Contract Soundness for Object-Oriented Languages

Robert Bruce Findler
Rice University
Computer Science Department
6100 S. Main; MS 132
Houston, TX 77006
USA

Matthias Felleisen
Northeastern University
Rice University

ABSTRACT

Checking pre- and post-conditions of procedures and methods at runtime helps improve software reliability. In the procedural world, pre- and post-conditions have a straightforward interpretation. If a procedure's pre-condition doesn't hold, the caller failed to establish the proper context. If a post-condition doesn't hold, the procedure failed to compute the expected result.

In the object-oriented world, checking pre- and post-conditions for methods, often called contracts in this context, poses complex problems. Because methods may be overridden, it is not sufficient to check only pre- and post-conditions. In addition, the contract hierarchy must be checked to ensure that the contracts on overridden methods are properly related to the contracts on overriding methods. Otherwise, a class hierarchy may violate the substitution principle, that is, it may no longer be true that an instance of a class is substitutable for objects of the super-class.

In this paper, we study the problem of contract enforcement in an object-oriented world from a foundational perspective. More specifically, we study contracts as refinements of types. Pushing the analogy further, we state and prove a contract soundness theorem that captures the essential properties of contract enforcement. We use the theorem to illustrate how most existing tools suffer from a fundamental flaw and how they can be improved.

1. INTRODUCTION

Checking pre- and post-conditions of procedures is an important technique for improving the reliability of software. It is available in many languages, including Ada [16], C [22], and Java [3, 4, 9, 10, 12, 17]. Several languages, including Eiffel [20], Sather [7], and Blue [11] have supported runtime-checked pre- and post-condition contracts since their inception. Eiffel's community has even developed the "design by contract" programming discipline, which is based on this idea [18, 19].

In C-like languages, enforcing pre- and post-condition contracts is a simple and effective matter. Functions are annotated with pre- and post-conditions. If a pre-condition is violated, the function's caller is to blame and if a post-condition is violated the function itself is to blame. Rosenblum's study [22] demonstrates the effectiveness of contracts in this world.

ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA 2001

In object-oriented languages, where contracts annotate method signatures, monitoring contracts is more complex than in procedural languages. The additional complexity is due to the subtyping relationships that are at the heart of object-oriented programming. Because instances of a type may be substituted in contexts expecting a supertype, the behavior of objects of a type must be related to the behavior of objects of the supertype. Further, because contracts represent key aspects of an object's behavior, the contracts on a given type must be related to the contracts of a supertype. Although a theory of object substitutability exists [15], a complementary theory of contract monitoring is missing and, as a result, existing contract monitoring tools are flawed.

Our starting point for a theory of run-time contracts is the theory of type systems. A major role for contracts is to specify properties similar to types, but that cannot be checked by the type checker [22]. For example, a Java method may accept an integer, intending it to be an array index that must be within a certain range. Because Java's type system cannot express the range constraint, a Java programmer must resort to contracts to state this fact. Additionally, type systems have benefited from a well-developed theory. In particular, good type systems satisfy a type soundness theorem, which ensures that the type checker respects the language's semantics.

Based on these observations, we conclude that contract checkers could benefit from a similarly well-developed theory. This paper develops a first formal foundation for a theory of contracts. We discuss how contracts must respect a program's behavior, and give a semantics that defines how contract hierarchies should be interpreted. Additionally, we state and prove a *contract soundness theorem*. The theorem shows that our contract checker discovers invalid contract hierarchies at method calls and returns.

The paper consists of seven sections. The next section motivates the contract checking theorem and explains it intuitively. Section 3 presents the syntax, type checking, semantics, and contract elaborator for Contract Java, a small model of (sequential) Java [8], extended with pre- and post-condition contracts. Section 4 states the contract soundness theorem, and proves that it holds for the contract elaborator of section 3. Section 5 discusses existing contract checking tools [2, 3, 4, 7, 9, 11, 12, 13, 17, 20] and explains how none of them (with the exception of Jass [2, 3]) satisfy the contract soundness theorem. Section 6 briefly discusses an implementation. The last section summarizes our results.

2. BEHAVIORAL SUBTYPING

In programs without subtyping, checking pre- and post-conditions is a simple matter. Consider this code, which implements a wrapper class for floats:

```
class Float {
  Float getValue() { ... }
  Float sqrt() { ... }
    @pre { getValue() > 0 }
    @post { Math.abs(sqrt * sqrt – this.getValue()) <= 0.1 }
}
```

In this case, the pre-condition for *sqrt* ensures that the method is only applied to positive numbers. The post-condition promises that the square of the result is within a certain tolerance of the original input. Following tradition, we use the name of the method to stand for its result in the body of the post-condition.

In the case of the *sqrt* method, the pre- and post-conditions fully specify its correctness. In practice, however, programmers do not use pre- and post-conditions to specify the entire behavior of the method; instead programmers use contracts to refine method type specifications. Consider this program:

```
interface IConsole {
  int getMaxSize();
    @post { getMaxSize > 0 }
  void display(String s);
    @pre { s.length() < this.getMaxSize() }
}

class Console implements IConsole {
  int getMaxSize() { ... }
    @post { getMaxSize > 0 }
  void display(String s) { ... }
    @pre { s.length() < this.getMaxSize() }
}
```

The *IConsole* interface contains the methods, types, and pre- and post-conditions for a small window that can display a message, and the *Console* class provides an implementation of *IConsole*. The *getMaxSize* method returns the limit on the message's size, and the *display* method changes the console's visible message. The post-condition for *getMaxSize* and the pre-condition for *display* merely guarantee simple invariants of the console; they do not ensure correctness.

As long as programs do not use inheritance, contract checking merely involves evaluating the conditions that the programmer stated. Once programs employ inheritance contract monitoring requires more sophistication than that. According to the notion of behavioral subtyping [1, 14, 15], an instance of a subtype must be substitutable for an instance of a supertype. For pre- and post-condition contracts, behavioral subtyping mandates that the pre-condition of a method in a type implies the pre-condition of the same method in each of its subtypes. Similarly, it requires that each post-condition in a subtype implies the corresponding post-condition in the original type. A contract checker for object-oriented languages must verify that the pre-condition and post-condition hierarchies meet this behavioral subtyping requirement.

As an example, consider this extension of *Console*:

```
class RunningConsole extends Console {
  void display(String s) {
    ... super.display
          (String.substring
            (s, ..., ... + getMaxSize())) ...
  }
    @pre { true }
}
```

The *display* method creates a thread that displays whatever portion of the string fits in the console and then updates the console's display, scrolling the message character by character. Since the pre-condition of *display* in *RunningConsole* is **true**, it is implied by the pre-condition in *Console*, and thus *RunningConsole* is a behavioral subtype of *Console*.

Not every pre-condition on a method turns a subclass into a behavioral subtype. Concretely, extensions of the *Console* class may have pre-conditions that are not implied by the supertype's pre-condition. Consider this example:

```
class PrefixedConsole extends Console {
  String getPrefix() {
    return ">> ";
  }
  void display(String s) {
    super.display(this.getPrefix() + s);
  }
    @pre { s.length() <
              this.getMaxSize() – this.getPrefix().length() }
}
```

In this case, the pre-condition on *PrefixedConsole* is not implied by the pre-condition on *Console*. Accordingly, code that is written to accept instances of *Console* may violate the pre-condition of *PrefixedConsole* without violating the pre-condition of *Console*. Clearly, the code that expects instances of *Console* should not be blamed, since that code fulfilled its obligations by meeting *Console*'s pre-condition. Instead, the blame must lie with the programmer of *PrefixedConsole* for failing to create a behavioral subtype of *Console*.

