Semantic Casts

Contracts and Structural Subtyping in a Nominal World

Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen

[1] University of Chicago; Chicago, IL, USA; robby@cs.uchicago.edu
[2] University of Utah; Salt Lake City, UT, USA; mflatt@cs.utah.edu
[3] Northeastern University; Boston, MA, USA; matthias@ccs.neu.edu

Abstract

Nominal subtyping forces programmers to explicitly state all of the subtyping relationships in the program. This limits component reuse, because programmers cannot anticipate all of the contexts in which a particular class might be used. In contrast, structural subtyping implicitly allows any type with appropriate structure to be used in a given context. Languagues with contracts exacerbate the problem. Since contracts are typically expressed as refinements of types, contracts in nominally typed languages introduce additional obstacles to reuse.

To overcome this problem we show how to extend a nominally typed language with semantic casts that introduce a limited form of structural subtyping. The new language must dynamically monitor contracts, as new subtyping relationships are exploited via semantic casts. In addition, it must also track the casts to properly assign blame in case interface contract are violated.

1 Enriching Nominal Subtypes with Semantic Casts

Conventional class-based object-oriented languages like C++ [43], C# [33], Eiffel [32], and Java [18] come with nominal typing systems. In such systems, a programmer explicitly names the superclass(es) and the implemented interfaces of a class. Thus, the declared type of any instance of a class must be one of the explicitly named interfaces or classes.

Language designers choose nominal type systems because they are easy to understand and easy to implement. A programmer doesn't need to investigate the structure of an interface *I* to find out whether an instance *o* of a class *C* can have type *I*; it suffices to check whether the definition of *C* mentions *I* as an implemented interface (or whether the superclasses and superinterfaces mention *I*). A compiler writer, in turn, can build a class graph and an interface graph and type check expressions and statements by comparing points in a graph.

Nominal typing, however, is also a known obstacle to software reuse. In particular, a programmer can only compose two objects if the creators of the two respective classes used the same (nominal) types. Unfortunately, in a world of software components where third-party programmers compose existing pieces of software, the implementor of a class cannot possibly anticipate all possible types for an object. Hence, programmers resort to casts and have invented adapter patterns to bridge the gap between third-party components.

One way to overcome this problem is to switch to a structural type system. The research community has long recognized this shortcoming of nominal subtype systems and that structural subtype systems do not suffer from this flaw. Some modern research languages like LOOM [3], OCaml [28], OML [39], PolyTOIL [4], and Moby [13] adopt structural subtype systems. Their designs demonstrate how their structural subtype systems empower their user communities to reuse classes in unanticipated situations.

Changing a language's subtype system from a nominal to a structural perspective is a drastic step. We therefore propose an alternative, smaller change to conventional languages that also overcomes the reuse problem. Specifically, our proposal is to introduce a "semantic cast" mechanism. The cast allows programmers to change the type of an object according to a structural subtype criteria. Thus, if an existing class *C* satisfies the needs of some interface *I* but doesn't explicitly implement it, a programmer can, even retroactively, specify that an instance of *C* is of type *I*.

Naturally, the programmer should only take such an action if the semantics of the class is that of the interface. We therefore allow the programmer to describe an executable approximation of the interface's semantics—called *contracts* here—and use that semantics to monitor the validity of the cast. If the cast object behaves according to the contracts, the execution proceeds as normal. Otherwise, the monitoring system raises an exception and attributes the misbehavior to a specific component, *i.e.*, either the object's use-context, the object itself, or the cast.

In this paper, we explain the need for these contract-based casts, their design, their implementation, and our experience with the contract system. We present the ideas in a Java-like setting to show how they can be adapted to conventional languages. Indeed, we only present the internal form of the new construct, rather than a surface syntax. Section 2 describes a common situation where nominal subtyping fails to support reuse effectively. Section 3 presents our semantic cast construct and reformulates the example from section 2 with this construct. Section 4 precisely specifies the new contract checker with a calculus. Section 5 discusses our implementation. The last three sections discuss related work, future work, and present our conclusions.

2 Contracts and Component Reuse

In this section, we introduce object-oriented contracts and illustrate how languages with contracts that augment a nominal subtyping hierarchy inhibit reuse.

Consider the canonical queue implementation in figure 1 (in Java syntax, using JML [26] notation for contracts). The queue supports three operations: *enq* to add an element to the queue, *deq* to remove an element from the queue, and *empty* to test if the queue contains any elements. The post-condition contract on *enq* guarantees that the queue is not empty after an element is added and the pre-condition contract on *deq* requires that there is an element in the queue to remove.

Enforcing pre- and post-conditions such as these is straightforward. When the *enq* method returns, the post-condition code is run and if it produces **false**, evaluation terminates and *enq* is blamed for breaking its contract. Similarly, when *deq* is called, the pre-condition code is run and if it produces **false**, evaluation terminates and *deq*'s caller is blamed for breaking *deq*'s contract. Although these contracts do not ensure that the

```
interface IQueue {
  void enq(int x);
  // @post !empty()

  void deq(int x);
  // @pre !empty()

  boolean empty();
}
```

```
class Q implements IQueue {
  void enq(int x) { ... }
  int deq() { ... }
  boolean empty() { ... }
}
```

Fig. 1. Queues

queue implementation is correct, experience has shown that such weak contracts provide a good balance between correctness and run-time overhead [**?**].

Object-oriented languages allow much more complex forms of interaction than those between the queue and its client. Since objects may be passed as arguments or returned as results from methods, the call structure of the program can depend on the flow of values in the program. Put differently, invoking an object's methods may trigger nested callbacks (a.k.a upcalls) between components [44].

```
class Q implements IQueue {
  IObserver o;
  void enq(int x) {
    ...
    if (o != null) o.onEnq(this, x);
  }
  int deq() {
    int hd = ...;
    if (o != null) o.onDeq(this, hd);
    ...
    return hd;
  }
  boolean empty() { ... }
  void registerObs(IObserver _o) {o=_o;}
}
```

```
interface IObserver {
  void onEnq(Queue q, int x);
  // @post !q.empty()

  void onDeq(Queue q, int x);
  // @pre !q.empty()
}
```

Fig. 2. Queues with Observers

Consider the revised queue class in figure 2; this variant of the class supports an observer. The additional method *registerObs* accepts an observer object. This observer object is saved in a field of the queue and its methods are invoked when an element is enqueued or dequeued from the queue.

Although this addition may seem innocuous at first, consider the misbehaved observer in figure 3. Instances of this observer immediately dequeue any objects added to the queue. Imagine that an instance of this observer were registered with an instance of the *Q* class. The first time the *enq* method is invoked, it adds an integer to the queue and then invokes the observer. Then the observer removes the integer, before the *enq* method returns. Due to the *onEnq* post-condition in the *IObserver* interface, however, *BadO* is immediately indicted, ensuring the *Q* class can always meet its contracts.

```
class BadO implements IObserver {
  ...
  onEnq(Queue q, int x) {
    q.deq(); }}
```

Fig. 3. Bad Observer

Programming language designers (including the authors of this paper) have historically been satisfied with contracts in interfaces and abstract classes [2, 8, 12, 17, 21–24, 31, 32]. Unfortunately, this design decision exacerbates the problems with software reuse in a nominally typed world. Independent producers of components cannot possibly foresee the precise contracts that some component should satisfy. Indeed, if they aim to produce software components that are as flexible as possible they must have the least constraining interface contracts (that are still safe). Accordingly, contract checkers must allow component programmers to refine a component's contracts. These refinements, in turn, allow programmers to rely on different extensions of a component's contracts when using it in different contexts.

Concretely, consider the interface *IPosQueue* and static method *ProcessManager* in figure 4. The interface limits the queue to contain only positive integers by adding a pre-condition to *enq* guaranteeing that its input is bigger than zero and a post-condition to *deq* promising that the result is bigger than zero. The static method *ProcessManager* accepts instances of *IPosQueue*. Clearly, the *Q* class satisfies the *IPosQueue* interface. Regardless, since interfaces must be declared when the class is declared, the code in figure 4 cannot be combined with the independently produced code in figure 2.

Programmers can work around this mismatch with several techniques, especially the adapter pattern. In this particular example, the programmer could derive a class from *Q* that inherits all the methods and superimposes the new, stronger contract interface. In general, however, the programmer that wishes to impose additional contracts to an object is not the programmer that originally created the object. In these other cases, a programmer may create an entirely new class that bridges the gap between the two components that are to be composed. No matter which solution the programmer chooses, however, the requirement to build and manually maintain an adapter, including error checking that catches and flags errors inside the adapter, is an obstacle to controlled composition of software. Worse, a programmer-produced mechanism for as-

```
interface IPosQueue {
  void enq(int x);
  // @pre x > 0
  // @post !empty()

  int deq();
  // @pre !empty()
  // @post deq > 0

  boolean empty();
}
```

```
class QueueClient {
  static void ProcessManager(IPosQueue q) {
    ...
  }
}
```

Fig. 4. Positive Queues, in a Separate Component

signing blame is ad-hoc and therefore less trustworthy than a mechanism designed into the programming language.

