LOOJ: Weaving LOOM into Java*

Kim B. Bruce[1] and J. Nathan Foster[2]

[1] Department of Computer Science
Williams College
Williamstown, MA USA 01267
email: `kim@cs.williams.edu`

[2] Department of Computer & Information Science
University of Pennsylvania
Philadelphia, PA USA 19104
email: `jnfoster@cis.upenn.edu`

Abstract. LOOJ is an extension of Java obtained by adding bounded parametric polymorphism and new type expressions `ThisClass` and `ThisType`, which are similar to *MyType* in LOOM. Through examples we demonstrate the utility of this language even over very expressive extensions such as GJ. The LOOJ compiler generates standard JVML code and supports `instanceof` and casts for all types including type variables and the other new type expressions. The core of the LOOJ type system is sound, as demonstrated by a soundness proof for an extension of Featherweight GJ. This paper also highlights difficulties that arise from the use of both classes and interfaces as types in Java.

Keywords: object-oriented language design, static type systems, *MyType*, `ThisClass`, `ThisType`, formal semantics.

1 Introduction

Modularity and reusability are two core concepts in object-oriented programming systems. Unfortunately, too often the type systems of object-oriented languages make it difficult to write modular, reusable code because they lack expressive power. Java's type system is often criticized because it does not allow explicit type abstractions and applications, making it impossible to write generic data structures within the static type system. The many proposals for adding parametric polymorphism to Java [AFM97,OW97,CGLS98,MBL97], including GJ [BOSW98], aim to overcome this deficiency.[3] However another limitation – the lack of a precise type for `this` – makes it difficult to write many useful programs. In this paper, we extend Java to a language LOOJ that includes two constructs analogous to *MyType*, as well as bounded polymorphism and support for weak reflection constructs.

* This work was partially supported by the National Science Foundation under grants CCR-0306486 and CCR-9988210.

[3] Parametric polymorphism based on GJ will be included in Java 1.5.

Contributions

We demonstrate that *MyType* can be added cleanly to an extension of Java alongside bounded parametric polymorphism, producing a very expressive type system. First, we introduce `ThisClass`, which closely captures the class type of `this`. We also introduce *exact types*, which are needed to ensure that uses of `ThisClass` are type safe. Second, we give a brief description of the LOOJ compiler that implements the language in a homogeneous, type-passing compilation style. Our implementation supports the use of `instanceof` and type casts for all type expressions including type variables. Third, we describe an extension to Featherweight GJ [IPW01], which models a core subset of LOOJ, and its type soundness proof. Fourth, we discuss some problems that result when `ThisClass` is used in combination with interfaces and introduce `ThisType` to stand for the interface type of `this`.

In the remainder of this section, we show some examples where the expressiveness of GJ falls short.

1.1 Motivation

We will present a formal definition of a binary method later; for now, consider a binary method to be a method that has a parameter whose type is intended to be the same as the type of the object it is used with. In what follows, we will write all examples using a Java-like syntax, though we write method types using → notation instead of the syntax of methods in Java interfaces.

As our first example of a binary method, consider the `setNext` method in a `Node` which represents singly-linked list elements with type:

```
setNext: Node → void
```

As the parameter has type `Node`, `setNext` meets our definition of binary method.

However, suppose we try to define a subclass `DoubleNode` of `Node` to represent doubly-linked nodes. Then the inherited version of `setNext` still takes a parameter with type `Node`, even though its parameter should have type `DoubleNode` if it is intended to remain a binary method in the subclass. The poor support for binary methods in Java's static type system allows programs to link singly-linked nodes as the next field of a doubly-linked node. Clearly this behavior is not what the programmer wants. She would be better off if she could specify that `setNext` should only ever be applied to an argument whose type is the *same* as the object that it is invoked on.

A similar problem arises in programs that create an object of the same type as the currently executing object. Java's standard library includes an interface `Cloneable` that is implemented by classes which may be cloned.[4] The `clone` method has the type:

```
clone: () → Object
```

[4] That classes which implement `Cloneable` can be safely cloned is enforced at runtime; the interface itself is empty.

which is not precise enough to describe the high-level behavior that programmers want of `clone`. The method's return type, `Object`, gives no information about the type of the object that it returns. As a result, if a programmer clones an object with type `C`, she must cast the result of the `clone` method to `C` before using it. Part of this problem could be fixed by allowing covariant changes in the return types of methods.[5] However, adding this flexibility does not solve a more fundamental problem. Even if class `C`'s `clone` method is defined with return type `C`, there is nothing to guarantee that each subclass will override the method with a definition whose return type is its own type. If the programmer neglects to override the method for a particular subclass, `D`, then she can invoke its `clone` method and get an object whose type is `C`, not `D`. Again, the type system is unable to express that a particular type, here the return type, is the same as the type of the object that it is used with.

2 Introducing `ThisClass` and exact types

The two examples above share a common trait: the lack of a name for the type of `this` in Java's type system makes it very difficult to express important properties about programs. In the example involving `setNext` and linked-list nodes, the desired property is that singly-linked nodes are only ever linked in to a list containing other singly-linked nodes. In the `clone` example, it is that `clone` always returns an object with the same type as the object that it is invoked on. Adding a primitive type for `this`, similar to the *MyType* construct in LOOM [BFP97,Bru02], gives the flexibility needed to solve both of these problems and others.

The `ThisClass` type in LOOJ is directly inspired by *MyType* in LOOM (LOOJ also has a type `ThisType`, described in Section 5). However, because the type system of LOOM differs from Java's in two key ways, adding a construct like *MyType* to Java requires some care. First, LOOM has structural type relations whereas Java's type relations are nominal. Second, in LOOM, classes and object types are distinct. In Java, the two are conflated and a class is commonly used both as the generator of an object and as its type. As a result of these differences, finding a clean way of adapting LOOM's *MyType* to Java so that the extended type system is a natural fit with existing programming styles and idioms is not immediately obvious.

We begin by introducing the type `ThisClass`, which stands for the class type of `this`. If a class `C` defines a method `m`, then when `m` is used with an object whose runtime type is `C`, any occurrences of `ThisClass` in the method's type signature may be safely assumed to be `C`. The power of `ThisClass` becomes apparent when `m` is inherited in a subclass, `D`. In this case, occurrences of `ThisClass` in the same method type are assumed to have all the features of `D`, *not* just those of `C`. This

[5] Java's type system does not allow changes to method signatures in subclasses; however, a covariant change in return type would be safe and is supported by the JVM; it is allowed in GJ and will be allowed in Java 1.5 [BCK^{+}01].

behavior – that `ThisClass` corresponds to the type of the runtime object that it is actually used with – is the hallmark behavior of *MyType*.

In order to ensure that methods type checked in superclasses are type safe when used in subclasses, the type checker analyzes all fields and methods of a class in a type context which includes the assumption that `ThisClass` extends the class type that is currently being checked. When it checks an individual method invocation, it substitutes the static class type of the object that the method is being invoked on for all occurrences of `ThisClass` in the method's type signature. As an example, suppose that `d` is an expression of type `D`, and that `binMeth` is a method defined in `D` with static type

```
binMeth: ThisClass → void
```

When the type checker analyzes the method invocation `d.binMeth(o)`, it treats the type signature of `binMeth` just like a method with type0

```
d.binMeth: D → void
```

That is, it checks that the static type of `o` extends `D`. Formal rules for type checking `ThisClass` in a core calculus are given in the appendix.

Henceforth, we use the term *binary method* to refer to methods where `ThisClass` appears in a negative (value consuming) position in its type signature.[6] In particular, every method where `ThisClass` appears in the type of one of its parameters is a binary method. In order to ensure that a binary method invocation is safe, we need to be able to determine the precise class type that the receiver will have at run-time. To see why this property is needed, consider the following declarations:

```
class C { public void binMeth(ThisClass tc) { ... } }
class D extends C {
   public void newMeth() {...};
   public void binMeth(ThisClass tc) { ... tc.newMeth() ... }
}

void problem(C c1, C c2) {
  c1.binMeth(c2);
}

problem(new D(), new C());   // error!
```

Note that method `newMeth()` does not occur in `C`, but is used in the redefinition of `binMeth` in `D`.

If the line labelled "`error`" were legal in the LOOJ type system, then the evaluation of `binMeth` in the body of `problem` would send the message `newMeth` to an object of type `C`, which has no such method.

[6] Later we extend this definition to include methods with parameter types involving `ThisType`.

To avoid this problem, we introduce *exact types* to LOOJ. Normally a Java expression with type `C` might at runtime contain an object with type `C` or any extension of `C`. For an exact type, denoted with the `@` symbol, we rule out this second case; an expression whose static type is `@C` always refers to an object with runtime type `C`. One can think of exact types as ruling uses of subsumption for specific expressions.

