Behavioral Interface Contracts for Java (Rice University CS TR00-366)

Robert Bruce Findler   Matthias Felleisen
Department of Computer Science; Rice University
6100 South Main; MS132
Houston TX, 77030; USA
Contact: <robby@cs.rice.edu>

August 25, 2000

Abstract

Programs should consist of off-the-shelf, interchangeable, black-box components that are produced by a network of independent software companies. These components should not only come with type signatures but also with contracts that describe other aspects of their behavior.

One way to express contracts is to state pre- and post- conditions for externally visible functions. These pre- and post-conditions should then be validated during evaluation or possibly even during compilation. If a function call fails to satisfy its contract, the run-time system should blame the faulty program component.

Behavioral contracts in the form of assertions are well-understood in the world of procedural languages. Their addition to class and interface hierarchies in object-oriented programming languages, however, raises many new and interesting questions. The most complicating factor is that objects can pass between components and trigger callbacks. Another problem is that object-oriented languages allow objects to satisfy several interfaces at once.

In this paper, we analyze existing approaches to adding contracts to class-based languages and show how they blame the wrong component in certain situations for breach of contract. We then present a conservative extension of Java that allows programmers to specify method contracts in interfaces. The extension is a compromise between a consistent enforcement of contracts and language design concerns. In the future, we plan to clarify the relationship between contracts and contract violations with a rigorous analysis.

1 Contracts and Assertions

Beugnard, Jézéquel, Plouzeau, and Watkins [1] suggest that software components should come with contracts. They describe four levels of contracts:

1. syntactic contracts (type systems),
2. behavorial contracts (invariants, pre- and post-conditions),
3. synchronization contracts, and
4. quality of service contracts.

Type systems and other syntactic contracts for components have been studied extensively in object-oriented, functional, and higher-order contexts [2, 10, 16]. Behavioral contracts have been studied in procedural contexts [11, 19, 20, 25, 26]. Behavioral contracts have also been added to object-oriented languages [4, 9, 12, 13, 14, 21, 22, 24]. Adding behavioral contracts to object-oriented languages, however, is subtle. Each of the referenced systems fails

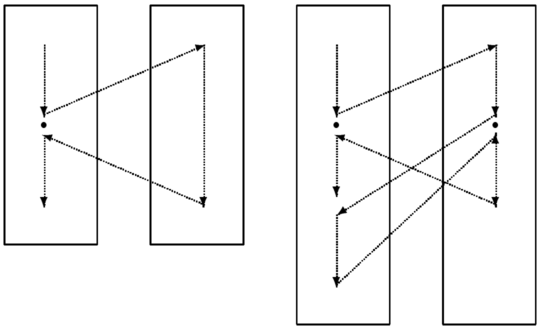

Figure 1: Component Interactions

```
interface Observer {
    void onEnq (Q q);
    void onDeq (Q q);
}

class Q {
    private Observer o;
    public boolean isEmpty() { ··· }

      /* @post isEmpty() = false; */
    public void enq(int i) { ··· }

      /* @pre isEmpty() = false; */
    public int deq() { ··· }

    public void setObserver(Observer o)
        { ··· }
}
```

```
class O implements Observer {
    public void onEnq(Q q) {
        q.deq();
    }

    public void onDeq(Q q) {
        ···
    }
}
```

```
class Main {
    public static void main () {
        Q q = new Queue();
        Observer o = new O();
        q.setObserver(o);
        q.enq(1);
    }
}
```

Figure 2: Queues in iContract

to interpret them properly. Some only allow contracts on methods, which means that they cannot catch contractual violations when objects are used in a higher-order manner, *e.g.* as callbacks. Others allow contracts on interfaces but, in certain situations, stop evaluation and blame components when no contractual violation has occurred.

In this paper, we present Contract Java, a version of Java with behavioral interface contracts. Our design goals for Contract Java are two-fold. First, the semantics of program without contracts must be identical under Contract Java and standard Java; even adding contracts that are always satisfied must not change a program's meaning. Second, code that is compiled with contracts must interoperate with legacy code that was compiled without contracts. In fact, the design of Contract Java is a compromise that sacrifices some degree of contract enforcement so that classes compiled with contracts can interact with legacy code.

We model Contract Java via a translation into plain Java. The compiler inserts code in the appropriate places to check the pre- and post-conditions. When feasible, a broken contract causes evaluation to halt and blames the component that broke the contract.

The next section analyzes previous work on contracts in object-oriented languages. Section 3 explains how to add behavioral contracts to Java interfaces, and how contracts in interfaces affect the design of the rest of the language. Section 4 describes Contract Java and shows how it compiles programs with contracts into standard Java. Finally, sections 5 and 6 discuss related and future work.

2 Contracts in Classes

Previous work [11, 20, 22, 26] on contracts suggests that programmers annotate the implementation of their methods with contracts, in particular, pre- and post-conditions. Unfortunately, such contracts cannot capture enough of a component's invariants. Consider figure 1. It contains two pictorial representations of component interactions that Szyperski mentions in his book [28]. The boxes indicate component boundaries. The dotted vertical lines represent the flow of control in a single component and the dotted diagonal lines represent control transfer to another component. The left-hand side of figure 1 is a pictorial representation of a simple interaction between two components. The left-hand component begins evaluation and during the course of that evaluation calls the right-hand component, exchanging flat values (numbers, characters, or strings). In this simple world, it is sufficient to associate contracts directly with methods.

The right-hand side of figure 1 describes a complex interaction between two components. As before, control begins in the left-hand component. This time, however, the left-hand component passes an object to the right-hand component. Then, instead of the right-hand component just returning some flat value, it calls a method of the object it received, transferring control back to the left-hand side component in a nested fashion. During this nested call, the left-hand component may modify some state, causing the post-condition of the right-hand component to fail. The blame should be cast on the left-hand component for providing a faulty object to the right-hand component. Unfortunately, when contracts are associated with implementations, there is no contract the right-hand component can write to ensure that the object it receives from the left-hand component is well-behaved.[1]

For a concrete example, consider a program in iContract[2] [14], an adaption of Eiffel's [22, 27] notion of contract to Java. Contracts in iContract are specified with @**pre** and @**post** clauses on methods. The @**pre** clause is a precondition that must be true before the method is called. If the @**pre** clause fails, iContract blames the method's caller. The @**post** clause is a postcondition of the method that must be true before the method returns. If the @**post** clause fails, iContract blames the method itself.

Figure 2 contains the iContract code for three components that implement the observer pattern [8] for a queue. The first component implements the queue class. The queue supports enqueue (*enq*) and dequeue (*deq*) operations and a test for emptiness (*isEmpty*). Additionally, the queue supports an observer. The *setObserver* method registers its argument as the current observer, and its methods *onEnq* and *onDeq* are called at the end of an *enq* operation and at the beginning of a *deq* operation, respectively.

In this example, the @**pre** and @**post** clauses dictate that dequeuing from an empty queue is illegal and, after an enqueue, the queue is not empty. In order to preserve these invariants, the behavior of the observer must also be restricted. In particular, the observer's *onEnq* method must not empty the queue. Unfortunately, iContract's contract language provides no way for *Q* to specify or enforce this restriction.

The second component is an observer that implements the *Observer* interface. The observer's *onEnq* method dequeues an element from the queue being observed. There is no mechanism for forcing the implementor of the *O* component to add postconditions that check the queue's invariants.

The final component, *Main*, instantiates *Q* and *O*, binds them to *q* and *o* respectively, and registers *o* with *q*. Then, when *Main* enqueues `1` to the queue, the observer immediately dequeues it, causing the post-condition on *enq* to fail. The blame should fall on the observer for improperly dequeuing the queue. The iContract tool, however, incorrectly blames *q*'s *enq* method for completing with an empty queue. If *Q* and *O* are implemented by different companies, this false blame can cause economic damage.

Because the contracts are specified in classes and not with interfaces, there is no appropriate contract for the argument of *setObserver*. If, instead, the contracts were associated with interfaces, the author of *Q* could impose a condition on observers, via the *Observer* interface. Then, the interface would contain a contract that ensures that *onEnq* does not empty the queue.

3 Contract Java

The lesson of the preceding section is that assertions specified with method implementations are insufficient to capture many component contracts. Therefore, we designed Contract Java, an extension of Java in which method contracts are

[1] Higher-order functions in language like SML [23] or Scheme [3] also exhibit this kind of behavior.

[2] To be fair, iContract does allow contracts to be specified on interfaces and it would properly assign blame if we were to rewrite it using interface contracts. We use iContract syntax here in an attempt to facilitate understanding. Section 5 returns to iContract and explains its shortcomings.

specified in interfaces. We identified three design goals.

1. First, Contract Java programs without contracts and programs with fully-satisfied contracts should behave as if they were run without contracts in Java.

2. Second, programs compiled with a conventional Java compiler must be able to interoperate with programs compiled under Contract Java.

3. Finally, unless a class declares that it meets a particular contract, it must never be blamed for failing to meet that contract. Abstractly, if the method m of an object with type t is called, the caller should only be blamed for the pre-condition contracts associated with t and m should only be blamed for post-condition contracts associated with t.

These design goals raise several interesting questions and demand decisions that balance language design with software engineering concerns. This section describes each of the major design issues, the alternatives, our decisions, our rationale, and the ramifications of the decisions. The decisions are not orthogonal; some of the later decisions depend on earlier ones.

3.1 Contracts: Specification and Enforcement

Contracts in Contract Java are decorations of methods signatures in interfaces. Each method declaration may come with a pre-condition expression and a post-condition expression; both expressions must evaluate to booleans. The pre-condition specifies what must be true when the method is called. If it fails, the context of the method call is to blame for not using the method in a proper context. The post-condition expression specifies what must be true when the method returns. If it fails, the method itself is to blame for not establishing the promised conditions.

Contract Java does not restrict the contract expressions. Still, good programming discipline dictates that the expressions should not contribute to the result of the program. In particular, the expressions should not have any side-effects.

Both the pre- and post-condition expressions are parameterized over the arguments of the method and the pseudo-variable *this*. The latter is bound to the current object. Additionally, the post-condition of the contract may refer to the name of the method, which is bound to the result of the method call.[3]

Contracts are enforced based on the type-context of the method call. If an object's type is an interface type, the method call must meet all of the contracts in the interface. For instance, if an object implements the interface I, a call to one of I's methods must check that pre-condition and the post-condition specified in I. If the object's type is a class type, the object has no contractual obligations. Since a programmer can always create an interface for any class, we leave objects with class types unchecked for efficiency reasons.

For an example, consider the interface *RootFloat*:

```
interface RootFloat {
  float getValue ();

  float sqRoot ();
  @pre { this.getValue() >= 0f }
  @post { Math.abs(sqRoot * sqRoot - this.getValue()) < 0.01f }
}
```

It describes the interface for a **float** wrapper class that provides a *sqRoot* method. The first method, *getValue*, has no contracts. It accepts no arguments and returns the unwrapped float. The *sqRoot* method also accepts no arguments, but has a contract. The pre-condition asserts that the unwrapped value is greater than or equal to zero. The result type of *sqRoot* is **float**. The post-condition states that the square of the result must be within 0.01 of the value of the float.

Even though the contract language is sufficiently strong to specify the complete behavior in some cases, such as the previous example, total or even partial correctness is not our goal in designing these contracts. Typically, the contracts cannot express the full behavior of a method. In fact, there is a tension between the amount of information revealed in the interface and the amount of validation the contracts can satisfy.