3 Contract Checking for Semantic Casts

The problem is that allowing contracts only in interfaces and classes means that each object supports only a fixed, pre-determined set of contracts, which prevents the direct use of a *Q* object as an *IPosQueue* object. To overcome this problem, we propose **semanticCast**, a new construct that allows programmers to cast an object to a structurally equivalent type.[4]

The shape of a **semanticCast** expression is:

[4] For the purposes of this paper, we treat **semanticCast** as a regular member of the programming language, to be written in programs at the programmer's whim. In fully integrated system, however, **semanticCast** expressions should only appear at component boundaries. For example, if Java's package system or some other form of module system were used to organize a program, **semanticCast** expressions should be inserted around each variable reference between modules. Abstractly, imagine that a module *A* refers to an export of module *B*, say *B.x*. The context of the variable reference expects it to match interface *I* but the actual type of the variable is a compatible, but different interface I'. The variable reference would be replaced by **semanticCast**($B.x : I', I$, "B", "A") allowing the user of the exported variable to refine the contracts in I' to *I*, while still ensuring that blame is properly assigned.

In a component model similar to Corba [35], components explicitly establish connections to each other via a function call protocol. To add contracts to this style of component system, **semanticCast** expressions would be added by the calls that establish the connections between the components.

Although each component system synthesizes **semanticCast** expressions in a different manner, all component systems can use some form of **semanticCast** expression. In essence, our intention is that a **semanticCast** expression *defines* the component boundaries, as far as our model is concerned. Accordingly, to understand its essence, we treat it as a feature in the programming language directly, with the understanding that it is only truly available to the programmer who implements the component mechanism.

semanticCast(*obj* : *t*, *Intf*, *in_str*, *out_str*)

It consists of four subexpressions: an object (annotated with its type), an interface, and two strings. The expression constructs an object that behaves like *obj*, except with type *Intf* (including the contracts in *Intf*). The typing rules guarantee that the type of *obj* has the same methods names and types as *Intf*, but does not require that *obj*'s class implements *Intf*, allowing *obj* to take on the contracts in *Intf*. In fact, the typing rules synthesize the type of *obj* from the context, but we include it explicitly here, for clarity. The string *in_str* represents the guilty party if *obj* is not treated as an *Intf* by the context, and the string *out_str* represents the guilty party if *o* itself does not behave according to the contracts in *Intf*. As a first approximation, *in_str* is blamed if a pre-condition in *Intf* is violated and *out_str* is blamed if a post-condition of *Intf* is violated.

Using **semanticCast**, we can now combine the code from figure 4 with the original *Q* class:

```
public static void Main(String argv[]) {
  Q q = new Q();
  IQueue iq = semanticCast(q : Q, IQueue, "Main", "Q");
  IPosQueue ipq = semanticCast(iq : IQueue, IPosQueue, "QueueClient", "Main");
  QueueClient.ProcessManager(ipq);
}
```

In the first line of its body, *Main* creates a *Q* object. In the second line, the **semanticCast** expression states that the new instance must behave according to the contracts in *IQueue*.[5] The third argument to **semanticCast** indicates that *Main* is responsible for any violations of *IQueue*'s pre-conditions. The fourth argument indicates that *Q* is responsible for any violations of *IQueue*'s post-conditions. The result of the first **semanticCast** is bound to *iq*.

In the third line, *Main* uses a **semanticCast** expression to add the contracts of *IPosQueue* to *iq*. The third argument to **semanticCast** indicates that *QueueClient* is responsible for pre-condition violations of the contracts in *IPosQueue*. The fourth argument to **semanticCast** indicates that *Main* is responsible for post-condition violations. The result of the second **semanticCast** expression is bound to *ipq*. Finally, in the fourth line, *ipq* is passed to *QueueClient.ProcessManager*.

Intuitively, the queue object itself is like the core of an onion, and each **semanticCast** expression corresponds to a layer of that onion. When a method is invoked, each layer of the onion is peeled back, and the corresponding pre-condition checked, to reveal the core. Upon reaching the core, the actual method is invoked. Once the method returns, the layers of the onion are restored as the post-condition checking occurs.

For instance, imagine that *QueueClient.ProcessManager* invokes its argument's *enq* method, with a positive number. First, the pre-condition on *enq* in *IPosQueue* is checked, since the last **semanticCast** expression added *IPosQueue*'s contracts to the

[5] Of course, the *Q* class declares that it implements the *IQueue* class and the contracts could have been compiled directly into its methods. Since we are focusing on semantic casts here, we assume that contracts are only checked with explicitly specified **semanticCast** expressions.

queue. The input is positive, so it passes. If it had failed, the blame would lie with the queue client. Next, that outer layer is peeled back to reveal an object that must meet *IQueue*'s contracts. Accordingly, the *enq* pre-condition in *IQueue* is checked. This pre-condition is empty, and thus trivially true. After removing this layer we reach the core, so the *enq* method in the *Q* class is invoked.

Once the *enq* method returns, its post-conditions are checked. First, the *enq* post-condition in *IQueue* is checked. If it fails, the blame lies with *Q*, since "Q" is the last argument to the innermost **semanticCast**. Assuming it succeeds, the post-condition on *enq* in *IPosQueue* is checked. If it fails, the blame lies with *Main*, since "Main" is the last argument to the outer **semanticCast** expression.

3.1 Supporting Positive Queues with Positive Observers

The code in figure 5 shows observers added to *IPosQueue*, mirroring the extension of the *IQueue* interface in figure 2. In addition to the *onEnq* and *onDeq* contracts from *IObserver*, the integer argument to both *onEnq* and *onDeq* is guaranteed to be positive.

```
interface IPosObserver {
  void onEnq(IPosQueue q, int x);
  // @pre x > 0
  // @post !q.empty()

  void onDeq(IPosQueue q, int x);
  // @pre x > 0
  // @pre !q.empty()
}
```

```
interface IPosQueue {
  ⋮
  void registerObs(IPosObserver o);
}
```

Fig. 5. Positive Queue with Observer

Imagine that the body of the *QueueClient.ProcessManager* static method creates an instance of some class that implements the *IPosObserver* interface and passes that object to the *registerObs* method of its argument:

```
class QueueClient {
  ⋮
  static void ProcessManager(IPosQueue ipq) {
    IPosObserver po = new ProcessObserver();
    ipq.registerObs(po);
    ipq.enq(5);
  }
}
```

Adding observers to the positive queue triggers additional, indirect contract obligations on the code that casts the queue object to a positive queue. To understand how the

indirect contracts are induced and who should be blamed if they fail, let us examine the sequence of steps that occur when *ipq.enq* is invoked in the body of *ProcessManager*. There are five key steps:

(1) *ipq.enq*(*5*)
⋮
(2) test *IPosQueue* pre-condition, blame *QueueClient* if failure
⋮
(3) *q.enq*(*5*)
⋮
(4) *po.onEnq*(*q*,*5*)
⋮
(5) test *IPosObserver* pre-condition, blame *Main* if failure.

In the first step, *ipq.enq* is invoked, with *5* as an argument. This immediately triggers a check of the *IPosQueue* pre-condition, according to the contract added in *Main*. The contract check succeeds because *5* is a positive number. If, however, the check had failed, blame would lie with *QueueClient* because *QueueClient* supplied the argument to *ipq*.

Next, in step three, the original *IQueue* object's *enq* method is invoked, which performs the actual work of enqueing the object into the queue. As part of this work, it calls the observer (recall figure 2). In this case, *QueueClient* registered the object *po* with the queue, so *po.onEnq* is invoked with the queue and with the integer that was just enqueued.

Since the observer is an *IPosObserver* object, its pre-condition must be established, namely the argument must be a positive number. Because the *Q* class's *enq* method supplies its input to *onEnq*, we know that the contract succeeds at this point. The interesting question, however, is who should be blamed if *Q* had negated the number and passed it to the observer, forcing the *onEnq* contract to fail.

Clearly, *Q* must not be blamed for a failure to establish this pre-condition, since *Q* did not declare that it meets the contracts in the *IPosQueue* interface and, in fact, *IPosQueue* was defined after *Q*. Additionally, *QueueClient* must not be blamed. It only agreed to enqueue positive integers into the queue; if the queue object mis-manages the positive integers before they arrive at the observer, this cannot be *QueueClient*'s fault.

That leaves *Main*. In fact, *Main* should be blamed if the *IPosObserver* object does not receive a positive integer, since *Main* declared that instances of *Q* behave like *IPosQueue* objects knowing that these objects must respect *IPosObserver*'s contracts. Put another way, if the *Q* class had declared it implemented the *IPosQueue* interface, it would have been responsible for the pre-conditions of *IPosQueue*. Accordingly, by casting an instance of *Q* to *IPosQueue*, *Main* is promising that *Q* does indeed live up to the contracts in *IPosQueue*, so *Main* must be blamed if *Q* fails to do so.

More generally, since objects have higher-order behavior, the third and fourth arguments to **semanticCast** do not merely represent who to blame for pre- and post-condition violations of the object with the contract. Instead, the last argument to a **semanticCast** expression indicates who is to blame for any contract that is violated as a

value flows *out* of the object with the contract, whether the value flows out as a result of a method or flows out by calling a method of an object passed into the original object. Conversely, the third argument to a **semanticCast** expression indicates who is to blame for any contract that is violated as a value flows *in* to the object, no matter if the bad value flows in by calling a method, or via a callback that returns the bad value.

This suggests that the casted objects must propagate contracts to method arguments and method results, when those arguments or results are themselves objects. The following equation roughly governs how **semanticCast** expressions propagate (assuming that the immediate pre and post-conditions are satisfied):

semanticCast(*o*: *I*, *J*, *in_str*, *out_str*).*m*(*x*)
=
semanticCast(*o*.*m*(**semanticCast**(*x* : *C*, *D*, *out_str*, *in_str*)) : *B*,
C,
in_str,
out_str)

if *I* and *J* have these shapes:

```
interface I {
  B m(D x);
}
```

```
interface J {
  C m(C x);
}
```

and *B* is a subtype of *C*, which is a subtype of *D*.