With exact types, we can design a sound type system by requiring that the receiver of each binary method invocation must have an exact type. This restriction eliminates problems such as the one above where a binary method call on a receiver whose type is not known exactly can lead to a hole in the static type system. We cannot write the `problem` method, because `c1` is used as the call site for a binary method but `c1`'s type is not exact. If we change the type of the first parameter of `problem` to `@C`, then the method body type checks, but the type checker will rule out the invocation of `problem` with a first parameter whose type is `D`.

We can summarize the typing properties of programs that use `ThisClass` as follows. When type checking methods in a class `C` we assume that:

- `this` has type `@ThisClass`,
- `ThisClass` extends `C`.[7]

When type-checking message sends,

- The receiver of a binary method invocation must have an exact type.
- The type of a binary method invocation is obtained from the method's type signature by substituting the receiver's static type for each occurrences of `ThisClass`.
- The type of a (non-binary) method invocation where `ThisClass` appears in the method's type signature, but the receiver is not exact, is safe if `ThisClass` appears only in positive (value producing) positions. We calculate the type of such a method call from the method's type signature by substituting the receiver's static type first for each occurrence of `@ThisClass` and then for each remaining occurrence of `ThisClass`.

Exact types have uses beyond type checking binary methods in LOOJ. They are also useful for writing *homogeneous* data structures. In general, exact type expressions can be used to more precisely describe the shape of program computations. They do so, of course, at the cost of flexibility and extensibility at each use because exact types prohibit uses of subtype polymorphism for particular expressions. In certain programs, this tradeoff between increased precision and flexibility of reuse may lean towards precision and away from flexibility – exact types provide a primitive for specifying exact typing constraints.

[7] To correctly model the semantics of the `private` access modifier, some care is required. Note that `this` and other expressions of type `ThisClass` have access to private instance variables and methods of `C`, while expressions whose type is just known to be an extension of `C` do not.

Java programs also use interfaces as types and we have also introduced a construct, `ThisType`, that stands for the interface of `this`. As there are some subtle negative interactions between interfaces and `ThisClass`, we postpone all discussion of interfaces and `ThisType` for now. We will address these issues in detail in Section 5. Meanwhile, we restrict our attention to Java programs that do not use interfaces.

3 Motivating Examples Revisited

Our two motivating examples which caused problems in GJ's type system are easily solved in LOOJ using `ThisClass`. For example, we can write `Node`'s `setNext` method as:

```
setNext: @ThisClass → void
```

In this program, the static type system ensures that `setNext` is only ever passed arguments whose type is the same as the object that the method is invoked on. In particular, we cannot link a `Node` into a list consisting of elements of type `DoubleNode`, as desired.

The example involving cloning also has a simple solution with `ThisClass`. Instead of declaring the return type of the `clone` method as `Object`, we can write it as `@ThisClass`. This declaration ensures that `clone` always returns an object whose type is the same as the object that it was invoked upon, even when it is used in a subclass. Note that with this type declaration, if `clone` is called on a receiver with type `@C` then the result will have static type `@C`, while if it is called on a receiver with type `C`, then the result will has static type `C`. This last point illustrates that `ThisClass` may safely appear positively in a method's type signature even when the method is invoked with an unexact receiver.

However, it is not immediately obvious what we can write in the body of `clone` in order to manufacture an object whose type is `@ThisClass`. We need an expression whose type is `C` when the method is used with a `C`, and `D` when used with a subclass `D`. We cannot simply call `C`'s constructor because an object with type `@C` is not a subtype of `@ThisClass`. However, using a Factory pattern [GHJV96], we can produce a new object with the correct type. Suppose that the interface of a factory is:

```
interface Factory<T> {
  @T create();
}
```

Then if `C` contains an instance variable `thisClassFactory` with type `Factory<ThisClass>`, we can write

```
thisClassFactory.create();
```

as the body of `clone`. We can then add a new parameter to the constructors of `C` to initialize the factory object:

```
public C(..., Factory<ThisClass> cfact) {
    ...
    thisClassFactory = cfact;
}
```

When we create an instance of `C`, we can initialize `cfact` by passing the constructor an argument with type `Factory<C>`.[8]

We can use factory classes to code up many useful expressions. For example, the expression `new X()` where `X` is a type variable, can be simulated by sending a `create` message to an object with type `Factory<X>`. A previous version of LOOJ supported special syntax for distinguished `This` constructors (an idea originally due to Bill Joy). These were just factories that returned an object with type `@ThisClass`. We have found that in practice, constructing the factories and passing them to standard constructors explicitly is not prohibitively burdensome, and so the current version of LOOJ does not support special `This` constructor syntax. It remains simple to write expressions that have the same behavior as `This` constructors in LOOJ using the Factory pattern with `ThisClass`.

4 Type safety in LOOJ and an implementation

LOOJ is more than a mere design; we have implemented a compiler and developed a proof of static type safety for the language.

In previous work we provided both translational semantics [Bru02] and high-level operational semantics [BFSvG03,BFP97] for object-oriented languages with a *MyType* construct. For LOOJ we have proved soundness for a subset of the language using the techniques introduced in Featherweight Java [IPW01]. Due to space constraints we do not include the full proof in this paper. Instead, we give the syntax, type-checking rules, and evaluation rules in the appendix, along with a proof sketch of type soundness. An extended version of this paper that contains the full type system and soundness proof is available as a companion technical report [BF04].

The LOOJ compiler supports the additional type `ThisClass` and exact types (as well as the new type `ThisType` to be discussed later). Moreover, unlike current versions of GJ,[9] our compiler supports `instanceof` expressions and type casts for all types, including those involving type variables and `ThisClass`.

All legal Java programs are legal programs of LOOJ and produce the same results. GJ and LOOJ differ on some minor points. In LOOJ, all instantiations of type variables must be written explicitly, as opposed to, GJ where instantiations of type abstractions private to methods are inferred. Our compiler does not

[8] The static type of the "receiver" of a `new` expression is just the exact type of the class type being created; hence, when we type check `new C(...)`, it is safe to substitute `C` for `ThisClass` in the constructor's type signature. Calls of `super` constructors in subclasses also use the subclass as the receiver type.

[9] Future implementations of GJ, along the lines of NextGen [CGLS98], will address these current limitations.

perform type reconstruction for invocations of polymorphic methods, as we prefer explicit instantiations of type variables. More substantially, the compiler differs in the translation style used to compile programs to Java bytecodes. Rather than using pure erasure, the LOOJ compiler uses a homogeneous translation originally proposed by Burstein [Bur98] that is based on erasure, but that also annotates classes with private instance variables representing the runtime values of type variables.

In our type-passing implementation, the private variables are initialized in constructors by passing representations of class types obtained from Java's reflection utilities. With this information, each object can determine the values of its type variables at runtime. This is useful for performing lightweight introspective operations such as dynamically-checked type casts and `instanceof` type tests. We believe that providing these operations for all types including generics, `ThisClass`, and exact types is a closer fit with Java's existing semantics. As an example, consider this simple class:

```
class C<T> {
  T myT;
  public C() { }
}
```

A compiler that uses pure erasure to implement generics translates this fragment to bytecodes equivalent to:

```
class C {
  Object myT;
  public C() { }
}
```

whereas the LOOJ compiler translates it to:

```
class C {
  Object myT;
  private PolyClass T$$class;
  public C(PolyClass T$$class) {
    this.T$$class = T$$class;
  }
  public boolean instanceOfC(PolyClass otherT$$class) {
    return T$$class.equals(otherT$$class);
  }
}
```

Here, `PolyClass` objects explicitly hold information about polymorphic types in the same way that `Class` objects in Java hold information about Java types. The LOOJ compiler uses these objects to implement operations that require runtime type information, such as `instanceof` expressions, and type casts. The translations of these expressions have the correct runtime semantics for most types.[10]

For example, the expression

```
obj instanceof C<T>
```

is not allowed in GJ, because its erased version would be true whenever `obj` is a `C`. It is translated in LOOJ to

```
((obj != null)
  && (obj instanceof C)
  && (((C)obj).instanceOfC(T$$class)))
```

It is easy to see that this expression is only true if `obj` is an instance of `C<T>`. Similar translations are used to implement these and many similar expressions:

```
obj instanceof @ThisClass
((C<T>)obj)...
```

Unfortunately, the same technique cannot be extended to array types as there is no uniform location where we can transparently hold the runtime representations of instantiations of type variables for array types. We provide a wrapper class for arrays with polymorphic element types that can be used to simulate the correct semantics for these operations if needed. Additionally, because the runtime type representations of type variables are available, expressions such as

```
new T[n];
new ThisClass[n];
```

can be translated to expressions using Java's reflection facilities to create an array of the correct base type at runtime. GJ statically translates the first expression (with a warning that the translation is unchecked) to a `new` array expression of `T`'s bound.

More details about the LOOJ compiler are available in honors theses by Burstein [Bur98] and Foster [Fos01].