For an example, consider this stack interface:

[3] At this time, no provisions are made for referring to the initial value of an argument in a post-condition.

```
interface Stack {
    void push (int i);
    int pop ();
}
```

With only *push* and *pop* operations available in the interface, it is impossible to specify that, after a *push*, the top element in the stack is the element that was just pushed. But, if we augment the interface with a *top* operation that reveals the topmost item on the stack (without removing it), then we can specify that *push* adds items to the top of the stack:

```
interface Stack {
    void push (int x);
    @post { x = this.top() }
    int pop ();
    int top ();
}
```

In summary, we do not restrict the language of contracts. This makes the contract language as flexible as possible; contract expression evaluation may even contribute to the final result of a computation. Despite the flexibility of the contract language, not all desirable contracts are expressible. Some contracts are inexpressible because they may involve checking undecidable properties, while others are inexpressible because the interface does not permit enough observations. Next we need to study how Java's class and interface type system interacts with contracts.

3.2 Multiple Interfaces

A single object may play several roles. For example, an object may be used in two different contexts with different methods, because the two different contexts may need to share state. Java, therefore, allows each class to implement more than one interface. In Contract Java, this means that a single method may have more than one pre-condition and more than one post-condition, because different interfaces may impose different contracts on the same method.

Since pre-conditions are enforced based on the type at the method application site and since each caller uses an object at a single type, only one of the pre-conditions for each method is required to be satisfied when the method is called. Consider this program:

```
interface I {
    int m (int x)
    @pre {x > 8}
}

interface J {
    int m (int x)
    @pre { x < 88 }
}

class C implements I, J {
    int m (int x) { · · · }
}
```

If an instance of class *C* has type *I*, the argument to *m* must be greater than 8. Conversely, if an instance of *C* is treated as a *J*, the argument to *m* must be less than 88. Due to Java's type system, however, an instance of C cannot have more than one type, which means that *x* never has to meet both pre-conditions simultaneously.

Analogously, only one post-conditions must ever be met. It is also determined by the calling context.

We used the "principle of least astonishment" to guide our choice in this decision. That is, if a client of a library only uses one aspect of library, the code should not trigger error messages about failed contracts for unused aspects of the library.

3.3 Substitutability

Java's type system defines a notion of object *substitutability*, that is, the ability for an instance of one class to take the place of an instance of another class or to play a role defined by some interface. Specifically, the type system allows an object to be used in any context when its type is:

- any superclass of the class,
- any interface that the object implements, or
- any interface that any superclass of the object implements.

Both the second and third form of substitutability pose problems in the presence of interface-based contracts. This section discusses a problem with the second item and section 3.8 discusses a different problem with the third item.

The following program demonstrates how using a class where an interface is expected complicates the semantics of Contract Java:

```
interface PosCounter {
   int getValue();
   @post { getValue > 0 }
   void inc();
   void dec();
   @pre { getValue() > 0 }
}

class Counter implements PosCounter {
   int val = 0;

   int getValue() { return val }
   void inc() { val = val + 1 }
   void dec() { val = val − 1 }
}

Main () {
   Client1 c1 = new Client1();
   Client2 c2 = new Client2();
   Client3 c3 = new Client3();
   PosCounter pc = c1.makeCounter();
   Counter c = c2.m(pc);
   c3.m(c);
}
```

```
class Client1 {
   PosCounter makeCounter() {
      return new Counter();
   }
}

class Client2 {
   Counter m(PosCounter c) {
      return (Counter)c;
   }
}

class Client3 {
   void m(Counter c) {
      c.dec();
   }
}
```

This program defines the class *Counter*, which implements the *PosCounter* interface. The *Counter* class keeps track of a single integer and supports increment and decrement operations for that integer. The *PosCounter* interface promises that the counter never delivers or contains a number below 1. It enforces that promise by adding a pre-condition to the *dec* method.

The *Client1* class creates instances of *Counter*, but it returns them with type *PosCounter*. The *Client2* accepts instances of *PosCounter*, casts them to *Counter* and returns them. Finally, *Client3* accepts instances of *Counter* and decrements them. When *Main* creates an instance of *Client1* and uses it to create a counter, *Main* receives a counter with the pre-condition constraint on *dec* from the *PosCounter* interface. Then, *Main* passes the *PosCounter* to *Client2*. Then, when the counter is passed to *Client3*, it is decremented and the pre-condition on *dec* fails.

According to our earlier description, the fault for post-condition failure should be the method with the post-condition, *dec*. In this case, however, the *PosCounter* was temporarily treated as a *Counter*, which is the true cause of the violation of *dec*'s post-condition. Rather than blaming *Client3*, the class that decremented the *Counter*, we should blame *Client2*, the class that cast the *PosCounter* to a *Counter*. That is, when *Client2* casts *c* to *Counter*, it should take the responsibility for any violations of *c*'s *PosCounter* contracts, from now on. So, when *Client3* decrements the counter, the blame should fall on *Client2*.

If we wish Contract Java to blame *Client2*, the language must track which class casts the counter object to *C*. One way to do this is to create a *C_Proxy* class:

```
class C_Proxy {
  PosCounter c;
  C_Proxy(PosCounter _c) { c = _c }

  void dec() {
    if (c.getValue() < 0)
      error("Client2")
      c.dec()
  }
  void inc() { c.inc() }
  int getValue() { c.getValue() }
}
```

The proxy class duplicates the pre-condition on *PosCounter*, but blames *Client2*, rather than the client, if the pre-condition fails.

These proxy classes must be created whenever an implicit or explicit cast lifts a pre-condition. That is, the compiler statically detects each place where these casts happen and inserts code to create the proxy class at that point. These proxys, however, are a subtle change to the semantics of Java programs. Java supports object identity, via the == operation. If Contract Java were to wrap objects, the unwrapped object and the wrapped object would no longer be equal under Java's == operation. A plain Java cast, however, does not affect the identity of an object. This problem can be fixed if every reference to == is changed into a method call that checks if the object is wrapped and, if it is, temporarily unwraps it to do the comparison. This solution is flawed for two reasons. First, pre-compiled legacy code does not call the new method that replaces ==. Second, a simple efficient check, namely pointer equality, is turned into an expensive method call.

Since contract enforcement across casts would introduce incompatibilities between Java and Contract Java, we omit it. That is, no blame would be assigned in the previous example, instead, the pre-condition on the counter is ignored for the dynamic extent of the call to *Client.m*.

3.4 Strengthening Pre-conditions

Strengthening the pre-condition of a method in a derived interface is quite useful. Consider the following pair of interfaces:

```
interface I {
  Object visit (IVisitor i)
}

interface J extends I {
  Object visit (IVisitor i)
  @pre {i instanceof JVisitor}
}
```

These two interfaces are extracted from the extensible visitor pattern [15]. The interface *I* defines a *visit* method that accepts instances of *IVisitor*. The *J* interface is derived from *I* and also has a *visit* method, except that it requires the arguments to be instances of *JVisitor*, an extension of *IVisitor*. Since the type system does not allow the derived *visit* method to have a different type than that of the super method, the programmer can only specify the input's type with a pre-condition.

Unfortunately, strengthening a pre-condition can cause problems. Consider the following continuation of the above program:

```
class Client {
  void m (I i)
    i.visit(ivisitor);
```

```
    }
}

class Jc implements J { ··· }

class Main {
    void main() {
        Client c = new Client();
        J j = new Jc();

        c.m(j);
    }
}
```

In this program, *Client* is written to use instances of the *I* interface with some unspecified visitor object (*ivisitor*). But, when *main* calls *c*'s method *m*, it passes an instance of *Jc*, which has been implicitly cast to *I*. The eventual call to *visit* violates the contract on *Jc*. Because the client issues the call to visit, it is blamed.

We could allow *J* to strengthen *I* by requiring each class implementing *J*, no matter if it is treated as instance of *I* or not, to check *J*'s pre-condition on *visit*. This violates our third design criterion. In this particular example, *Client* was only designed to match *I*, so it should not be blamed for the contract violation of a *J* object.

Another approach is to adapt the proxy solution from the section 3.3. This technique would enable us to properly blame *main*. We reject this alternative again for the same reasons as discussed above.

Contract Java, instead, disallows pre-condition strengthening, because there seems to be no way of supporting it without compromising our goal of Java compatibility.

3.5 Weakening Pre-conditions

As with strengthening pre-conditions, it is important to be able to weaken them. Consider this program:

```
interface Shape { ··· }
class Moon implements Shape { ··· }
class Square implements Shape { ··· }

interface ConcaveOnly {
        public void ConcaveOnlyOp(Shape s);
        @pre s instanceof Square
}
```

It consists of a *Shape* datatype with two variants: *Moon* and *Square*. It supports a single operation, *ConcaveOnlyOp* which only makes sense on concave shapes. We omit the details of such an operation; it is important that the precondition for *ConcaveOnly* ensures that only the concave shapes are allowed as inputs for the operation.

Then, when another programmer extends the *Shape* datatype with another concave shape, he must also extend the pre-condition on *ConcaveOnlyOp*, as below:

```
class Circle implements Shape { ··· }

interface ConcaveOnly-Circle extends ConcaveOnly {
        public void ConcaveOnlyOp(Shape s);
        @pre (s instanceof Square) || (s instanceof Circle)
}
```

This precondition implies the pre-condition of *ConcaveOnly*.

Because Contract Java's contract expressions are arbitrary Java expressions and not in a special-purpose sublanguage, there is no decision procedure to determine if one of Contract Java's contracts is weaker than another, so Contract Java cannot simultaneously provide pre-condition weakening and dis-allow pre-condition strengthening. So, Contract Java requires that pre-conditions for derived methods are syntactically the same as their overridden counterparts.

3.6 Strengthening Post-conditions

As with pre-conditions, sometimes it is important to strengthen a method's post-condition. To see why, consider the factory method design pattern [8]. An instantiation of the pattern consists of a single method that abstracts over the creation of instances of some class. Here is a program fragment illustrating a use of the pattern:

```
interface I {
  I createInstance ()
}

class C implements I {
  I createInstance () {
    new C()
  }
}
```

In our example, the *createInstance* method of *C* produces classes that implement *I*. This pattern is used so that derived classes can override *createInstance* to create instances of the derived class:

```
interface J extends I {
   I createInstance ()
   @post { createInstance instanceof J }
}

class D extends C implements J {
  C createInstance () {
    return new D()
  }
}
```

In this case, the *D* class overrides *createInstance* and creates an instance of *D*, instead of *C*. Due to the typing discipline of Java, the overriding definition of *createInstance* in *D* must have the same type as the method it overrides in *C*. Similarly, *J*'s method *createInstance* must have *I* as the result type. So, if the programmer is to provide a stronger guarantee for components that use the *J* interface, the guarantee must be in a contract. In this case, the programmer has strengthened *createInstance* by mandating that its result must be an instance of *J*.

This kind of strengthening does not cause the same problems that pre-condition strengthening does and is easily accommodated in Contract Java.

3.7 Weakening Post-conditions

Supporting weakened post-conditions poses problems for Contract Java. Consider this program (a variation of the square root example in section 3.1):

```
interface Number { ··· }
class FloatN implements Number {
  float getFloat()
  ...
}

interface RootFloat {
  float getValue();

  Number sqRoot();
  @post {(sqRoot instanceof FloatN) &&
          (sqRoot.getFloat() > 0f)}
}
class Root implements RootFloat { ··· }
```

It consists of a *Number* interface and a *FloatN* class that implements the *Number* interface. It also has a *RootFloat* class that calculates the square root of a number. The post-condition of the *sqRoot* method ensures that square roots are always positive.