Informally, the equation says that when a method *m* of an objected casted to *I* is invoked, the cast is distributed to *m*'s argument and *m*'s result. Further, the distribution is based on *m*'s signature in *I*.

Notice that the blame strings are reversed in the cast around the argument object and stay in the same order in the cast around the result. This captures the difference between values that flow into and out of the object. That is, if a value flows into the argument object, it is flowing out of the original object and if a value flows out of the argument object, it is flowing into the original object. In contrast, when the context invokes methods on the result (assuming it is an object), the sense of the blame is like the original. The reversal corresponds to the standard notion of contra-variance for method or function arguments.

4 Calculus

This section presents a calculus for a core sequential Java (without reflection), enriched with **semanticCast** expressions, and it gives meaning to the semantic cast expressions via a translation to the calculus without them.

For familiarity, this paper builds on our model of Java [10, 16], but the core ideas carry over to any class-based object-oriented language, including C++, C#, Eiffel, or even MzScheme's class-based object system.

Fig. 6. Syntax; before and after contracts are compiled away

4.1 Syntax

Figure 6 contains the syntax for our enriched 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. The contract compiler elaborates syntax (b) to syntax (c). It elaborates the pre- and post-conditions and **semanticCast** expressions into monitoring code; the result is accepted by the evaluator for plain Java.

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. An interface consists of method specifications and their contracts. The contracts are arbitrary Java expressions that have type **boolean**. To simplify the model, we do not allow classes as types. This is not a true restriction to Java, however, since each class can be viewed

as (implicitly) defining a interface based on its method signatures. This interface can be used everywhere the class was used as a type.

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; instance variables are initialized to **null**. The **view** form represents Java's casting expressions and **instanceof** tests if an object has membership in a particular type. The **let** forms represent the capability for binding variables locally. The **if** expressions test the value of the first expression, if it is **true** the **if** expression results in the value of the second subexpression and if it is **false** the **if** expression results in the value of the third subexpression.[6] The **==** operator compares objects by their location in the heap. The || and ! operators are the boolean operations disjunction and negation, respectively. Expressions of the form { *e* ; *e* } are used for sequencing. The first expression is executed for its effect and the result of the entire expression is the result of the second expression. Finally, *str* stands for the string literals.

The expressions following **@pre** and **@post** in a method interface declaration are the pre- and post-conditions for that method, respectively. The method's argument variables are bound in both the expressions and the name of the method is bound to the result of calling the method, but only in the post-condition expression.

In the code fragments presented in this paper, we use several shorthands. We omit the **extends** and **implements** clauses when nothing would appear after them. We write sequencing expressions such as { e_1 ; e_2 ; e_3 ; . . . } to stand for { e_1 ; {e_2 ; {e_3 ; . . . }}} and sometimes add extra {} to indicate grouping. For field declarations, we write *t* fd_1, fd_2 to stand for *t* fd_1 ; *t* fd_2.

The type checker translates syntax (a) to syntax (b). It inserts additional information (underlined in the figure) to be used by the evaluator. In particular, field update and field reference are annotated with the class containing the field, and calls to **super** are annotated with the class.

The contract compiler produces syntax (c) and the evaluator accepts it. The **@pre** and **@post** conditions are removed from interfaces, and inserted into wrapper classes. Syntax (c) also adds the **blame** construct to the language, which is used to signal contract violations. This construct is only available to the programmer indirectly via the compilation process, to preserve the integrity of blame assignment (assuming correct synthesis of blame strings for **semanticCast**).

4.2 Relations and Predicates

A valid program satisfies a number of simple predicates and relations; these are described in figures 7 and 8. 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 (. . .) indicate a repeated pattern or contin-

[6] The **if** in our calculus matches *e* ? *e* : *e* expressions in Java, rather than Java's **if** statements.

$\prec^{\mathsf{c}}_P$ Class is declared as an immediate subclass

$$c \prec^{\mathsf{c}}_P c' \Leftrightarrow \textbf{class}\ c\ \textbf{extends}\ c' \cdots \{ \cdots \}\ \text{is in } P$$

$\leq^{\mathsf{c}}_P$ Class is a subclass

$$\leq^{\mathsf{c}}_P \equiv \text{the transitive, reflexive closure of } \prec^{\mathsf{c}}_P$$

$\in^{\mathsf{c}}_P$ Method is declared in class

$$\langle md, (t_1 \ldots t_n \longrightarrow t), (var_1 \ldots var_n), e\rangle \in^{\mathsf{c}}_P c$$
$$\Leftrightarrow \textbf{class}\ c \cdots \{ \cdots t\ md\ (t_1\ var_1 \ldots t_n\ var_n)\ \{e\} \cdots \} \text{ is in } P$$

$\in^{\mathsf{c}}_P$ Method is contained in a class

$$\langle md, T, V, e\rangle \in^{\mathsf{c}}_P c$$
$$\Leftrightarrow (\langle md, T, V, e\rangle \in^{\mathsf{c}}_P c' \text{ and } c' = \min\{c'' \mid c \leq^{\mathsf{c}}_P c'' \text{ and } \exists e', V' \text{ s.t. } \langle md, T, V', e'\rangle \in^{\mathsf{c}}_P c''\})$$

$\in^{\mathsf{c}}_P$ Field is declared in a class

$$\langle c.fd, t\rangle \in^{\mathsf{c}}_P c \Leftrightarrow \textbf{class}\ c \cdots \{ \cdots t\ fd \cdots \} \text{ is in } P$$

$\in^{\mathsf{c}}_P$ Field is contained in a class

$$\langle c'.fd, t\rangle \in^{\mathsf{c}}_P c$$
$$\Leftrightarrow \langle c'.fd, t\rangle \in^{\mathsf{c}}_P c' \text{ and } c' = \min\{c'' \mid c \leq^{\mathsf{c}}_P c'' \text{ and } \exists t' \text{ s.t. } \langle c''.fd, t'\rangle \in^{\mathsf{c}}_P c''\}$$

$\prec_P$ Interface is declared as an immediate subinterface

$$i \prec_P i' \Leftrightarrow \textbf{interface}\ i\ \textbf{extends} \cdots i' \cdots \{ \cdots \}\ \text{is in } P$$

$\leq_P$ Interface is a subinterface

$$\leq_P \equiv \text{the transitive, reflexive closure of } \prec_P$$

$\in^{\mathsf{i}}_P$ Method is declared in an interface

$$\langle md, (t_1 \ldots t_n \longrightarrow t), (var_1 \ldots var_n), e_b, e_a\rangle \in^{\mathsf{i}}_P i$$
$$\Leftrightarrow \textbf{interface}\ i \cdots \{ \cdots t\ md(t_1\ var_1, \ldots t_n\ var_n)\ \textbf{@pre}\ \{\ e_b\ \}\ \textbf{@post}\ \{\ e_a\ \} \cdots \}\ \text{is in } P$$

$\in^{\mathsf{i}}_P$ Method is contained in an interface

$$\langle md, T, V, e_b, e_a\rangle \in^{\mathsf{i}}_P i \Leftrightarrow \exists i' \text{ s.t. } i \leq_P i' \text{ and } \langle md, T, V, e_b, e_a\rangle \in^{\mathsf{i}}_P i'$$

$\in_P$ Field or Method is in a type (method/interface)

$$\langle md, T\rangle \in_P i \Leftrightarrow \exists V, e_b, e_a \text{ s.t.} \langle md, T, V, e_b, e_a\rangle \in^{\mathsf{i}}_P i$$

$\in_P$ Field or Method is in a type (field/type)

$$\langle c.fd, t\rangle \in_P c \Leftrightarrow \langle c.fd, t\rangle \in^{\mathsf{c}}_P c$$

$\lhd^{\mathsf{c}}_P$ Class declares implementation of an interface

$$c \lhd^{\mathsf{c}}_P i \Leftrightarrow \textbf{class}\ c \cdots \textbf{implements} \cdots i \cdots \{ \cdots \}\ \text{is in } P$$

$\ll^{\mathsf{c}}_P$ Class implements an interface

$$c \ll^{\mathsf{c}}_P i \Leftrightarrow \exists c', i' \text{ s.t. } c \leq^{\mathsf{c}}_P c' \text{ and } i' \leq_P i \text{ and } c' \lhd^{\mathsf{c}}_P i'$$

$\circledS_P$ Structural subtyping for interfaces

$$i \circledS_P i' \Leftrightarrow$$
$$\forall \langle md, T, V, e_b, e_a\rangle \in^{\mathsf{i}}_P i', \exists \langle md, T', V', e'_b, e'_a\rangle \in^{\mathsf{i}}_P i, \text{ such that } T \circledS\!\!\circledS_P T'$$

$\circledS_P$ Structural subtyping for other types

$$\textbf{String} \circledS_P \textbf{String} \qquad \textbf{boolean} \circledS_P \textbf{boolean}$$

$\circledS\!\!\circledS_P$ Structural subtyping for method type specifications

$$t_1 \ldots t_n \longrightarrow t \circledS\!\!\circledS_P t'_1 \ldots t'_n \longrightarrow t' \Leftrightarrow t'_1 \circledS_P t_1, ..., t'_n \circledS_P t_n, t \circledS_P t'$$

Fig. 7. Relations on enriched Java programs

ued sequence, while centered ellipses ($\cdots$) indicate arbitrary missing program text (not spanning a class or interface definition).