5 Interfaces, `ThisClass`, and `ThisType`

Thus far, we have described how LOOM's *MyType* can be mapped onto class types in Java's type system in `ThisClass`. However, we have carefully avoided mentioning how `ThisClass` interacts with Java interfaces. In this section we discuss some of the problems that result when `ThisClass` is used together with interface types. We conclude that `ThisClass` should not appear in interfaces and introduce the new type expression `ThisType`, to represent the *interface* of `this`. Later we show how `ThisType` can be used to solve problems that arise when programs using the the Visitor pattern are extended.

[10] Runtime operations involving array types are not currently supported, as discussed below.

5.1 `ThisClass` in interfaces

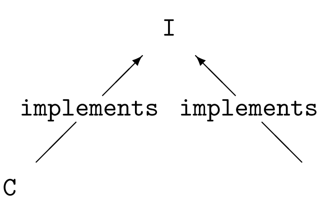
As a first attempt at expressing binary methods in interfaces, we might try to write `ThisClass` directly in an interface. Unfortunately, it is not clear what such a use of `ThisClass` would mean. Consider these source fragments:

```
interface I {
  void binMeth(ThisClass tc);
}
class C implements @I {
  // instance variable declarations
  void binMeth(ThisClass tc) { ... }
}
class D implements @I {
  // instance variable declarations
  public void binMeth(ThisClass tc) { ... }
}

@I i1 = new D();
@I i2 = new C();
i1.binMeth(i2);
```

Notice that the class declarations state that both `C` and `D` implement `@I`. In LOOJ, a class implements an interface exactly if its public methods are *exactly* the methods in the interface. A class declaration may only declare a single exact interface type (this restriction is a consequence of Java's nominal type relations: multiple exact interfaces would be structurally identical, but their names would be distinct and hence, the two would be unrelated in the type system). As in Java, classes may also implement as many non-exact interfaces as are desired. The annotation of the classes with an exact interface informs the type checker that objects from classes `C` and `D` can be used in contexts expecting objects of type `@I`. As a result, the assignments above to `i1` and `i2` are legal.

In the example above, `C` and `D` are unrelated in Java's by name inheritance hierarchy except that they both implement `I`:



The code for the implementations of `binMeth` in `C` and `D` may access their respective (and different) instance variables. Hence, it is not safe to invoke the `binMeth` method of one on an object of the other type. But we cannot determine from its type, `@I`, if `i1` is a `C` or a `D`. We reluctantly conclude that we cannot reliably identify the class type of the receiver of a binary method call at compile time if the static type of the receiver is an interface. As a result, the last line of code above is not legal in the LOOJ type system.[11]

It is often useful, however, to be able to write interfaces that include declarations of binary methods. To facilitate this, we introduce a second version of *MyType*, `ThisType`, which stands for the *public interface type* of the definition where it occurs.

We can rewrite the above example, replacing `ThisClass` with `ThisType`:

```
interface I {
  void binMeth(ThisType tc);
}
class C implements @I {
  void binMeth(ThisType tc) { ... }
}
class D implements @I {
  public void binMeth(ThisType tc) { ... }
}
@I i1, i2;
i1.binMeth(i2);
```

Now the message send in the last line is safe because the parameter of `binMeth` is an interface type, `ThisType`. As such, the method bodies of `binMeth` in `C` and `D` may not access instance variables or non-public methods. They may, however, invoke methods that are included in their exact interface from inside `binMeth`.

As with `ThisClass`, when type checking a method call involving `ThisType` on an object `obj`, each occurrence of `ThisType` in the signature is replaced by the *interface type* of the receiver. If the type of the receiver is a class, then the exact interface associated with that class is used in the substitution.[12] Note that if `I` is extended by an interface `J`, then the meaning of `ThisType` within inherited methods changes, just as the meaning of `ThisClass` changes when it is used in subclasses.

The typing properties of programs provided earlier can be extended to a type system with `ThisType` as follows. When type checking methods in a class `C` with exact interface `I`, we are allowed to assume that:

- `this` has type `@ThisClass`,
- `ThisClass` extends `C`.
- `ThisType` extends `I`,
- `ThisClass` implements `@ThisType`, and

[11] It would be type safe to allow `ThisClass` in parameter positions of methods in interfaces if we also required that the receiver of any invocation of that method must be an exact *class* type. In LOOJ, we use a conceptually simpler rule that forbids the use of `ThisClass` in interfaces and instead will provide `ThisType` as a *MyType* construct for interface types.

[12] In our implementation, the compiler synthesizes the public interface of a class if `ThisType` is used with the class, but an exact interface is not declared by the programmer.

– `C` implements `@I`.

A consequence of the last two items is that a value with type `@C` can be used in a context expecting a value of type `@I` and a value with type `@ThisClass` can be used in a context expecting a value of type `@ThisType`.

5.2 Visitors and ThisType

The addition of `ThisType` provides a natural way to write programs containing both binary methods and interfaces. We illustrate this point by showing an example where using `ThisType` and interfaces solves a challenging problem.

Suppose that we wish to write a statically type-safe, extensible interpreter for integer expressions in a GJ-like language. By extensible, we mean that when we extend the language with some new syntactic forms, the new interpreter can reuse all of the existing code for interpreting the initial language without modification. This example is adapted from [Bru03]; a survey of various solutions to the problem can be found in Torgersen's recent paper [Tor04].

The initial language we interpret is a very simple language of integer constants and negations. We use the Visitor pattern [GHJV96] to implement our interpreter and each visitor returns values of type `int`. The classes `ConstForm` and `NegForm`, representing constants and negations, respectively, both implement the interface

```
interface Form {
   int visit(Visitor v);
}
```

A visitor is an interface that has methods for processing each of the forms in the language, producing an integer value as its result:[13]

```
interface Visitor {
  int constCase(ConstForm cf);
  int negCase(NegForm nf);
}
```

The `visit` method in each formula class sends a message to the corresponding method of the visitor. For example, the code for `visit` method in `NegForm` is

```
public int visit(Visitor v) {
  return v.negCase(this);
}
```

When a `visit` message is sent to a formula, dynamic dispatch results in a message being sent to the appropriate case in the visitor.

The advantage of separating the syntax from the visitors that implement particular high level operations (e.g., evaluation, type checking, pretty printing)

[13] More generally, the `Visitor` interface would be parameterized by the return type.

is that it is very easy to add new operations on syntax trees by adding new visitors. However, it is difficult to extend the syntax with new forms without modifying the original definitions. For example, if we want to add a new form `PlusForm`, representing abstractly the concrete syntax $e_1 + e_2$, we cannot reuse our existing visitor. To see why it is hard, first consider how we might write the new form and extended visitor to process it:

```
class PlusForm implements @Form {
  @Form lhs, rhs;
  public int visit(ExtVisitor ev) {
    ev.plusCase(this);
  }
}
interface ExtVisitor extends Visitor {
  int plusCase(PlusForm pf);
}
```

Unfortunately as written, this code will not pass the GJ type checker because the interface of all syntactic forms, `Form`, requires each class that implements it to define a method named `visit` with type

```
visit: Visitor → int
```

and the `visit` method of `PlusForm` has a different type. But if we write the `visit` method for `PlusForm` with the required type, then we cannot invoke `plusCase` from the visitor on it because the `Visitor` interface does not include it!

A natural next step is to introduce type variables to abstract the type of the visitor that is used to process each syntactic element. Following this approach, we might rewrite the definitions of visitors and syntactic forms like this:

```
interface Visitor {
  ...
  int negCase(NegForm<Visitor> nf);
}
interface Form<V extends Visitor> {
  int visit(V v);
}
class NegForm<V extends Visitor> implements @Form<V> {
  @Form exp;
  public int visit(V v) {
    v.negCase(this);  // error!
  }
}
```

However when we try to type-check this code, we get an error because of GJ's invariant subtyping rule for parameterized class types. The visitor has a method `negCase` with type:

```
negCase: NegForm<Visitor> → int
```

and the occurrence of `this` in the expression

```
v.negCase(this)
```

has type `NegForm<V>`,[14] and `NegForm<V>` is not a subtype of `NegForm<Visitor>`.

Thus introducing a type variable to stand for the type of the visitor does not lead to a natural solution to the extensible interpreter problem.[15]

Though the problem does not obviously contain binary methods, it can be solved by using `ThisType` as the instantiation of the type variable `V` in the definitions of visitors. We can revise the definition of the visitor class to the following:

```
interface Visitor {
  int constCase(ConstForm<ThisType> cf);
  int negCase(NegForm<ThisType> nf);
}
```

Now the `visit` methods for each of the syntactic forms in the language can be written as follows (we give the case for `NegForm<V>` only, the others are similar):

```
class NegForm<V extends Visitor> implements @Form<V> {
  ...
  public int visit(@V v) { return v.negCase(this); }
}
```

The problematic expression from above, `v.negCase(this)`, no longer leads to a type error as witnessed by the following reasoning steps:

- The method `negCase` has type

  ```
  negCase: NegForm<ThisType> → int
  ```

- As `v` has type `@V`,[16] where `V` is a type variable with bound `Visitor`, when we type check the message send to `v`, we replace `ThisType` with `V`:

  ```
  [V/ThisType](NegForm<ThisType> → int) = NegForm<V> → int
  ```

- Recall that the class `NegForm<V>` is type checked under the assumptions that `this` has type `@ThisClass` and `ThisClass` extends `NegForm<V>`. Thus `this` can be used as the parameter to `v.negCase` in the `visit` method, as desired.