Next, the program is extended with complex numbers:

```
class ComplexN implements Number { ··· }

interface ComplexRootFloat extends RootFloat {
  float getValue();

  Number sqRoot();
  @post {(sqRoot instanceof ComplexN) ||
         (sqRoot instanceof FloatN && sqRoot.getFloat() > 0f)}
}

class ComplexRoot implements ComplexRootFloat { .. }

Main () {
  ((RootFloat)(new ComplexRoot(-1.0f))).sqRoot();
}
```

The extension adds a *ComplexN* class to represent complex numbers and an extension of the *RootFloat* class to calculate complex roots. Thus, the post condition on the *sqRoot* method must be weakened to allow complex results.

In Java, however, an instance of *ComplexRootFloat* can be implicitly cast to *RootFloat*. This allows instances of *ComplexRoot* class to be cast to *RootFloat*. So, these instances can be forced to meet the more stringent requirements of *RootFloat*, and are incorrectly blamed for returning complex results.

One solution to this dilemma is to use proxy classes, as described in section 3.3. These proxies could be used to blame the class that casts the instance of *ComplexRoot* to *RootFloat*, instead of *ComplexRoot* itself. As before, we reject this solution in order to maintain compatibility with Java. Instead, we disallow post-condition weakening.

3.8 Implied Interface Inheritance

Section 3.3 described three ways in which Java's type system defines substitutability and a problem with the second. This section describes another problem with Java's notion of substitutability. A typical Java programmer will derive one class from another to re-use the implementation of the original class. For example, one class may implement a list; a derived class may re-use the list to implement a finite mapping. Similarly, a bignum class may extend the class for arrays, because bignums are most easily implemented as arrays of digits.

Even though this style of inheritance is designed for code re-use, Java's type-system also treats it as substitutability with respect to interfaces. That is, Java allows the finite mapping to be treated as a list and the bignum to be treated as an array. The following picture demonstrates the issue in the abstract:

```
programmer: C implements I                       java: C implements I
                 ‖              ⟶                          ‖
                 ‖          elaboration                    ‖
            D implements I                            D implements I,J
```

The left-hand side shows what the programmer writes down: an interface *I*, a class *C* that implements *I*, and a class *D* that is derived from *C*. Java, however, translates this program into the right-hand side diagram. Thus, the class *D* also implements *I*, even if this is not the programmer's intent.

Implicit interface inheritance induces several problems for contracts. Consider these two interfaces:

```
interface Array^i {
  int getSize ()

  Object lookup (int i)
  @pre { i < this.getSize() }
```

```
    void update : (Object o, int i)
    @pre { i < this.getSize () }
}

interface ResizingArray^i {
    int getSize ()

    Object lookup (int i)
    @pre { i < getSize() }

    void update (Object o, int i)
}
```

The first specifies a standard array class. The second is the same as the first, except it lifts the restriction that *update*'s argument be in range; thus implementations can dynamically resize the array as new elements are added. *A priori*, these two interfaces are unrelated.

It is natural to want to reuse a class implementing $Array^i$, say *AC*, for a class that implements $ResizingArray^i$. For this, a programmer would create a new class *RAC* that extends *AC*. Unfortunately, according to the semantics of Java, *RAC* must also implement $Array^i$. Thus, the derived class cannot eliminate the pre-condition on *update*.

Contract Java could fix this problem by changing the semantics of Java's induced interface inheritance. That is, we could say that in Contract Java *RAC* does not implement *AC*. Since this violates one of our design goals for Contract Java, we do not pursue this option.

Since the semantics of Java require *RAC* to implement both $Array^i$ and $ResizingArray^i$, Contract Java must also enforce both $Array^i$ and $ResizingArray^i$'s contracts on *RAC*. That is, *RAC* must support both the pre-condition on from $Array^i$ and the (empty) pre-condition from $ResizingArray^i$. As discussed in section 3.5, however, it is illegal in Contract Java for a single method to have two different pre-conditions, so this program is syntactically illegal in Contract Java.

4 Implementing Contract Java

To specify Contract Java, we define an elaboration into plain Java. More specifically, we modify the type elaboration phase of a conventional Java model [7] so that the compilation of classes and method calls can account for contracts in interfaces. The other changes follow from this first one. The following subsections explain the major ideas; the appendix specifies the details.

4.1 Elaborating Plain Contracts

Figure 3 illustrates how Contract Java is elaborated into Java. The top half represents the Contract Java version of the example of section 2. The contracts for *enq* and *deq* have been shifted to the *Queue* interface; the *Observer* interface has been enriched with contracts for the *onDeq* method.

The bottom half of figure 3 represents the elaboration of the classes in the example. The figure illustrates that a method contract in an interface causes the compiler to insert a new method into the class that implements the interface. We call this new method an *enforcer*. For example, the method contract for *enq* generates the *enq_Queue* enforcer; the contract for *onDeq* leads to the *onDeq_Observer* enforcer. The enforcers test the pre- and post-conditions as necessary and signal errors as necessary. Those enforcers that test a pre-condition consume an additional argument: information about the source of the call. Error messages use this source information.

Each method call is elaborated according to the type of the object. If the type is an interface and the method is annotated in that interface, the matching enforcer is called instead. Further, if the annotation is a pre-condition, the call also consumes information about the call context for potential blame assignment. The calls to *deq* in *onEnq* and *enq* in *Main* illustrate this kind of elaboration.

An evaluation of this program will detect that *onEnq* violates its contract and will correctly blame class *O* for the failure.

```
interface Observer {
    void onEnq (Queue q);
    @post { ! q.isEmpty() }
    void onDeq (Queue q);
}

interface Queue {
    boolean isEmpty();
    void enq(int i);
    @post { ! isEmpty() }
    void deq();
    @pre { ! isEmpty() }
    void setObserver(Observer o);
}

class Q implements Queue {
    private Observer o;
    public boolean isEmpty() { ··· }
    public void enq(int i) {
        ···
        o.onEnq(this)
    }
    public int deq() { ··· }
    public void setO(Observer o) {
      ··· }
}
```

```
class O implements Observer {
    public void onEnq(Queue q) {
        q.deq()
    }

    public void onDeq(Queue q) { ··· }
}
```

```
class Main {
    public static void main () {
        Queue q = new Q();
        Observer o = new O();
        q.setO(o);
        q.enq(1)
    }
}
```

```
class Q implements Queue {
    private Observer o;
    public boolean isEmpty() { ··· }
    public void enq_Queue(int i) {
      this.enq(i);
      if (isEmpty())
          error("Q violates Queue");
    }
    public void enq(int i) {
        ···
        o.onEnq_Observer(this)
    }

    public void deq_Queue(string source) {
      if (isEmpty())
          error(source);
      this.deq();
    }
    public int deq() { ··· }

    public void setO(Observer o) { ··· }
}
```

```
class O implements Observer {
    public void onEnq_Observer(Queue q) {
      this.onEnq(q);
      if (isEmpty())
          error("O violates Observer");
    }
    public void onEnq(Queue q) {
        q.deq_Queue("O")
    }

    public void onDeq(Queue q) { ··· }
}
```

```
class Main {
    public static void main () {
        Queue q = new Q();
        Observer o = new O();
        q.setO(o);
        q.enq_Queue(1)
    }
}
```

Figure 3: Compiling Contract Java Programs

4.2 Syntactic Errors

Contract Java introduces a new form of syntactic error. If interface *J* extends interface *I*, then *J* may not mention one of *I*'s methods with new pre-conditions. As sections 3.4 and 3.5 explain, modifying preconditions violates basic expectations concerning contracts and contract enforcement.

4.3 Classes with Multiple Interfaces

A class can implement several interfaces: those mentioned in an **implements** clauses and those inherited via an **extends** clause. For each method with a contract in each implemented interface, the class must contain an enforcer. Thus, for

class *C* **implements** *I*, *J* $\{ \cdots \}$

C contains two enforcers for method *m* if both *I* and *J* have contracts for *m*. Similarly, for

class *C* **implements** *I* $\{ \cdots \}$

class *D* **extends** *C* **implements** *J* $\{ \cdots \}$

D contains two enforcers for *m* as well, if both *I* and *J* specify contracts for *m*.

4.4 Collecting Postconditions

As the elaborator checks the interface DAG, it must collect the postconditions on all methods along each derivation path. That is, if *J* extends *I* and if *J* adds a post-condition for some method *m* from *I*, then the enforcer for *m* relative to *J* must check both post-conditions. The conditions are merely **and**ed together into the enforcer.

A naive implementation of the last two steps could lead to a quadratic increase in the size of the program. Factoring out the post-conditions in separate methods overcomes this problem in a natural manner.

5 Related Work

The software engineering literature contains a large number of publications on integrating assertions into programs. Their focus typically concerns the relationship to program specifications, the usefulness of certain classes of assertions, and their efficient implementation in all kinds of programming languages.

Our focus is narrower. We explore a mechanism for adding contracts to class-based object-oriented programming languages. In this regard, the research literature is sparser. The relevant work concerns ADL [24], Eiffel [22, 27], Handshake [4], iContract [14], jContractor [12], and JMSAssert [21]

Eiffel and ADL permit programmers to specify contracts for methods in classes, not in interfaces. As outlined in section 2, this approach suffers from serious problems. Technically, an Eiffel programmer can avoid some of these problems by using both multiple inheritance and abstract classes. First, an abstract class can implement a template-and-hook pattern and thus superimpose contracts on a method in derived concrete classes. Second, multiple inheritance is needed to mimic Java's "implements" clause. This solution, however, is complex and error prone. In addition, it is only a programming protocol and not a compiler enforced mechanism.

Of the remaining systems, iContract and JMSAssert are the most advanced ones. Unlike Eiffel and ADL, iContract and JMSAssert do permit programmers to add contracts to interfaces. Unfortunately, they interpret interface contracts incorrectly. In particular, when a class declares that it implements an interface, all of the interface's pre- and post-condition checks are compiled into the corresponding methods of the class. This leads to incorrectly assigned blame. Consider the example from section 3.2. In this example, it is important that a method that accepts an instance of *I* not be blamed for failing to meet *J*'s pre-conditions, since *I* and *J* are unrelated interfaces. In both iContract and JMSAssert, however, a class that implements both *I* and *J* will blame callers for violations of *J*'s pre-conditions, even if they treat the instance exclusively as an *I*. Even worse, if *I* and *J* have mutually unsatisfiable contracts, it is impossible to write a class that implements both *I* and *J* in either iContract and JMSAssert.

Also, iContract and JMSAssert fail to handle the example in section 3.3 properly. They both blame *Client3* for the violation, instead of *Client2*. Even worse, since *Client2* is no longer on the stack, they give the programmer no help in tracking down the true violator of the broken contract.

We believe Handshake falls into the same category as Eiffel and that jContractor falls into the same category as iContract. Unfortunately, the design documents [4, 12] leave this point open. Worse, no implementations are available for testing.

From a wider perspective, Rosenblum's work [26] on programming with assertions is the most closely related research. Based on his experience of building an assertion preprocessor for C and using it on several projects, he identifies the use of assertions as contracts for functions as critical. Furthermore, he identifies several important classes of contracts, especially, consistency checks across arguments, value dependencies, and subrange (or subset) membership of data. Our work can be interpreted as an adaption of Rosenblum's work to Java. The adaption is complex due to our desire to deal with Java's class and interface hierarchies.