In addition to classes, interfaces describe another type hierarchy. Like the class type hierarchy, the interface type hierarchy must also specify a hierarchy of behavioral subtypes. Thus, the blame for a malformed hierarchy can fall on the author of code that contains only interfaces. Consider the following two-part Java program:

```
// Written by Guy
interface I {
  void m(int a);
    @pre { a > 0 }
}


interface J extends I {
  void m(int a);
    @pre { a > 10 }
}
```

```
// Written by James
class C implements J {
  void m(int a) { ... }
    @pre { a > 10 }

  public static void
  main(String argv[]) {
    I i = new C();
    i.m(5);
  }
}
```

Imagine that two different programmers, Guy and James, wrote the two different parts of the program. First, James's *main* method creates an instance of *C*, but with type *I*. Then, it invokes *m* with *5*. According to the contracts for *I*, this is perfectly valid input. According to the contract on *J*, however, this is an illegal input. The behavioral subtyping condition tells us that *J* can only be a subtype of *I* if it is substitutable for *I* in every context. This is not true, however, as *J*'s *m* accepts fewer arguments than *I*'s *m*. In particular, *J*'s *m* does not accept *1*, *2*, ..., *10* but *I*'s *m* does. Guy's claim that *J* **extends** *I* is wrong. When the method call from James's code fails, the blame for the contractual violation must lie with Guy.

The preceding examples suggest that contract checking systems for object-oriented languages should signal *three* kinds of errors: pre-condition violations, post-condition violations, and hierarchy errors. The latter distinguishes contract checking in the procedural world from contract checking in the object-oriented world. A hierarchy error signals that some subclass or extending interface is not a behavioral subtype, either because the hierarchy of pre-conditions or the hierarchy of post-conditions is malformed.

The goal of our paper is to develop a theoretical framework in which we can state and prove claims about a contract monitoring system for Java. Following the reasoning of the introduction, we develop our framework in analogy to Milner's work on type systems and type soundness theorems [21]. A type soundness theorem has two parts. First, it specifies what kind of errors (or run-time exceptions) the evaluation of a well-typed program can trigger. Second, it implies that certain properties hold for the evaluation of subexpressions. For example, an addition operation in an ML program will always receive two numbers, and thus ML programs never terminate with errors due to the misuse of the addition operation. Similarly, an array indexing operation will always receive an integer as an index, but the integer may be out of the array's range. Hence, an ML program may terminate due to a misuse of an array primitive.

Here we show that a contract checking system can satisfy a contract soundness theorem. Like a type soundness theorem, the contract soundness theorem spells out two properties. First, it states what kind of errors the evaluation of a monitored program may signal. Second, it implies that the specified hierarchy of interfaces and classes satisfies implications between the stated pre- and post-conditions of overridden methods.

To formalize these intuitions, we develop a model of Java with contracts in the form of a calculus. The calculus specifies the syntax, type system, and semantics of a small Java-like language with mechanisms for simple contract specifications. Based on this calculus we specify contract checking as a translation from the full language into a small kernel. Using the calculus, we state and prove a contract soundness theorem.

3. CONTRACT JAVA

Contract Java extends the Classic Java calculus [6] with pre- and post-condition contracts on methods. This section presents the Contract Java calculus, it syntax and semantics. Section 3.1 presents the syntax and type checker. Section 3.2 is the focal point of this section; it presents the contract elaborator. Finally, section 3.3 presents the operational semantics.

3.1 Syntax and Type Checking

Figure 1 contains the syntax for Contract Java. The syntax is divided into three parts. Programmers use syntax (a) to write their programs. The type checker elaborates syntax (a) to syntax (b), which contains type annotations for use by the evaluator and contract compiler. The contract compiler elaborates syntax (b) to syntax (c). It elaborates the pre- and post-conditions into monitoring code; the result is accepted by the evaluator.

A program *P* is a sequence of class and interface definitions followed by an expression that represents the body of the *main* method. Each class definition consists of a sequence of field declarations followed by a sequence of method declarations and their contracts. An interface consists of method specifications and their contracts. The contracts are arbitrary Java expressions that have type **boolean**.[1] 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 in Java, the body of a method is just an expression whose result is the result of the method. Like in Java, classes are instantiated with the **new** operator, but there are no class constructors[2] in Contract Java; instance variables are initialized to **null**. Finally, the **view** and **let** forms represent Java's casting expressions and the capability for binding variables locally. In the code examples presented in this paper, we omit the **extends** and **implements** clauses when nothing would appear after them.

The type checker translates syntax (a) to syntax (b). It inserts additional information (underlined in the figure) to used by the contract elaborator and the evaluator. To support contract elaboration, method calls are annotated with the type of the object whose method is called. 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.

The contract elaborator produces syntax (c) and the evaluator accepts it. The **@pre** and **@post** conditions are removed from interfaces and classes, and inserted elsewhere in the elaborated program. Syntax (c) also adds three constructs to our language: **preErr**, **postErr**, and **hierErr**. These constructs are used to signal contract violations.

[1] We could have carried out our study in a more complex contract specification language, but plain Java expressions suffice to express many important contracts. Additionally, using a single language for pre-conditions, post-conditions, and expressions simplifies the presentation and proofs.

[2] Pre- and post-condition contracts for constructors can be treated as contracts on methods that are never overridden.

```
      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 }
              @pre { e } @post { e }
  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
```

(a) 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 }
              @pre { e } @post { e }
  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 }

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
```

(b) Typed Contract 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* )

    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 }
            | return : t, c { e }
            | preErr(e) | postErr(e)
            | hierErr(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
```

(c) Core Syntax

Figure 1: Contract Java syntax; before and after contracts are compiled away

Expressions of the shape:

$$\textbf{return} : t,\ c\ \{\ e\ \}$$

mark method returns. The type t indicates the type of the object whose method was invoked, in parallel to the type annotations on method calls, and the class name, c, is the class that defined the invoked method. Unlike standard Java, in Contract Java the programmer does not write **return** expressions in the program. Instead, the evaluator introduces **return** expressions as it executes the program. These annotations are used in the statement and the proof of the contract soundness theorem.

There are three important relations on the abstract syntax: $\leq_P$, PRE_P, and POST_P. The first, $\leq_P$, defines the subtyping relationship. A type t is a subtype of another type t' in a program P, written $t \leq_P t'$, when one of these conditions holds:

- t and t' are the same type,
- t and t' are both classes, t is derived from t'' in P, and $t'' \leq_P t'$,
- t and t' are both interfaces, t is an extension of t'' in P (also written $t \prec^{\mathsf{i}}_P t''$), and $t'' \leq_P t'$, or
- t is a class and t' is an interface, and either
  - t implements t' in P,
  - t implements an interface i in P and $i \leq_P t'$, or
  - t is derived from a class c in P and $c \leq_P t'$.

The relations PRE_P and POST_P relate expressions with pairs of methods and types. An expression e is the pre-condition for m in t in the program P, $e\ \mathsf{PRE}_P\ \langle t, m\rangle$, if the expression e appears in the program P, declared as a precondition of m in t. Similarly an expression e is a postcondition of m in t in the program P if $e\ \mathsf{POST}_P\ \langle t, m\rangle$.

3.2 Contract Elaboration

Contract checking is modeled as a translation, called T, from syntax (b) to syntax (c). Since contract checking is triggered via method calls, we need to understand how T deals with those. Consider the following code fragment:

IConsole o = *ConsoleFactory*(...);

... *o.display*("It's crunch time.") ...

Since the programmer cannot know what kind of console o represents at run-time, he can ensure only that the preconditions for *display* that *IConsole* specifies. Hence, the code that T produces for the method call must first test the preconditions for *display* in *IConsole*. If this test fails, the author of the method call has made a mistake. If the test succeeds, the contract monitoring code can check the ancestor portion of the class and interface hierarchy that is determined by o's class tag.[3] These hierarchy checks ensure that the precondition of an overriding method implies the precondition of the overridden method, and that the postcondition of an overridden method implies the postcondition of each overriding method.

To perform both forms of checking, T adds new classes to check the subtype hierarchy and inserts methods into existing classes to check pre- and post-conditions. For each method of a class, the elaborator inserts several wrapper methods, one for each type that instances of the class might have. These wrapper methods perform the pre- and post-condition checking and call the hierarchy checkers. Additionally, the elaborator redirects each method call so it invokes the appropriate wrapper method, based on the static type of the object whose method is invoked. So, in the above invocation, the elaborator inserts a *display_IConsole* wrapper method into the each console class since each console class can be cast to *IConsole*. Additionally, it rewrites the call to the *display* method to call the *display_IConsole* method, since o's type is *IConsole*. Each *display_IConsole* method checks *IConsole*'s pre-condition and the pre-condition hierarchy from the instantiated class upwards. Then the *display_Console* method calls the original *display* method. When it returns, the *display_IConsole* method checks *IConsole*'s post-condition and the post-condition hierarchy from the instantiated class upwards. The rest of this subsection presents the elaborator both concretely via the example of the console classes and interfaces, and abstractly via judgements that define the elaborator.