Extending the interpreter to handle `PlusForm` is now straightforward:

```
interface ExtVisitor extends Visitor {
  int plusCase(PlusForm<ThisType> pf);
}
```

[14] More accurately, `this` has some type that extends `NegForm<V>`.

[15] A more subtle error arises if `Visitor` takes a type parameter and `Form` uses F-bounded polymorphism. See [Bru03] for more on various attempts at type-checking this example.

[16] We declare `v`'s type exactly because it is used as the call site for a binary method.

The definition of `PlusForm<V>` also follows naturally:

```
class PlusForm<V extends ExtVisitor> implements @Form<V> {
  ...
  public int visit(@V v) { return v.plusCase(this); }
}
```

This class can be type checked using similar derivation steps as described above.

The interpreters are written as classes which implement the appropriate visitor interface. An interpreter for the smaller language looks like this:

```
class Interp implements @Visitor {
  public int constCase(ConstForm<ThisType> cf) {
    return cf.value;
  }
  public int negCase(NegForm<ThisType> nf) {
    return (0 - nf.exp.visit(this));
  }
}
```

`ThisType` is used in the instantiations of the classes representing syntactic forms in the methods so that the extended interpreter can reuse the code for interpreting all of the old forms. This satisfies the extensibility requirement that was set out in the beginning. The extended interpreter is also easy to write:

```
class ExtInterp extends Interp implements @ExtVisitor {
  public int plusCase(PlusForm<ThisType> pf) {
    int lval = pf.lhs.visit(this);
    int rval = pf.rhs.visit(this);
    return lval + rval;
  }
}
```

In fact, the solution to the visitor problem with `ThisType` is not yet ideal, as we would also like to be able to parameterize visitors by their return type in order to write other visitors (e.g., type checkers, pretty printers, etc.). A more detailed discussion of this problem along with a suggested solution using a generalization of *MyType* to mutually recursive types is given in [Bru03].

5.3 Bounded polymorphism and `ThisType`

In GJ, one can declare type variables with bounds that restrict the types that can be used to instantiate them. GJ's rules for bounded type variable instantiation state that if the bound is a class, then any class that extends that class may instantiate its variable. If the bound is an interface, then any interface that extends the type, or any class that implements the interface type may instantiate it. In GJ, a type variable such as `V` in the `Form` class might be instantiated with an interface type or a class type at runtime.

This convenient conflating of the notions of an interface extending another and a class implementing an interface runs into difficulties in the presence of `ThisType`. As a result, in LOOJ, if `I` is an interface and a class or interface declares a type variable `T` extending `I`, then only interfaces can be used to instantiate `T`. For example, with the `Form` interface, we may instantiate `V` with `Visitor` or `ExtVisitor`, but *not* with a class `C`.

The following example illustrates why this restriction is needed:

```
interface Iter {
  @ThisType getNext();
  void setNext(@ThisType newNext);
}
class C implements @Iter { ... }
class D implements @Iter { ... }
class E<X extends Iter> {
  @X x;
  public void setX(@X newX) { this.x = newX; }
  public @X getX() { return x; }
  public @X peekAhead() {
    return x.getNext();
  }
}
```

This code is excerpted from a program that iterates across some values; similar examples come up in many different data structures.

The body of method `peekAhead` is clearly type safe: as `x` has type `@X`, `x.getNext()` also has type `@X` (as always, we replace occurrences of `ThisType` by the interface type of the receiver, `X`). However, this leads to problems if we attempt to instantiate type variable `X` of class `E` with a class type. Consider the following program fragment:

```
void hole(@E<C> e) {
   @C c;
   @D d;
   c.setNext(d);      // (1)
   e.setX(c);         // (2)
   c = e.peekAhead(); // (3)
}
```

The line marked `(1)` type checks because `setNext`, when invoked on an object with type `@C`, has type

```
c.setNext: @I → void
```

and `d` has type `@D`, and hence `@I`. The line marked `(2)` is unproblematic because `setX` has the following type when used with an `@E<C>`:

```
e.setX: @C → void
```

and `c` has type `@C`. The line marked `(3)` also type checks, because

```
e.peekAhead: void → @C
```

and `c` has type `@C`. But note that here, `peekAhead` actually returns an object with type `@D`! Thus, this code allows us to assign a `D` to a `C` – a hole.

To avoid this problem we must modify the rules from GJ for instantiating type variables. Unlike GJ's more flexible rule, which allows type variables that are bounded by interface types to be instantiated with either classes or interfaces, in LOOJ such a type variable may only be instantiated with an interface type.[17]

We believe that the loss in expressiveness at the level of bounded polymorphism is not prohibitively great, while the gains from making the restriction on instantiation, which allows us to use `ThisType` with receivers that are type variables, are significant. In practice, we have found that a natural programming style is to use either `ThisClass` or `ThisType` to describe the type of `this`, but rarely to mix both types in programs. The first style rarely uses interfaces, as is common with many Java programs; the second only uses classes to generate objects and rarely uses them as types, emulating the programming style of a language like LOOM. Instead all classes are declared with the exact interfaces that they define and interfaces are used as types in the program.

6 Summary and Related Work

In this paper we have described an extension to Java, LOOJ, that supports bounded polymorphism, exact types and new type expressions `ThisClass` and `ThisType` to represent, respectively, the class and interface of `this`. The language and type system is an extension of that of GJ except that parameterized methods must be instantiated with a parameter (they are not inferred as in GJ) and, if a type parameter is declared with a bound that is an interface, then it may only be instantiated with interfaces. Unlike GJ, LOOJ supports the use of `instanceof` and type casts with types that involve type variables. Our LOOJ compiler generates standard bytecodes that can be run on any standard JVM. We also include a sketch of the proof of soundness of Featherweight LOOJ, an extension of Featherweight Java that includes `ThisClass` and exact types.

This extension was directly inspired by our earlier work on the languages LOOM [BFP97] and PolyTOIL [BSvG95], which both support a *MyType* construct. (See also [Bru02] for more on *MyType*). Because Java allows both classes and interfaces to be used as types, we added both `ThisClass` and `ThisType` to the language. The use of both classes and interfaces as types added complications to LOOJ that did not arise in the design of LOOM or PolyTOIL. In

[17] There are several additional points in the design space; in particular, we could make a distinction between class and interface types in the bound declaration syntax. For example, we considered introducing the syntax `C<T implements I>`. However, in LOOJ we chose the simplest design and instead require that type variables bounded by an interface type are only ever instantiated with interface types.

particular, occurrences of `ThisClass` in the parameter types of methods in interfaces are difficult to make sense of. Another complication that did not arise in LOOM stems from the useful notational ambiguity of GJ that allows programs to instantiate type variables whose bound is an interface type with a class type. Both of these complications are a result of allowing both classes and interfaces to be used as types. We would prefer that only interfaces be used as types, but we realize that many programmers like the convenience of using classes as types.

There have been many proposals for extensions of Java involving bounded polymorphism. including GJ [BOSW98], Pizza [OW97], NextGen [CGLS98], and PolyJ [MBL97], however none of these has included constructs similar to `ThisClass` or `ThisType`. Our use of instance variables to hold `PolyClass` objects for each type variable in order to support `instanceof` and type casts was originally proposed in Burstein [Bur98], and further refined in Foster [Fos01]. Viroli and Natali [VN00] independently proposed a similar scheme. Their scheme provided optimizations that could be adopted in our implementation to improve efficiency. NextGen, and it more recent offspring, MixGen [ABC03], both support `instanceof` and type casts using a relatively efficient heterogeneous translation that results in a different (though compact) class being generated for each instantiation of a parameterized class.

We have also designed languages with a generalized *MyType* construct for groups of mutually recursive classes. An early version is reported in [BV99], while a more powerful version is sketched in [Bru03], which also includes much more detail on solutions to typing visitors (the so-called "Expression problem").

In related work, Gonzalez [Gon03] has designed and implemented a verifier for the JVM that accepts annotated bytecode generated by a variant of the LOOJ compiler. After verifying the bytecode, all type information is stripped away and standard JVML bytecode is executed. As the type variable information is available to the verifier, we can eliminate many of the casts inserted by the GJ compiler that the type system guarantees will succeed. While we have not yet run careful benchmarks, we expect that this change will result in increased performance over the GJ compiler. An advantage of this approach is that the efficiency of existing high performance JVML JITs can be improved by replacing their existing verifiers with verifiers that are aware of the extended type system.

Acknowledgements We wish to thank the anonymous ECOOP reviewers, Stephen Freund, and Alan Schmitt for very helpful comments on an earlier draft.

References

[ABC03] Eric Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In *Proc. OOPSLA 2003*, pages 96–114, 2003.

