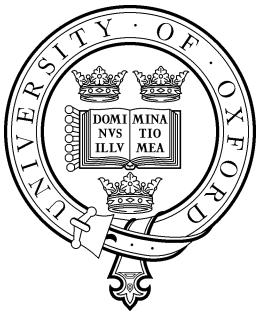
Formal Relationships in Sequential Object Systems



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 as well as 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 [29] defines sound invariants between two object types using classical Design-by-Contract [35] 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 [11].

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 [29, 30, 31], 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, and Dr. Alessandra Cavarra. 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 threaded 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 [31]. 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 running 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 [35] (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 any program state.

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 or not these invariants hold for all relevant states of running programs. Given a module whose reachable states adhere to the invariant conditions when appropriate, 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 [20, 21] 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 {
...
  ensures size() == 0;
  public void empty();

  public int size() { ... }
  public ListIterator iterator() { ... }
}
```

```
class ListIterator {
  private List list;
  private int position, last;

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

In this example, the *ListIterator* instances are responsible for their own invariants, however the *List* instances they rely must on also ensure their state satisfies the given constraint. Expecting the methods of *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 prevent 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.clear(); // 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, but 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* is correct 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 prove 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 [35], 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 [5, 25, 26, 36, 41, 43, 46, 53], or there must be some approach to ensuring mutation abides by the present constraints.

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 allows 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 when they complete, 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 [4, 14, 24, 37] 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, both passive and active, 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. Invariant soundness in the CoJava sense means that, when a method is called and obeys its contract, any invariant that held before the call will continue to hold afterward. Without ownership and other techniques, this is not always so as the Indirect Invariant Effect demonstrates.

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 ([14, 37] to name a few) in that it relies on simpler type definitions, fewer concepts, has a more rigid definition of encapsulation, and is defined in simpler formal terms.

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 [8] devised for the Spec# Language [7], this technique relies on explicit bidirectional aliasing by the colleague objects and a means of generating additional contract support. Colleagues however does not require the explicit definition of ancillary invariants or conditions to ensure correctness. Another means to addressing the same problem is to employ ownership primarily but weaken its constraints in certain situations and introduce additional proof obligations [38].

These disparate techniques all 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 are general in form. 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 CoJava language. 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 and threaded objects are implemented as standard Java output. Combined with a runtime assertion checking technique based on AspectJ [32] aspects, the tool implements an effective software development methodology capable to compiling, running, and checking simple CoJava examples as well as larger complex systems.

The chapters of this thesis are broken down as follows:

Chapter **??** discusses challenges of specifying object systems in relation to relationships. This will include 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

Chapter **??** will define the sequential CoJava language by providing its abstract syntax, type and operation rules. This semantic description 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. An overview of the Design-by-Contract methodology is given. CoJava concrete syntax uses a subset of JML [33, 34] for specification, and this section will outline those elements from JML used to specify CoJava.

Chapter **??** will define the CoJava ownership type system. This will detail how it is used to enforce encapsulation and build abstract data types whose internal components are insulated against damaging mutations. The specific properties ownership provides will be proved to always hold for well-typed programs, as well as allowing invariants to soundly rely on owned objects.

Chapter **??** introduces the Colleague Technique that allows two objects to soundly rely on each other for their invariant conditions. This technique involves specialized types and significant tool support, but as a consequence allows invariants that would otherwise be susceptible to adverse external mutations.

Chapter 2

CoJava

The CoJava language is defined in this chapter as a subset of the Java language [20, 21] which includes the features of interest. The language is defined as an extension to the Lightweight Java[1] [50, 51] language whose type-safety has been proven using Isabelle/HOL [39] based on proof obligations generated by the Ott Tool [49].

CoJava represents a small sequential subset of Java. Functional subsets of Java have been defined in other work [19, 27, 45] as a basis for the formalization of certain concepts, however including statements 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.

2.1 Formalizing Java

The primary definition for Java is the official Java Language Specification [21]. 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 [47]. 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 [17, 18, 40, 42, 52]. 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 [1, 2, 3, 14, 37]. These definitions of subset languages aim to capture the 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 [6, 10] presented a new challenge to type safety. Again definitions for subset languages are devised [27] for which type safety is proved. Extending this

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

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 [15, 16] 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 [44] (small-step) or Natural Semantic [28] (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 [48] style, which describe semantics in terms of functions over state, or axiomatic [23] 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 [22] are suited for defining Java where the execution at runtime often diverges considerably from what the syntax of the code would indicate [9]. 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 Isabelle/HOL.