Formally, our contract elaborator is defined by these judgements:

$\vdash P \rightharpoonup_{\mathsf{p}} P'$
The program P compiles to the program P'.
$P \vdash \mathit{defn} \rightharpoonup_{\mathsf{d}} \mathit{defn}'\ \mathit{defn}_{pre}\ \mathit{defn}_{post}$
defn compiles to *defn'* with checkers defn_{pre} and defn_{post} in P.

$P \vdash \mathit{imeth} \rightharpoonup_{\mathsf{i}} \mathit{imeth}'$
imeth compiles to *imeth'*.

$P, c \vdash \mathit{meth} \rightharpoonup_{\mathsf{m}} \mathit{meth}'$
meth compiles to *meth'* in class c.
$P, c, t \vdash \mathit{meth} \rightharpoonup_{\mathsf{w}} \mathit{meth}'$
meth' checks the pre- and post-conditions for t's *meth*, which blames c for contract violations.
$P, c \vdash e \rightharpoonup_{\mathsf{e}} e'$
e compiles to e', which blames c for contract violations.

$P, t \vdash \mathit{imeth} \rightharpoonup_{\mathsf{pre}} \mathit{imeth}'$
imeth' checks the hierarchy for the pre-condition of *imeth* in t.
$P, t \vdash \mathit{imeth} \rightharpoonup_{\mathsf{post}} \mathit{imeth}'$
imeth' checks the hierarchy for the post-condition of *imeth* in t.

The first judgement, $\rightharpoonup_{\mathsf{p}}$, is the program elaboration judgement. The $\rightharpoonup_{\mathsf{d}}$ judgement builds three definitions for each definition in the original program. The first is derived from the original definition. The second and third are the pre- and post-condition hierarchy checking classes, respectively.

The $\rightharpoonup_{\mathsf{i}}$ judgements erases interface method contracts. The $\rightharpoonup_{\mathsf{m}}$, $\rightharpoonup_{\mathsf{e}}$, and $\rightharpoonup_{\mathsf{w}}$ judgements produce the annotated class. The $\rightharpoonup_{\mathsf{w}}$

[3] Following ML tradition, we use the word "type" to refer only to the static type determined by the type checker. We use the words "class tag" to refer to the so-called dynamic or run-time type.

defn$^{\mathsf{i}}$

$$\frac{\vdash imeth_j \rightharpoonup_{\mathsf{i}} imeth'_j \qquad P, c \vdash imeth_j \rightharpoonup_{\mathsf{pre}} meth_{j_{pre}} \qquad P, c \vdash imeth_j \rightharpoonup_{\mathsf{post}} meth_{j_{post}} \qquad \text{for } j \in [1, n]}{\begin{array}{l} P \vdash \textbf{interface}\ i\ \textbf{extends}\ i_1 \ldots i_l\ imeth_1 \ldots imeth_n \rightharpoonup_{\mathsf{d}} \textbf{interface}\ i\ \textbf{extends}\ i_1 \ldots i_l\ imeth'_1 \ldots imeth'_n \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{class}\ check\_i\_pre\ imeth_{1_{pre}} \ldots imeth_{n_{pre}} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{class}\ check\_i\_post\ imeth_{1_{post}} \ldots imeth_{n_{post}} \end{array}}$$

defn$^{\mathsf{c}}$

$$\frac{\begin{array}{l} P, c \vdash meth_j \rightharpoonup_{\mathsf{m}} meth'_j \qquad P, c \vdash meth_j \rightharpoonup_{\mathsf{pre}} meth_{j_{pre}} \qquad P, c \vdash meth_j \rightharpoonup_{\mathsf{post}} meth_{j_{post}} \qquad \text{for } j \in [1, n] \\ P, c, t \vdash meth_j \rightharpoonup_{\mathsf{w}} wrap\_method_j \qquad \text{for } j \in [1, n], \text{ and } t \text{ such that } c \leq_P t \end{array}}{\begin{array}{ll} P \vdash \textbf{class}\ c\ \textbf{extends}\ c'\ \textbf{implements}\ i_1 \ldots i_l & \rightharpoonup_{\mathsf{d}} \textbf{class}\ c\ \textbf{extends}\ c'\ \textbf{implements}\ i_1 \ldots i_l \\ \qquad meth_1 \ldots meth_n & \qquad meth'_1 \ldots meth'_n \\ & \qquad wrap\_method_1 \ldots wrap\_method_n \ldots \\ & \textbf{class}\ check\_c\_pre\ \textbf{extends Object}\ meth_{1_{pre}} \ldots meth_{n_{pre}} \\ & \textbf{class}\ check\_c\_post\ \textbf{extends Object}\ meth_{1_{post}} \ldots meth_{n_{post}} \end{array}}$$

wrap

$$\frac{e_b\ \textsf{PRE}_P\ \langle t, md \rangle \qquad e_a\ \textsf{POST}_P\ \langle t, md \rangle \qquad P, c \vdash e_b \rightharpoonup_{\mathsf{e}} e'_b \qquad P, c \vdash e_a \rightharpoonup_{\mathsf{e}} e'_a}{\begin{array}{l} P, c, t \vdash t'\ md\ (t_1\ x_1, \ldots, t_j\ x_j) \ldots \rightharpoonup_{\mathsf{w}} \\ \quad t'\ \_md\ (t_1\ x_1,\ \ldots,\ t_j\ x_j,\ string\ cname)\ \{ \\ \quad\ \textbf{if}\ (e'_b)\ \{ \\ \qquad\quad (\textbf{new}\ check\_c\_pre()).md(this,\ x_1,\ \ldots,\ x_j); \\ \qquad\quad \textbf{let}\ \{\ md = this.md(x_1,\ \ldots,\ x_j)\ \} \\ \qquad\quad \textbf{in}\ \{\ \textbf{if}\ (e'_a) \\ \qquad\qquad\qquad (\textbf{new}\ check\_c\_post()).md(\text{“dummy”},\ \textbf{true},\ this,\ md,\ x_1,\ \ldots,\ x_j); \\ \qquad\qquad\quad \textbf{else} \\ \qquad\qquad\qquad \textbf{postErr}(c); \\ \qquad\qquad\quad md\ \} \\ \quad\ \}\ \textbf{else}\ \{ \\ \qquad \textbf{preErr}(cname); \\ \quad\ \} \\ \quad \} \end{array}}$$

pre$^{\mathsf{i}}$

$$\frac{\text{for all } i' \text{ such that } i \prec^{\mathsf{i}}_P i'}{\begin{array}{l} P, i \vdash t\ md\ (t_1\ var_1 \ldots t_n\ var_n)\ \{\ e\ \}\ \textbf{@pre}\ \{\ e_b\ \}\ \textbf{@post}\ \{\ e_a\ \} \rightharpoonup_{\mathsf{pre}} \textbf{boolean}\ md\ (i\ this,\ t_1\ var_1,\ \ldots\ t_n\ var_n)\ \{ \\ \qquad\qquad \textbf{let}\ \{\ next = (\textbf{new}\ check\_i'\_pre()).md(this,\ var_1,\ \ldots,\ var_n)\ \|\ \ldots \\ \qquad\qquad\qquad res = e_b\ \} \\ \qquad\qquad \textbf{in if}\ (!next\ \|\ res)\ //\ next \Rightarrow res \\ \qquad\qquad\qquad res \\ \qquad\qquad\quad \textbf{else} \\ \qquad\qquad\qquad \textbf{hierErr}(i) \\ \qquad\quad \} \end{array}}$$

post$^{\mathsf{i}}$

$$\frac{\text{for all } i' \text{ such that } i \prec^{\mathsf{i}}_P i'}{\begin{array}{l} P, i \vdash t\ md\ (t_1\ var_1 \ldots t_n\ var_n)\ \{\ e\ \}\ \textbf{@pre}\ \{\ e_b\ \}\ \textbf{@post}\ \{\ e_a\ \} \rightharpoonup_{\mathsf{post}} \textbf{boolean}\ md\ (\textbf{String}\ tbb,\ \textbf{boolean}\ last,\ i\ this,\ t\ md,\ t_1\ var_1,\ \ldots,\ t_n\ var_n)\ \{ \\ \qquad\qquad \textbf{let}\ \{\ res = e_a\ \} \\ \qquad\qquad \textbf{in if}\ (!last\ \|\ res)\ //\ last \Rightarrow res \\ \qquad\qquad\qquad (\textbf{new}\ check\_i'\_post()).md(c,\ res,\ this,\ md,\ var_1,\ \ldots,\ var_n)\ \&\&\ \ldots \\ \qquad\qquad\quad \textbf{else} \\ \qquad\qquad\qquad \textbf{hierErr}(tbb) \\ \qquad\quad \} \end{array}}$$

call$^{\mathsf{i}}$

$$\frac{P, c \vdash e \rightharpoonup_{\mathsf{e}} e' \qquad P, c \vdash e_j\underline{:\ i} \rightharpoonup_{\mathsf{e}} e'_j \ \text{for } j \in [1, n]}{P, c \vdash e\underline{:\ i}.md\ (e_1, \ldots, e_n) \rightharpoonup_{\mathsf{e}} e'.i\_md(e'_1, \ldots, e'_n)}$$