Figure 7 is separated into four groups: relations for classes, relations for interfaces, relations that relate classes and interfaces, and finally the structural subtyping relations. As an example relation, the ClassesOnce(P) predicate states that each class name is defined at most once in the program P. The relation $\prec^{\mathsf{c}}_P$ associates each class name in P to the class it extends, and the (overloaded) $\in^{\mathsf{c}}_P$ relations capture the field and method declarations of the classes in 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) predicate holds and the WellFoundedClasses(P) predicate holds. In this case, the classes declared in P form a tree that has **Object** at its root.

ClassesOnce(P) Each class name is declared only once
class $c \cdots$ **class** $c' \cdots$ is in $P \implies c \neq c'$

FieldOncePerClass(P) Field names in each class declaration are unique
class $\cdots \{ \cdots fd \cdots fd' \cdots \}$ is in $P \implies fd \neq fd'$

MethodOncePerClass(P) Method names in each class declaration are unique
class $\cdots \{ \cdots md (\cdots) \{ \cdots \} \cdots md' (\cdots) \{ \cdots \} \cdots \}$ is in $P \implies md \neq md'$

InterfacesOnce(P) Each interface name is declared only once
interface $i \cdots$ **interface** $i' \cdots$ is in $P \implies i \neq i'$

MethodArgsDistinct(P) Each method argment name is unique
$md\ (t_1\ var_1 \ldots t_n\ var_n)\ \{ \cdots \}$ is in $P \implies var_1, \ldots var_n$, and *this* are distinct

CompleteClasses(P) Classes that are extended are defined
$\mathrm{rng}(\prec_P^{\mathsf{c}}) \subseteq \mathrm{dom}(\prec_P^{\mathsf{c}}) \cup \{\textbf{Object}\}$

WellFoundedClasses(P) Class hierarchy is an order
$\leq_P^{\mathsf{c}}$ is antisymmetric

ClassMethodsOK(P) Method overriding preserves the type
$(\langle md, T, V, e\rangle \in_P^{\mathsf{c}} c$ and $\langle md, T', V', e'\rangle \in_P^{\mathsf{c}} c') \implies (T = T'$ or $c \not\leq_P^{\mathsf{c}} c')$

CompleteInterfaces(P) Extended/implemented interfaces are defined
$\mathrm{rng}(\prec_P) \cup \mathrm{rng}(\prec_P^{\mathsf{c}}) \subseteq \mathrm{dom}(\prec_P) \cup \{\textbf{Empty}\}$

WellFoundedInterfaces(P) Interface hierarchy is an order
$\leq_P$ is antisymmetric

InterfaceMethodsOK(P) Interface inheritance or redeclaration of methods is consistent
$\langle md, T, V, e_b, e_a\rangle \in_P^{\mathsf{i}} i$ and $\langle md, T', V', e'_b, e'_a\rangle \in_P^{\mathsf{i}} i'$
$\implies (T = T'$ or $\forall i''(i'' \not\leq_P i$ or $i'' \not\leq_P i'))$

ClassesImplementAll(P) Classes supply methods to implement interfaces
$c \prec_P^{\mathsf{c}} i \implies (\forall md, T\ \langle md, T, V, e_b, e_a\rangle \in_P^{\mathsf{i}} i \implies \exists e, V'$ s.t. $\langle md, T, V', e\rangle \in_P^{\mathsf{c}} c)$

Fig. 8. Predicates on enriched Java programs

If the program describes a tree of classes, we can associate 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 $\in_P^{\mathsf{c}}$ relations. The $\in_P^{\mathsf{c}}$ 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 subinterface declaration relation $\prec_P$ induces a subinterface relation $\leq_P$. Unlike classes, a single interface can have multiple proper superinterfaces, so the subinterface order forms a DAG instead of a tree. The set of methods of an interface, as described by $\in_P^{\mathsf{i}}$, 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 $\prec_P^{\mathsf{c}}$ relation.

The structural subtyping predicate $\circledcirc_P$ relates types in a structural manner. The base types **boolean** and **String** are only related to themselves. Two interface types are related if one has a subset of the methods of the other and the corresponding method arguments and result are related. Note that the relation is contra-variant for method arguments and co-variant for method results.

The type system uses $\circledcirc_P$ to ensure that **semanticCast** expressions are well-formed.

4.3 Types

Type elaboration is defined by the following judgments:

$\vdash_{\mathsf{p}} P \Rightarrow P' : t$	P elaborates to P' with type t
$P \vdash_{\mathsf{d}} defn \Rightarrow defn'$	$defn$ elaborates to $defn'$
$P, c \vdash_{\mathsf{m}} mth \Rightarrow mth'$	mth in c elaborates to mth'
$P, i \vdash_{\mathsf{i}} imth \Rightarrow imth'$	$imth$ in i elaborates to $imth'$
$P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t$	e elaborates to e' with type t in Γ
$P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t$	e elaborates to e' with type t in Γ, using subsumption
$P \vdash_{\mathsf{t}} t$	t is a well-formed type in P

Type elaboration for complete programs ensures that the properties described in the previous section hold for the complete program and ensures that each subexpression in the program is properly typed. Type checking for classes and interface definitions merely ensures that each expression mentioned in each method is properly typed and that the types written in the method specifications are well-formed and that the method specifications match up with the bodies of the methods.

For each form of expression, the intended use dictates the types of its constituents. For example, the arguments to || and ! must be booleans. Similarly, the type of the first subexpression of **if** must be a boolean and the types of the two branches must match. There are four places where subsumption is allowed: at field assignment, method invocations (for the arguments), super calls, and, of course, **view** expressions. These correspond to the places where implicit or explicit casts occur. These rules are the same as in our prior work [16] and many are omitted here.

The typing rule for interface methods ensures that pre- and post-conditions use appropriate variables and have type boolean. The typing rule for **semanticCast** ensures that the last two arguments are both strings and that the first argument is an object. Further, the static type of the object must have the same structure as the type in the second argument, as determined by the $\Subset_P$ relation.

4.4 Contract Compilation

The contract compiler eliminates **semanticCast** expressions from the program and inserts **blame** expressions. The **blame** expression is a primitive mechanism that, when evaluated, aborts the program and assigns blame to a specific **semanticCast** for a contract violation.

The contract compiler, $C[\![\cdot]\!]$, is defined by the following judgments:

$C[\![P]\!] = P'$ if $\vdash_{\mathsf{p}} P \rightharpoonup P'$	
$\vdash_{\mathsf{p}} P \rightharpoonup P'$	P elaborates to P'
$\vdash_{\mathsf{d}} defn, defs \rightharpoonup defn', defs'$	$defn$ elaborates to $defn'$ extending $defs$ to $defs'$
$\vdash_{\mathsf{b}} body, defs \rightharpoonup body', defs'$	$body$ elaborates to $body'$ extending $defs$ to $defs'$
$\vdash_{\mathsf{e}} e, defs \rightharpoonup e', defs'$	e elaborates to e' extending $defs$ to $defs'$

The $\vdash_{\mathsf{p}}$ judgment rewrites a complete program from the second syntax in figure 6 to the third syntax. The other three judgments rewrite definitions, bodies, and expressions,

$\vdash_{\mathsf{p}}$

$$\frac{\textsc{ClassesOnce}(P)\ \textsc{InterfacesOnce}(P)\ \textsc{MethodOncePerClass}(P)\ \textsc{FieldOncePerClass}(P)\ \textsc{CompleteClasses}(P)\ \textsc{WellFoundedClasses}(P)\ \textsc{CompleteInterfaces}(P)\ \textsc{WellFoundedInterfaces}(P)\ \textsc{InterfaceMethodsOK}(P)\ \textsc{MethodArgsDistinct}(P)\ \textsc{ClassesImplementAll}(P) \quad P \vdash_{\mathsf{d}} defn_j \Rightarrow defn'_j \text{ for } j \in [1,n] \quad P, [\,] \vdash_{\mathsf{e}} e \Rightarrow e' : t}{\vdash_{\mathsf{p}} defn_1 \ldots defn_n\ e \Rightarrow defn'_1 \ldots defn'_n\ e' : t} \text{ where } P = defn_1 \ldots defn_n\ e$$

$\vdash_{\mathsf{d}}$

$$\frac{P \vdash_{\mathsf{t}} t_j \text{ for } j \in [1,n] \qquad P, c \vdash_{\mathsf{m}} mth_k \Rightarrow mth'_k \text{ for } k \in [1,p]}{P \vdash_{\mathsf{d}} \textbf{class}\ c\ \cdots\ \{\ t_1\ fd_1\ \ldots\ t_n\ fd_n\ mth_1\ \ldots\ mth_p\ \} \Rightarrow \textbf{class}\ c\ \cdots\ \{\ t_1\ fd_1\ \ldots\ t_n\ fd_n\ mth'_1\ \ldots\ mth'_p\ \}}$$

$$\frac{P \vdash_{\mathsf{i}} imth_j \Rightarrow imth_j \text{ for } j \in [1,p]}{P, i \vdash_{\mathsf{d}} \textbf{interface}\ i\ \cdots\ \{\ imth_1\ \ldots\ imth_p\ \} \Rightarrow \textbf{interface}\ i\ \cdots\ \{\ imth_1\ \ldots\ imth_p\ \}}$$