2.2 Lightweight Java

The abstract syntax for Lightweight Java (LJ) is given in Figure 2.1, 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 they and others are not essential 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 [12, 13], therefore primitive types with their associated arithmetic operations can

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 2.1: Lightweight Java Abstract Syntax

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*.

Figure 2.2 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 $\bar{\iota}$. 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 $\varnothing$.

$\bar{\iota}$	$::=$	$[] \mid \iota \ldots \iota$	– 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 2.2: Lightweight Java List Syntax

The overline syntax is also used to define universal quantification and define lists through predicates. 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 $\{\iota : \overline{\iota_k}^k \mid v(\iota, \iota') \bullet \iota'\}$.

2.2.1 Types, Type Environment, and State

Figure 2.3 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 2.2.1 (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 2.2.2 (Variable State Function L)
The function L recalls the state of variables, hence it relates variable names to values: $Var \to Val$.
$L[x \mapsto v]$ associates the variable x with the value v.

Definition 2.2.3 (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: $oid \to (\tau, f \to v)$.

- *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 $(H\ oid).1$.*

2.2.2 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.

A definition of configurations is given here which represents a slight extension to that in LJ by introducing the concepts of initial and final states:

---

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 2.3: Lightweight Java Formal Definition Elements

Definition 2.2.4 (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.

2.2.3 Type Information

The following 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 inherited as well as defined as new.

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($cl\ meth(\overline{vd})\{meth\_body\}) = meth$

Given a program P, this predicate states that all the classes it defines have distinct names:

$$\frac{\begin{array}{l}P = \overline{cld_k}^{\,k} \\ \overline{\textbf{class\_name}(cld_k) = dcl_k}^{\,k} \\ \textbf{distinct}(\overline{dcl_k}^{\,k})\end{array}}{\textbf{distinct\_names}(P)}$$

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:

$$\frac{}{\textbf{find\_cld}([], ctx, dcl) = \varnothing} \qquad \frac{\begin{array}{l}P = cld : \overline{cld} \\ cld = \textbf{class}\ dcl\ \textbf{extends}\ cl\ \{\overline{fd}\ \overline{meth\_def}\}\end{array}}{\textbf{find\_cld}(P, ctx, dcl) = (ctx, cld)}$$

$$\frac{\begin{array}{l}P = cld : \overline{cld} \\ cld = \textbf{class}\ dcl'\ \textbf{extends}\ cl\ \{\overline{fd}\ \overline{meth\_def}\} \\ dcl \neq dcl' \\ \textbf{find\_cld}(\overline{cld}, ctx, dcl) = ctxcld_{opt}\end{array}}{\textbf{find\_cld}(P, ctx, dcl) = ctxcld_{opt}}$$

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 **acyclic_clds**(P) states that the inheritance relation between classes defined in P is acyclic, such that no class inherits in anyway from itself:

$$\frac{\forall\ ctx\ \ dcl \bullet \textbf{find\_cld}(P, ctx, dcl) \neq \varnothing \Rightarrow (\exists\ nn \bullet (P, ctx, dcl, nn) \in \textbf{path\_length})}{\textbf{acyclic\_clds}(P)}$$

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`:

$$\frac{}{\textbf{find\_path\_rec}(P, ctx, \texttt{Object}, \overline{ctxcld}) = \overline{ctxcld}}$$

$$\frac{(\neg\ \textbf{acyclic\_clds}(P)) \lor \textbf{find\_cld}(P, ctx, dcl) = \varnothing}{\textbf{find\_path\_rec}(P, ctx, dcl, \overline{ctxcld}) = \varnothing}$$

$$\frac{\begin{array}{l}\textbf{acyclic\_clds}(P)\\ \textbf{find\_cld}(P, ctx, dcl) = (ctx', cld)\\ \textbf{superclass\_name}(cld) = cl\\ \textbf{find\_path\_rec}(P, ctx, cl, \overline{ctxcld} \frown [(ctx', cld)]) = \overline{ctxcld}_{opt}\end{array}}{\textbf{find\_path\_rec}(P, ctx, dcl, \overline{ctxcld}) = \overline{ctxcld}_{opt}}$$