Parnas [25] was the first to recognize that languages for building large systems must support a linguistic mechanism for components and that components should come with contracts in the form of total correctness assertions. Anna [20] can be understood as a representative implementation of Parnas's proposal on modules. The ML family of programming languages [17, 23] provides the most sophisticated form of module mechanisms. Alas, the designers of ML almost exclusively focus on type specifications in module interfaces and ignore other forms of assertions. We consider our work complementary to this work on modules and module interfaces, especially, because module languages cannot express class and interface hierarchies.

6 Conclusion and Future Work

In this paper, we have analyzed existing integrations of contracts as assertions into class-based languages and have proposed an alternative approach. Our proposal represents a compromise between language design and software engineering concerns. Although we believe that our result is superior to previous work along these lines, it leaves open many questions.

The first question we plan to address is how practical Contract Java really is. Using an implementation of Contract Java we wish to uncover a discipline of contracts similar to that developed by Rosenblum [26]. Furthermore, we plan to study the expressiveness of our assertion language, its effectiveness in finding bugs, and the cost of contracts in realistic programs.

Our second question concerns the design of a Java variant that is better suited for the specification of contracts. The design of Contract Java required several compromises concerning the enforcement of contractual invariants for the sake of consistency with Java's semantics. Conversely, we can study how Java should be changed to allow a stricter enforcement of contracts.

Finally, we are also interested in developing a specialized language of contracts and a contract proof system in the spirit of our static debugger [6]. A static debugger uses a program analysis to paint each primitive operation (*e.g.*, arithmetic operations, function applications, method calls) in red or green. A green primitive operation means the operation is provably used according to its stated invariant. A red operation indicates that the analysis cannot prove that the operation's safety conditions are met. In that case, the static debugger can, at the programmer's request, draw the value flow graph on top of the program text, indicating what flow of values might trigger the potential error. The programmer can then determine if this is truly an error or if the analysis is too weak to prove the safety of the application. Our contract-based proof system would generalize this form safety analysis to component and class contracts.

Acknowledgments: Thanks to Matthew Flatt and Clemens Szyperski for helpful discussions.

References

[1] Beugnard, A., J.-M. Jézéquel, N. Plouzeau and D. Watkins. Making components contract aware. In *IEEE Software*, pages 38–45, june 1999.

[2] Cardelli, L. Type systems. In Tucker, A. B., editor, *The Computer Science and Engineering Handbook*, pages 2208–2236. CRC Press, 1997.

[3] Clinger, W. and J. Rees. The revised^4 report on the algorithmic language Scheme. *ACM Lisp Pointers*, 4(3), July 1991.

[4] Duncan, A. and U. Hölze. Adding contracts to java with handshake. Technical Report TRCS98-32, The University of California at Santa Barbara, December 1998.

[5] Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. In *Proceedings of Theoretical Computer Science*, pages 235–271, 1992.

[6] Flanagan, C., M. Flatt, S. Krishnamurthi, S. Weirich and M. Felleisen. Catching bugs in the web of program invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 23–32, May 1996.

[7] Flatt, M., S. Krishnamurthi and M. Felleisen. Classes and mixins. In *ACM Conference Principles of Programming Languages*, Janurary 1998.

[8] Gamma, E., R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[9] Gomes, B., D. Stoutamire, B. Vaysman and H. Klawitter. *A Language Manual for Sather 1.1*, August 1996.

[10] Harper, R. and J. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2):211–252, 1993. Earlier version appears as "The Essence of ML" in *Proc. 15th ACM Symp. on Principles of Programming Languages,* 1988, pp. 28–46.

[11] Holt, R. C. and J. R. Cordy. The Turing programming language. In *Communications of the ACM*, volume 31, pages 1310–1423, December 1988.

[12] Karaorman, M., U. Hölzle and J. Bruno. jContractor: A reflective java library to support design by contract. In *Proceedings of Meta-Level Architectures and Reflection*, volume 1616 of *Lecture Notes in Computer Science*, July 1999.

[13] Kölling, M. and J. Rosenberg. *Blue: Language Specification, version 0.94*, 1997.

[14] Kramer, R. iContract — the Java design by contract tool. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, 1998.

[15] Krishnamurthi, S., M. Felleisen and D. P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *European Conference on Object-Oriented Programming*, pages 91–113, July 1998.

[16] Leroy, X. Manifest types, modules, and separate compilation. In *ACM Conference Principles of Programming Languages*, pages 109–122, Janurary 1994.

[17] Leroy, X. *The Objective Caml system, documentation and user's guide*, 1997.

[18] Liskov, B. H. and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, November 1994.

[19] Luckham, D. *Programming with Specifications*. Springer-Verlag, 1990.

[20] Luckham, D. C. and F. von Henke. An overview of Anna, a specification language for Ada. In *IEEE Software*, volume 2, pages 9–23, March 1985.

[21] Man Machine Systems. Design by contract for java using jmsassert. http://www.mmsindia.com/DBCForJava.html, 2000.

[22] Meyer, B. *Eiffel: The Language*. Prentice Hall, 1992.

[23] Milner, R., M. Tofte and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[24] Obayashi, M., H. Kubota, S. P. McCarron and L. Mallet. The assertion based testing tool for OOP: ADL2. In *International Conference on Software Engineering*, 1998.

[25] Parnas, D. L. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[26] Rosenblum, D. S. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Janurary 1995.

[27] Switzer, R. *Eiffel: An Introduction*. Prentice Hall, 1993.

[28] Szyperski, C. *Component Software*. Addison-Wesley, 1998.

A Semantics for Contract Java

```
P       = defn* e
defn    = class c extends c implements i*
            { field* meth* }
          | interface i extends i* { imeth* }
field   = t fd
meth    = t md ( arg* ) { body }
imeth   = t md ( arg* )
            @pre { e } @post { e }
arg     = t var
body    = e | abstract
e       = new c | var | null | e.fd | e.fd = e
          | e.md (e* ) | super.md (e* )
          | view t e | let { binding* } in e
          | if ( e ) e else e | true | false
          | { e ; e }
binding = var = e
var     = a variable name or this
c       = a class name or Object
i       = interface name or Empty
fd      = a field name
md      = a method name
t       = c | i | boolean
```
Surface Syntax

```
P       = defn* e
defn    = class c extends c implements i*
            { field* meth* }
          | interface i extends i* { imeth* }
field   = t fd
meth    = t md ( arg* ) { body }
imeth   = t md ( arg* )
            @pre { e } @post { e }
arg     = t var
body    = e | abstract
e       = new c | var | null
          | e : c .fd | e : c .fd = e
          | e: t .md (e* )
          | super ≡ this : c .md (e* )
          | view t e | let { binding* } in e
          | if ( e ) e else e | true | false
          | { e ; e } | { andall e }
binding = var = e
var     = a variable name or this
c       = a class name or Object
i       = interface name or Empty
fd      = a field name
md      = a method name
t       = c | i | boolean
```
After typechecking and elaboration

```
P       = defn* e
defn    = class c extends c implements i*
            { field* meth* }
          | interface i extends i* { imeth* }
field   = t fd
meth    = t md ( arg* ) { body }
imeth   = t md ( arg* )
arg     = t var
body    = e | abstract
e       = new c | var | null
          | e : c .fd | e : c .fd = e
          | e.md (e* )
          | super ≡ this : c .md (e* )
          | view t e | let { binding* } in e
          | if ( e ) e else e | true | false
          | { e ; e } | { andall e }
          | contractviolated(e)
binding = var = e
var     = a variable name or this
c       = a class name or Object
i       = interface name or Empty
fd      = a field name
md      = a method name
t       = c | i | boolean
```
After contracts compiled away

Figure 4: Contract Java syntax; before and after contracts are compiled away

A.1 Syntax

The syntax for our model of Contract Java is shown in Figure 4. A program P is a sequence of class and interface definitions followed by an expression. Each class definition consists of a sequence of field declarations and a sequence of method declarations, while an interface consists of method specification and their contracts. A method body in a class can be **abstract**, indicating that the method must be overridden in a subclass before the class is instantiated. Unlike Java, the body of a method is an expression whose result is the result of the method. As in Java, classes are instantiated with the **new** operator, but there are no class constructors in Contract Java; instance variables are always initialized to null. Finally, the **view** and **let** forms represent Java's casting expressions and the capability for bindings vbariables locally.

The syntax is divided up into three parts. The first is the surface syntax that the programmer uses. The second includes additional information (underlined in the figure) added by elaboration and type checking. To support evaluation, field update and field reference are annotated with the class containing the field, and calls to **super** are annotated with the class. Also, method calls are annotated with types in order to insert calls to the enforcer methods. The second syntax also adds a new form: **andall**, which is used to compile post-conditions.

The third syntax is produced by the contract compiler and is accepted by the evaluator. The **@pre** and **@post** conditions are removed from the interfaces. The third syntax also has the **contractviolated** call which blames its argument for a contract violation.

A.2 Predicates and Relations

A valid Contract Java program satisfies a number of simple predicates and relations; these are described in Figures 5 and 6. For example, the ClassesOnce(P) predicate states that each class name is defined at most once in the program

The sets of names for variables, classes, interfaces, fields, and methods are assumed to be mutually distinct. The meta-variable T is used for method signatures ($t \ldots \longrightarrow t$), V for variable lists ($var \ldots$), and Γ for environments mapping variables to types. Ellipses on the baseline ($\ldots$) indicate a repeated pattern or continued sequence, while centered ellipses ($\cdots$) indicate arbitrary missing program text (not spanning a class or interface definition).