$\vdash_{\mathsf{m}}$

$$\frac{P \vdash_{\mathsf{t}} t \qquad P \vdash_{\mathsf{t}} t_j \text{ for } j \in [1,n] \qquad P, [this : t_o, var_1 : t_1, \ldots var_n : t_n] \vdash_{\mathsf{s}} e \Rightarrow e' : t}{P, t_o \vdash_{\mathsf{m}} t\ md\ (t_1\ var_1 \ldots t_n\ var_n)\ \{\ e\ \} \Rightarrow t\ md\ (t_1\ var_1 \ldots t_n\ var_n)\ \{\ e'\ \}}$$

$$\frac{P \vdash_{\mathsf{t}} t \qquad P \vdash_{\mathsf{t}} t_j \text{ for } j \in [1,n]}{P, t_o \vdash_{\mathsf{m}} t\ md\ (t_1\ var_1 \ldots t_n\ var_n)\ \{\ \textbf{abstract}\ \} \Rightarrow t\ md\ (t_1\ var_1 \ldots t_n\ var_n)\ \{\ \textbf{abstract}\ \}}$$

$\vdash_{\mathsf{i}}$

$$\frac{\begin{array}{ll} P \vdash_{\mathsf{t}} t & P, [this : i, var_1 : t_1, \ldots var_n : t_n] \vdash_{\mathsf{e}} e_b \Rightarrow e'_b : \textbf{boolean} \\ P \vdash_{\mathsf{t}} t_j \text{ for } j \in [1,n] & P, [this : i, md : t, var_1 : t_1, \ldots var_n : t_n] \vdash_{\mathsf{e}} e_a \Rightarrow e'_a : \textbf{boolean} \end{array}}{\begin{array}{ll} P, i \vdash_{\mathsf{i}} t\ md\ (t_1\ arg_1\ \ldots\ t_n\ arg_n) & \Rightarrow t\ md\ (t_1\ arg_1\ \ldots\ t_n\ arg_n) \\ \quad \textbf{@pre}\ \{\ e_b\ \} & \quad \textbf{@pre}\ \{\ e'_b\ \} \\ \quad \textbf{@post}\ \{\ e_a\ \} & \quad \textbf{@post}\ \{\ e'_a\ \} \end{array}}$$

$\vdash_{\mathsf{e}}$

$$\frac{P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t}{P, \Gamma \vdash_{\mathsf{e}} \textbf{view}\ t\ e \Rightarrow e' : t} \qquad \frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad t \in \mathrm{dom}(\prec_P) \cup \{\textbf{Empty}\}}{P, \Gamma \vdash_{\mathsf{e}} \textbf{view}\ t\ e \Rightarrow \textbf{view}\ t\ e' : t}$$

$$\frac{P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t}{P, \Gamma \vdash_{\mathsf{e}} e\ \textbf{instanceof}\ t \Rightarrow \{e'\ ;\ \textbf{true}\} : \textbf{boolean}} \qquad \frac{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad t \in \mathrm{dom}(\prec_P) \cup \{\textbf{Empty}\}}{P, \Gamma \vdash_{\mathsf{e}} e\ \textbf{instanceof}\ t \Rightarrow e'\ \textbf{instanceof}\ t : \textbf{boolean}}$$

$$\frac{P, \Gamma \vdash_{\mathsf{e}} pos \Rightarrow pos' : \textbf{String} \qquad P, \Gamma \vdash_{\mathsf{e}} neg \Rightarrow neg' : \textbf{String} \qquad P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t' \qquad t' \circledS_P t}{P, \Gamma \vdash_{\mathsf{e}} \textbf{semanticCast}(e, t, pos, neg) \Rightarrow \textbf{semanticCast}(e' : t', t, pos', neg') : t}$$

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

$$\frac{P, \Gamma \vdash_{\mathsf{s}} e \Rightarrow e' : t' \qquad t' \leq_P t}{P, \Gamma \vdash_{\mathsf{e}} e \Rightarrow e' : t} \qquad \frac{t \in \mathrm{dom}(\prec_P) \cup \{\textbf{Empty}, \textbf{boolean}, \textbf{String}\}}{P \vdash_{\mathsf{t}} t}$$

Fig. 9. Context-sensitive checks and type elaboration rules

respectively. Each accepts a term and a set of definitions and produces the rewritten term and a new set of definitions.

The rules for the judgements are given in figures **??** and **??**. The rule for $\vdash_{\mathsf{p}}$ rewrites the definitions and expressions in the program, threading the sets of definitions through the rewriting of the subterms. Its result is the rewritten definitions and expressions, combined with the final set of threaded definitions. With the exception of the rule for **semanticCast**, all of the other rules produces the same term they accept, carrying forward the definitions sets from their subexpressions.

The **semanticCast** rule for booleans and strings merely removes the semantic cast. For interfaces, however, it adds the elaborated definition for the class *Cast_i′_i* to the

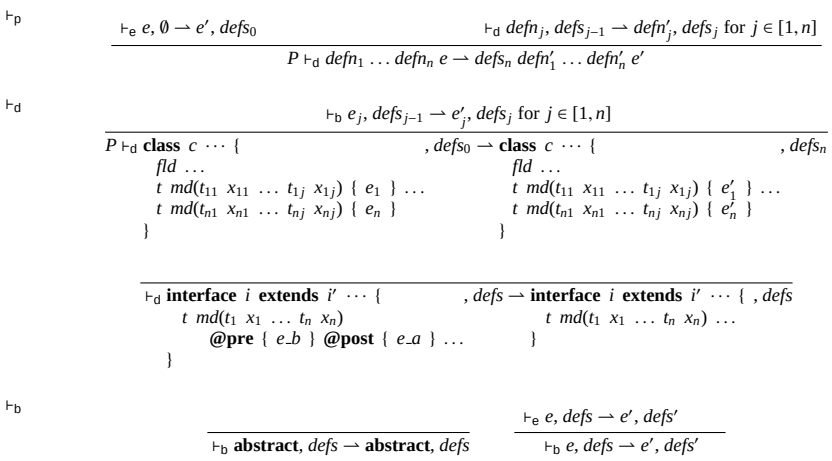

Fig. 10. Contract Elaboration, part 1

set of definitions it produces (without duplication), and replaces the semantic cast with code that creates and initializes an instance of *Cast_i′_i*.

Figure **??** shows the full definition of the *Cast_i′_i* classes, where the interfaces *i′* and *i* match the interface schemas shown. The class contains all of the methods of *i*, plus three instance variables. Two instance variables are strings representing the classes that are to be blamed for values flowing in to and out of the object, respectively. The other instance variable holds the unwrapped object. Whenever a method is called through the wrapper object, the following tasks are performed:

- The pre-condition contract is checked and *inBlame* is blamed if it fails.
- All of the arguments are wrapped, according to their types.
- The method of the unwrapped object is invoked with the newly wrapped arguments and the result is stored in a variable with the same name as the method.
- The post-condition contract is checked and *outBlame* is blamed if it fails.
- The result of the unwrapped call is wrapped according to the result type of the method and the new wrapper object is the result of the method.

Note that the wrapper classes contain **semanticCast** expressions. Thus, compiling these expressions generates new classes. This means that an implementation of the contract compiler must not generate new classes for each occurrence of **semanticCast** it encounters, or it would not terminate. Instead, it is must only generate one *Cast_i′_i* class for each unique pair of interfaces, *i′* and *i*.

4.5 Operational Semantics

The operational semantics is defined as a contextual rewriting system on pairs of expressions and stores [16, 45]. Each evaluation rule has this shape:

$\vdash_e$

$$\frac{}{\vdash_e \textbf{new}\ c, defs \rightharpoonup \textbf{new}\ c, defs} \qquad \frac{}{\vdash_e var, defs \rightharpoonup var, defs} \qquad \frac{}{\vdash_e \textbf{null}, defs \rightharpoonup \textbf{null}, defs}$$

$$\frac{\vdash_e e, defs \rightharpoonup e', defs'}{\vdash_e e : c\ .fd, defs \rightharpoonup e' : c\ .fd, defs'} \qquad \frac{\vdash_e e, defs \rightharpoonup e', defs' \qquad \vdash_e e_v, defs' \rightharpoonup e'_v, defs''}{\vdash_e e : c\ .fd = e_v, defs \rightharpoonup e' : c\ .fd = e'_v, defs''}$$

$$\frac{\vdash_e e, defs \rightharpoonup e', defs_0 \qquad \vdash_e e_j, defs_{j-1} \rightharpoonup e'_j, defs_j \text{ for } j \in [1,n]}{\vdash_e e : c\ .md\ (e_1 \ldots e_n), defs \rightharpoonup e' : c\ .md\ (e'_1 \ldots e'_n), defs_n}$$

$$\frac{\vdash_e e_j, defs_{j-1} \rightharpoonup e'_j, defs_j \text{ for } j \in [1,n]}{\vdash_e \textbf{super}{\equiv}this{:}c.md(e_1 \ldots e_n), t \rightharpoonup defs_0, \textbf{super}{\equiv}this{:}c.md(e'_1 \ldots e'_n)\ defs_n}$$

$$\frac{\vdash_e e, defs \rightharpoonup e', defs'}{\vdash_e \textbf{view}\ t\ e, defs \rightharpoonup \textbf{view}\ t\ e', defs'} \qquad \frac{\vdash_e e, defs \rightharpoonup e', defs'}{\vdash_e e\ \textbf{instanceof}\ t, defs \rightharpoonup e'\ \textbf{instanceof}\ t, defs'}$$

