \textbf{ftype\_in\_path}(P, \overline{ctxcld_k}^k, f) = \tau_{opt}}{\textbf{ftype\_in\_path}(P, (ctx, cld) : \overline{ctxcld_k}^k k, f) = \tau_{opt}}$$

ftype represents the type of the attribute f as declared in τ or in one of its supertypes:

$$\frac{\textbf{find\_path}(P, \tau) = \overline{ctxcld} \quad \textbf{ftype\_in\_path}(P, \overline{ctxcld}, f) = \tau'}{\textbf{ftype}(P, \tau, f) = \tau'}$$

Given a list of methods and a method name, **find_meth_def_in_list** is the definition for the method with that name as given in the list, or $\varnothing$ if the method is not in the list:

$$\frac{}{\textbf{find\_meth\_def\_in\_list}([], meth) = \varnothing}$$

$$\frac{meth\_def = cl\ meth(\overline{vd})\{meth\_body\}}{\textbf{find\_meth\_def\_in\_list}(meth\_def : \overline{meth\_def_k}^k, meth) = meth\_def}$$

$$\frac{meth\_def = cl\ meth'(\overline{vd})\{meth\_body\} \quad meth \neq meth' \quad \textbf{find\_meth\_def\_in\_list}(\overline{meth\_def_k}^k, meth) = meth\_def_{opt}}{\textbf{find\_meth\_def\_in\_list}(meth\_def : \overline{meth\_def_k}^k, meth) = meth\_def_{opt}}$$

Given a list of types representing the inheritance hierarchy from the first to `Object` and a method name *meth*, **find_meth_def_in_path** represents the definition of the method with this name as declared in one of the classes in the list, or $\varnothing$ if no definition can be found:

$$\frac{}{\textbf{find\_meth\_def\_in\_path}([], meth) = \varnothing}$$

$$\frac{\begin{array}{l}\textbf{class\_methods}(cld) = \overline{meth\_def} \\ \textbf{find\_meth\_def\_in\_list}(\overline{meth\_def}, meth) = meth\_def\end{array}}{\textbf{find\_meth\_def\_in\_path}((ctx, cld) : \overline{ctxcld_k}^k, meth) = (ctx, meth\_def)}$$

$$\frac{\begin{array}{l}\textbf{class\_methods}(cld) = \overline{meth\_def} \\ \textbf{find\_meth\_def\_in\_list}(\overline{meth\_def}, meth) = \varnothing \\ \textbf{find\_meth\_def\_in\_path}(\overline{ctxcld_k}^k, meth) = ctxmeth\_def_{opt}\end{array}}{\textbf{find\_meth\_def\_in\_path}((ctx, cld) : \overline{ctxcld_k}^k, meth) = ctxmeth\_def_{opt}}$$

Given a type and a method name, **find_meth_def** represents the method definition for the method with that name as defined in the given type or one of its supertypes:

$$\frac{\textbf{find\_path}(P, \tau) = \varnothing}{\textbf{find\_meth\_def}(P, \tau, meth) = \varnothing}$$

$$\frac{\begin{array}{l}\textbf{find\_path}(P, \tau) = \overline{ctxcld} \\ \textbf{find\_meth\_def\_in\_path}(\overline{ctxcld}, meth) = ctxmeth\_def_{opt}\end{array}}{\textbf{find\_meth\_def}(P, \tau, meth) = ctxmeth\_def_{opt}}$$

mtype represents the type of the method *meth* as declared in τ or one of its supertypes:

$$\frac{\begin{array}{l}\textbf{find\_meth\_def}(P, \tau, meth) = (ctx, meth\_def) \\ meth\_def = cl\ meth(\overline{cl_k\ var_k}^k)\{meth\_body\} \\ \textbf{find\_type}(P, ctx, cl) = \tau' \\ \overline{\textbf{find\_type}(P, ctx, cl_k) = \tau_k}^k \\ \pi = \overline{\tau_k}^k \to \tau'\end{array}}{\textbf{mtype}(P, \tau, meth) = \pi}$$

Bibliography

[1] J.-R. Abrial. *The B Book: Assigning Programs to Meanings.* Cambridge University Press, Aug. 1992.