CLASSESONCE(P)	Each class name is declared only once
	class $c \cdots$ **class** $c' \cdots$ is in $P \Longrightarrow c \neq c'$
FIELDONCEPERCLASS(P)	Field names in each class declaration are unique
	class $\cdots \{ \cdots fd \cdots fd' \cdots \}$ is in $P \Longrightarrow fd \neq fd'$
METHODONCEPERCLASS(P)	Method names in each class declaration are unique
	class $\cdots \{ \cdots md\,(\cdots)\,\{\cdots\} \cdots md'\,(\cdots)\,\{\cdots\} \cdots \}$ is in $P \Longrightarrow md \neq md'$
INTERFACESONCE(P)	Each interface name is declared only once
	interface $i \cdots$ **interface** $i' \cdots$ is in $P \Longrightarrow i \neq i'$
METHODARGSDISTINCT(P)	Each method argment name is unique
	$md\,(t_1\ var_1 \ldots t_n\ var_n)\,\{\cdots\}$ is in $P \Longrightarrow var_1, \ldots var_n$, and **this** are distinct
$\prec^{\mathsf{c}}_P$	Class is declared as an immediate subclass
	$c \prec^{\mathsf{c}}_P c' \Leftrightarrow$ **class** c **extends** $c' \cdots \{\cdots\}$ is in P
$\Subset^{\mathsf{c}}_P$	Field is declared in a class
	$\langle c.fd, t\rangle \Subset^{\mathsf{c}}_P c \Leftrightarrow$ **class** $c \cdots \{ \cdots t\ fd \cdots \}$ is in P
$\Subset^{\mathsf{c}}_P$	Method is declared in class
	$\langle md, (t_1 \ldots t_n \longrightarrow t), (var_1 \ldots var_n), e\rangle \Subset^{\mathsf{c}}_P c$
	$\Leftrightarrow$ **class** $c \cdots \{ \cdots t\ md\,(t_1\ var_1 \ldots t_n\ var_n)\,\{e\} \cdots \}$ is in P
$\prec^{\mathsf{i}}_P$	Interface is declared as an immediate subinterface
	$i \prec^{\mathsf{i}}_P i' \Leftrightarrow$ **interface** i **extends** $\cdots i' \cdots \{\cdots\}$ is in P
$\Subset^{\mathsf{i}}_P$	Method is declared in an interface
$\langle md, (t_1 \ldots t_n \longrightarrow t)\rangle \Subset^{\mathsf{i}}_P i$	
$\Leftrightarrow$ **interface** $i \cdots \{ \cdots md : (t_1\ arg_1) \ldots (t_n\ arg_n)\ \{ e^{\cdot}b \} \longrightarrow (t\ arg)\ \{ e^{\cdot}a \} \cdots \}$ is in P	
$\prec\!\!\!\prec^{\mathsf{c}}_P$	Class declares implementation of an interface
	$c \prec\!\!\!\prec^{\mathsf{c}}_P i \Leftrightarrow$ **class** $c \cdots$ **implements** $\cdots i \cdots \{\cdots\}$ is in P
$\leq^{\mathsf{c}}_P$	Class is a subclass
	$\leq^{\mathsf{c}}_P \equiv$ the transitive, reflexive closure of $\prec^{\mathsf{c}}_P$
COMPLETECLASSES(P)	Classes that are extended are defined
	$\mathrm{rng}(\prec^{\mathsf{c}}_P) \subseteq \mathrm{dom}(\prec^{\mathsf{c}}_P) \cup \{\mathit{Object}\}$
WELLFOUNDEDCLASSES(P)	Class hierarchy is an order
	$\leq^{\mathsf{c}}_P$ is antisymmetric
CLASSMETHODSOK(P)	Method overriding preserves the type
	$(\langle md, T, V, e\rangle \Subset^{\mathsf{c}}_P c$ and $\langle md, T', V', e'\rangle \Subset^{\mathsf{c}}_P c') \Longrightarrow (T = T'$ or $c \not\leq^{\mathsf{c}}_P c')$
$\in^{\mathsf{c}}_P$	Field is contained in a class
$\langle c'.fd, t\rangle \in^{\mathsf{c}}_P c$	
$\Leftrightarrow \langle c'.fd, t\rangle \Subset^{\mathsf{c}}_P c'$ and $c' = \min\{c'' \mid c \leq^{\mathsf{c}}_P c''$ and $\exists t'$ s.t. $\langle c''.fd, t'\rangle \Subset^{\mathsf{c}}_P c''\}$	
$\in^{\mathsf{c}}_P$	Method is contained in a class
$\langle md, T, V, e\rangle \in^{\mathsf{c}}_P c$	
$\Leftrightarrow (\langle md, T, V, e\rangle \Subset^{\mathsf{c}}_P c'$ and $c' = \min\{c'' \mid c \leq^{\mathsf{c}}_P c''$ and $\exists e', V'$ s.t. $\langle md, T, V', e'\rangle \Subset^{\mathsf{c}}_P c''\})$	

Figure 5: Predicates and relations in the model of Contract Java (Part I)

P. The relation $\prec^{\mathsf{c}}_P$ associates each class name in P to the class it extends, and the (overloaded) $\Subset^{\mathsf{c}}_P$ relations capture the field and method declarations of P.

The syntax-summarizing relations induce a second set of relations and predicates that summarize the class structure of a program. The first of these is the subclass relation $\leq^{\mathsf{c}}_P$, which is a partial order if the COMPLETECLASSES(P) and WELLFOUNDEDCLASSES(P) predicates hold. In this case, the classes declared in P form a tree that has *Object* at its root.

If the program describes a tree of classes, we can "decorate" each class in the tree with the collection of fields and methods that it accumulates from local declarations and inheritance. The source declaration of any field or method in a class can be computed by finding the *minimum* superclass (*i.e.*, farthest from the root) that declares the field or method. This algorithm is described precisely by the $\in^{\mathsf{c}}_P$ relations. The $\in^{\mathsf{c}}_P$ relation retains information about the source class of each field, but it does not retain the source class for a method. This reflects the property of Java classes that fields cannot be overridden (so instances of a subclass always contain the field), while methods can be overridden (and may become inaccessible).

Interfaces have a similar set of relations. The superinterface declaration relation $\prec^{\mathsf{i}}_P$ induces a subinterface relation $\leq^{\mathsf{i}}_P$. Unlike classes, a single interface can have multiple proper superinterfaces, so the subinterface order forms a DAG instead of a tree. The set methods of an interface, as described by $\in^{\mathsf{i}}_P$, is the union of the interface's declared methods and the methods of its superinterfaces.

Classes and interfaces are related by **implements** declarations, as captured in the $\prec\!\!\!\prec^{\mathsf{c}}_P$ relation. This relation is a

$\leq^{\mathsf{i}}_P$	Interface is a subinterface $\leq^{\mathsf{i}}_P \equiv$ the transitive, reflexive closure of $\prec^{\mathsf{i}}_P$
COMPLETEINTERFACES(P)	Extended/implemented interfaces are defined $\mathrm{rng}(\prec^{\mathsf{i}}_P) \cup \mathrm{rng}(\prec\!\!\prec^{\mathsf{c}}_P) \subseteq \mathrm{dom}(\prec^{\mathsf{i}}_P) \cup \{\textit{Empty}\}$
WELLFOUNDEDINTERFACES(P)	Interface hierarchy is an order $\leq^{\mathsf{i}}_P$ is antisymmetric
$\ll^{\mathsf{c}}_P$	Class implements an interface $c \ll^{\mathsf{c}}_P i \Leftrightarrow \exists c', i'$ s.t. $c \leq^{\mathsf{c}}_P c'$ and $i' \leq^{\mathsf{i}}_P i$ and $c' \prec\!\!\prec^{\mathsf{c}}_P i'$
INTERFACEMETHODSOK(P)	Interface inheritance or redeclaration of methods is consistent $\langle md, T\rangle \in\!\!\in^{\mathsf{i}}_P i$ and $\langle md, T'\rangle \in\!\!\in^{\mathsf{i}}_P i'$ $\Longrightarrow (T = T'$ or $\forall i''(i'' \not\leq^{\mathsf{i}}_P i$ or $i'' \not\leq^{\mathsf{i}}_P i'))$
$\in^{\mathsf{i}}_P$	Method is contained in an interface $\langle md, T\rangle \in^{\mathsf{i}}_P i \Leftrightarrow \exists i'$ s.t. $i \leq^{\mathsf{i}}_P i'$ and $\langle md, T\rangle \in\!\!\in^{\mathsf{i}}_P i'$
CLASSESIMPLEMENTALL(P)	Classes supply methods to implement interfaces $c \prec\!\!\prec^{\mathsf{c}}_P i \Longrightarrow (\forall md, T\ \langle md, T\rangle \in^{\mathsf{i}}_P i \Longrightarrow \exists e, V'$ s.t. $\langle md, T, V', e\rangle \in^{\mathsf{c}}_P c)$
NOABSTRACTMETHODS(P, c)	Class has no **abstract** methods (can be instantiated) $\langle md, T, V, e\rangle \in^{\mathsf{c}}_P c \Longrightarrow e \neq \textbf{abstract}$
$\leq_P$	Type is a subtype $\leq_P \equiv \leq^{\mathsf{c}}_P \cup \leq^{\mathsf{i}}_P \cup \ll^{\mathsf{c}}_P$
$\in_P$	Field or Method is in a type (method/interface) $\langle md, T\rangle \in_P i \Leftrightarrow \langle md, T\rangle \in^{\mathsf{i}}_P i$
$\in_P$	Field or Method is in a type (method/class) $\langle md, T\rangle \in_P c \Leftrightarrow \exists T, V\, s.t. \langle md, T, V, e\rangle \in^{\mathsf{c}}_P c$
$\in_P$	Field or Method is in a type (field/type) $\langle c.fd, t\rangle \in_P c \Leftrightarrow \langle c.fd, t\rangle \in^{\mathsf{c}}_P c$
$\cdot\!\propto_P$	Pre-condition contract is in method in interface $e \cdot\!\propto_P \langle i,md\rangle \Leftrightarrow$ **interface** $i \{\cdots t\ md\ arg \ldots$ @**pre** $\{ e \}$ @**post** $\{ e' \} \cdots \}$ is in P
$\propto\!\cdot_P$	Post-condition contract is in method in interface $e \propto\!\cdot_P \langle i,md\rangle \Leftrightarrow$ **interface** $i \{\cdots t\ md\ arg \ldots$ @**pre** $\{ e' \}$ @**post** $\{ e \} \cdots \}$ is in P
$\odot_P$	Contract is declared in an interface $\langle md, \langle\langle t_1, arg_1\rangle, \ldots, \langle t_n, arg_n\rangle\rangle, t, e_b, e_a\rangle \odot_P i$ $\Leftrightarrow$ **interface** $i \cdots \{ \cdots t\ md : (t_1\ arg_1) \ldots (t_n\ arg_n)$ @**pre** $\{ e_b \}$ @**post** $\{ e_a \} \cdots \}$ is in P
$\otimes_P$	Contract is contained in an interface $\langle md, \langle\langle t_1, arg_1\rangle, \ldots, \langle t_n, arg_n\rangle\rangle, t, e_b, e_a\rangle \otimes_P i$ $\Leftrightarrow \exists i'$ s.t. $i \leq^{\mathsf{i}}_P i'$ and $\langle md, \langle\langle t_1, arg_1\rangle, \ldots, \langle t_n, arg_n\rangle\rangle, t, e_b, e_a\rangle \odot_P i'$
OVERRIDENCONTRACTSIDENTICAL(P)	Overriden method's pre-condition contracts are the same $e \cdot\!\propto_P \langle i, meth\rangle$ and $e' \cdot\!\propto_P \langle i', meth'\rangle \Longrightarrow (e = e'$ or $\forall j. j \leq^{\mathsf{i}}_P i \Longrightarrow j \not\leq^{\mathsf{i}}_P i')$

Figure 6: Predicates and relations in the model of Contract Java (Part II)

set of edges joining the class tree and the interface graph, completing the sub*type* picture of a program. A type in the full graph is a subtype of all of its ancestors.

Finally, the pre and post-condition relation OVERRIDENCONTRACTSIDENTICAL(P) ensures that if two interfaces declare the same method, they also have the same pre-conditions.

A.3 Contract Java Type Elaboration

The type elaboration rules for Contract Java are defined by the following judgements:

$\vdash_{\mathsf{p}} P \Rightarrow P' : t$	P elaborates to P' with type t
$P \vdash_{\mathsf{d}} \textit{defn} \Rightarrow \textit{defn}'$	*defn* elaborates to *defn*$'$
$P, c \vdash_{\mathsf{m}} \textit{meth} \Rightarrow \textit{meth}'$	*meth* in c elaborates to *meth*$'$
$P, i \vdash_{\mathsf{i}} \textit{imeth} \Rightarrow \textit{imeth}'$	*imeth* in i elaborates to *imeth*$'$
$P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t$	e elaborates to e' with type t in Γ
$P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t$	e has type t using subsumption in Γ
$P \vdash_{\mathsf{t}} t$	t exists

The type elaboration rules translate expressions that access a field, call a **super** method, or call a normal method into annotated expressions (see the underlined parts of Figure 4). For field uses, the annotated expression contains the compile-time type of the instance expression, which determines the class containing the declaration of the accessed field. For **super** method invocations, the annotated expression contains the compile-time type of **this**, which determines the class that contains the declaration of the method to be invoked. For regular method calls, the annotatation contains the type of the object being called.