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the java language. In *Proc. OOPSLA 1997*, pages 49–65, 1997.

[BCK+01] Gilad Bracha, Norman Cohen, Christian Kemper, Steve Marx, Martin Odersky, Sven-Eric Panitz, David Stoutamire, Kresten Thorup, and Philip Wadler. Adding generics to the Java programming language. `http://jcp.org/aboutJava/communityprocess/review/jsr014/index.html`, 2001.

[BF04] Kim B. Bruce and J. Nathan Foster. LOOJ: Weaving LOOM into Java. Technical Report MS-CIS-03-38, University of Pennsylvania, 2004. `http://www.cis.upenn.edu/~jnfoster/papers/MS-CIS-03-38.ps`.

[BFP97] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *Proc. ECOOP 1997*, pages 104–127. LNCS 1241, Springer-Verlag, 1997.

[BFSvG03] Kim B. Bruce, Adrien Fiech, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. *ACM TOPLAS*, 25(2):225–290, 2003.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proc. OOPSLA 1998*, 1998.

[Bru02] Kim B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, Cambridge, MA, 2002.

[Bru03] Kim B. Bruce. Some challenging typing issues in object-oriented languages. In *Electronic notes in Theoretical Computer Science*, volume 82(8), 2003.

[BSvG95] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language, extended abstract. In *Proc. of ECOOP 1995*, pages 27–51. LNCS 952, Springer-Verlag, 1995.

[Bur98] Jon Burstein. *Rupiah: An extension to Java supporting match-bounded parametric polymorphism, ThisType, and exact typing*. Williams College Senior Honors Thesis, 1998.

[BV99] Kim B. Bruce and Joseph C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Electronic notes in Theoretical Computer Science*, volume 20, 1999.

[CGLS98] Robert Cartwright and Jr. Guy L. Steele. Compatible genericity with runtime types for the Java programming language, 1998.

[Fos01] John N. Foster. *Rupiah: Towards an Expressive Static Type System for Java*. Williams College Senior Honors Thesis, 2001.

[GHJV96] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1996.

[Gon03] Robert Gonzalez. *In the World of Type Checking, Smarter Is Faster*. Williams College Senior Honors Thesis, 2003.

[IPW01] Atushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, May 2001. An earlier version appeared in Proc. OOPSLA 1999.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proc. POPL 1997*, pages 132–145, 1997.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proc. POPL 1997*, pages 146–159, 1997.

[Tor04] Mads Torgersen. The expression problem revisited. In *Proc. of ECOOP 2004*, 2004. To appear.

[VN00] Mirko Viroli and Antonio Natali. Parametric polymorphism in Java: an approach to translation based on reflective features. In *Proc. OOPSLA 2000*, pages 146–165, 2000.

Featherweight LOOJ

The formal language Featherweight Java only models a small subset of Java – notably it does not include imperative features or interfaces – but has a correspondingly simple dynamic semantics and soundness proof. Its main virtue is that it is easy to extend. In their original paper, its designers extend it immediately by modelling the core features of GJ in the Featherweight GJ (FGJ) calculus.

In this section, we describe an extension to FGJ with `ThisClass` and exact types. The extended calculus, Featherweight LOOJ (FLJ), formalizes the core of the LOOJ type system. Much of what follows is similar to and assumes familiarity with FGJ. That FLJ is such a modest extension to FGJ is, we claim, a virtue, and suggests that `ThisClass` would be easy to add to many Java-like languages. To keep the presentation compact, we highlight the important differences while eliding some of the less interesting details that are the same in both calculi. As noted earlier, the full system and soundness proof is available in an accompanying technical report [BF04].

The essential difference between GJ and LOOJ is that the latter includes `ThisClass`. Accordingly, the most substantial differences in the core calculi FGJ and FLJ appear in the parts of the type system that deal with fields and methods whose type involves `ThisClass`. As we have described in earlier sections, uses of `ThisClass` are type checked statically by substituting the type of the receiver of a message send (or field access) for occurrences of `ThisClass`. Hence, in the calculi, the key differences arise in the auxiliary definitions that define the operations for looking up the fields (*fields*), method type and method body (*mtype* and *mbody*) of members of a class. Other major differences have to do with tracking exact types in the type system. In particular we do not wish to allow instantiation of type variables by exact types, or to allow doubly-exact types. Ensuring these well-formedness constraints induces some notational complexity. Most of the rest of the complexity of FLJ is directly inherited from FGJ.

The syntax of FLJ is shown in Figure 1. Following the syntactic conventions of FJ, we abbreviate `extends` with $\triangleleft$, and `return` with $\uparrow$; $\overline{\mathtt{T}}$ is used as shorthand for $\mathtt{T}_1, ...\mathtt{T}_n$, $\overline{\mathtt{T}}\ \overline{\mathtt{f}}$ abbreviates $\mathtt{T}_1\mathtt{f}_1, ...\mathtt{T}_n\mathtt{f}_n$, etc. Term contexts Γ are partial functions from variables to types; similarly, type contexts Δ map type variables and `ThisClass` to their bounds. The class table CT is assumed to be fixed, and maps class names to their definitions.

We adopt the syntactic convention introduced in FJ that `x` ranges over the set of variable names and the special variable `this`. Similarly, `X` ranges over the set of type variables and the special type `ThisClass`. The metavariables `Y` and `Z`, however, only range over type variable names. For example, the declared type variables of a class, $\overline{\mathtt{Z}}$, may not include `ThisClass`. The syntax also enforces the following restriction: the bound of a type variable declaration may not contain `ThisClass`, `ThisType` or an exact type. Otherwise the syntax is a straightforward extension of FGJ.

The auxiliary definitions *fields* and *mtype* define the fields of a class and the type of a method respectively. Note that some of the definitions make use of a

Syntax

Classes	CL	$::= \texttt{class}\ \texttt{C}\langle\overline{\texttt{Z}} \lhd \overline{\texttt{N}}\rangle \lhd \texttt{D}\langle\overline{\texttt{N}}\rangle\ \{\ \overline{\texttt{T}}\ \overline{\texttt{f}};\ \texttt{K}\ \overline{\texttt{M}}\ \}$
Constructors	K	$::= \texttt{C}(\overline{\texttt{S}}\ \overline{\texttt{g}}, \overline{\texttt{T}}\ \overline{\texttt{f}})\ \{\ \texttt{super}(\overline{\texttt{g}});\ \texttt{this}.\overline{\texttt{f}} = \overline{\texttt{f}};\ \}$
Methods	M	$::= \langle\overline{\texttt{Z}} \lhd \overline{\texttt{N}}\rangle\ \texttt{T}\ \texttt{m}(\overline{\texttt{T}}\ \overline{\texttt{x}})\ \{\ \uparrow \texttt{e};\ \}$
Expressions	e	$::= \texttt{x} \mid \texttt{e.f} \mid \texttt{x.m}\langle\overline{\texttt{H}}\rangle(\overline{\texttt{e}}) \mid \texttt{new}\ \texttt{C}\langle\overline{\texttt{H}}\rangle(\overline{\texttt{e}}) \mid (\texttt{T})\texttt{e}$
Types	T	$::= \texttt{H} \mid @\texttt{H}$
Hash Types	H	$::= \texttt{X} \mid \texttt{C}\langle\overline{\texttt{H}}\rangle$
Bound Types	N	$::= \texttt{C}\langle\overline{\texttt{Z}}\rangle \mid \texttt{C}\langle\overline{\texttt{N}}\rangle$

Field Lookup

$$\mathit{fields}(\texttt{Object}, \_) = \emptyset \qquad \text{(F-Obj)}$$

$$\frac{\mathsf{CT}(\texttt{C}) = \texttt{class}\ \texttt{C}\langle\overline{\texttt{Z}} \lhd \overline{\texttt{N}}\rangle \lhd \texttt{D}\langle\overline{\texttt{U}}\rangle\ \{\ \overline{\texttt{S}}\ \overline{\texttt{f}};\ \ldots\ \} \quad \mathit{fields}(\texttt{D}\langle\overline{\texttt{U}}\rangle, @\texttt{ThisClass}) = \overline{\texttt{Y}}\ \overline{\texttt{g}}}{\mathit{fields}(\texttt{C}\langle\overline{\texttt{T}}\rangle, @\texttt{R}) = [\overline{\texttt{T}}/\overline{\texttt{Z}}][\texttt{R}/\texttt{ThisClass}](\overline{\texttt{Y}}\ \overline{\texttt{g}}, \overline{\texttt{S}}\ \overline{\texttt{f}})} \ \text{(F-@Cl)}$$

$$\frac{\begin{array}{c}\texttt{R}\ \text{not exact} \qquad \mathsf{CT}(\texttt{C}) = \texttt{class}\ \texttt{C}\langle\overline{\texttt{Z}} \lhd \overline{\texttt{N}}\rangle \lhd \texttt{D}\langle\overline{\texttt{U}}\rangle\ \{\ \overline{\texttt{S}}\ \overline{\texttt{f}};\ \ldots\ \} \\ \mathit{fields}(\texttt{D}\langle\overline{\texttt{U}}\rangle, @\texttt{ThisClass}) = \overline{\texttt{Y}}\ \overline{\texttt{g}} \qquad \mathit{pos}(\overline{\texttt{Y}}\ \overline{\texttt{S}})\end{array}}{\mathit{fields}(\texttt{C}\langle\overline{\texttt{T}}\rangle, \texttt{R}) = [\overline{\texttt{T}}/\overline{\texttt{Z}}][\texttt{R}/@\texttt{ThisClass}, \texttt{ThisClass}](\overline{\texttt{Y}}\ \overline{\texttt{g}}, \overline{\texttt{S}}\ \overline{\texttt{f}})} \ \text{(F-Cl)}$$