[2] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, and P. H. Schmitt. The KeY Tool. Technical report in computing science no. 2003-5, Department of Computing Science, Chalmers University and Göteborg University, Göteborg, Sweden, Feb. 2003.

[3] J. Aldrich. Using Types to Enforce Architectural Structure. In *WICSA '08: Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture (WICSA 2008)*, pages 211–220, Washington, DC, USA, 2008. IEEE Computer Society.

[4] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *in ECOOP 2004 – Object-Oriented Programming*, pages 1–25, 2004.

[5] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations For Program Understanding. In *In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 311–330. ACM Press, 2002.

[6] P. S. Almeida. Balloon Types: Controlling Sharing of State in Data Types. *Lecture Notes in Computer Science*, 1241:32, 1997.

[7] J. Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, July 2007.

[8] M. Atkins, R. Pike, and H. Trickey. *The Inferno Programming Book: An Introduction to Programming for the Inferno Distributed System.* John Wiley & Sons, 2005.

[9] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, Jan. 2006.

[10] S. Balzer, P. T. Eugster, and B. Meyer. Can Aspects Implement Contracts? In *In: Proceedings of RISE 2006 (Rapid Implementation of Engineering Techniques)*, pages 13–15, 2006.

[11] J. A. Bank, B. Liskov, and A. C. Myers. Parameterized Types and Java. In *In Principles of Programming Languages (POPL*, pages 132–145, 1997.

[12] J. Barnes. *High Integrity Ada: The Spark Approach.* Addison-Wesley, 1997.

[13] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *FMCO 2005*, LNCS. Springer, 2006.

[14] M. Barnett, R. DeLine, M. Fahndrich, K. Rustan, M. Leino, and W. Schulte. Verification of Object-oriented Programs With Invariants. *Journal of Object Technology*, 0(0):1–30, 2003.

[15] M. Barnett, R. DeLine, B. Jacobs, M. Fhndrich, K. R. M. Leino, W. Schulte, , and H. Venter. The Spec# Programming System: Challenges and Directions. In *VSTTE 2005*, 2005.

[16] M. Barnett, K. R. M. Leino, K. Rustan, M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

[17] M. Barnett and D. Naumann. Friends Need a Bit More: Maintaining Invariants Over Shared State. In D. Kozen and C. Shankland, editors, *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings*, volume 3125 of *Lecture Notes in Computer Science*, pages 54–84. Springer, 2004.

[18] A. Bhorkar. A Run-Time Assertion Checker For Java Using JML. Technical report, Iowa State University, 2000.

[19] D. Björner and C. B. Jones. *The Vienna Development Method: The Meta-Language*, volume 61 of *Lecture Notes in Computer Science*. Springer Verlag, 1978.

[20] E. Börger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In *Formal Syntax and Semantics of Java, LNCS*, pages 353–404. Springer, 1999.

[21] A. Borgida. The Frame Problem in Object-Oriented Specifications: An Exhibition of Problems and Approaches. Technical report, Rutgers University, Dept. of Computer Science, 1992.

[22] A. Borgida, J. Mylopoulos, and R. Reiter. On the Frame Problem in Procedure Specifications. *IEEE transactions in Software Engineering*, 21(10):785–798, 1995.

[23] C. Boyapati, R. Lee, and M. Rinard. Ownership Types For Safe Programming: Preventing Data Races And Deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 37, pages 211–230, New York, NY, USA, Nov. 2002. ACM Press.

[24] C. Boyapati, B. Liskov, and L. Shrira. Ownership Types For Object Encapsulation. *SIGPLAN Not.*, 38(1):213–223, 2003.

[25] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the Future Safe For the Past: Adding Genericity to the Java Programming Language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 183–200, New York, NY, USA, 1998. ACM.

[26] L. Cardelli and R. Pike. Squeak: a Language for Communicating with Mice. In *Computer Graphics*, pages 199–204, 1985.

[27] D. Caromel. Toward a Method of Object-Oriented Concurrent Programming. *Communications of the ACM*, 36(9):90–102, 1993.