$\vdash_{\mathsf{p}}$

$$\frac{\begin{array}{c}\textsc{ClassesOnce}(P) \quad \textsc{InterfacesOnce}(P) \quad \textsc{MethodOncePerClass}(P) \quad \textsc{FieldOncePerClass}(P) \quad \textsc{CompleteClasses}(P) \\ \textsc{WellFoundedClasses}(P) \quad \textsc{CompleteInterfaces}(P) \quad \textsc{WellFoundedInterfaces}(P) \quad \textsc{InterfaceMethodsOK}(P) \quad \textsc{MethodArgsDistinct}(P) \\ \textsc{ClassesImplementAll}(P) \quad P \vdash_{\mathsf{d}} \mathit{defn}_j \Rightarrow \mathit{defn}'_j \text{ for } j \in [1,n] \quad P,[\,] \vdash_{\mathsf{e}} e \Rightarrow e' : t \quad \text{where } P = \mathit{defn}_1 \ldots \mathit{defn}_n\ e \end{array}}{\vdash_{\mathsf{p}} \mathit{defn}_1 \ldots \mathit{defn}_n\ e \Rightarrow \mathit{defn}'_1 \ldots \mathit{defn}'_n\ e' : t}\,[\textbf{prog}^{\mathsf{c}}]$$

$\vdash_{\mathsf{d}}$

$$\frac{P \vdash_{\mathsf{t}} t_j \text{ for } j \in [1,n] \qquad P, c \vdash_{\mathsf{m}} \mathit{meth}_k \Rightarrow \mathit{meth}'_k \text{ for } k \in [1,p]}{P \vdash_{\mathsf{d}} \textbf{class}\ c \cdots \begin{array}{l}\{\ t_1\ \mathit{fd}_1 \ \ldots\ t_n\ \mathit{fd}_n \\ \quad \mathit{meth}_1 \ \ldots\ \mathit{meth}_p\ \}\end{array} \Rightarrow \textbf{class}\ c \cdots \begin{array}{l}\{\ t_1\ \mathit{fd}_1 \ \ldots\ t_n\ \mathit{fd}_n \\ \quad \mathit{meth}'_1 \ \ldots\ \mathit{meth}'_p\ \}\end{array}}\,[\textbf{defn}^{\mathsf{c}}]$$

$$\frac{P \vdash_{\mathsf{i}} \mathit{imeth}_j \Rightarrow \mathit{imeth}_j \text{ for } j \in [1,p]}{P, i \vdash_{\mathsf{d}} \textbf{interface}\ i \cdots \{\ \mathit{imeth}_1 \ \ldots\ \mathit{imeth}_p\ \} \Rightarrow \textbf{interface}\ i \cdots \{\ \mathit{imeth}_1 \ \ldots\ \mathit{imeth}_p\ \}}\,[\textbf{defn}^{\mathsf{i}}]$$

$\vdash_{\mathsf{m}}$

$$\frac{P \vdash_{\mathsf{t}} t \qquad P \vdash_{\mathsf{t}} t_j \text{ for } j \in [1,n] \qquad P,[\textbf{this} : t_o, \mathit{var}_1 : t_1, \ldots \mathit{var}_n : t_n] \vdash_{\mathsf{s}} e \Rightarrow e' : t}{P, t_o \vdash_{\mathsf{m}} t\ \mathit{md}\,(t_1\ \mathit{var}_1 \ \ldots\ t_n\ \mathit{var}_n)\ \{\ e\ \} \Rightarrow t\ \mathit{md}\,(t_1\ \mathit{var}_1 \ \ldots\ t_n\ \mathit{var}_n)\ \{\ e'\ \}}\,[\textbf{meth}]$$

$$\frac{P \vdash_{\mathsf{t}} t \qquad P \vdash_{\mathsf{t}} t_j \text{ for } j \in [1,n]}{P, t_o \vdash_{\mathsf{m}} t\ \mathit{md}\,(t_1\ \mathit{var}_1 \ \ldots\ t_n\ \mathit{var}_n)\ \{\ \textbf{abstract}\ \} \Rightarrow t\ \mathit{md}\,(t_1\ \mathit{var}_1 \ \ldots\ t_n\ \mathit{var}_n)\ \{\ \textbf{abstract}\ \}}\,[\textbf{abs}]$$

$\vdash_{\mathsf{i}}$

$$\frac{\begin{array}{c} P \vdash_{\mathsf{t}} t \qquad P \vdash_{\mathsf{t}} t_j \text{ for } j \in [1,n] \qquad P,[\textbf{this} : i, \mathit{var}_1 : t_1, \ldots \mathit{var}_n : t_n] \vdash_{\mathsf{e}} e_b \Rightarrow e'_b : \textbf{boolean} \\ P,[\textbf{this} : i, \mathit{md} : t, \mathit{var}_1 : t_1, \ldots \mathit{var}_n : t_n] \vdash_{\mathsf{e}} e \Rightarrow e' : \textbf{boolean} \text{ for each } e \text{ in } \{ e \mid e \propto_P \langle i', \mathit{md} \rangle \text{ and } i \leq^{\mathsf{i}}_P i' \} \end{array}}{P, i \vdash_{\mathsf{i}} t\ \mathit{md}\,(\mathit{arg}_1\ t_1 \ \ldots\ \mathit{arg}_n\ t_n)\ \textbf{@pre}\ \{\ e_b\ \}\ \textbf{@post}\ \{\ e_a\ \} \Rightarrow t\ \mathit{md}\,(\mathit{arg}_1\ t_1 \ \ldots\ \mathit{arg}_n\ t_n)\ \textbf{@pre}\ \{\ e'_b\ \}\ \textbf{@post}\ \{\ \textbf{andall}\ e' \ \ldots\ \}}\,[\textbf{imeth}]$$

$\vdash_{\mathsf{e}}$

$$\frac{P \vdash_{\mathsf{t}} c \qquad \textsc{NoAbstractMethods}(P, c)}{P, \Gamma \vdash_{\mathsf{e}} \textbf{new}\ c \Rightarrow \textbf{new}\ c : c}\,[\textbf{new}^{\mathsf{c}}] \qquad \frac{\mathit{var} \in \mathrm{dom}(\Gamma)}{P, \Gamma \vdash_{\mathsf{e}} \mathit{var} \Rightarrow \mathit{var} : \Gamma(\mathit{var})}\,[\textbf{var}]$$

$$\frac{P \vdash_{\mathsf{t}} t}{P, \Gamma \vdash_{\mathsf{e}} \mathsf{null} \Rightarrow \mathsf{null} : t}\,[\textbf{null}] \qquad \frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad \langle c.\mathit{fd}, t \rangle \in_P t'}{P, \Gamma \vdash_{\mathsf{e}} e.\mathit{fd} \Rightarrow e' \underline{: c}\,.\mathit{fd} : t}\,[\textbf{get}^{\mathsf{c}}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad \langle c.\mathit{fd}, t \rangle \in_P t' \qquad P, \Gamma \vdash_{\mathsf{s}} e_v \Rightarrow e'_v : t}{P, \Gamma \vdash_{\mathsf{e}} e.\mathit{fd} = e_v \Rightarrow e' \underline{: c}\,.\mathit{fd} = e'_v : t}\,[\textbf{set}^{\mathsf{c}}]$$

Figure 7: Context-sensitive checks and type elaboration rules for Contract Java (Part I)

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad \langle \mathit{md}, (t_1 \ldots t_n \longrightarrow t) \rangle \in_P t' \qquad P, \Gamma \vdash_{\mathsf{s}} e_j \Rightarrow e'_j : t_j \text{ for } j \in [1,n]}{P, \Gamma \vdash_{\mathsf{e}} e.\mathit{md}\,(e_1 \ldots e_n) \Rightarrow e' \underline{: t'}\,.\mathit{md}\,(e'_1 \ldots e'_n) : t}\,[\textbf{call}^{\mathsf{c}}]$$

$$\frac{\begin{array}{c} P, \Gamma \vdash_{\mathsf{e}} \textbf{this} \Rightarrow \textbf{this} : c' \qquad c' \prec^{\mathsf{c}}_P c \qquad \langle \mathit{md}, (t_1 \ldots t_n \longrightarrow t), (\mathit{var}_1 \ldots \mathit{var}_n), e_b \rangle \in^{\mathsf{c}}_P c \\ P, \Gamma \vdash_{\mathsf{s}} e_j \Rightarrow e'_j : t_j \text{ for } j \in [1,n] \qquad e_b \neq \textbf{abstract} \end{array}}{P, \Gamma \vdash_{\mathsf{e}} \textbf{super}.\mathit{md}(e_1 \ldots e_n) \Rightarrow \textbf{super} \underline{\equiv \textbf{this} : c}\,.\mathit{md}(e'_1 \ldots e'_n) : t}\,[\textbf{super}^{\mathsf{c}}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t}{P, \Gamma \vdash_{\mathsf{e}} \textbf{view}\ t\ e \Rightarrow e' : t}\,[\textbf{wcast}^{\mathsf{c}}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad t \leq_P t' \text{ or } t \in \mathrm{dom}(\prec^{\mathsf{i}}_P) \text{ or } t' \in \mathrm{dom}(\prec^{\mathsf{i}}_P)}{P, \Gamma \vdash_{\mathsf{e}} \textbf{view}\ t\ e \Rightarrow \textbf{view}\ t\ e' : t}\,[\textbf{ncast}^{\mathsf{c}}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e_j \Rightarrow e'_j : t_j \text{ for } j \in [1,n] \qquad P, \Gamma[\mathit{var}_1 : t_1] \ldots [\mathit{var}_n : t_n] \vdash_{\mathsf{e}} e \Rightarrow e' : t}{P, \Gamma \vdash_{\mathsf{e}} \textbf{let}\ \{\ \mathit{var}_1 = e_1 \ \ldots\ \mathit{var}_n = e_n\ \}\ \textbf{in}\ e \Rightarrow \textbf{let}\ \{\ \mathit{var}_1 = e'_1 \ \ldots\ \mathit{var}_n = e'_n\ \}\ \textbf{in}\ e' : t}\,[\textbf{let}]$$

$$\frac{}{P, \Gamma \vdash_{\mathsf{e}} \textbf{true} \Rightarrow \textbf{true} : \textbf{boolean}}\,[\textbf{true}] \qquad \frac{}{P, \Gamma \vdash_{\mathsf{e}} \textbf{false} \Rightarrow \textbf{false} : \textbf{boolean}}\,[\textbf{false}]$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e_1 \Rightarrow e'_1 : \textbf{boolean} \quad P, \Gamma \vdash_{\mathsf{e}} e_2 \Rightarrow e'_2 : t \quad P, \Gamma \vdash_{\mathsf{e}} e_3 \Rightarrow e'_3 : t}{P, \Gamma \vdash_{\mathsf{e}} \textbf{if}\ (e_1)\ e_2\ \textbf{else}\ e_3 \Rightarrow \textbf{if}\ (e'_1)\ e'_2\ \textbf{else}\ e'_3 : t}\,[\textbf{if}] \qquad \frac{P, \Gamma \vdash_{\mathsf{e}} e_1 \Rightarrow e'_1 : t' \qquad P, \Gamma \vdash_{\mathsf{e}} e_2 \Rightarrow e'_2 : t}{P, \Gamma \vdash_{\mathsf{e}} \{\ e_1\ ;\ e_2\ \} \Rightarrow \{\ e_1\ ;\ e_2\ \} : t}\,[\textbf{seq}]$$

$\vdash_{\mathsf{s}}, \vdash_{\mathsf{t}}$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad t' \leq_P t}{P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t}\,[\textbf{sub}^{\mathsf{c}}] \qquad \frac{t \in \mathrm{dom}(\prec^{\mathsf{c}}_P) \cup \mathrm{dom}(\prec^{\mathsf{i}}_P) \cup \{\mathit{Object}, \mathit{Empty}, \textbf{boolean}\}}{P \vdash_{\mathsf{t}} t}\,[\textbf{type}^{\mathsf{c}}]$$

Figure 8: Context-sensitive checks and type elaboration rules for Contract Java (Part II)

The complete typing rules are shown in Figures 7 and 8. A program is well-typed if its class definitions and final expression are well-typed. A definition, in turn, is well-typed when its field and method declarations use legal types and the method body expressions are well-typed. Finally, expressions are typed and elaborated in the context of an environment that binds free variables to types. For example, the $\mathbf{get}^{\mathsf{c}}$ and $\mathbf{set}^{\mathsf{c}}$ rules for fields first determine the type of the instance expression, and then calculate a class-tagged field name using $\in_P$; this yields both the type of the field and the class for the installed annotation. In the $\mathbf{set}^{\mathsf{c}}$ rule, the right-hand side of the assignment must match the type of the field, but this match may exploit subsumption to coerce the type of the value to a supertype.