$$\frac{\vdash_e e_1, defs \rightharpoonup e'_1, defs' \qquad \vdash_e e_2, defs' \rightharpoonup e'_2, defs''}{\vdash_e e_1 == e_2, defs \rightharpoonup e'_1 == e'_2, defs''}$$

$$\frac{\vdash_e e, defs \rightharpoonup e', defs_0 \qquad \vdash_e e_j, defs_{j-1} \rightharpoonup e'_j, defs_j \text{ for } j \in [1,n]}{\vdash_e \textbf{let}\ \{\ var_1 = e_1 \ldots var_n = e_n\ \}\ \textbf{in}\ e, defs \rightharpoonup \textbf{let}\ \{\ var_1 = e'_1 \ldots var_n = e'_n\ \}\ \textbf{in}\ e', defs_n}$$

$$\frac{}{\vdash_e \textbf{true}, defs \rightharpoonup \textbf{true}, defs} \qquad \frac{}{\vdash_e \textbf{false}, defs \rightharpoonup \textbf{false}, defs} \qquad \frac{}{\vdash_e str, defs \rightharpoonup str, defs}$$

$$\frac{\vdash_e e_1, defs \rightharpoonup e'_1, defs' \qquad \vdash_e e_2, defs' \rightharpoonup e'_2, defs'' \qquad \vdash_e e_3, defs'' \rightharpoonup e'_3, defs'''}{\vdash_e \textbf{if}\ (e_1)\ e_2\ \textbf{else}\ e_3, defs \rightharpoonup \textbf{if}\ (e'_1)\ e'_2\ \textbf{else}\ e'_3, defs'''}$$

$$\frac{\vdash_e e_1, defs \rightharpoonup e'_1, defs' \qquad \vdash_e e_2, defs' \rightharpoonup e'_2, defs''}{\vdash_e \{\ e_1\ ;\ e_2\ \}, defs \rightharpoonup \{\ e'_1\ ;\ e'_2\ \}, defs''}$$

$$\frac{\vdash_e e_1, defs \rightharpoonup e'_1, defs'}{\vdash_e !\ e, defs \rightharpoonup !\ e', defs'} \qquad \frac{\vdash_e e_1, defs \rightharpoonup e'_1, defs' \qquad \vdash_e e_2, defs' \rightharpoonup e'_2, defs''}{\vdash_e e_1 \parallel e_2, defs \rightharpoonup e'_1 \parallel e'_2, defs''}$$

$$\frac{\vdash_e e_1, defs \rightharpoonup e'_1, defs' \qquad \vdash_e e_2, defs' \rightharpoonup e'_2, defs'' \qquad \vdash_e e_3, defs'' \rightharpoonup e'_3, defs''' \qquad \text{t is } \textbf{boolean} \text{ or } \textbf{String} \text{ and } x \text{ not free in } e'_2 \text{ or } e'_3}{\vdash_e \textbf{semanticCast}(e_1 : t, t, e_2, e_3), defs \rightharpoonup \textbf{let}\ \{x = e'_1\}\ \textbf{in}\ \{\ e'_2\ ;\ e'_3\ ;\ x\ \}, defs'''}$$

$$\frac{\vdash_e e_1, defs \rightharpoonup e'_1, defs' \quad \vdash_e e_2, defs' \rightharpoonup e'_2, defs'' \quad \vdash_e e_3, defs'' \rightharpoonup e'_3, defs''' \quad \vdash_d \textbf{class}\ Cast\_i'\_i \ldots, defs''' \rightharpoonup defn, defs''''}{\vdash_e \textbf{semanticCast}(e_1 : i', i, e_2, e_3), defs \rightharpoonup \textbf{let}\ \{\ x = e'_1\ \ i = e'_2\ \ o = e'_3 \qquad , defn \uplus defs''''}$$
$$w = \textbf{new}\ Cast\_i'\_i()\ \}$$
$$\textbf{in}\ \{\ w : Cast\_i'\_i\ .unwrapped = x;$$
$$w : Cast\_i'\_i\ .inBlame = i;$$
$$w : Cast\_i'\_i\ .outBlame = o;$$
$$w\}$$

Fig. 11. Contract Elaboration, part 2

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

A store ($\mathcal{S}$) is a mapping from *objects* (a set of identifiers distinct from the program variables) 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

```
class Cast_i'_i implements i {
  String inBlame, outBlame;
  i' unwrapped;
  t md(t_1 x_1 ... t_n x_n) {
    if (e_b) {
      let {md = unwrapped.md(semanticCast(x_1 : t_1, t'_1, outBlame, inBlame) ...
                             semanticCast(x_n : t_n, t'_n, outBlame, inBlame))}
      in { if (e_a) {
             semanticCast(md : t', t, inBlame, outBlame);
           } else { blame(outBlame); }}
    } else { blame(inBlame); }}
  ...
}

where i and i' match:

interface i extends ... { t md(t_1 x_1 ... t_n x_n) @pre { e_b } @post { e_a } ...}
interface i' extends ... { t' md(t'_1 x_1 ... t'_n x_n) ... }
```

Fig. 12. Compiler-generated Wrapper Classes

```
e = ... | object
v = object | null
  | true | false
  | str

E = [ ] | E : c .fd | E : c .fd = e | v : c .fd = E
  | E .md(e ...) | v.md(v ... E e ...)
  | super≡v:c.md(v ... E e ...)
  | view t E | let var = v ... var = E var = e ... in e
  | if ( E ) e else e | E instanceof i | E == e | v == E
  | E || e | ! E | { E ; e } | blame(E)
```

Fig. 13. Expressions, values, and contexts

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 6. Figure 12 shows the contexts where reductions can occur.

The complete evaluation rules are in Figure 13. 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.

The **blame** expressions terminate the program by throwing away the context and reducing to a configuration containing just an error, just like mis-use of **null** or a bad cast.

4.6 Soundness

A naïve soundness theorem for a contract compiler would guarantee that the additional code that the contract compiler adds to the program changes the behavior of the program only by signaling contract errors. Put positively, if no contract violations are signaled,

$P \vdash \langle \mathsf{E}[object : t.md(v_1, \ldots, v_n)], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[object/this, v_1/var_1, \ldots v_n/var_n]], \mathcal{S}\rangle$ [**call**]
where $\mathcal{S}(object) = \langle c, \mathcal{F}\rangle$ and $\langle md, (t_1 \ldots t_n \longrightarrow t), (var_1 \ldots var_n), e\rangle \in^{c}_{P} c$

$P \vdash \langle \mathsf{E}[\mathbf{super}{\equiv}object{:}c.md(v_1, \ldots, v_n)], \mathcal{S}\rangle$ [**super**]
$\hookrightarrow \langle \mathsf{E}[e[object/this, v_1/var_1, \ldots v_n/var_n]], \mathcal{S}\rangle$
where $\langle md, (t_1 \ldots t_n \longrightarrow t), (var_1 \ldots var_n), e\rangle \in^{c}_{P} c$

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

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

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

$P \vdash \langle \mathsf{E}[\mathbf{view}\ t'\ object], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[object], \mathcal{S}\rangle$ [**cast**]
where $\mathcal{S}(object) = \langle c, \mathcal{F}\rangle$ and $c \ll^{c}_{P} t'$

$P \vdash \langle \mathsf{E}[object\ \mathbf{instanceof}\ t'], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathbf{true}], \mathcal{S}\rangle$ [**ipass**]
where $\mathcal{S}(object) = \langle c, \mathcal{F}\rangle$ and $c \ll^{c}_{P} t'$

$P \vdash \langle \mathsf{E}[object\ \mathbf{instanceof}\ t'], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathbf{false}], \mathcal{S}\rangle$ [**ifail**]
where $\mathcal{S}(object) = \langle c, \mathcal{F}\rangle$ and $c \not\ll^{c}_{P} t'$

$P \vdash \langle \mathsf{E}[bool\ \mathbf{==}\ bool'], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathbf{if}\ (bool)\ bool'\ \mathbf{else}\ !\ bool'], \mathcal{S}\rangle$ [**==b**]
$P \vdash \langle \mathsf{E}[\mathbf{let}\ var_1 = v_1 \ldots var_n = v_n\ \mathbf{in}\ e], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e[v_1/var_1 \ldots v_n/var_n]], \mathcal{S}\rangle$ [**let**]
$P \vdash \langle \mathsf{E}[\mathbf{if}\ (\ \mathbf{true}\ )\ e_1\ \mathbf{else}\ e_2], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e_1], \mathcal{S}\rangle$ [**iftrue**]
$P \vdash \langle \mathsf{E}[\mathbf{if}\ (\ \mathbf{false}\ )\ e_1\ \mathbf{else}\ e_2], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e_2], \mathcal{S}\rangle$ [**iffalse**]
$P \vdash \langle \mathsf{E}[\mathbf{true} \parallel e], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathbf{true}], \mathcal{S}\rangle$ [**ortrue**]
$P \vdash \langle \mathsf{E}[\mathbf{false} \parallel e], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e], \mathcal{S}\rangle$ [**orfalse**]
$P \vdash \langle \mathsf{E}[!\ \mathbf{true}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathbf{false}], \mathcal{S}\rangle$ [**nottrue**]
$P \vdash \langle \mathsf{E}[!\ \mathbf{false}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[\mathbf{true}], \mathcal{S}\rangle$ [**notfalse**]
$P \vdash \langle \mathsf{E}[\{\ v\ ;\ e\ \}], \mathcal{S}\rangle \hookrightarrow \langle \mathsf{E}[e], \mathcal{S}\rangle$ [**seq**]