Figure 2: Blame Compilation

constructs the wrapper methods that check the contracts. The $\rightharpoonup_{\mathsf{m}}$ judgement re-writes methods and erases class method contracts. The $\rightharpoonup_{\mathsf{e}}$ judgement rewrites expressions so that method calls are re-directed to the wrapper methods, based on the type of the call. The final two judgements, $\rightharpoonup_{\mathsf{pre}}$ and $\rightharpoonup_{\mathsf{post}}$, produce the methods for the pre- and post-condition hierarchy checkers.

The important clauses for those judgements are given in figure 2. The remainder of this section illustrates how the judgements work, using the console example from section 2.

As the **[defn$^{\mathsf{c}}$]** rule and the **[defn$^{\mathsf{i}}$]** rule show, each definition in the original program generates a definition and two additional classes. The first definition corresponds to the original definition, with the contracts erased and, in the case of classes, wrapper methods inserted. These wrapper methods check for pre-condition and post-condition violations, and invoke the hierarchy checkers. The elaborator inserts wrapper methods based on the types that instances of the class might have.

Consider the *Console* class of section 2. The elaboration adds two wrapper methods for *getMaxSize*, because instances of *Console* can have two types: *IConsole* and *Console*. The elaborator also adds two wrapper methods for *display*:

```
class Console implements IConsole {
  int getMaxSize() { ... } @post { ... }
  int getMaxSize_IConsole ...
  int getMaxSize_Console ...

  void display(String s) { ... } @pre { ... }
  void display_IConsole ...
  void display_Console ...
}
```

Similarly, for *RunningConsole* and *PrefixedConsole*, T adds three methods, since instances of each of those classes may take on three types. Here is *RunningConsole*:

```
class RunningConsole extends Console {
  int getMaxSize() { ... } @post { ... }
  int getMaxSize_IConsole ...
  int getMaxSize_Console ...
  int getMaxSize_RunningConsole ...

  void display(String s) { ... } @pre { ... }
  void display_IConsole ...
  void display_Console ...
  void display_RunningConsole ...
}
```

The **[wrap]** rule specifies the shape of the wrapper methods. It uses the program P, the class c where the wrapper method appears, the type t at which the method is being called, and the method header t' md $(t_1\ x_1,\ \ldots,\ t_j\ x_j)$. The wrapper method accepts the same arguments that the original method did, plus one extra argument naming the class whose program text contains the method call. The wrapper method first checks the pre-condition e'_b. If it fails, it blames the calling context for not establishing the required pre-condition. If the pre-condition succeeds, the wrapper calls the pre-condition hierarchy checker for c. The pre-condition hierarchy checker traverses the class and interface hierarchy, making sure that each subtype is a behavioral subtype, for the pre-conditions. If the hierarchy checking succeeds, the wrapper method calls the original method. After the method returns, it saves the result in the variable *md*, checks the post-condition, e'_a and calls the post-condition hierarchy checker. Like the pre-condition hierarchy checkers, the post-condition hierarchy checker ensures that each subtype is a behavioral subtype, for the post-conditions. Finally, if the post-condition checking succeeds, the wrapper method delivers the result of the wrapped method.

Additionally, the **[wrap]** rewrites the contract expressions themselves so that pre- and post-condition of methods invoked by the contracts are also checked.

Here is *Console*'s *display_Console* wrapper method.[4]

```
void display_Console(String s, string cname) {
  if ( s.length() < this.getMaxSize() ) {
    (new check_Console_pre()).display(this, s);
    let { display = this.display(s) }
    in {
      (new check_Console_post())
        .display("dummy", true, this, display, s);
    }
  } else {
    preErr(cname);
  }
}
```

The original console class's *display* method has no post-condition, so the **if**-expression from the **[wrap]** rule is eliminated. The variable *display* is bound to the result of the method, for the post-condition. The first two arguments to *check_Console_post* are initial values for accumulators and are explained below.

The second and third classes definitions introduced by the **[defn$^{\mathsf{i}}$]** and **[defn$^{\mathsf{c}}$]** rules are the hierarchy checkers. Each hierarchy checker is responsible for checking a portion of the hierarchy and combining its result with the rest of the hierarchy checkers. Unlike pre- and post-condition checking, hierarchy checking begins at the class tag for the object; it is not based on the static type of the object. As an example, consider the hierarchy diagram in figure 3 and this code fragment:

```
I o = new C();
o.m();
```

When m is invoked, the hierarchy checkers must ensure that the hierarchy is well-formed. Since instances of C can never be cast to D or K, only the boxed portion of the hierarchy in figure 3 is checked. Thus, when o's m is invoked, the hierarchy checking classes ensure that I's pre-condition implies C's pre-condition and that J's pre-condition also implies C's pre-condition. Similarly, when m returns, only I, J, and C's post-conditions are checked to ensure the post-condition hierarchy is well-formed, too.

[4] We omit the annotations inserted by the type-checker to clarify the presentation.

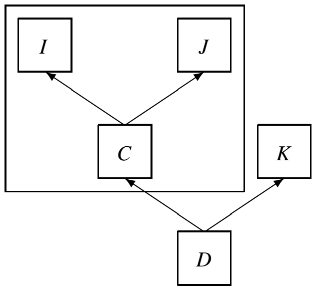


Figure 3: Example Hierarchy Diagram

In our running console example, the following classes are generated:

- *check_IConsole_pre*,
- *check_Console_pre*,
- *check_RunningConsole_pre*,
- *check_PrefixedConsole_pre*,
- *check_IConsole_post*,
- *check_Console_post*,
- *check_RunningConsole_post*, and
- *check_PrefixedConsole_post*.

Each of the hierarchy-checking classes has a method for each method in the original class. The methods in the hierarchy-checking classes have the same names as the methods in the original class, although their purpose is different. The hierarchy checking methods check the pre- or post-condition for that method. Then they combine that result with the results of the rest of the hierarchy checking to determine if there are any hierarchy violations. In our example, each hierarchy checking class contains a *getMaxSize* method and a *display* method.

The **[pre^i]** rule produces the pre-condition hierarchy checker for the *md* method of the interface *i*. The resulting method accepts the same arguments that *md* accepts, plus a binding for *this*. The *this* argument is passed along so the contract checking code can test the state of the object. The hierarchy checking method returns the result of the pre-condition for *md*. First, it recursively calls the hierarchy checkers for each of the immediate super-interfaces of *i* and combines their results in a disjunction. Second, it evaluates the pre-condition for *md*. Finally, the checker ensures that the hierarchy is well-formed by checking that the pre-conditions for the super-methods imply the current pre-condition. If the implication holds, the checker returns *res*, the value of this pre-condition. If the implication does not hold, the hierarchy checker method signals a hierarchy error and blames *i*, the extending interface. The rule for classes is analogous.