[28] D. Caromel, W. Klauser, and J. Vayssiere. Towards Seamless Computing and Metacomputing in Java. In G. C. Fox, editor, *Concurrency Practice and Experience*, volume 10, pages 1043–1061. Wiley & Sons, Ltd., Sept. 1998. http://www-sop.inria.fr/oasis/proactive/.

[29] Y. Cheon and G. Leavens. A Runtime Assertion Checker For the Java Modeling Language. In *International Conference on Software Engineering Research and Practice (SERP '02), Las Vegas, Nevada*, pages 322–328. CSREA Press, June 2002.

[30] A. Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[31] A. Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, 1941.

[32] D. Clarke, M. Richmond, and J. Noble. Saving the World from Bad Beans: Deployment-Time Confinement Checking. In *In Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA)*, pages 374–387. ACM Press, 2003.

[33] D. Clarke,, T. Wrigstad,, J. Östlund,, and E. B. Johnsen,. Minimal Ownership for Active Objects. In *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 139–154, Berlin, Heidelberg, 2008. Springer-Verlag.

[34] D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33:10 of *ACM SIGPLAN Notices*, pages 48–64, New York, Oct. 1998. ACM Press.

[35] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model For Static Analysis of Programs By Construction or Approximation of Fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[36] D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universe Types for Race Safety. In *VAMP 07*, pages 20–51, Aug. 2007.

[37] J. Davies, C. Crichton, E. Crichton, D. Neilson, and I. H. Sørensen. Formality, Evolution, and Model-Driven Software Engineering. *Electronic Notes in Theoretical Computer Science*, 130:39–55, 2005.

[38] D. L. Detlefs, K. R. M. Leino, K. Rustan, M. Leino, G. Nelson, and J. B. Saxe. Extended Static Checking. Technical Report SRC-RR-159, HP, 1998.

[39] W. Dietl, S. Drossopoulou, and P. Müller. Formalization of Generic Universe Types. Technical Report 532, ETH Zurich, 2006.

[40] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In E. Ernst, editor, *ECOOP 2007 – Object-Oriented Programming*, volume 4609 of *LNCS*, pages 28–53. Springer, 2007.

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

[42] D. Distefano and M. J. Parkinson J. jStar: Towards Practical Verification for Java. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, pages 213–226, New York, NY, USA, 2008. ACM.

[43] S. Drossopoulou and S. Eisenbach. Java is Type Safe — Probably. *Lecture Notes in Computer Science*, 1241:389–418, 1997.

[44] S. Drossopoulou, S. Eisenbach, and S. Khurshid. Is the Java Type System Sound? *Theory and Practice of Object Systems*, 5(1):3–24, 1999.

[45] D. Faitelson, J. Welch, and J. Davies. From Predicates to Programs: The Semantics of a Method Language. In *Proceedings of SBMF 2005*, volume 184, pages 171–187. Electronic Notes in Theoretical Computer Science, 2007.

[46] Y. A. Feldman, O. Barzilay, and S. Tyszberowicz. Jose: Aspects for Design by Contract. *International Conference on Software Engineering and Formal Methods*, 0:80–89, 2006.

[47] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, volume 37, pages 234–245, June 2002.

[48] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking For Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245, New York, NY, USA, 2002. ACM.

[49] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. In *In Principles of Programming Languages (POPL*, pages 171–183. ACM Press, 1998.

[50] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[51] J. Gosling et al. *The Java Language Specification*. GOTOP Information Inc., 5F, No.7, Lane 50, Sec.3 Nan Kang Road Taipei, Taiwan, 1996.

[52] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[53] O. M. Group. *Common Object Request Broker Architecture: Core Specification*. OMG, 2004.

[54] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. *Specification and Validation Methods*, pages 9–36, 1995.

[55] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[56] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier Science Inc., New York, NY, USA, 1977.

[57] C. A. R. Hoare. An Axiomatic Basis For Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[58] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[59] C. A. R. Hoare. The Verifying Compiler: A Grand Challenge For Computing Research. *J. ACM*, 50(1):63–69, 2003.