Method Type Lookup

$$\frac{\mathsf{CT}(\texttt{C}) = \texttt{class}\ \texttt{C}\langle\overline{\texttt{Z}} \lhd \overline{\texttt{N}}\rangle \lhd \texttt{D}\langle\overline{\texttt{U}}\rangle\ \{\ \ldots\ \overline{\texttt{M}}\ \} \qquad \langle\overline{\texttt{Y}} \lhd \overline{\texttt{O}}\rangle\texttt{V}\ \texttt{m}(\overline{\texttt{V}}\ \overline{\texttt{x}})\{\uparrow \texttt{e};\} \in \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C}\langle\overline{\texttt{T}}\rangle, @\texttt{R}) = [\overline{\texttt{T}}/\overline{\texttt{Z}}][\texttt{R}/\texttt{ThisClass}](\langle\overline{\texttt{Y}} \lhd \overline{\texttt{O}}\rangle\overline{\texttt{V}} \to \texttt{V})} \ \text{(MT-@Cl)}$$

$$\frac{\begin{array}{c}\mathsf{CT}(\texttt{C}) = \texttt{class}\ \texttt{C}\langle\overline{\texttt{Z}} \lhd \overline{\texttt{N}}\rangle \lhd \texttt{D}\langle\overline{\texttt{U}}\rangle\ \{\ \ldots\ \overline{\texttt{M}}\ \} \qquad \langle\overline{\texttt{Y}} \lhd \overline{\texttt{O}}\rangle\texttt{V}\ \texttt{m}(\overline{\texttt{V}}\ \overline{\texttt{x}})\{\uparrow \texttt{e};\} \in \overline{\texttt{M}} \\ \texttt{R}\ \text{not exact} \qquad \texttt{ThisClass}\ \text{does not appear in}\ \overline{\texttt{V}} \qquad \mathit{pos}(\texttt{V})\end{array}}{\mathit{mtype}(\texttt{m}, \texttt{C}\langle\overline{\texttt{T}}\rangle, \texttt{R}) = [\overline{\texttt{T}}/\overline{\texttt{Z}}][\texttt{R}/@\texttt{ThisClass}, \texttt{ThisClass}](\langle\overline{\texttt{Y}} \lhd \overline{\texttt{O}}\rangle\overline{\texttt{V}} \to \texttt{V})} \ \text{(MT-Cl)}$$

$$\frac{\mathsf{CT}(\texttt{C}) = \texttt{class}\ \texttt{C}\langle\overline{\texttt{Z}} \lhd \overline{\texttt{N}}\rangle \lhd \texttt{D}\langle\overline{\texttt{U}}\rangle\ \{\ \ldots\ \overline{\texttt{M}}\ \} \qquad \langle\overline{\texttt{Y}} \lhd \overline{\texttt{O}}\rangle\texttt{V}\ \texttt{m}(\overline{\texttt{V}}\ \overline{\texttt{x}})\{\uparrow \texttt{e};\} \not\in \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C}\langle\overline{\texttt{T}}\rangle, @\texttt{R}) = [\overline{\texttt{T}}/\overline{\texttt{Z}}][\texttt{R}/\texttt{ThisClass}](\mathit{mtype}(\texttt{m}, \texttt{D}\langle\overline{\texttt{U}}\rangle, @\texttt{ThisClass}))} \ \text{(MT-@Sup)}$$

$$\frac{\begin{array}{c}\mathsf{CT}(\texttt{C}) = \texttt{class}\ \texttt{C}\langle\overline{\texttt{Z}} \lhd \overline{\texttt{N}}\rangle \lhd \texttt{D}\langle\overline{\texttt{U}}\rangle\ \{\ \ldots\ \overline{\texttt{M}}\ \} \\ \langle\overline{\texttt{Y}} \lhd \overline{\texttt{O}}\rangle\texttt{V}\ \texttt{m}(\overline{\texttt{V}}\ \overline{\texttt{x}})\{\uparrow \texttt{e};\} \not\in \overline{\texttt{M}} \qquad \mathit{mtype}(\texttt{m}, \texttt{D}\langle\overline{\texttt{U}}\rangle, @\texttt{ThisClass}) = \langle\overline{\texttt{Y}} \lhd \overline{\texttt{O}}\rangle\overline{\texttt{V}} \to \texttt{V} \\ \texttt{R}\ \text{not exact} \qquad \texttt{ThisClass}\ \text{does not appear in}\ \overline{\texttt{V}} \qquad \mathit{pos}(\texttt{V})\end{array}}{\mathit{mtype}(\texttt{m}, \texttt{C}\langle\overline{\texttt{T}}\rangle, \texttt{R}) = [\overline{\texttt{T}}/\overline{\texttt{Z}}][\texttt{R}/@\texttt{ThisClass}, \texttt{ThisClass}]\langle\overline{\texttt{Y}} \lhd \overline{\texttt{O}}\rangle\overline{\texttt{V}} \to \texttt{V}} \ \text{(MT-Sup)}$$

Positive Occurrences of `ThisClass`

$$\mathit{pos}(\texttt{Object}) \qquad \mathit{pos}(\texttt{X}) \qquad \frac{\mathit{pos}(\texttt{T})}{\mathit{pos}(@\texttt{T})} \qquad \frac{\texttt{ThisClass}\ \text{does not appear in}\ \overline{\texttt{T}}}{\mathit{pos}(\texttt{C}\langle\overline{\texttt{T}}\rangle)}$$

Fig. 1. FLJ Syntax and Auxiliary Definitions

non-standard substitution operation. When we write the substitution expression:

$$[\mathtt{R}/@\mathtt{ThisClass}, \mathtt{ThisClass}]\mathtt{T}$$

we mean the operation that substitutes `R` for each occurrence of @`ThisClass` in `T` and `R` for each *remaining* `ThisClass` in `T`.[18]

The definitions of *mbody*, which looks up the body of a method, bound_Δ, which looks up the bound of a type in Δ, and *override*, which ensures that a class correctly overrides a method, are very similar to their versions in FGJ and are elided here. Figure 1 shows the definitions of *fields*, *mtype*, and *pos*. The definition *mtype* takes three arguments: the method's name, the class where we start searching for its definition, and the type of the receiver of the method invocation and returns the method's type. The definition of *fields* is similar; it takes the type of the class and the type of the receiver of the field access and returns the types and names of the fields. In each of these definitions, occurrences of `ThisClass` in the signature are replaced by the type of the the receiver. Finally, *pos* asserts that `ThisClass` only appears positively in a type. In particular *pos*(`ThisClass`) and *pos*(@`ThisClass`) are defined but *pos*($\mathtt{C}\langle\mathtt{ThisClass}\rangle$) is not, because $\mathtt{C}\langle\mathtt{Z}\rangle$ might have a method that takes a parameter of type `Z` and thus, use `ThisClass` negatively.[19]

Note that when used with an unexact receiver, *mtype* and *fields* can be undefined if, for example, `ThisClass` appears in a negative position in the type signature.

The evaluation relation for FLJ is given in Figure 2. Note the minor change from FGJ that a cast expression reduces if the object's type (which at runtime is known exactly) is a subtype of the specified type, `T`, in an empty type context.[20] We elide the congruence rules EC-Field, EC-Invk-Recv, EC-Invk-Arg, and EC-New-Arg, which are all similar to the versions given in FGJ.

Well-Formed Types The rules for well-formed types are given in Figure 2. The interesting cases here involve exact types: rule WF-@ states that an exact type is well-formed if its non-exact version is well-formed and not an exact type (this prevents us from forming doubly-exact types such as `@@T`). Similarly, the types used to instantiate a class's type variables must not be exact, as specified by rule WF-Class.

---

[18] This is not quite the same as sequential or simultaneous substitution. In the compiler, this special substitution operation is realized by first renaming occurrences of `ThisClass` in `R` to a fresh name, performing the two substitutions simultaneously, and then renaming the occurrences in `R` back to `ThisClass`.

[19] A more complicated definition of *pos* could allow *pos*($\mathtt{C}\langle\mathtt{ThisClass}\rangle$) by examining the definition of $\mathtt{C}\langle\mathtt{Z}\rangle$ and ensuring that `Z` is only used positively in `C`. We use the simpler but more restrictive rule.