The pre-condition checkers for *RunningConsole* and *Console display* methods are:

```
class check_RunningConsole_pre extends Object {
  boolean display (RunningConsole this, String s) {
    let { next = (new check_Console_pre()).display(this, s)
          res = true }
    in if (!next || res)              // next ⇒ res
         res
       else
         hierErr("RunningConsole")
  }
}
```

and

```
class check_Console_pre extends Object {
  boolean display (Console this, String s) {
    let { next = (new check_IConsole_pre()).display(this, s)
          res = s.length() < this.getMaxSize() }
    in if (!next || res)              // next ⇒ res
         res
       else
         hierErr("Console")
  }
}
```

The **[post^i]** rule specifies the post-condition hierarchy checking method. The post-condition hierarchy checker is similar to the pre-condition checker. Rather than returning the truth value of each condition, however, the post-condition checker accumulates the results of the conditions in the *last* argument. Using an accumulator in this fashion means the post-condition checker uses the same recursive traversal of the type hierarchy as the pre-condition checker, but checks the implications in the reverse direction. The *tbb* argument is also an accumulator. It represents the subclass to be blamed if the implication does not hold. As mentioned above, the initial values for the accumulators *tbb* and *last* are "dummy" and **false**, respectively. Since the post-condition checker for a particular class actually blames a subclass for a hierarchy violation, the first post-condition checker never assigns blame. The initial **false** passed via *last* guarantees that no blame is assigned in the first checker and that "dummy" is ignored. Additionally, the highest class or interface in the hierarchy can never be blamed, since it cannot possibly violate the hierarchy.

Here is the code for the post-condition hierarchy checker for *getMaxSize* in both *RunningConsole* and *Console*:

```
class check_RunningConsole_post extends Object {
  boolean getMaxSize (String tbb, boolean last,
                      RunningConsole this, int getMaxSize) {
    let { res = getMaxSize > 0 }
    in if (!last || res)              // last ⇒ res
         (new check_Console_post())
           .getMaxSize("RunningConsole", res, this, getMaxSize)
       else
         hierErr(tbb)
  }
}
```

and

```
class check_Console_post extends Object {
  boolean getMaxSize (String tbb, boolean last,
                      Console this, int getMaxSize) {
    let { res = getMaxSize > 0 }
    in if (!last || res)          // last ⇒ res
         (new check_IConsole_post())
           .getMaxSize("Console", res, this, getMaxSize)
       else
         hierErr(tbb)
  }
}
```

Finally, the **[call^i]** rule shows how the elaboration re-writes method calls. Each method call becomes a call to a wrapper method, based on the type of the object whose method is invoked. For example, this code fragment:

```
IConsole o = ConsoleFactory(...);
o.display("It's crunch time");
```

is rewritten to this:

```
IConsole o = ConsoleFactory(...);
o.display_IConsole("It's crunch time");
```

Figure 4 gathers the code fragments of our running example. The left column contains the proper interfaces and classes, enriched with wrapper methods. The right column contains the hierarchy checking classes, plus the translation of the method call.

3.3 Evaluation

The operational semantics for Contract Java is defined as a contextual rewriting system on pairs of expressions and stores [6, 23]. Each evaluation rule has this shape:

$$P \vdash \langle e, \mathcal{S} \rangle \hookrightarrow \langle e, \mathcal{S} \rangle \quad \textbf{[reduction rule name]}$$

A store ($\mathcal{S}$) is a mapping from *object*s to class-tagged field records. A field record ($\mathcal{F}$) is a mapping from field names to values. We consider configurations of expressions and stores equivalent up to α-renaming; the variables in the store bind the free variables in the expression. Each *e* is an expression and P is a program, as defined in figure 1.

The complete evaluation rules are in Figure 5. For example, the **call** rule models a method call by replacing the call expression with the body of the invoked method and syntactically replacing the formal parameters with the actual parameters. 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.

Both **call** and **super** expressions reduce to **return** expressions. The **return** expressions are markers that signal where post-condition contract violations might occur. They are inserted by method call and super call reductions for the statement of the contract soundness theorem.

4. CONTRACT SOUNDNESS

A contract monitoring system must have two properties. First, at a minimum, it must preserve the semantics of the programming language. Second, contract monitoring must guarantee certain properties for the evaluation of a program.

Since contracts in our model are arbitrary Java expressions, they may have side-effects or raise errors and may thus affect the behavior of the underlying program. Considering the role of contracts, this is undesirable. We therefore restrict our attention to contracts that are effect-free.[5]

DEFINITION 1 (EFFECT-FREE EXPRESSION). *An expression e is* effect-free *if for any store S such that the free variables of e are included in dom(S), there exists a value v such that $\langle e, S\rangle \hookrightarrow^* \langle v, S\rangle$.*

The key to this definition is that the effect-free expressions evaluate to a value without changing the store or signalling an error. This does not mean that *e* never allocates, however. Since garbage collection is modeled as a non-deterministic reduction step, a contract expression *e* may allocate as long as the newly allocated objects are garbage when the evaluation of the contract produces a value.

Kleene equality, another supplementary definition, is used as the equality on programs. Informally, it states that terminating programs are equivalent when both produce the same answer or the same error. Additionally, all non-terminating programs are equivalent.

DEFINITION 2 (KLEENE EQUALITY).

Two programs P_1 and P_2 are Kleene equal *if one of the following conditions holds:*

- $\langle P_1, \emptyset\rangle \Uparrow$ *and* $\langle P_2, \emptyset\rangle \Uparrow$,
- $\langle P_1, \emptyset\rangle \hookrightarrow^* \langle v, S\rangle$ *and*
  $\langle P_2, \emptyset\rangle \hookrightarrow^* \langle v, S\rangle$ *for some store S and value v*
- $\langle P_1, \emptyset\rangle \hookrightarrow^* \langle \text{error: } str, S_1\rangle$ *and*
  $\langle P_2, \emptyset\rangle \hookrightarrow^* \langle \text{error: } str, S_2\rangle$ *for some stores S_1 and S_2 and error message str.*

Definition 4 specifies coherence. Intuitively, coherence guarantees that the contract checker preserves the meaning of programs that do not violate any contracts. To define coherence, we use a trivial contract elaborator, *Erase*, which merely erases the contracts and does not insert any calls to the error-signalling primitives.

DEFINITION 3 (ELABORATION COHERENCE). *An elaboration T from annotated Contract Java (syntax (b) in figure 1) to unannotated Contract Java (syntax (c) in figure 1) is coherent if for any program P whose pre- and post-conditions are all effect-free expressions, one of the following conditions holds:*

- *$T(P)$ is Kleene equal to Erase(P),*

[5] In practice, there are many approaches to enforcing this restriction, each with different pros and cons.

```
interface IConsole {
  int getMaxSize();
  void display(String s);
}

class Console implements IConsole {
  int getMaxSize() { ... }
  int getMaxSize_IConsole ...
  int getMaxSize_Console ...

  void display(String s) { ... }
  void display_IConsole ...
  void display_Console(String s, string cname) {
    if ( s.length() < this.getMaxSize() ) {
      (new check_Console_pre()).display(this, s);
      let { display = this.display(s) }
      in {
        (new check_Console_post()).display
          ("dummy", true, this, display, s);
      }
    } else {
      preErr(cname);
    }
  }
}

class RunningConsole extends Console {
  int getMaxSize_IConsole ...
  int getMaxSize_Console ...
  int getMaxSize_RunningConsole ...
  void display(String s) {
    ... super.display
          (String.substring
            (s, ..., ... + getMaxSize())) ...
  }
  void display_IConsole ...
  void display_Console ...
  void display_RunningConsole ...
}

class PrefixedConsole extends Console {
  int getMaxSize_IConsole ...
  int getMaxSize_Console ...
  int getMaxSize˙PrefixedConsole ...
  String getPrefix() {
    return ">> ";
  }
  void display(String s) {
    super.display(this.getPrefix() + s)
  }
  void display_IConsole ...
  void display_Console ...
  void display˙PrefixedConsole ...
}
```

```
class check_IConsole_pre { ... }

class check_Console_pre extends Object {
  boolean display (Console this, String s) {
    let { next = (new check_IConsole_pre()).display(this, s)
          res = s.length() < this.getMaxSize() }
    in if (!next || res)          // next ⇒ res
          res
       else
         hierErr("Console")}}

class check_RunningConsole_pre extends Object {
  boolean display (RunningConsole this, String s) {
    let { next = (new check_Console_pre()).display(this, s)
          res = true }
    in if (!next || res)          // next ⇒ res
          res
       else
         hierErr("RunningConsole")
  }
}
class check_PrefixedConsole_pre { ... }

class check_IConsole_post { ... }

class check_Console_post extends Object {
  boolean getMaxSize (String tbb, boolean last,
                      Console this, int getMaxSize) {
    let { res = getMaxSize > 0 }
    in if (!last || res)          // last ⇒ res
          (new check_IConsole_post())
            .getMaxSize("Console", res, this, getMaxSize)
       else
          hierErr(tbb)
  }
}

class check_RunningConsole_post extends Object {
  boolean getMaxSize (String tbb, boolean last,
                      RunningConsole this, int getMaxSize) {
    let { res = getMaxSize > 0 }
    in if (!last || res)          // last ⇒ res
          (new check_Console_post())
            .getMaxSize("RunningConsole", res, this, getMaxSize)
       else
          hierErr(tbb)
  }
}

class check_PrefixedConsole_post { ... }

IConsole o = ConsoleFactory(...);
o.display_IConsole("It's crunch time");
```