$P \vdash \langle \mathsf{E}[\mathbf{blame}(s)], \mathcal{S}\rangle \hookrightarrow \langle$error: s violated contract, $\mathcal{S}\rangle$ [**blame**]
$P \vdash \langle \mathsf{E}[\mathbf{view}\ t'\ object], \mathcal{S}\rangle \hookrightarrow \langle$error: bad cast, $\mathcal{S}\rangle$ [**xcast**]
where $\mathcal{S}(object) = \langle c, \mathcal{F}\rangle$ and $c \not\ll^{c}_{P} t'$

$P \vdash \langle \mathsf{E}[\mathbf{view}\ t'\ \mathbf{null}], \mathcal{S}\rangle \hookrightarrow \langle$error: bad cast, $\mathcal{S}\rangle$ [**ncast**]
$P \vdash \langle \mathsf{E}[\mathbf{null} : c.fd], \mathcal{S}\rangle \hookrightarrow \langle$error: dereferenced null, $\mathcal{S}\rangle$ [**nget**]
$P \vdash \langle \mathsf{E}[\mathbf{null} : c.fd = v], \mathcal{S}\rangle \hookrightarrow \langle$error: dereferenced null, $\mathcal{S}\rangle$ [**nset**]
$P \vdash \langle \mathsf{E}[\mathbf{null}.md(v_1, \ldots, v_n)], \mathcal{S}\rangle \hookrightarrow \langle$error: dereferenced null, $\mathcal{S}\rangle$ [**ncall**]
$P \vdash \langle \mathsf{E}[\mathbf{null}\ \mathbf{instanceof}\ i], \mathcal{S}\rangle \hookrightarrow \langle$error: dereferenced null, $\mathcal{S}\rangle$ [**nisa**]
$P \vdash \langle \mathsf{E}[\mathbf{null}\ \mathbf{==}\ v], \mathcal{S}\rangle \hookrightarrow \langle$error: dereferenced null, $\mathcal{S}\rangle$ [**n==l**]
$P \vdash \langle \mathsf{E}[v\ \mathbf{==}\ \mathbf{null}], \mathcal{S}\rangle \hookrightarrow \langle$error: dereferenced null, $\mathcal{S}\rangle$ [**n==r**]

Fig. 14. Operational semantics

the original program with the contracts erased and the contract compiled program must behave identically.

Unfortunately, that theorem is too strong, for two reasons. First, the contract expressions themselves may change the behavior of the program (via side-effects or non-termination). So, we only consider a class of contracts that do not affect the behavior of the program, captured by this definition:

Definition 1 (Effect Free). *An expression e is said to be* effect free *if, for any store $\mathcal{S}$ and program P (that bind the free variables in e),*

$$P \vdash \langle e, \mathcal{S}\rangle \hookrightarrow^{*} \langle v, \mathcal{S}\rangle$$

for some value, v.

Second, because semantic casts allow structural subtyping, simply removing them would yield a type-incorrect Java program. Accordingly, we must replace semantic casts with wrapper classes that perform the type adaptation, but do not check contracts.

Definition 2 (Erasure).
The function $\mathcal{E}[\![\cdot]\!]$ behaves just like the contract compiler, except for **semanticCast** *expressions, where it instantiates adapter classes instead of the wrapper classes, according to this rule:*

$$\frac{\vdash_{\mathsf{e}} e_1, defs \rightharpoonup e_1', defs' \qquad \vdash_{\mathsf{e}} e_2, defs' \rightharpoonup e_2', defs'' \qquad \vdash_{\mathsf{e}} e_3, defs'' \rightharpoonup e_3', defs''' \qquad \vdash_{\mathsf{d}} \textbf{class } Adapt\_i'\_i \ldots, defs''' \rightharpoonup defn, defs''''}{\vdash_{\mathsf{e}} \textbf{semanticCast}(e_1 : i', i, e_2, e_3), defs \rightharpoonup \textbf{let } \{ x = e_1' \; i = e_2' \; o = e_3' \; w = \textbf{new } Adapt\_i'\_i() \} \textbf{ in } \{ w : Adapt\_i'\_i .unwrapped = x; w\}, defn \uplus defs''''}$$

```
class Adapt_i'_i implements i {
  i' unwrapped;
  t md(t_1 x_1 ... t_n x_n) {
    let {md = unwrapped.md(semanticCast(x_1 : t_1, t'_1, "", "") ...
                           semanticCast(x_n : t_n, t'_n, "", ""))}
    in { semanticCast(md : t', t, "", ""); }} ...
}
```

Since the adapter classes never signal contract violations, the third and fourth arguments to **semanticCast** are ignored by the replacement $\vdash_{\mathsf{e}}$ rule. Similarly, the adapter class can safely use bogus string arguments to the nested **semanticCast** expressions.

In order to meaningfully state a soundness result, we must also ensure that the contract compiler and the erasure procedure both preserve the typing structure of programs.

Lemma 1. *For any program P = defn ... e that type checks:*

$$\vdash_{\mathsf{p}} P \Rightarrow P' : t$$

the erased and compiled versions of P must also type check and have the same type:

$$\vdash_{\mathsf{p}} C[\![P]\!] \Rightarrow P'' : t \qquad\qquad \vdash_{\mathsf{p}} \mathcal{E}_p[\![P]\!] \Rightarrow P''' : t$$

Proof (sketch). Both the erasure and contract compilation leave all expressions intact, except for **semanticCast** expressions. Inspection of the compiler and erasure definitions shows that they produce well-typed expressions and the typing rule for **semanticCast** gives the same types that erasure and the compiler give. □

With that background, we can now formulate a soundness theorem for our contract checker.

Theorem 1. *Let P = defn ... e be a program such where all the contract expressions are effect free. Let $C[\![P]\!] = P_c$ = $defn_c$ ... e_c and let $\mathcal{E}[\![P]\!] = P_e$ = $defn_e$ ... e_e. One of the following situation occurs:*

- $P_c \vdash \langle e_c, \emptyset\rangle \hookrightarrow^* \langle \textbf{blame}(s), \mathcal{S}\rangle$
- $P_c \vdash \langle e_c, \emptyset\rangle \hookrightarrow^* \langle \mathsf{error}\text{: } str, \mathcal{S}\rangle$ *and* $P_e \vdash \langle e_e, \emptyset\rangle \hookrightarrow^* \langle \mathsf{error}\text{: } str, \mathcal{S}\rangle$

- $P_c \vdash \langle e_c, \emptyset\rangle \hookrightarrow^* \langle v, \mathcal{S}\rangle$ *and* $P_e \vdash \langle e_e, \emptyset\rangle \hookrightarrow^* \langle v', \mathcal{S}\rangle$ *where either* $v = v'$ *and* v *is not an object, or both* v *and* v' *are objects.*
- *For each* e' *such that* $P_c \vdash \langle e_c, \emptyset\rangle \hookrightarrow^* \langle e', \mathcal{S}\rangle$ *there exists an* e'' *such that* $P_c \vdash \langle e', \mathcal{S}\rangle \hookrightarrow \langle e'', \mathcal{S}\rangle$ *and for each* e' *such that* $P_e \vdash \langle e_e, \emptyset\rangle \hookrightarrow^* \langle e', \mathcal{S}\rangle$ *there exists an* e'' *such that* $P_e \vdash \langle e', \mathcal{S}\rangle \hookrightarrow \langle e'', \mathcal{S}\rangle$

The formal statement of the theorem is divided into four cases, based on the behavior of the contract compiled program. If the contract compiled program produces a blame error, the theorem does not say anything about the erased program. If, however, the contract compiled program produces a value, a safety error, or does not terminate, the original program and the contract-free program must behave in the same manner. Note that if the contract compiled program produces an object, the erased program only has to produce an object, not necessarily the same object. If the contract compiled program results in a boolean or a string, the erased program must produce the same boolean or string.

Proof (sketch). The proof operates by relating the reduction sequences of the original program to the compiled program. Clearly, if there are no **semanticCast** expressions in the program, the new program contains extra definitions, but they are unused. Accordingly, the two programs reduce in lockstep until the first **semanticCast** expression. At that point, the erased program produces the adapter object and the compiled program produces a wrapper object. If the wrapper object ever signals a contract violation, we know that the theorem holds. If it does not, we can see from the definition of the wrapper objects that they behave identically to the adapter object when a method is invoked, because the contract expressions are effect free, by assumption. □

5 Implementation Status

We have implemented this contract checker as part of DrScheme [9], a 200,000 line MzScheme [15] program. Although the class system of MzScheme is not statically typed, its design is otherwise similar to the design of Java's class system. That is, the safety properties that Java's type system guarantees, *e.g.*, each method call has a receiver, are all also guaranteed, but the enforcement is entirely dynamic and implemented in terms of runtime checks. Accordingly, MzScheme benefits from contracts just as we have described in this paper.

Although the contract system described in this paper (with extensions to support all of the details of MzScheme's class system) has been implemented and is part of the current pre-release of DrScheme, the contract checker for the object sub-language is not widely used yet. In addition to contracts on objects, however, DrScheme also has a contract checker for higher-order functions. As far as contracts are concerned, a function is essentially an object with a single method.