find_path(P, ctx, cl) is simply shorthand for **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:

$$\frac{}{\textbf{fields\_in\_path}([]) = []} \qquad \frac{\begin{array}{l}\textbf{class\_fields}(cld) = \overline{cl_j\ f_j\ ;}^{\,j}\\ \textbf{fields\_in\_path}(\overline{ctxcld_k}^{\,k}) = \overline{f}\\ \overline{f}' = \overline{f_j}^{\,j} \frown \overline{f}\end{array}}{\textbf{fields\_in\_path}((ctx, cld) : \overline{ctxcld_k}^{\,k}) = \overline{f}'}$$

fields is the collection of all defined and inherited attributes for a given type τ in program P:

$$\frac{\textbf{find\_path}(P, \tau) = \varnothing}{\textbf{fields}(P, \tau) = \varnothing} \qquad \frac{\begin{array}{l}\textbf{find\_path}(P, \tau) = \overline{ctxcld}\\ \textbf{fields\_in\_path}(\overline{ctxcld}) = \overline{f}\end{array}}{\textbf{fields}(P, \tau) = \overline{f}}$$

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 $\bot$:

$$\frac{}{\textbf{ftype\_in\_fds}(P, ctx, [], f) = \varnothing}$$

$$\frac{\textbf{find\_type}(P, ctx, cl) = \varnothing}{\textbf{ftype\_in\_fds}(P, ctx, cl\ f : \overline{fd_k}^{\,k}, f) = \bot}$$

$$\frac{\textbf{find\_type}(P, ctx, cl) = \tau}{\textbf{ftype\_in\_fds}(P, ctx, cl\ f : \overline{fd_k}^{\,k}, f) = \tau}$$

$$\frac{\begin{array}{l}f \neq f' \\ \textbf{ftype\_in\_fds}(P, ctx, fd_2 \ldots fd_k, f') = \tau^{\bot}_{opt}\end{array}}{\textbf{ftype\_in\_fds}(P, ctx, cl\ f : \overline{fd_k}^{\,k}, f') = \tau^{\bot}_{opt}}$$

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.

$$\frac{}{\textbf{ftype\_in\_path}(P, [], f) = \varnothing}$$

$$\frac{\begin{array}{l}\textbf{class\_fields}(cld) = \overline{fd} \\ \textbf{ftype\_in\_fds}(P, ctx, \overline{fd}, f) = \bot\end{array}}{\textbf{ftype\_in\_path}(P, (ctx, cld) : \overline{ctxcld_k}^{\,k}, f) = \varnothing}$$

$$\frac{\begin{array}{l}\textbf{class\_fields}(cld) = \overline{fd} \\ \textbf{ftype\_in\_fds}(P, ctx, \overline{fd}, f) = \tau\end{array}}{\textbf{ftype\_in\_path}(P, (ctx, cld) : \overline{ctxcld_k}^{\,k}, f) = \tau}$$

$$\frac{\textbf{class\_fields}(cld) = \overline{fd} \quad \textbf{ftype\_in\_fds}(P, ctx, \overline{fd}, f) = \varnothing \quad \textbf{ftype\_in\_path}(P, \overline{ctxcld_k}^k, f) = \tau_{opt}}{\textbf{ftype\_in\_path}(P, (ctx, cld) : \overline{ctxcld_k}^k k, f) = \tau_{opt}}$$

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:

$$\frac{}{\textbf{find\_meth\_def\_in\_path}([], meth) = \varnothing}$$

$$\frac{\textbf{class\_methods}(cld) = \overline{meth\_def} \quad \textbf{find\_meth\_def\_in\_list}(\overline{meth\_def}, meth) = meth\_def}{\textbf{find\_meth\_def\_in\_path}((ctx, cld) : \overline{ctxcld_k}^k, meth) = (ctx, meth\_def)}$$

$$\frac{\textbf{class\_methods}(cld) = \overline{meth\_def} \quad \textbf{find\_meth\_def\_in\_list}(\overline{meth\_def}, meth) = \varnothing \quad \textbf{find\_meth\_def\_in\_path}(\overline{ctxcld_k}^k, meth) = ctxmeth\_def_{opt}}{\textbf{find\_meth\_def\_in\_path}((ctx, cld) : \overline{ctxcld_k}^k, meth) = ctxmeth\_def_{opt}}$$