Figure 4: Elaborated Console Example

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

$$P \vdash \langle \mathsf{E}[\mathit{object} : t.\mathit{md}(v_1, \ldots, v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\textbf{return} : t, c\ \{e[\mathit{object}/\mathit{this}, v_1/\mathit{var}_1, \ldots\ v_n/\mathit{var}_n]\}], \mathcal{S}\rangle \qquad [\textbf{call}]$$
where $\mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle$ and $\langle \mathit{md}, (t_1 \ldots t_n \longrightarrow t), (\mathit{var}_1 \ldots \mathit{var}_n), e\rangle \in^{\varsigma}_{P} c$

$$P \vdash \langle \mathsf{E}[\textbf{super}\,\underline{\equiv \mathit{object} : c}\,.\mathit{md}(v_1, \ldots, v_n)], \mathcal{S}\rangle \qquad [\textbf{super}]$$
$$\hookrightarrow \langle \mathsf{E}[\textbf{return} : c, c\ \{e[\mathit{object}/\mathit{this}, v_1/\mathit{var}_1, \ldots\ v_n/\mathit{var}_n]\}], \mathcal{S}\rangle$$
where $\langle \mathit{md}, (t_1 \ldots t_n \longrightarrow t), (\mathit{var}_1 \ldots \mathit{var}_n), e\rangle \in^{\varsigma}_{P} c$

$$P \vdash \langle \mathsf{E}[\textbf{return} : t, c\ \{\ v\ \}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}\rangle \qquad [\textbf{return}]$$

$$P \vdash \langle \mathsf{E}[\textbf{new}\ c], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathit{object}], \mathcal{S}[\mathit{object} \mapsto \langle c, \mathcal{F}\rangle]\rangle \qquad [\textbf{new}]$$
where $\mathit{object} \notin \mathrm{dom}(\mathcal{S})$ and $\mathcal{F} = \{c'.\mathit{fd} \mapsto \textbf{null} \mid c \leq^{\varsigma}_{P} c' \text{ and } \exists t \text{ s.t. } \langle c'.\mathit{fd}, t\rangle \in^{\varsigma}_{P} c'\}$

$$P \vdash \langle \mathsf{E}[\mathit{object}\underline{\;:c'\;}.\mathit{fd}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}\rangle \qquad [\textbf{get}]$$
where $\mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle$ and $\mathcal{F}(c'.\mathit{fd}) = v$

$$P \vdash \langle \mathsf{E}[\mathit{object}\underline{\;:c'\;}.\mathit{fd} = v], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[v], \mathcal{S}[\mathit{object} \mapsto \langle c, \mathcal{F}[c'.\mathit{fd} \mapsto v]\rangle]\rangle \qquad [\textbf{set}]$$
where $\mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle$

$$P \vdash \langle \mathsf{E}[\textbf{view}\ t'\ \mathit{object}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathit{object}], \mathcal{S}\rangle \qquad [\textbf{cast}]$$
where $\mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle$ and $c \leq_P t'$

$$P \vdash \langle \mathsf{E}[\textbf{let}\ \mathit{var}_1 = v_1\ \ldots\ \mathit{var}_n = v_n\ \textbf{in}\ e], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[v_1/\mathit{var}_1 \ldots v_n/\mathit{var}_n]], \mathcal{S}\rangle \qquad [\textbf{let}]$$

$$P \vdash \langle \mathsf{E}[\textbf{if}\ (\ \textbf{true}\ )\ e_1\ \textbf{else}\ e_2], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e_1], \mathcal{S}\rangle \qquad [\textbf{iftrue}]$$

$$P \vdash \langle \mathsf{E}[\textbf{if}\ (\ \textbf{false}\ )\ e_1\ \textbf{else}\ e_2], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e_2], \mathcal{S}\rangle \qquad [\textbf{iffalse}]$$

$$P \vdash \langle \mathsf{E}[\{\ v\ ;\ e\ \}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e], \mathcal{S}\rangle \qquad [\textbf{seq}]$$

$$P \vdash \langle \mathsf{E}[\textbf{preErr}(c)], \mathcal{S}\rangle \hookrightarrow \langle \text{error: } c \text{ violated pre-condition}, \mathcal{S}\rangle \qquad [\textbf{pre}]$$

$$P \vdash \langle \mathsf{E}[\textbf{postErr}(c)], \mathcal{S}\rangle \hookrightarrow \langle \text{error: } c \text{ violated post-condition}, \mathcal{S}\rangle \qquad [\textbf{post}]$$

$$P \vdash \langle \mathsf{E}[\textbf{hierErr}(t)], \mathcal{S}\rangle \hookrightarrow \langle \text{error: } t \text{ is a bad extension}, \mathcal{S}\rangle \qquad [\textbf{hier}]$$

$$P \vdash \langle \mathsf{E}[\textbf{view}\ t'\ \mathit{object}], \mathcal{S}\rangle \hookrightarrow \langle \text{error: bad cast}, \mathcal{S}\rangle \qquad [\textbf{xcast}]$$
where $\mathcal{S}(\mathit{object}) = \langle c, \mathcal{F}\rangle$ and $c \not\leq_P t'$

$$P \vdash \langle \mathsf{E}[\textbf{view}\ t'\ \textbf{null}], \mathcal{S}\rangle \hookrightarrow \langle \text{error: bad cast}, \mathcal{S}\rangle \qquad [\textbf{ncast}]$$

$$P \vdash \langle \mathsf{E}[\textbf{null}\underline{\;:c\;}.\mathit{fd}], \mathcal{S}\rangle \hookrightarrow \langle \text{error: dereferenced null}, \mathcal{S}\rangle \qquad [\textbf{nget}]$$

$$P \vdash \langle \mathsf{E}[\textbf{null}\underline{\;:c\;}.\mathit{fd} = v], \mathcal{S}\rangle \hookrightarrow \langle \text{error: dereferenced null}, \mathcal{S}\rangle \qquad [\textbf{nset}]$$

$$P \vdash \langle \mathsf{E}[\textbf{null}.\mathit{md}(v_1, \ldots, v_n)], \mathcal{S}\rangle \hookrightarrow \langle \text{error: dereferenced null}, \mathcal{S}\rangle \qquad [\textbf{ncall}]$$

Figure 5: Operational semantics for Contract Java

- $\langle T(P), \emptyset\rangle \hookrightarrow^* \langle$ error: c violated pre-condition$, S\rangle$, *for some store* S;
- $\langle T(P), \emptyset\rangle \hookrightarrow^* \langle$ error: c violated post-condition$, S\rangle$, *for some store* S;
- $\langle T(P), \emptyset\rangle \hookrightarrow^* \langle$ error: t is a bad extension$, S\rangle$, *for some store* S.

Our elaboration does not change any expressions except method calls. Furthermore, the wrapper methods have no effect if the contracts have no effect and evaluate to **true**. If any contract does evaluate to **false**, our contract checker is guaranteed to signal one of the three errors listed above. Thus, our elaboration T is coherent.

Definition 4 specifies contract soundness. Intuitively, soundness guarantees that elaborated programs respect the contracts of the *original* program. More concretely, if a program that the contract elaborator produces does not signal a contract error, the contract-erased program must be contract sound at each step of its evaluation.