The rule for interface methods folds all of the post-conditions for super methods of the same name into each post-condition, using the **andall** form.

$\vdash P \rightharpoonup_{\mathsf{p}} P'$	The program P compiles to the program P'
$P \vdash \mathit{defn} \rightharpoonup_{\mathsf{d}} \mathit{defn}'$	*defn* compiles to *defn*$'$ in the program P
$P, c \vdash \mathit{intrfce} \rightharpoonup_{\mathsf{n}} \mathit{meth} \ldots$	*meth* ... check *intrfce*'s pre- and post-conditions.
$\vdash \mathit{imeth} \rightharpoonup_{\mathsf{i}} \mathit{imeth}'$	*imeth* compiles to *imeth*$'$
$P, i, c \vdash \mathit{meth} \rightharpoonup_{\mathsf{m}} \mathit{meth}'$	*meth* compiles to *meth*$'$ which enforces i's post-condition on *meth*, if any, and blames c.
$P, c \vdash e \rightharpoonup_{\mathsf{e}} e'$	e compiles to e' which blames c for contract violations

Figure 9: Contract Java Judgements

A.4 Contract Compilation

The contract compilation rules for Contract Java are defined by the judgements in figure 9.

Figure 6 contains some predicates that are used during compilation. The most important ones are: OVERRIDENCONTRACTSIDENTICAL(P), $\otimes_P$, and $\ll_P^{\varsigma}$ The first, OVERRIDENCONTRACTSIDENTICAL(P), ensures that all the preconditions for any given method are syntactically the same. The second, $\otimes_P$ provides contract information used by the translation itself. The last, $\ll_P^{\varsigma}$, gives all of the contracts for post-conditions and for objects that implement multiple interfaces. The other predicates are used to define the important three.

Figure 10 contains the rules that translate Java Contract programs into Java programs. For each method that has either a pre- or post-condition, an enforcer method is generated, via the [**interface**] rule. The enforcer accepts the same arguments as the original method, plus an additional argument that names the class that is calling the method. The enforcer first tests the pre-condition. If it fails, the enforcer blames the calling method. If it suceeds, the enforcer calls the original method. Then, when the original method returns, the enforcer checks the post-conditions (which have been folded into a single method using **andall** expressions). If they fail, the enforcer blames the class itself. If they suceed, the enforcer returns the result of the methd.

Then, all of the calls to methods whose objects have interface types are changed to calls to the enforcer, by the [**call**] rule. [Matthis writes: "how?"]

A.5 Contract Java Evaluation

The operational semantics for Contract Java is defined as a contextual rewriting system on pairs of expressions and stores. A store $\mathcal{S}$ is a mapping from **object**s to class-tagged field records. A field record $\mathcal{F}$ is a mapping from elaborated field names to values. The evaluation rules are a straightforward adaption of those for imperative Scheme [5].

The complete evaluation rules are in Figure 11. For example, the *call* rule invokes a method by rewriting the method call expression to the body of the invoked method, syntactically replacing argument variables in this expression with the supplied argument values. The dynamic aspect of method calls is implemented by selecting the method based on the run-time type of the object (in the store). In contrast, the *super* reduction performs **super** method selection using the class annotation that is statically determined by the type-checker.

$\rightharpoonup_{\mathsf{p}}$

$$\frac{P \vdash \mathit{defn}_j \rightharpoonup_{\mathsf{d}} \mathit{defn}'_j \text{ for } j \in [1,n] \quad \textsc{OverridenContractsIdentical}(P) \quad P, \mathit{main} \vdash e \rightharpoonup_{\mathsf{e}} e' \text{ where } P = \mathit{defn}_1 \ldots \mathit{defn}_n\ e}{\vdash \mathit{defn}_1 \ldots \mathit{defn}_n\ e \rightharpoonup_{\mathsf{p}} \mathit{defn}'_1 \ldots \mathit{defn}'_n\ e'}\,[\mathbf{prog}]$$

$\rightharpoonup_{\mathsf{d}}$

$$\frac{\vdash \mathit{imeth}_j \rightharpoonup_{\mathsf{i}} \mathit{imeth}'_j \text{ for } j \in [1,n]}{P \vdash \textbf{interface } i \textbf{ extends } i_1 \ldots i_l\ \mathit{imeth}_1 \ldots \mathit{imeth}_n \rightharpoonup_{\mathsf{d}} \textbf{interface } i \textbf{ extends } i_1 \ldots i_l\ \mathit{imeth}'_1 \ldots \mathit{imeth}'_n}\,[\mathbf{defn}^{\mathsf{i}}]$$

$$\frac{P, c \vdash \mathit{meth}_i \rightharpoonup_{\mathsf{m}} \mathit{meth}'_i \qquad P, c \vdash i \rightharpoonup_{\mathsf{n}} \mathit{pre\_meth}_1 \ldots \mathit{pre\_meth}_n \text{ for } i \in \{i \mid c \ll^{\mathsf{c}}_P i\}}{P \vdash \textbf{class } c \textbf{ implements } i_1 \ldots i_l\ \mathit{meth}_1 \ldots \mathit{meth}_n \rightharpoonup_{\mathsf{d}} \textbf{class } c \textbf{ implements } i_1 \ldots i_l\ \mathit{pre\_meth}_1 \ldots \mathit{pre\_meth}_n\ \mathit{meth}'_1 \ldots \mathit{meth}'_n}\,[\mathbf{defn}^{\mathsf{c}}]$$

$\rightharpoonup_{\mathsf{n}}$

$$\frac{P, c \vdash e_i \rightharpoonup_{\mathsf{e}} e''_i \text{ for } i \in [1,n] \qquad P, c \vdash e'_i \rightharpoonup_{\mathsf{e}} e'''_i \text{ for } i \in [1,n]}{\begin{array}{l} P, c \vdash \textbf{interface } i\ t_1\ md_1\ (\, t_{11}\ x_{11} \ldots t_{1j}\ x_{1j}\,)\ \textbf{@pre}\ \{\, e_1 \,\}\ \textbf{@post}\ \{\, e'_1 \,\} \ldots t_n\ md_n\ (\, t_{n1}\ x_{n1} \ldots t_{nj}\ x_{nj}\,)\ \textbf{@pre}\ \{\, e_n \,\}\ \textbf{@post}\ \{\, e'_n \,\} \rightharpoonup_{\mathsf{n}} \\ \quad \begin{array}{lcl} i\_md_1\ (\ t_{11}\ x_{11} \ldots t_{1j}\ x_{1j}\ \textbf{string}\ \mathit{cname})\ \{ & & i\_md_n\ (\ t_{n1}\ x_{n1} \ldots t_{nj}\ x_{nj}\ \textbf{string}\ \mathit{cname})\ \{ \\ \quad \textbf{if}\ (e''_1) & & \quad \textbf{if}\ (e''_1) \\ \quad\ \{\ \textbf{let}\ \{\ md = md(x_{11} \ldots x_{1j})\ \} & & \quad\ \{\ \textbf{let}\ \{\ md = md(x_{n1} \ldots x_{nj})\ \} \\ \quad\quad \textbf{in} & \ldots & \quad\quad \textbf{in} \\ \quad\quad\ \{\ \textbf{if}\ (e'''_1)\ \textbf{true else contractviolated}(c)\ ; & & \quad\quad\ \{\ \textbf{if}\ (e'''_n)\ \textbf{true else contractviolated}(c)\ ; \\ \quad\quad\quad md\ \}\} & & \quad\quad\quad md\ \}\} \\ \quad \textbf{contractviolated}(\mathit{cname}) & & \quad \textbf{contractviolated}(\mathit{cname}) \\ \} & & \} \end{array} \end{array}}\,[\mathbf{interface}]$$

$\rightharpoonup_{\mathsf{i}}$

$$\frac{}{P \vdash t\ md\ (t_1\ \mathit{var}_1) \ldots (t_n\ \mathit{var}_n)\ \textbf{@pre}\ \{\, e_b \,\}\ \textbf{@post}\ \{\, e_a \,\} \rightharpoonup_{\mathsf{i}} t\ md\ (t_1\ \mathit{var}_1) \ldots (t_n\ \mathit{var}_n)}\,[\mathbf{imeth}]$$

$\rightharpoonup_{\mathsf{m}}$

$$\frac{P, c \vdash e \rightharpoonup_{\mathsf{e}} e'}{P, c \vdash t\ md\ (\, t_1\ \mathit{arg}_1 \ldots t_n\ \mathit{arg}_n\,)\ \{\, e \,\} \rightharpoonup_{\mathsf{m}} t\ md\ (\, t_1\ \mathit{arg}_1 \ldots t_n\ \mathit{arg}_n\,)\ \{\, e' \,\}}\,[\mathbf{meth}]$$