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:

$$\frac{\textbf{find\_path}(P, \tau) = \varnothing}{\textbf{find\_meth\_def}(P, \tau, meth) = \varnothing}$$

$$\frac{\textbf{find\_path}(P, \tau) = \overline{ctxcld} \quad \textbf{find\_meth\_def\_in\_path}(\overline{ctxcld}, meth) = ctxmeth\_def_{opt}}{\textbf{find\_meth\_def}(P, \tau, meth) = ctxmeth\_def_{opt}}$$

mtype represents the type of the method *meth* as declared in τ or one of its supertypes:

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

2.2.4 Subtyping

The subtyping relation $\prec$ is defined as a partial order on types. When a class inherits from another, then any type derived from that class is a subtype of that derived from the superclass. The following rules define the relation.

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

$$\frac{\textbf{find\_path}(P, \tau) = \overline{ctxcld} \quad \tau = ctx.cld}{P \vdash \tau \prec ctx.\texttt{Object}}$$

Given a type τ and a list of types representing the inheritance path from τ to *ctx*.`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, hence $\tau \prec \tau''$ will be true. Since τ is also in the path, then this implies that $\tau \prec \tau$ is also true.

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

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

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

$$\frac{\overline{\tau} = \overline{\tau_k}^k \quad \overline{\tau}' = \overline{\tau'_k}^k \quad \overline{P \vdash \tau_k \prec \tau'_k}^k}{P \vdash \overline{\tau} \prec \overline{\tau}'}$$

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:

$$\frac{\tau_{opt} = \tau \quad \tau_{opt}{}' = \tau' \quad P \vdash \tau \prec \tau'}{P \vdash \tau_{opt} \prec \tau_{opt}{}'}$$

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}:

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

2.2.5 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:

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

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:

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

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

$$\frac{\begin{array}{l}\vdash P \\ P \vdash H \\ P, \Gamma, H \vdash L\end{array}}{\Gamma \vdash (P, L, H, \mathit{Exception})}$$ WF_ExConfig

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

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

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

$$\frac{\overline{P, \Gamma \vdash s_k}^{k}}{P, \Gamma \vdash \{\overline{s_k}^{k}\}}$$ WF_StmtList

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

$$\frac{P \vdash \Gamma(x) \prec \Gamma(var)}{P, \Gamma \vdash var = x;}$$ 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:

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

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

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

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:

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

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:

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

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:

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

A method is well-formed 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 it:

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

A well-formed type definition 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:

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

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:

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

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:

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

2.2.6 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 2.2.5 (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.

$$\frac{\overline{\theta \vdash s_k \rightsquigarrow s'_k}^{k}}{\theta \vdash \{\overline{s_k}^{k}\} \rightsquigarrow \{\overline{s'_k}^{k}\}} \qquad \frac{\theta(var) = var' \quad \theta(x) = x'}{\theta \vdash var = x; \rightsquigarrow var' = x';}$$

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

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

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

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

2.2.7 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.

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

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

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

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

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

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

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

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

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

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

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

2.3 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 require to describe logical properties. Additionally, a `Boolean` type with `True` and `False` subtypes is introduced to describe these conditions in terms of true or false.

2.3.1 Ownership

Ownership enforces encapsulation through the type system. By constraining what operations with owned types are permitted and relationship between owned and non-owned types, an object can control what clients access those objects it owns. The set of owned objects forms a series rooted acyclic digraphs when considering the aliasing relationship as edges between 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.

The restrictions which impose these properties are summarized here:

- Owned values cannot be assigned to non-owned variables and attributes, or vice versa. This ensures that owned objects can only be accessed through owned variables.
- Methods with owned arguments can be called only when the receiver is **this**. This prevents external clients from creating cycles in the ownership structure.
- Owned attributes can only be assigned to when the receiver is **this**. This also is necessary to prevent the formation of cycles.
- Methods returning owned references and owned attributes can only be accessed through an owned receiver. Since owned objects are only accessible through owned references, a client cannot access an object's owned objects.
- Within the method of a class *dcl*, **this** is typed as the owned version of *dcl*. This prevents an owned object from exposing a non-owned reference to itself to its clients.