DEFINITION 4 (CONTRACT SOUNDNESS). *An elaboration* T *is* contract sound *if for any program* P *whose pre- and post-conditions are effect-free expressions, one of the following conditions holds:*

- $\langle T(P), \emptyset\rangle \hookrightarrow^* \langle$error: c violated pre-condition$, S\rangle$, *for some store, S,*
- $\langle T(P), \emptyset\rangle \hookrightarrow^* \langle$error: c violated post-condition$, S\rangle$, *for some store, S,*
- $\langle T(P), \emptyset\rangle \hookrightarrow^* \langle$error: t is a bad extension$, S\rangle$, *for some store, S, or*
- *For each state* $\langle P', S'\rangle$ *such that* $\langle Erase(P), \emptyset\rangle \hookrightarrow^* \langle P', S'\rangle$, $\langle P', S'\rangle$ *is locally contract sound with respect to* P *(recall that* $Erase(P)$ *is just* P*, but with the contract annotations erased).*

Roughly, local contract soundness for a configuration $\langle e, S\rangle$ means that in the given store S, the contracts about e hold and that the necessary relations between contracts in e hold as well. More precisely, all states that do not perform a method call or a method return are locally contract sound. A state that is about to evaluate a method call is locally sound if the two conditions are true. First, the pre-condition on the method must be satisfied. Second, the pre-condition hierarchy must be behaviorally well-formed. That is, each type's pre-condition must imply each of its subtypes' pre-conditions, for the method about to be invoked. Similarly, a state that is about to perform a method return is locally sound if the post-condition on the method is satisfied and the post-condition hierarchy is behaviorally well-formed.

DEFINITION 5 (LOCAL CONTRACT SOUNDNESS). *A program state* $\langle e, S\rangle$ *is* locally contract sound *with respect to a Contract Java program* P*, if one of the following conditions holds:*

- $e = E[o.m : t\ (v_1, v_2, \ldots, v_k)]$

  and $S(o) = \langle c, \mathcal{F}\rangle$

  and *if there exists a* y *such that* y PRE_P $\langle t, m\rangle$, *then* $\langle y, S\rangle \hookrightarrow^* \langle \mathbf{true}, U\rangle$ *for some store* U.

  and *for any* s, s' *such that* $c \leq_P s \leq_P s'$, *if there exists an* x *and* x' *such that* x PRE_P $\langle s, m\rangle$, x' PRE_P $\langle s', m\rangle$, *then* $\langle x, S\rangle \hookrightarrow^* \langle b, T\rangle$, $\langle x', S\rangle \hookrightarrow^* \langle b', T'\rangle$, *and* $b' \Rightarrow b$

- $e = E[\mathbf{return} : t,\ c\ \{\ v\ \}]$

  and *if there exists a* y *such that* y POST_P $\langle t, m\rangle$, *then* $\langle y, S\rangle \hookrightarrow^* \langle \mathbf{true}, U\rangle$ *for some store* U.

  and *for any* s, s' *such that* $c \leq_P s \leq_P s'$, *if there exists an* x *and* x' *such that* x POST_P $\langle s, m\rangle$, x' POST_P $\langle s', m\rangle$, *then* $\langle x, S\rangle \hookrightarrow^* \langle b, T\rangle$, $\langle x', S\rangle \hookrightarrow^* \langle b', T'\rangle$, *and* $b \Rightarrow b'$.

- *e is neither a method call or method return.*

THEOREM 5. *The elaboration* T *is contract sound.*

PROOF SKETCH. Let P be a program. Assume that $T(P)$ does not signal a contract error. If $T(P)$ is to be contract sound, we must show that each reduction step of $Erase(P)$ is locally contract sound.

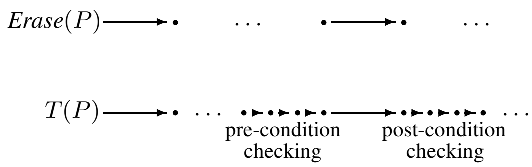
Lemma: $T(P)$ *takes every reduction step that* $Erase(P)$ *takes.* Since the elaboration does not change any expressions except method calls, $Erase(P)$ and $T(P)$ are synchronized, as long as there are no method calls. Let us consider the first method call. The reductions for $Erase(P)$ look like this:

$$\begin{array}{lll} \langle Erase(P), \emptyset\rangle & \hookrightarrow & \ldots \\ & \hookrightarrow & \langle E[o.m : t(v_1, \ldots, v_n)]\ , S\rangle \\ & \hookrightarrow & \langle\ E[\mathbf{return} : t, c\ b[x_1/v_1 \ldots\ x_n/v_n], S\rangle \end{array}$$

Since the ellipses do not contain any method calls, the reductions up to the first method call are identical for $T(P)$. Then, the elaborated version calls the wrapper method. We know that the wrapper method does not have any effects, since their contract expressions are effect-free and $T(P)$ does not signal a pre-condition error or a hierarchy error. Thus, the reduction sequence looks like this:

$$\begin{array}{lll} \langle T(P), \emptyset\rangle & \hookrightarrow & \ldots \\ & \hookrightarrow & \langle E[o.m\_t(v_1, \ldots, v_n)]\ , S\rangle \\ & \hookrightarrow & \ldots \\ & \hookrightarrow & \langle E[F[o.m : t(v_1, \ldots, v_n)]]\ , S\rangle \\ & \hookrightarrow & \langle\ E[F[\mathbf{return} : t, c\ b[x_1/v_1 \ldots\ x_n/v_n]], S\rangle \end{array}$$

The extra context, *F*, is the remainder of the wrapper method that checks the post-conditions and the post-condition hierarchy. Since the post-conditions are effect-free and $T(P)$ does not raise a hierarchy error or a post-condition error, that code has no effect on the computation. Pictorially, the two reduction sequences look like this:



for the first method call. The smaller arrows are the extra steps that $T(P)$ takes, before and after each method call. By an inductive argument, we can conclude for each reduction that *Erase*(P) takes, $T(P)$ also takes the same reduction step, possibly with extra context that is the contract enforcement. Thus, the lemma holds.

Now, using the lemma, we can prove the theorem. Let $\langle e, S\rangle$ be a step in the reduction sequence starting from *Erase*(P). If *e* does not decompose into some evaluation context and a method call or some evaluation context and a return instruction, it is locally hierarchy sound. Assume that it does decompose into a context and a method call. Now, we must show that the first bullet from definition 4 is true. Since $T(P)$ reached the same method call by the previous argument, we know that the wrapper method was invoked. From the **[wrap]** rule in figure 2, we can see that the pre-condition check must have succeeded. All that remains is to show this:

for any s, s' such that $c \leq_P s \leq_P s'$,
  if there exists an x and x' such that
    $x \;\textsf{PRE}_P\; \langle s, m\rangle, x' \;\textsf{PRE}_P\; \langle s', m\rangle$,
  then $\langle x, S\rangle \hookrightarrow^* \langle b, T\rangle$,
    $\langle x', S\rangle \hookrightarrow^* \langle b', T'\rangle$, and
    $b' \Rightarrow b$

This states that if there are two types, s, and s' with pre-conditions x and x' that evaluate to b and b', we must have $b \Rightarrow b'$. Since the hierarchy checkers traverse the entire hierarchy checking that the pre-condition of each type implies the pre-condition of each of its subtypes, this holds. Thus, this step is locally hierarchy sound.

Similarly, if *e* decomposes into a context and a method return, $T(P)$ must also have **return**ed and the wrapper method's code must have been invoked, so this step is also locally contract sound. $\square$

5. RELATED WORK

The calculus of Contract Java and its contract soundness theorem show what it means to monitor and enforce contracts in an object-oriented world. They are analogous to typed lambda calculi with constants and primitives and type soundness theorems. Using type soundness theorems, we can analyze the type systems of existing languages, say ML, and pinpoint their flaws. Similarly, we can study existing contract enforcement mechanisms and determine how well they monitor the contracts in a program.