[60] J. Hogg. Islands: Aliasing Protection In Object-oriented Languages. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 271–285, New York, NY, USA, 1991. ACM Press.

[61] R. Holzapfel and G. Winterstein. VDM++ – A Formal Specification Language For Object-Oriented Designs. In *Ada in Industry, Proceedings of the Ada-Europe Conference 1988*. Cambridge University Press, Great Brittain, 1989.

[62] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23:279–295, 1997.

[63] G. J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, Sept. 2003.

[64] P. Hudak, P. Wadler, A. Brian, B. J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, S. P. Jones, M. Reeve, D. Wise, and J. Young. Report On the Programming Language Haskell: A Non-strict, Purely Functional Language. *ACM SIGPLAN Notices*, 27, 1992.

[65] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[66] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. In L. Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[67] G. Kahn. Natural Semantics. In *STACS '87: Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science*, pages 22–39, London, UK, 1987. Springer-Verlag.

[68] E. Kerfoot and S. McKeever. Maintaining Invariants Through Object Coupling Mechanisms. In T. Wrigstad, editor, *3rd International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO), in conjunction with ECOOP 2007*, Berlin, Germany, July 2007.

[69] E. Kerfoot and S. McKeever. Checking Concurrent Contracts with Aspects. In *SAC 2010*. ACM, 2010.

[70] E. Kerfoot, S. McKeever, and F. Torshizi. Deadlock freedom through object ownership. In *IWACO '09: International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming*, pages 1–8, New York, NY, USA, 2009. ACM.

[71] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Commun. ACM*, 44(10):59–65, 2001.

[72] G. Kiczales,, E. Hilsdale,, J. Hugunin,, M. Kersten,, J. Palm,, and W. G. Griswold,. An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[73] G. A. Kildall. A Unified Approach To Global Program Optimization. In *In Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206. ACM Press, 1973.

[74] R. G. Lavender and D. C. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. *Pattern Languages of Program Design 2*, pages 483–499, 1996.

[75] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A Notation for Detailed Design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[76] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, and J. Kiniry. JML Reference Manual. `http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html`, May 2008.

[77] K. R. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Institute of Technology, Pasadena, CA, USA, 1995.

[78] K. R. M. Leino. Data Groups: Specifying The Modification Of Extended State. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, And Applications*, pages 144–153, New York, NY, USA, 1998. ACM Press.

[79] B. Liskov, and L. Shrira,. Promises: Linguistic Support For Efficient Asynchronous Procedure Calls In Distributed Systems. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 260–267, New York, NY, USA, 1988. ACM.

[80] B. Liskov and J. Wing. Family Values: A Behavioral Notion of Subtyping. Technical Report MIT/LCS/TR-562b, ACM Transactions on Programming Languages and Systems, 1994.

[81] B. Liskov and J. Wing. Behavioral Subtyping Using Invariants And Constraints, 1999.

[82] B. Liskov and S. Zilles. Programming With Abstract Data Types. *SIGPLAN Not.*, 9(4):50–59, 1974.

[83] K.-P. Löhr and M. Haustein. The JAC System: Minimizing the Differences between Concurrent and Sequential Java Code. *Journal of Object Technology*, 5(7), 2006.

[84] C. Lopes, M. Lippert, and E. Hilsdale. Design By Contract With Aspect-oriented Programming, 2002. U.S. Patent No. 06,442,750, Issued August 27,2002.

[85] C. V. Lopes. A Study on Exception Detection and Handling Using Aspect-Oriented Programming. In *In Proceedings of the 22nd International Conference on Software Engineering*, pages 418–427. ACM Press, 2000.

[86] D. H. Lorenz and T. Skotiniotis. Contracts and Aspects. Technical Report NU-CCIS-03-13, College of Computer and Information Science, Northeastern University, Boston, MA 02115, Dec. 2003.

[87] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, first edition, 1988.

[88] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[89] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[91] C. Morgan. *Programming From Specifications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[92] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[93] P. Müller and A. Poetzsch-Heffter. Universes: A Type System for Alias and Dependency Control. Technical Report 279, Fernuniversität Hagen, 2001.