The context construct in LJ is intended to be used for extensions to the language. Ownership can be represented with this by introducing two contexts representing regular and owned types: **reg** and **owned**. The *ctx* definition in Figure 2.3 is thus modified to reflect this:

ctx	::=	– Context
	|	– Regular types with no context
	| **owned**	– Owned types

The syntax of LJ is also modified to include context with attribute, argument and method definitions. Figure 2.4 gives the syntax for CoJava with the added context values, as well as the

local variable statement. 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 be altered.

P	::=	$(\overline{cld}, s)$	– 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
	|	$var = \mathbf{new}_{ctx}$ $cl()$;	– Object creation
	|	$var = x.met(\overline{y})$;	– Method call
	|	ctx cl var;	– Declare new variable
$TVar, x, y$	::=		– Term variables
	|	var	– Regular variables
	|	**this**	– Ref. to current object

Figure 2.4: CoJava Abstract Syntax

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

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

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

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 $\textbf{owned}.cld \prec \textbf{reg}.cld'$ 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-owed 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.

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:

$$\frac{\tau = (ctx, cld) \quad ctx = \textbf{owned}}{\textbf{owned}(\tau)} \qquad \frac{\tau = (ctx, cld) \quad ctx \neq \textbf{owned}}{\textbf{reg}(\tau)}$$

The following rules replace those previously defined in this chapter:

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

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. If x were not owned but f was, then x's internal representation would be exposed once the statement was executed.

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

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

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

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. 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 type. 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:

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

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. By requiring that the type of the returned value y be a subtype of the method's return type τ, a method with an owned return type cannot return a non-owned reference, or vice versa. 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.

2.3.2 DbC Specification

Specifying Java programs with JML or other Design-by-Contract 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 2.3.2 is annotated with CoJava 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. 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.

Specifying CoJava programs require predicates to be introduced to the language:

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

```
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 2.5: Counter Java+JML Example

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

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

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:

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

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

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

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

Figure 2.6: CoJava Predicate Syntax

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

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

$$\frac{P, \Gamma \vdash_{\tau, meth} pr}{P, \Gamma \vdash_{\tau, meth} old(pr)} \quad \text{WF\_Old}$$

$$\frac{\begin{array}{l}P, \Gamma \vdash_{\tau, meth, \tau'} prv \\ P, \Gamma \vdash_{\tau, meth, \tau''} prv' \\ P \vdash \tau' \prec \tau'' \vee P \vdash \tau'' \prec \tau'\end{array}}{P, \Gamma \vdash_{\tau, meth} prv == prv'} \quad \text{WF\_Eq}$$

Only methods which do not change the state of the program can be called in predicates, that is they must be pure. The predicate $\mathbf{pure}(ctx, cld)$ (explained subsequently) 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:

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

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

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

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

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

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

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 discuss the relationships between the states. Predicates are normally evaluated in terms of the second heap, except for predicates within **old** predicates which are evaluated in terms of the first.

The judgment $H, H', L \vdash pr$ 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. For values, $H, H', L \vdash prv = v$ states that the attribute or argument prv evaluates to the value v. The following rules define the conditions required for a predicate to be evaluated to true for given states:

$$\frac{\begin{array}{l} H, H', L \vdash pr \\ H, H', L \vdash pr' \end{array}}{H, H', L \vdash pr\ \&\&\ pr'}$$

$$\frac{H, H', L \vdash pr \vee H, H', L \vdash pr'}{H, H', L \vdash pr \;\; || \;\; pr'}$$

$$\frac{H, H', L \vdash pr \Rightarrow H, H', L \vdash pr'}{H, H', L \vdash pr \;\; ==> \;\; pr'}$$

$$\frac{\forall\, oid : \mathrm{dom}\, H \cup \mathrm{dom}\, H' \mid H, H', L[var \mapsto oid] \vdash pr \bullet H, H', L[var \mapsto oid] \vdash pr'}{H, H', L \vdash forall(ctx \;\; cl \;\; var;\;\; pr;\;\; pr')}$$

$$\frac{\exists\, oid : \mathrm{dom}(H) \cup \mathrm{dom}(H') \mid H, H', L[var \mapsto oid] \vdash pr \bullet H, H', L[var \mapsto oid] \vdash pr'}{H, H', L \vdash exists(ctx \;\; cl \;\; var;\;\; pr;\;\; pr')}$$

$$\frac{H', H', L \vdash pr}{H, H', L \vdash old(pr)}$$

$$\frac{\begin{array}{l} H, H', L \vdash prv = v \\ H, H', L \vdash prv' = v' \\ v = v' \end{array}}{H, H', L \vdash prv == prv'}$$

$$\frac{\begin{array}{l} L[\mathbf{this}] = oid \\ x \notin \mathrm{dom}\, L \\ var \notin \mathrm{dom}\, L \\ \overline{y_k \notin \mathrm{dom}\, L}^{\,k} \\ \overline{H, H', L \vdash prv_k = v_k}^{\,k} \\ L' = L[x \mapsto oid][var \mapsto \mathbf{null}][\overline{y_k \mapsto v_k}^{\,k}] \\ (P, L', H, var = x.meth(\overline{y_k}^{\,k});\;) \longrightarrow (P, L'', H'', \varnothing) \\ L''(var) = oid' \\ H''(oid') = \mathbf{True} \end{array}}{H, H', L \vdash \mathbf{this}.meth(\overline{prv_k}^{\,k})}$$

$$\frac{\begin{array}{l} L[w] = oid \\ x \notin \mathrm{dom}\, L \\ var \notin \mathrm{dom}\, L \\ \overline{y_k \notin \mathrm{dom}\, L}^{\,k} \\ \overline{H, H', L \vdash prv_k = v_k}^{\,k} \\ L' = L[x \mapsto oid][var \mapsto \mathbf{null}][\overline{y_k \mapsto v_k}^{\,k} \\ (P, L', H, var = x.meth(\overline{y_k}^{\,k});\;) \longrightarrow (P, L'', H'', \varnothing) \\ L''(var) = oid' \\ H''(oid') = \mathbf{True} \end{array}}{H, H', L \vdash w.meth(\overline{prv})}$$

$$\frac{L(\mathbf{this}) = oid \quad H(oid, x) = v}{H, H', L \vdash \mathbf{this}.x = v}$$

$$\frac{L(w) = v}{H, H', L \vdash w = v}$$

$$\frac{\mathbf{result} \in \mathrm{dom}\, L}{H, H', L \vdash \mathbf{result} = L(\mathbf{result})}$$

Java is specified with JML annotations added within comments embedded in the code as shown int he *Counter* example. The alternative approach is to define a series of functions which relate CoJava constructs to the predicates and annotations which specify them:

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

Additional functions will be later defined to describe other properties and annotations. Consider the *Counter* class as an example of how these functions define specifications, starting with the CoJava version of the class:

The invariant for this type would thus be defined by $\mathbf{inv}(ctx.Counter)$, which would evaluate to $\mathbf{this}.value >= 0$ && $\mathbf{this}.value <= \mathbf{this}.max$ (noting that operators are shorthand for *Boolean*-returning methods of *int*). Similarly the precondition for method `inc()` would be defined by $\mathbf{pre}(ctx.Counter, inc)$ and would equal $\mathbf{this}.value < \mathbf{this}.max$. The full specification for *Counter* is given here:

$\mathbf{inv}(ctx.Counter) = \mathbf{this}.value != \mathbf{null}$ && $\mathbf{this}.max != \mathbf{null} ==>$
$\mathbf{this}.value >= 0$ && $\mathbf{this}.value <= \mathbf{this}.max$
$\mathbf{post}(ctx.Counter, init) = \texttt{this}.max == max$
$\mathbf{pre}(ctx.Counter, inc) = \mathbf{this}.value < \mathbf{this}.max$
$\mathbf{post}(ctx.Counter, inc) = \mathbf{this}.value = \mathbf{this}.value + 1$
$\mathbf{pre}(ctx.Counter, add) = (\mathbf{this}.value + n) >= 0$ && $(\mathbf{this}.value + n) <= \mathbf{this}.max$
$\mathbf{post}(ctx.Counter, add) = \mathbf{this}.value == old(\mathbf{this}.value + n)$
$\mathbf{post}(ctx.Counter, get) = result == \mathbf{this}.value$
$\mathbf{pure}(ctx.Counter) = [get]$

With these functions representing the specification for a class and the defined evaluation of predicates, a definition of program correctness can be given. When a correct method is called with a receiver whose invariant holds and arguments satisfying the precondition, the state of the program will satisfy the receiver's invariant and the method's postcondition. The judgment

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

  Counter init(int max) {
    this.value=0;
    this.max = max;
    return this;
  }

  Counter inc() {
    int v;
    v = this.value;
    this.value = v + 1;
    return this;
  }

  Counter add(int n){
    int v;
    v = this.value;
    this.value = v + n;
    return this;
  }

  int get()  { return this.value; }
}
```