We have studied the performance impact of contracts in DrScheme. An instrumented version of DrScheme counts the number of functions and function contracts. After starting up DrScheme and opening a few windows and Help Desk, there are 27962 reachable functions and only 507 wrappers, *i.e.*, slightly less than 2% of the functions are wrapped. With a different accounting annotation, DrScheme can also determine the

number of function calls and calls to contract functions; for basically the same start-up action, the program performs 2,142,000 calls to user-defined functions, of which 1425 are calls to contract wrappers. That is, 0.06% of the calls to user-defined functions are calls to wrappers. Unfortunately, it is difficult to generalize these experiments, because it is a major undertaking to write contracts for a large system of components. Still, the experiment with DrScheme suggests that well-chosen contracts have little performance impact on a large program. Based on our experience, the number of contracts in a component rarely exceeds 10% of the number of functions proper. Yet, even if our system were to contain that many wrapper functions, our experiments suggest that only .3% of the function calls would be calls to wrapper functions. In short, we don't expect semantic casts to affect the overall system performance in a noticeable manner.

6 Related Work

Contracts have a long history. In 1972, Parnas [36] first suggested equipping module interfaces with contracts. His objective was to state the purpose of his proposed units of reuse in a formal manner. Soon thereafter, contracts appeared in a range of programming languages, including ADA [30], Euclid [25], and Turing [20]. In the 1980s, the designers of OO programming languages began to incorporate contracts [32] and OO researchers investigated the meaning of contracts in an OO context [1, 29]. By now, a fair number of OO languages support contracts either directly or as add-on packages [2, 5, 6, 8, 17, 21–24, 31, 32, 37, 38].

Over the past three years, we have investigated the theory and practice of contract and contract checking. Thus far, our theoretical research has focused on the soundness of contract checking in class hierarchies and in the presence of higher-order functions [10–12]. Our practical efforts have led to the implementation of a contract checking system for our Scheme class and mixin system. Experience with contracts in our DrScheme product suggested the proposal for a semantic cast in this paper.

Beyond contract checking systems, researchers are also investigating notations, theorem provers, and other tools for supporting contracts. For example, JML [26] is a notation for stating and reasoning about contracts. We use it in this paper to notate our contracts. JML is also used for many tools that go well beyond mere dynamically checked behavioral contracts. For example, ESC/Java [7, 14] is a theorem-prover that can validate theorems about JML contracts. In addition to ESC/Java, EML [40, 41] and Larch [19] are systems that statically verify contracts. Although our work focuses on dynamic validation, we believe that a static validation of such contracts is feasible and useful. Specifically, we hope that existing extended static checking efforts, like those of Flanagan et al [14], can be modified to account for semantic casts.

ML's module and signature language [34, 27] has been a different source of inspiration. It has long supported *signature ascription*, the ability to refine an existing ML structure's interface with the rest of the program. Our work can be seen as an extension of signature ascription to dynamically checked contracts.

The implementation of **semanticCast** with wrapper objects is suggestive of creating a denotational retract [42]. Although this intuition does not carry over directly, it

suggested certain directions for our investigations. Also, our wrapper classes are reminiscent of the coercions that Henglein considers in his work [**?**].

7 Future Work

So far, we have only explored semantic up casts, that is, casts from a subtype to a supertype. It may, however, be useful to permit some form of semantic down casts. In particular, if an object were first cast to a super type, it is often useful to be able to cast it back to its original type. For example, when using container classes, the type of the container is some supertype of all of the objects that may ever be stored in the container. Accordingly, when retrieving objects from the container, it may be sensible to down cast them to a type with more information.

Clearly, one simple way to support semantic down casts is to remove layers of wrapping from the downcast object. Unfortunately, this would circumvent the contract checking. In general, components depend on contracts being enforced on the objects that play a role in their communication with other components. That is, if a downcast were to remove the contract checking code from some object, one component's contract violation may not be detected, leading to another component being blamed for a subsequent contract violation, or perhaps even erroneous output.

We have not yet found a consistent, simple extension to a nominally typed language design that manages to both support semantic down casts and preserves contract checking.

8 Conclusion

This paper introduces semantic casts, a modest extension to languages with nominal subtype systems. A semantic cast enables programmers to reuse classes and interfaces that match structurally but not nominally. Our calculus validates that doing so is compatible with conventional languages such as C++ [43], C# [33], Eiffel [32], and Java [18]. In the future, we plan to continue our investigations of how contracts can overcome the limitations of conventional type systems in a safe manner.

Acknowledgments

Thanks to Adam Wick for instrumenting his garbage collector so we could collect wrapper and function counts. Thanks also to the anonymous ECOOP reviewers for their comments.

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., 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.
3. Bruce, K. B., A. Fiech and L. Petersen. Subtyping is not a good "match" for object-oriented languages. In *Proceedings of European Conference on Object-Oriented Programming*, pages 104–127, 1997.
4. Bruce, K. B., A. Schuett and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. *Lecture Notes in Computer Science*, 952:27–51, 1995.
5. Carrillo-Castellon, M., J. Garcia-Molina, E. Pimentel and I. Repiso. Design by contract in smalltalk. *Journal of Object-Oriented Programming*, 7(9):23–28, 1996.
6. Cheon, Y. A runtime assertion checker for the Java Modelling Language. Technical Report 03-09, Iowa State University Computer Science Department, April 2003.
7. Detlefs, D. L., K. Rustan, M. Leino, G. Nelson and J. B. Saxe. Extended static checking. Technical Report 158, Compaq SRC Research Report, 1998.
8. Duncan, A. and U. Hölzle. Adding contracts to Java with handshake. Technical Report TRCS98-32, The University of California at Santa Barbara, December 1998.
9. Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002. A preliminary version of this paper appeared in PLILP 1997, LNCS volume 1292, pages 369–388.
10. Findler, R. B. and M. Felleisen. Contract soundness for object-oriented languages. In *Object-Oriented Programming, Systems, Languages, and Applications*, 2001.
11. Findler, R. B. and M. Felleisen. Contracts for higher-order functions. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, 2002.
12. Findler, R. B., M. Latendresse and M. Felleisen. Behavioral contracts and behavioral subtyping. In *Proceedings of ACM Conference Foundations of Software Engineering*, 2001.
13. Fisher, K. and J. H. Reppy. The design of a class mechanism for Moby. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1999.
14. Flanagan, C., K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe and R. Stata. Extended static checking for Java. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, 2002.
15. Flatt, M. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997. http://www.mzscheme.org/.
16. Flatt, M., S. Krishnamurthi and M. Felleisen. Classes and mixins. In *Proceedings of the ACM Conference Principles of Programming Languages*, pages 171–183, Janurary 1998.
17. Gomes, B., D. Stoutamire, B. Vaysman and H. Klawitter. *A Language Manual for Sather 1.1*, August 1996.
18. Gosling, J., B. Joy and J. Guy Steele. *The Java(tm) Language Specification*. Addison-Wesley, 1996.
19. Guttag, J. V. and J. J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.
20. Henglein, F. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, 1994.
21. Holt, R. C. and J. R. Cordy. The Turing programming language. In *Communications of the ACM*, volume 31, pages 1310–1423, December 1988.
22. 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.
23. Kizub, M. Kiev language specification. http://www.forestro.com/kiev/, 1998.
24. Kölling, M. and J. Rosenberg. *Blue: Language Specification, version 0.94*, 1997.

25. Kramer, R. iContract: The Java design by contract tool. In *Technology of Object-Oriented Languages and Systems*, 1998.
26. Lampson, B. W., J. J. Horning, R. L. London, J. G. Mitchell and G. J. Popek. Report on the programming language Euclid. *ACM Sigplan Notices*, 12(2), Feburary 1977.
27. 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.
28. Leroy, X. Applicative functors and fully transparent higher-order modules. In *Proceedings of the ACM Conference Principles of Programming Languages*, pages 142–153. ACM Press, 1995.
29. Leroy, X. *The Objective Caml system, Documentation and User's guide*, 1997.
30. 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.
31. 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.
32. Man Machine Systems. Design by contract for Java using JMSAssert. http://www.mmsindia.com/DBCForJava.html, 2000.
33. Meyer, B. *Eiffel: The Language*. Prentice Hall, 1992.
34. Microsoft Corporation. *Microsoft C# Language Specifications*. Microsoft Press, 2001.
35. Milner, R., M. Tofte and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
36. Object Management Group. The object management architecture guide, 1997. http://www.omg.org/.
37. Parnas, D. L. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.
38. Plösch, R. Design by contract for Python. In *IEEE Proceedings of the Joint Asia Pacific Software Engineering Conference*, 1997. http://citeseer.nj.nec.com/257710.html.
39. Plösch, R. and J. Pichler. Contracts: From analysis to C++ implementation. In *Technology of Object-Oriented Languages and Systems*, pages 248–257, 1999.
40. Rémy, D. and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proceedings of the ACM Conference Principles of Programming Languages*, pages 40–53, January 1997.
41. Rosenblum, D. S. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, Janurary 1995.
42. Sannella, D. Formal program development in Extended ML for the working programmer. In *Proc. 3rd BCS/FACS Workshop on Refinement; Springer Workshops in Computing*, pages 99–130, 1991.
43. Sannella, D. and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997. http://www.dcs.ed.ac.uk/home/dts/eml/.
44. Scott, D. S. Data types as lattices. *Society of Industrial and Applied Mathematics (SIAM) Journal of Computing*, 5(3):522–586, 1976.
45. Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, 1997.
46. Szyperski, C. *Component Software*. Addison-Wesley, second edition, 1998.
47. 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.

This research is partially supported by the National Science Foundation.