[20] To retain consistency with FJ, we use the word "subtype" and symbol <: to stand for the transitive closure of the extension operator in Java. However, the actual relation defined is more similar to matching (see [Bru02,BFP97]) than subtype.

Evaluation

$$\frac{\mathsf{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{Z}} \triangleleft \overline{\mathtt{N}}\rangle \triangleleft \mathtt{D}\langle\overline{\mathtt{U}}\rangle\ \{\ \ldots\ \} \qquad \mathit{fields}(\mathtt{C}\langle\overline{\mathtt{T}}\rangle, @\mathtt{C}\langle\overline{\mathtt{T}}\rangle) = \overline{\mathtt{Q}}\ \overline{\mathtt{f}}}{\mathtt{new}\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle(\overline{\mathtt{e}}).\mathtt{f_i} \longrightarrow \mathtt{e_i}}\ \text{(E-Field)}$$

$$\frac{\mathsf{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{Z}} \triangleleft \overline{\mathtt{N}}\rangle \triangleleft \mathtt{D}\langle\overline{\mathtt{U}}\rangle\ \{\ \ldots\ \} \qquad \mathit{mbody}(\mathtt{m}\langle\overline{\mathtt{Y}}\rangle, \mathtt{C}\langle\overline{\mathtt{T}}\rangle, @\mathtt{C}\langle\overline{\mathtt{T}}\rangle) = (\overline{\mathtt{x}}, \mathtt{e_0})}{\mathtt{new}\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle(\overline{\mathtt{e}}).\mathtt{m}\langle\overline{\mathtt{Y}}\rangle(\overline{\mathtt{d}}) \longrightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new}\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle(\overline{\mathtt{e}})/\mathtt{this}]\mathtt{e_0}}\ \text{(E-Invk)}$$

$$\frac{\emptyset \vdash @\mathtt{C}\langle\overline{\mathtt{T}}\rangle <: \mathtt{T}}{(\mathtt{T})\mathtt{new}\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle(\overline{\mathtt{e}}) \longrightarrow \mathtt{new}\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle(\overline{\mathtt{e}})}\ \text{(E-Cast)}$$

Well-Formed Types

$$\Delta \vdash \mathtt{Object}\ \mathit{ok} \quad \text{(WF-Obj)} \qquad\qquad \frac{\mathtt{X} \in \mathit{dom}(\Delta)}{\Delta \vdash \mathtt{X}\ \mathit{ok}}\ \text{(WF-Var)}$$

$$\frac{\Delta \vdash \mathtt{T}\ \mathit{ok} \qquad \mathtt{T}\ \text{not exact}}{\Delta \vdash @\mathtt{T}\ \mathit{ok}}\ \text{(WF-@)}$$

$$\frac{\begin{array}{c}\mathsf{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{Z}} \triangleleft \overline{\mathtt{N}}\rangle \triangleleft \mathtt{D}\langle\overline{\mathtt{U}}\rangle\ \{\ \ldots\ \} \\ \Delta \vdash \overline{\mathtt{T}}\ \mathit{ok} \qquad \Delta \vdash \overline{\mathtt{T}} <: [\overline{\mathtt{T}}/\overline{\mathtt{Z}}]\overline{\mathtt{N}} \qquad \text{None of } \overline{\mathtt{T}} \text{ exact}\end{array}}{\Delta \vdash \mathtt{C}\langle\overline{\mathtt{T}}\rangle\ \mathit{ok}}\ \text{(WF-Class)}$$

Subtyping

$$\Delta \vdash \mathtt{T} <: \mathtt{T} \quad \text{(S-Refl)} \qquad \Delta \vdash \mathtt{X} <: \Delta(\mathtt{X}) \quad \text{(S-Var)} \qquad \Delta \vdash @\mathtt{T} <: \mathtt{T} \quad \text{(S-Exact)}$$

$$\frac{\Delta \vdash \mathtt{S} <: \mathtt{T} \qquad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{S} <: \mathtt{U}}\ \text{(S-Trans)}$$

$$\frac{\mathsf{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{Z}} \triangleleft \overline{\mathtt{N}}\rangle \triangleleft \mathtt{D}\langle\overline{\mathtt{U}}\rangle\ \{\ \ldots\ \}}{\Delta \vdash \mathtt{C}\langle\overline{\mathtt{T}}\rangle <: [\overline{\mathtt{T}}/\overline{\mathtt{Z}}]\mathtt{D}\langle\overline{\mathtt{U}}\rangle}\ \text{(S-Super)}$$

Method Typing

$$\frac{\begin{array}{c}\Delta = \overline{\mathtt{Z}} <: \overline{\mathtt{N}}, \overline{\mathtt{Y}} <: \overline{\mathtt{O}}, \mathtt{ThisClass} <: \mathtt{C}\langle\overline{\mathtt{Z}}\rangle \\ \Delta \vdash \overline{\mathtt{T}}, \mathtt{T}, \overline{\mathtt{O}}\ \mathit{ok} \qquad \Delta; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : @\mathtt{ThisClass} \vdash \mathtt{e_0} : \mathtt{S} \\ \Delta \vdash \mathtt{S} <: \mathtt{T} \qquad \mathsf{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{Z}} \triangleleft \overline{\mathtt{N}}\rangle \triangleleft \mathtt{D}\langle\overline{\mathtt{U}}\rangle\ \{\ \ldots\ \} \qquad \mathit{override}(\mathtt{m}, \mathtt{D}\langle\overline{\mathtt{U}}\rangle, \langle\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{O}}\rangle\overline{\mathtt{T}} \rightarrow \mathtt{T})\end{array}}{\langle\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{O}}\rangle\ \mathtt{T}\ \mathtt{m}(\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \{\ \uparrow \mathtt{e_0};\ \}\ \text{OK in } \mathtt{C}\langle\overline{\mathtt{Z}} \triangleleft \overline{\mathtt{N}}\rangle}$$

Class Typing

$$\frac{\begin{array}{c}\overline{\mathtt{Z}} <: \overline{\mathtt{N}} \vdash \overline{\mathtt{N}}, \mathtt{D}\langle\overline{\mathtt{V}}\rangle, \overline{\mathtt{T}}\ \mathit{ok} \qquad \mathit{fields}(\mathtt{D}\langle\overline{\mathtt{V}}\rangle, @\mathtt{ThisClass}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}} \\ \overline{\mathtt{M}}\ \text{OK in } \mathtt{C}\langle\overline{\mathtt{Z}} \triangleleft \overline{\mathtt{N}}\rangle \qquad \mathtt{K} = \mathtt{C}(\overline{\mathtt{U}}\ \overline{\mathtt{g}}, \overline{\mathtt{T}}\ \overline{\mathtt{f}})\ \{\ \mathtt{super}(\overline{\mathtt{g}});\ \mathtt{this}.\overline{\mathtt{f}} = \overline{\mathtt{f}};\ \}\end{array}}{\mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{Z}} \triangleleft \overline{\mathtt{N}}\rangle \triangleleft \mathtt{D}\langle\overline{\mathtt{V}}\rangle\ \{\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\ \}\ \text{OK}}$$

Fig. 2. FLJ Semantics, Well-formedness, Subtyping, Class and Method Typing

Expression Typing

$$\Delta;\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \quad \text{(T-Var)}$$

$$\frac{\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \mathit{fields}(\mathrm{bound}_\Delta(\mathtt{T}_0), \mathtt{T}_0) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{f}_\mathtt{i} : \mathtt{T}_\mathtt{i}} \text{ (T-Field)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \mathit{mtype}(m, \mathrm{bound}_\Delta(\mathtt{T}_0), \mathtt{T}_0) = \langle \overline{\mathtt{Y}} \triangleleft \overline{\mathtt{O}} \rangle \overline{\mathtt{U}} \to \mathtt{U} \\ \Delta \vdash \overline{\mathtt{V}}\ ok \qquad \Delta \vdash \overline{\mathtt{V}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{O}} \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} <: [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{m}\langle \overline{\mathtt{V}} \rangle (\overline{\mathtt{e}}) : [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}} \text{ (T-Invk)}$$

$$\frac{\begin{array}{c}\mathsf{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\langle \overline{\mathtt{Z}} \triangleleft \overline{\mathtt{N}} \rangle \triangleleft \mathtt{D}\langle \overline{\mathtt{U}} \rangle\ \{\ \ldots\ \} \\ \Delta \vdash \mathtt{C}\langle \overline{\mathtt{T}} \rangle\ ok \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} : \overline{\mathtt{S}} \qquad \mathit{fields}(\mathtt{C}\langle \overline{\mathtt{T}} \rangle, @\mathtt{C}\langle \overline{\mathtt{T}} \rangle) = \overline{\mathtt{R}}\ \overline{\mathtt{f}} \qquad \Delta \vdash \overline{\mathtt{S}} <: \overline{\mathtt{R}}\end{array}}{\Delta;\Gamma \vdash \mathtt{new}\ \mathtt{C}\langle \overline{\mathtt{T}} \rangle (\overline{\mathtt{e}}) : @\mathtt{C}\langle \overline{\mathtt{T}} \rangle} \text{ (T-New)}$$

$$\frac{\Delta \vdash \mathtt{T}\ ok \qquad \Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}_0 <: \mathtt{T}}{\Delta;\Gamma \vdash (\mathtt{T})\mathtt{e}_0 : \mathtt{T}} \text{ (T-UCast)}$$

$$\frac{\Delta \vdash \mathtt{T}\ ok \qquad \Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T} <: \mathtt{T}_0 \qquad \mathtt{T}_0 \neq \mathtt{T}}{\Delta;\Gamma \vdash (\mathtt{T})\mathtt{e}_0 : \mathtt{T}} \text{ (T-DCast)}$$

$$\frac{\Delta \vdash \mathtt{T}\ ok \qquad \Delta;\Gamma \vdash \mathtt{e}_0 : \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}_0 \not<: \mathtt{T}, \mathtt{T} \not<: \mathtt{T}_0 \qquad \mathit{stupid\ warning}}{\Delta;\Gamma \vdash (\mathtt{T})\mathtt{e}_0 : \mathtt{T}} \text{ (T-SCast)}$$