[94] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular Invariants for Layered Object Structures. Technical Report 424, ETH Zurich, Mar. 2005.

[95] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[96] T. Nipkow and D. von Oheimb. Java$_{\ell ight}$ is Type-Safe — Definitely. In *Proc. 25th ACM Symp. Principles of Programming Languages*, pages 161–170. ACM Press, New York, 1998.

[97] J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. *Lecture Notes in Computer Science*, 1445:158–185, 1998.

[98] C. Okasaki. *Purely Functional Data Structures*. PhD thesis, Pittsburgh, PA, USA, 1996. Chair-Lee, Peter.

[99] M. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, Aug. 2005.

[100] S. Peyton Jones et al. *The Haskell 98 Language and Libraries: The Revised Report*, volume 13. Jan. 2003.

[101] R. Pike. Newsqueak: A Language for Communicating with Mice. Technical Report 143, Bell Labs, Aug. 1993.

[102] G. D. Plotkin. A Structural Approach to Operational Semantics, 1981.

[103] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Featherweight Generic Confinement. *J. Funct. Program.*, 16(6):793–811, 2006.

[104] H. Rebêlo, S. Soares, R. Lima, L. Ferreira, and M. Cornélio. Implementing Java Modeling Language Contracts With AspectJ. In *SAC '08: Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 228–233, New York, NY, USA, 2008. ACM.

[105] J. Reynolds. Separation Logic: A Logic For Shared Mutable Data Structures, 2002.

[106] A. W. Roscoe. *A Classical Mind: Essays in Honour of C. A. R. Hoare*, chapter Model-checking CSP, pages 353–378. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

[107] A. W. Roscoe, C. A. R. Hoare, and R. Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[108] K. Rustan, M. Leino, and P. Müller. Object invariants in dynamic contexts, 2004.

[109] A. Sabry. What Is A Purely Functional Language? *J. Funct. Program.*, 8(1):1–22, 1998.

[110] V. Saraswat. Java is Not Type-Safe. Manuscript, AT&T Research, 1997.

[111] D. S. Scott and C. Strachey. Toward a Mathematical Semantics for Computer Languages. In J. Fox, editor, *Computers and Automata*, pages 19–46. Wiley, New York, 1972.

[112] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective Tool Support For The Working Semanticist. *SIGPLAN Not.*, 42(9):1–12, 2007.

[113] J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, 2 edition, 1992.

[114] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *European Conference on Object Oriented Programming ECOOP 2008*, 2008.

[115] R. Strniša. Fixing the Java Module System, in Theory and in Practice. In *FTfJP '08: 10th Workshop on Formal Techniques for Java-like Programs*, July 8, 2008.

[116] R. Strniša, P. Sewell, and M. Parkinson. The Java Module System: Core Design and Semantic Definition. In *OOPSLA '07: Proceedings of the 22nd Cnnual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, pages 499–514, New York, NY, USA, 2007. ACM.

[117] D. Syme. Proving Java Type Soundness. In *Formal Syntax and Semantics of Java*, pages 83–118, London, UK, 1999. Springer-Verlag.

[118] D. A. Turner. Miranda: A Non-strict Functional Language With Polymorphic Types. In *Proc. of a Conference On Functional Programming Languages And Computer Architecture*, pages 1–16, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[119] J. Vitek and B. Bokowski. Confined Types. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 82–96, New York, NY, USA, 1999. ACM.

[120] E. F. Walker, R. Floyd, and P. Neves. Asynchronous Remote Operation Execution in Distributed Systems. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 253–259, May 1990.

[121] D. Wampler. Contract4J for Design by Contract in Java: Design Pattern-Like Protocols and Aspect Interfaces. In Y. Coady, D. H. Lorenz, O. Spinczyk, and E. Wohlstadter, editors, *Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 27–30, Bonn, Germany, Mar. 20 2006. Published as University of Virginia Computer Science Technical Report CS–2006–01.

[122] T. Zhao, J. Palsberg, and J. Vitek. Lightweight Confinement for Featherweight Java, 2003.