Figure 2.7: Counter CoJava Example

$P, \mathbf{inv}, \mathbf{pre}, \mathbf{post}, \mathbf{pure} \vdash_\tau meth$ 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:

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

This defines the criteria which a method definition must meet if it is to be considered correct in terms of its specification. Assuming that all methods meet this definition of correctness, all object invariants will hold in a running program if all method calls meet the precondition requirements, and if attributes are always assigned values satisfying the receiver's invariant. This ensures that any op-

eration which affects the state of an object must respect the appropriate specifications. Consequently objects can only transition from one state satisfying the invariant to another such state.

Bibliography

[1] 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.

[2] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *in ECOOP 2004 – Object-Oriented Programming*, pages 1–25, 2004.

[3] 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.

[4] P. S. Almeida. Balloon Types: Controlling Sharing of State in Data Types. *Lecture Notes in Computer Science*, 1241:32, 1997.

[5] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007.

[6] J. A. Bank, B. Liskov, and A. C. Myers. Parameterized Types and Java. In *In Principles of Programming Languages (POPL*, pages 132–145, 1997.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] A. Church. An Unsolvable Problem of Elementary Number Theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[13] A. Church. *The Calculi of Lambda-Conversion*, volume 6 of *Annals of Mathematical Studies*. Princeton University Press, 1941.

[14] 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.

[15] W. Dietl, S. Drossopoulou, and P. Müller. Formalization of Generic Universe Types. Technical Report 532, ETH Zurich, 2006.

[16] 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.

[17] S. Drossopoulou and S. Eisenbach. Java is Type Safe — Probably. *Lecture Notes in Computer Science*, 1241:389–418, 1997.

[18] S. Drossopoulou, S. Eisenbach, and S. Khurshid. Is the Java Type System Sound? *Theory and Practice of Object Systems*, 5(1):3–24, 1999.

[19] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *In Principles of Programming Languages (POPL*, pages 171–183. ACM Press, 1998.

[20] J. Gosling et al. *The Java Language Specification*. GOTOP Information Inc., 5F, No.7, Lane 50, Sec.3 Nan Kang Road Taipei, Taiwan, 1996.

[21] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[22] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. *Specification and Validation Methods*, pages 9–36, 1995.

[23] C. A. R. Hoare. An Axiomatic Basis For Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[24] 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.

[25] 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.

[26] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[27] 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.

[28] 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.

[29] 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.

[30] E. Kerfoot and S. McKeever. Checking Concurrent Contracts with Aspects. In *SAC 2010*. ACM, 2010.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[36] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[37] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[38] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular Invariants for Layered Object Structures. Technical Report 424, ETH Zurich, Mar. 2005.

[39] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[40] 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.

[41] C. Okasaki. *Purely Functional Data Structures*. PhD thesis, Pittsburgh, PA, USA, 1996. Chair-Lee, Peter.

[42] M. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, Aug. 2005.

[43] S. Peyton Jones et al. *The Haskell 98 Language and Libraries: The Revised Report*, volume 13. Jan. 2003.

[44] G. D. Plotkin. A Structural Approach to Operational Semantics, 1981.

[45] A. Potanin, J. Noble, D. Clarke, and R. Biddle. Featherweight Generic Confinement. *J. Funct. Program.*, 16(6):793–811, 2006.

[46] A. Sabry. What Is A Purely Functional Language? *J. Funct. Program.*, 8(1):1–22, 1998.

[47] V. Saraswat. Java is Not Type-Safe. Manuscript, AT&T Research, 1997.

[48] 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.

[49] 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.

[50] 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.

[51] R. Strniša, P. Sewell, and M. Parkinson. The java module system: core design and semantic definition. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 499–514, New York, NY, USA, 2007. ACM.

[52] D. Syme. Proving Java Type Soundness. In *Formal Syntax and Semantics of Java*, pages 83–118, London, UK, 1999. Springer-Verlag.

[53] 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.