$\rightharpoonup_{\mathsf{e}}$

$$\frac{}{P, c \vdash \textbf{new}\ c' \rightharpoonup_{\mathsf{e}} \textbf{new}\ c'}\,[\mathbf{new}^{\mathsf{c}}] \qquad \frac{}{P, c \vdash \mathsf{null} \rightharpoonup_{\mathsf{e}} \mathsf{null}}\,[\mathbf{null}] \qquad \frac{}{P, c \vdash \mathit{var} \rightharpoonup_{\mathsf{e}} \mathit{var}}\,[\mathbf{var}]$$

$$\frac{P, c \vdash e \rightharpoonup_{\mathsf{e}} e' \qquad P, c \vdash e_v \rightharpoonup_{\mathsf{e}} e'_v}{P, c \vdash e\underline{\ :c'\ }.fd = e_v \rightharpoonup_{\mathsf{e}} e'\underline{\ :c'\ }.fd = e'_v}\,[\mathbf{set}^{\mathsf{c}}] \qquad \frac{P, c \vdash e \rightharpoonup_{\mathsf{e}} e'}{P, c \vdash e\underline{\ :c'\ }.fd \rightharpoonup_{\mathsf{e}} e'\underline{\ :c'\ }.fd}\,[\mathbf{get}^{\mathsf{c}}]$$

$$\frac{P, c \vdash e \rightharpoonup_{\mathsf{e}} e' \qquad P, c \vdash e_j \rightharpoonup_{\mathsf{e}} e'_j \text{ for } j \in [1,n]}{P, c \vdash e\underline{: i}.md\,(e_1 \ldots e_n) \rightharpoonup_{\mathsf{e}} e.i\_md\ (e_1 \ldots e_n\ t)}\,[\mathbf{call}^{\mathsf{i}}]$$

$$\frac{P, c \vdash e \rightharpoonup_{\mathsf{e}} e' \qquad P, c \vdash e_j \rightharpoonup_{\mathsf{e}} e'_j \text{ for } j \in [1,n]}{P, c \vdash e\underline{: c}.md\,(e_1 \ldots e_n) \rightharpoonup_{\mathsf{e}} e.md\,(e_1 \ldots e_n\ t)}\,[\mathbf{call}^{\mathsf{c}}]$$

$$\frac{P, c \vdash e_j \rightharpoonup_{\mathsf{e}} e'_j \text{ for } j \in [1,n]}{P, c \vdash \textbf{super}\underline{\ \equiv \textbf{this} : c'\ }.md(e_1 \ldots e_n) \rightharpoonup_{\mathsf{e}} \textbf{super}\underline{\ \equiv \textbf{this} : c'\ }.md(e'_1 \ldots e'_n)}\,[\mathbf{super}^{\mathsf{c}}]$$

$$\frac{P, c \vdash e \rightharpoonup_{\mathsf{e}} e'}{P, c \vdash \textbf{view}\ t\ e \rightharpoonup_{\mathsf{e}} \textbf{view}\ t\ e'}\,[\mathbf{cast}^{\mathsf{c}}] \qquad \frac{P, c \vdash e_j \rightharpoonup_{\mathsf{e}} e'_j \text{ for } j \in [1,n] \qquad P, c \vdash e \rightharpoonup_{\mathsf{e}} e'}{P, c \vdash \textbf{let}\ \{\, \mathit{var}_1 = e_1 \ldots \mathit{var}_n = e_n \,\}\ \textbf{in}\ e \rightharpoonup_{\mathsf{e}} \textbf{let}\ \{\, \mathit{var}_1 = e'_1 \ldots \mathit{var}_n = e'_n \,\}\ \textbf{in}\ e'}\,[\mathbf{let}]$$

$$\frac{}{P, c \vdash \textbf{true} \rightharpoonup_{\mathsf{e}} \textbf{true}}\,[\mathbf{true}] \qquad \frac{}{P, c \vdash \textbf{false} \rightharpoonup_{\mathsf{e}} \textbf{false}}\,[\mathbf{false}]$$

$$\frac{P, c \vdash e_1 \rightharpoonup_{\mathsf{e}} e'_1 \qquad P, c \vdash e_2 \rightharpoonup_{\mathsf{e}} e'_2 \qquad P, c \vdash e_3 \rightharpoonup_{\mathsf{e}} e'_3}{P, c \vdash \textbf{if}\ (e_1)\ e_2\ \textbf{else}\ e_3 \rightharpoonup_{\mathsf{e}} \textbf{if}\ (e'_1)\ e'_2\ \textbf{else}\ e'_3}\,[\mathbf{if}] \qquad \frac{P, c \vdash e_1 \rightharpoonup_{\mathsf{e}} e'_1 \qquad P, c \vdash e_2 \rightharpoonup_{\mathsf{e}} e'_2}{P, c \vdash \{\, e_1 \,;\, e_2 \,\} \rightharpoonup_{\mathsf{e}} \{\, e'_1 \,;\, e'_2 \,\}}\,[\mathbf{seq}]$$

$$\frac{P, c \vdash e_i \rightharpoonup_{\mathsf{e}} e'_i \text{ for each } i \in [1,n]}{P, c \vdash \{\ \textbf{andall}\ e_1 \ldots e_n\ \} \rightharpoonup_{\mathsf{e}} \{\ \textbf{andall}\ e'_1 \ldots e'_n\ \}}\,[\mathbf{andall}]$$

Figure 10: Blame Compilation

$$
\begin{array}{rcl}
e & = & \ldots \mid \textbf{object} \\
v & = & \textbf{object} \mid \mathsf{null} \\
  &   & \textbf{true} \mid \textbf{false}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{E} & = & [\,] \mid \mathsf{E}\underline{\;:c\;}.\mathit{fd} \mid \mathsf{E}\underline{\;:c\;}.\mathit{fd} = e \mid v\underline{\;:c\;}.\mathit{fd} = \mathsf{E} \\
 & \mid & \mathsf{E}.\mathit{md}(e\ \ldots) \mid v.\mathit{md}(v\ \ldots\ \mathsf{E}\ e\ \ldots) \\
 & \mid & \textbf{super}\ \underline{\equiv v : c}\ .\mathit{md}(v\ \ldots\ \mathsf{E}\ e\ \ldots) \\
 & \mid & \textbf{view}\ t\ \mathsf{E} \mid \textbf{if}\ (\ \mathsf{E}\ )\ e\ \textbf{else}\ e \mid \{\ \mathsf{E}\ ;\ e\ \} \\
 & \mid & \textbf{let}\ \mathit{var} = v\ \ldots\ \mathit{var} = \mathsf{E}\ \ \mathit{var} = e\ \ldots\ \textbf{in}\ e \\
 & \mid & \textbf{andall}\ \mathsf{E}\ \ e\ \ldots
\end{array}
$$

$$
\begin{array}{ll}
P \vdash \langle \mathsf{E}[\textbf{new}\ c], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\textbf{object}], \mathcal{S}[\textbf{object} \mapsto \langle c, \mathcal{F}\rangle]\rangle & [\mathit{new}] \\
\quad \text{where } \textbf{object} \notin \mathrm{dom}(\mathcal{S}) \text{ and } \mathcal{F} = \{c'.\mathit{fd} \mapsto \mathsf{null} \mid c \leq^{\mathsf{c}}_P c' \text{ and } \exists t \text{ s.t. } \langle c'.\mathit{fd}, t\rangle \in^{\mathsf{c}}_P c'\} & \\
P \vdash \langle \mathsf{E}[\textbf{object}\underline{\;:c'\;}.\mathit{fd}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}\rangle & [\mathit{get}] \\
\quad \text{where } \mathcal{S}(\textbf{object}) = \langle c, \mathcal{F}\rangle \text{ and } \mathcal{F}(c'.\mathit{fd}) = v & \\
P \vdash \langle \mathsf{E}[\textbf{object}\underline{\;:c'\;}.\mathit{fd} = v], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}[\textbf{object} \mapsto \langle c, \mathcal{F}[c'.\mathit{fd} \mapsto v]\rangle]\rangle & [\mathit{set}] \\
\quad \text{where } \mathcal{S}(\textbf{object}) = \langle c, \mathcal{F}\rangle & \\
P \vdash \langle \mathsf{E}[\textbf{object}.\mathit{md}(v_1, \ldots v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[\textbf{object}/\textbf{this}, v_1/\mathit{var}_1, \ldots v_n/\mathit{var}_n]], \mathcal{S}\rangle & [\mathit{call}] \\
\quad \text{where } \mathcal{S}(\textbf{object}) = \langle c, \mathcal{F}\rangle \text{ and } \langle \mathit{md}, (t_1 \ldots t_n \longrightarrow t), (\mathit{var}_1 \ldots \mathit{var}_n), e\rangle \in^{\mathsf{c}}_P c & \\
P \vdash \langle \mathsf{E}[\textbf{super}\ \underline{\equiv \textbf{object} : c'}\ .\mathit{md}(v_1, \ldots v_n)], \mathcal{S}\rangle & [\mathit{super}] \\
\qquad \hookrightarrow \langle \mathsf{E}[e[\textbf{object}/\textbf{this}, v_1/\mathit{var}_1, \ldots v_n/\mathit{var}_n]], \mathcal{S}\rangle & \\
\quad \text{where } \langle \mathit{md}, (t_1 \ldots t_n \longrightarrow t), (\mathit{var}_1 \ldots \mathit{var}_n), e\rangle \in^{\mathsf{c}}_P c' & \\
P \vdash \langle \mathsf{E}[\textbf{view}\ t'\ \textbf{object}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\textbf{object}], \mathcal{S}\rangle & [\mathit{cast}] \\
\quad \text{where } \mathcal{S}(\textbf{object}) = \langle c, \mathcal{F}\rangle \text{ and } c \leq_P t' & \\
P \vdash \langle \mathsf{E}[\textbf{let}\ \mathit{var}_1 = v_1\ \ldots\ \mathit{var}_n = v_n\ \textbf{in}\ e], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[v_1/\mathit{var}_1 \ldots v_n/\mathit{var}_n]], \mathcal{S}\rangle & [\mathit{let}] \\
P \vdash \langle \mathsf{E}[\textbf{if}\ (\ \textbf{true}\ )\ e_1\ \textbf{else}\ e_2], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e_1], \mathcal{S}\rangle & [\mathit{iftrue}] \\
P \vdash \langle \mathsf{E}[\textbf{if}\ (\ \textbf{false}\ )\ e_1\ \textbf{else}\ e_2], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e_2], \mathcal{S}\rangle & [\mathit{iffalse}] \\
P \vdash \langle \mathsf{E}[\{\ v\ ;\ e\ \}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e], \mathcal{S}\rangle & [\mathit{seq}] \\
P \vdash \langle \mathsf{E}[\{\ \textbf{andall}\ e\ \}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e], \mathcal{S}\rangle & [\mathit{andalldone}] \\
P \vdash \langle \mathsf{E}[\{\ \textbf{andall true}\ e\ \ldots\ \}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\textbf{andall}\ e\ \ldots], \mathcal{S}\rangle & [\mathit{andalltrue}] \\
P \vdash \langle \mathsf{E}[\{\ \textbf{andall false}\ e\ \ldots\ \}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\textbf{false}], \mathcal{S}\rangle & [\mathit{andallfalse}] \\
 & \\
P \vdash \langle \mathsf{E}[\textbf{contractviolated}(\mathrm{c})], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error: } c \text{ violated contract}, \mathcal{S}\rangle & [\mathit{contract}] \\
P \vdash \langle \mathsf{E}[\textbf{view}\ t'\ \textbf{object}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error: bad\ cast}, \mathcal{S}\rangle & [\mathit{xcast}] \\
\quad \text{where } \mathcal{S}(\textbf{object}) = \langle c, \mathcal{F}\rangle \text{ and } c \not\leq_P t' & \\
P \vdash \langle \mathsf{E}[\textbf{view}\ t'\ \mathsf{null}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error: bad\ cast}, \mathcal{S}\rangle & [\mathit{ncast}] \\
P \vdash \langle \mathsf{E}[\mathsf{null}\underline{\;:c\;}.\mathit{fd}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error: dereferenced\ null}, \mathcal{S}\rangle & [\mathit{nget}] \\
P \vdash \langle \mathsf{E}[\mathsf{null}\underline{\;:c\;}.\mathit{fd} = v], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error: dereferenced\ null}, \mathcal{S}\rangle & [\mathit{nset}] \\
P \vdash \langle \mathsf{E}[\mathsf{null}.\mathit{md}(v_1, \ldots v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{error: dereferenced\ null}, \mathcal{S}\rangle & [\mathit{ncall}]
\end{array}
$$

Figure 11: Operational semantics for Contract Java