Fig. 3. FLJ Expression Typing

The subtyping rules for FLJ are given in Figure 2. The only interesting difference here is the addition of S-Exact. It says that an exact type is a subtype of its unexact version. Intuitively, this makes sense: we can use an object of type `@T` anywhere that we expected an object whose type was `T`.

The rule for typing classes and methods are given in Figure 2. Note that methods are type checked in a context where `ThisClass` extends the class that the method appears in, and where `this` is assumed to have type `@ThisClass`.

Figure 3 gives the rules for typing expressions. The most interesting cases are for field access and method invocation. Because they are similar, we explain only method invocation here. Recall that a binary method may only be invoked on a receiver whose type is known exactly. In rule T-Invk, the type of the receiver, `T_0`, of the method call is passed to *mtype*. If `T_0` is exact, then *mtype* substitutes it for `ThisClass` in the signature before returning the method type. If `T_0` is not exact and `ThisClass` appears in a negative position, then *mtype* is undefined. Otherwise it substitutes `T_0` for `@ThisClass` and then substitutes `T_0` for `ThisClass`. Finally, we check that the types of the actual parameters are subtypes of the types of the formal parameters. Typing for fields is similar. Rule T-New states that a new expression has the exact class type of the object created. The rule T-DCast is more flexible than the GJ version because LOOJ is not implemented by erasure, so unlike GJ, we do not need to rule out casts that would fail at the source level but succeed in their compiled versions.

Featherweight LOOJ Properties Because of space limitations, we only give the statements of the relevant lemmas and theorems.

Lemma 1 (Inversion).

1. If $\Delta \vdash \texttt{S} <: @\texttt{T}$ with no exact types appearing in Δ, then $\texttt{S}$ is $@\texttt{T}$.
2. If $pos(\texttt{T})$ then either $\texttt{T}$ is `ThisClass` or `@ThisClass`, or else `ThisClass` does not appear in $\texttt{T}$.

Lemma 2 (Fields Lookup). If $\Delta \vdash \texttt{S} <: \texttt{T}$ and $fields(\text{bound}_\Delta(\texttt{T}), \texttt{T}) = \overline{\texttt{C}}\ \overline{\texttt{f}}$ then $fields(\text{bound}_\Delta(\texttt{S}), \texttt{S}) = \overline{\texttt{D}}\ \overline{\texttt{g}}$ and $\forall i \leq \#(\overline{\texttt{f}}), \Delta \vdash \texttt{D}_\texttt{i} <: \texttt{C}_\texttt{i}$ and $\texttt{f}_\texttt{i} = \texttt{g}_\texttt{i}$

Lemma 3 (MType Lookup). If $\Delta \vdash \texttt{S} <: \texttt{T}$ and $mtype(\texttt{m}, \text{bound}_\Delta(\texttt{T}), \texttt{T}) = \langle \overline{\texttt{Y}} \triangleleft \overline{\texttt{O}} \rangle \overline{\texttt{C}} \rightarrow \texttt{C}$ then $mtype(\texttt{m}, \text{bound}_\Delta(\texttt{S}), \texttt{S}) = \langle \overline{\texttt{Y}} \triangleleft \overline{\texttt{O}} \rangle \overline{\texttt{C}} \rightarrow \texttt{C}'$ and $\Delta, \overline{\texttt{Y}} <: \overline{\texttt{O}} \vdash \texttt{C}' <: \texttt{C}$

Lemma 4 (Substitution). If $\Delta_1, \overline{\texttt{X}} <: \overline{\texttt{N}}, \Delta_2 \vdash \texttt{S} <: \texttt{T}$ and $\Delta_1 \vdash \overline{\texttt{U}} <: [\overline{\texttt{U}}/\overline{\texttt{X}}]\overline{\texttt{N}}$ and $\Delta_1 \vdash \overline{\texttt{U}}\ ok$ and none of $\overline{\texttt{U}}$ exact and none of $\overline{\texttt{X}} \in \Delta_1$ then

1. $\Delta_1, [\overline{\texttt{U}}/\overline{\texttt{X}}]\Delta_2 \vdash [\overline{\texttt{U}}/\overline{\texttt{X}}]\texttt{S} <: [\overline{\texttt{U}}/\overline{\texttt{X}}]\texttt{T}$
2. $\Delta_1, [\overline{\texttt{U}}/\overline{\texttt{X}}]\Delta_2 \vdash [\overline{\texttt{U}}/\overline{\texttt{X}}]\texttt{T}\ ok$
3. Additionally, if $\Delta_1, \overline{\texttt{X}} <: \overline{\texttt{N}}, \Delta_2; \Gamma \vdash \texttt{e} : \texttt{T}$ then $\Delta_1, [\overline{\texttt{U}}/\overline{\texttt{X}}]\Delta_2; [\overline{\texttt{U}}/\overline{\texttt{X}}]\Gamma \vdash [\overline{\texttt{U}}/\overline{\texttt{X}}]\texttt{e} : \texttt{S}$ for some $\texttt{S}$ such that $\Delta_1, [\overline{\texttt{U}}/\overline{\texttt{X}}]\Delta_2 \vdash \texttt{S} <: [\overline{\texttt{U}}/\overline{\texttt{X}}]\texttt{T}$
4. And if $\Delta; \Gamma, \overline{\texttt{x}} : \overline{\texttt{T}} \vdash \texttt{e} : \texttt{T}$ and $\Delta; \Gamma \vdash \overline{\texttt{d}} : \overline{\texttt{S}}$ and $\Delta \vdash \overline{\texttt{S}} <: \overline{\texttt{T}}$ then $\Delta; \Gamma \vdash [\overline{\texttt{d}}/\overline{\texttt{x}}]\texttt{e} : \texttt{S}$ for some $\texttt{S}$ such that $\texttt{S} <: \texttt{T}$

Theorem 1 (Preservation). If $\Delta; \Gamma \vdash \texttt{e} : \texttt{T}$ and $\texttt{e} \rightarrow \texttt{e}'$ then $\Delta; \Gamma \vdash \texttt{e}' : \texttt{T}'$ and $\Delta \vdash \texttt{T}' <: \texttt{T}$.

Theorem 2 (Progress). Suppose that $\vdash \texttt{e} : \texttt{T}$

1. If the expression $\texttt{e}$ contains $\texttt{new C}\langle \overline{\texttt{T}} \rangle (\overline{\texttt{e}}).\texttt{f}$ as a sub-expression then $fields(\texttt{C}\langle \overline{\texttt{T}} \rangle, @\texttt{C}\langle \overline{\texttt{T}} \rangle) = \overline{\texttt{T}}\ \overline{\texttt{f}}$ and $\texttt{f} \in \overline{\texttt{f}}$.
2. If the expression $\texttt{e}$ contains $\texttt{new C}\langle \overline{\texttt{T}} \rangle (\overline{\texttt{e}}).\texttt{m}\langle \overline{\texttt{V}} \rangle (\overline{\texttt{d}})$ as a sub-expression then $mbody(\texttt{m}\langle \overline{\texttt{V}} \rangle, \texttt{C}\langle \overline{\texttt{T}} \rangle, @\texttt{C}\langle \overline{\texttt{T}} \rangle) = (\overline{\texttt{x}}, \texttt{e}_\texttt{0})$ and $\#(\overline{\texttt{x}}) = \#(\overline{\texttt{d}})$.

Theorem 3 (Type Soundness). If $\vdash \texttt{e} : \texttt{T}$ and $\texttt{e} \rightarrow^* \texttt{e}'$ and $\texttt{e}'$ is a normal form, then either $\texttt{e}'$ is a value, or it contains a failed cast.

While Featherweight LOOJ omits imperative features, typing problems with instance variables generally also arise with parameters, so we are confident that the addition of instance variables will add no new difficulties.