Many existing contract checking systems [4, 7, 9, 10, 11, 12, 17, 20] collect all pre-conditions of a method and its super-methods in a disjunction.[6] Similarly, they collect the post-conditions of a method and its super-methods in a conjunction. Since

$$pre_{super} \Longrightarrow pre_{super} \;\textbf{or}\; pre_{type}$$

is a tautology and since

$$post_{super} \;\textbf{and}\; post_{type} \Longrightarrow post_{super}$$

is also a tautology, these tools cannot detect malformed pre-condition or post-condition hierarchies. Put differently, when the tools rewrite the programmer's contracts, they assume that *a programmer does not make mistakes concerning the relationship of pre-conditions on methods and their super-methods or post-conditions on methods and their super-methods.* Instead they assume that the implementor of a class has a perfect understanding of the superclass's contracts and merely adds additional specifications. We believe that it is ironic that tools that check a programmer's capability to maintain contracts for procedure-like code trust the programmer when it comes to the even more subtle task of forming correct hierarchies of contracts.

Some tool designers are aware of the problem. For example, Kandorman et al. [9] point out that re-writing the programmer's contracts in the above-mentioned manner leads to undetected contract violations and malformed type hierarchies. Here is their example:

```
interface I {
  int m(int a);
    @pre { a > 0 }
}
```

The interface contains a single method, *m*, with the pre-condition requirement that *a* is greater than *0*. Now, imagine this extension:

```
interface J extends I {
  int m (int a);
    @pre { a > 10 }
}
```

Since *J* extends *I*, the pre-condition for *I* should imply the pre-condition for *J*. That is not the case here. A witness to the failure of the implementation is a method call where *a* is *5*. Thus, the interface extension is faulty, when judged with the programmer's original contracts.

Unfortunately, rewriting a series of pre-conditions into a disjunction cannot reveal the problem. Because

$$(a > 0) \Longrightarrow (a > 0) \;\textbf{or}\; (a > 10)$$

is always true, none of these contract enforcement tools reveal that *J*'s implementor did not properly understand the precondition of *m* in the interface *I*. In short, contract monitoring systems that rewrite hierarchy contracts into conjunctions and disjunctions are not contract sound in the sense of our theorem.

While this example may look artificial at first glance, it is just an abstract version of the console example from section 2. As a test, we translated *IConsole*, *Console* and *PrefixedConsole* to iContract syntax [12], using 4 as the maximum and a dummy *display* routine that just prints to stdout. We created a *main* method that invokes the *PrefixedConsole*'s *display* method with the string "abc", as follows:

```
new PrefixedConsole().display("abc");
```

[6] Kiev's manual [10] seems to state that it treats pre- and post-condition on derived methods in this manner, but we were unable to get Kiev to run.

This call is erroneous, since the pre-condition on *PrefixedConsole*'s *display* method requires the input to be a string of at most one character. iContract responded with this error message:[7]

```
java.lang.RuntimeException:
error: precondition violated
(Console::display(String)):
(/*declared in IConsole::display(String)*/
(s.length() < this.getMaxSize()))
   at Console.display
   at PrefixedConsole.display
   at Main.main
```

iContract blames the call to **super**.*display* inside *PrefixedConsole*'s *display* method, rather than blaming the hierarchy or the caller of *PrefixedConsole*'s *display*. In general, any tool that rewrites the programmer's pre- and post-conditions in this manner would produce an analogous result for this program, and would cause programmers to look for errors in the wrong place.

Jass [2, 3] is the only contract checker for Java that can discover errors in the type hierarchy. To discover hierarchy errors with Jass, a programmer must specify a simulation method that creates an object of a supertype from the current object. The contract checker uses the simulation method to create a supertype object each time a method is called or a method returns. Then, it checks that the relevant contracts of the supertype object and the original object are related via the proper implications. If not, the contract checker signals a hierarchy error.

Jass and Contract Java differ in many respects. First, subtypes in Contract Java must function on the same state space as their respective supertypes. This implies that the programmer doesn't have to define a simulation method, and that the checks are significantly cheaper than Jass's because no new objects are created. Second, Contract Java's notion of subtyping naturally extends to interfaces, unlike Jass's. Third, Contract Java checks the entire portion of a hierarchy atop a given method, rather than just a single step. Finally, our model comes with a contract soundness theorem.

6. IMPLEMENTATION FOR FULL JAVA

The Contract Java elaborator given in section 3.2 produces inefficient Java code. There are several changes to the calculus that would improve the efficiency of the generated code. First, many of the hierarchy checking classes can be eliminated. For class definitions, hierarchy checking methods can be added directly to the class. For interface definitions, a single class with static methods can hold all of the hierarchy checkers. Second, in the case of a single inheritance hierarchy, the hierarchy checking can be turned into a loop. Finally, the value of the initial pre-condition test can be saved and re-used during the pre-condition hierarchy checking. Similarly, the value of the post-condition test can be re-used during the post-condition hierarchy checking. Some of these issues are considered in more detail in a companion paper [5].

7. CONCLUSION

This paper presents Contract Java, a calculus of Java programs with contracts. These contracts come as pre- and post-conditions on methods in classes and interfaces. The calculus specifies how programs with contracts are elaborated into plain Java programs with code that enforces the integrity of contracts. This code specifically enforces four properties of method calls and returns, respectively: the pre-conditions hold for a method call; the post-conditions hold for a method return; the pre-conditions of a method imply the pre-conditions of overriding methods in the subtypes; and the post-conditions of a method are implied by the post-conditions of overridden methods in the supertypes. Our contract soundness theorem guarantees that the last two items are properly enforced as far as the programmer-stated contracts are concerned.

Contract Java is the first complete semantic account of run-time enforced contracts in an object-oriented setting. The contract soundness theorem is a formal summary of its properties, especially with respect to behavioral subtyping. Our work thus fills a gap between the established theory of behavioral subtyping [1, 14, 15] and established practice of contract checking for Java [2, 3, 4, 9, 10, 12, 17] and object-oriented languages in general [7, 11, 20].

[7] iContract's response has been edited for clarity and formatting.

ACKNOWLEDGEMENTS

Thanks to Philippe Meunier for his detailed comments on a draft of this paper. We also thank Daniel Jackson, Shriram Krishnamurthi, and Clemens Szyperski for valuable discussions about this work. In addition, we gratefully acknowledge the support of the NSF (CCR-9619756) and Texas ATP (#003604-0126-1999).

8. REFERENCES

[1] America, P. Designing an object-oriented programming language with behavioural subtyping. In *Proceedings of Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 60–90. Springer-Verlag, 1991.

[2] Bartetzko, D. Parallelität und Vererbung beim Programmieren mit Vertrag. Diplomarbeit, Universität Oldenburg, April 1999.

[3] Bartetzko, D., C. Fischer, M. Moller and H. Wehrheim. Jass - Java with assertions. In *Workshop on Runtime Verification*, 2001. held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01.

[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] Findler, R. B., M. Latendresse and M. Felleisen. Behavioral contracts and behavioral subtyping. In *Proceedings of ACM Conference Foundations of Software Engineering*, 2001.

[6] Flatt, M., S. Krishnamurthi and M. Felleisen. Classes and mixins. In *Proceedings of ACM Conference Principles of Programming Languages*, pages 171–183, Janurary 1998.

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

[8] Gosling, J., B. Joy and G. Steele. *The Java(tm) Language Specification*. Addison-Wesley, 1996.

[9] 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 *lncs*, July 1999.

[10] Kizub, M. Kiev language specification. http://www.forestro.com/kiev/, 1998.

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

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

[13] Leavens, G. T., K. R. M. Leino, E. Poll, C. Ruby and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *Object-Oriented Programming, Systems, Languages, and Applications Companion*, pages 105–106, 2000. Also Department of Computer Science, Iowa State University, TR 00-15, August 2000.

[14] Liskov, B. H. and J. Wing. Behavioral subtyping using invariants and constraints. Technical Report CMU CS-99-156, School of Computer Science, Carnegie Mellon University, July 1999.

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

[16] 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.

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

[18] Meyer, B. *Object-oriented Software Construction*. Prentice Hall, 1988.

[19] Meyer, B. Applying design by contract. *IEEE Computer*, 25(10):40–51, October 1992.

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

[21] Milner, R. A theory of type polymorphism in programming. *Journal of Computer Systems Science*, 17:348–375, 1978.

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

[23] Wright, A. and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, pages 38–94, 1994. first appeared as Technical Report TR160, Rice University, 1991.