CZ: Multiple Inheritance Without Diamonds

Donna Malayeri
Carnegie Mellon University
donna@cs.cmu.edu

Jonathan Aldrich
Carnegie Mellon University
aldrich@cs.cmu.edu

Abstract

Multiple inheritance has long been plagued with the "diamond" inheritance problem, leading to solutions that restrict expressiveness, such as mixins and traits. Instead, we address the diamond problem directly, considering two difficulties it causes: ensuring a correct semantics for object initializers, and typechecking multiple dispatch in a modular fashion—the latter problem arising even with multiple interface inheritance. We show that previous solutions to these problems are either unsatisfactory or cumbersome, and suggest a novel approach: supporting multiple inheritance but forbidding diamond inheritance. Expressiveness is retained through two features: a "requires" construct that provides a form of subtyping without inheritance (inspired by Scala [40]), and a dynamically-dispatched "super" call similar to that found in traits. Through examples, we illustrate that inheritance diamonds can be eliminated via a combination of "requires" and ordinary inheritance. We provide a sound formal model for our language and demonstrate its modularity and expressiveness.

Categories and Subject Descriptors D3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages

Keywords Multiple inheritance, diamond problem, multimethods, modularity

1. Introduction

Single inheritance, mixins [12, 5, 24, 22, 4, 9], and traits [19, 23, 40] each have disadvantages: single inheritance restricts expressiveness, mixins must be linearly applied, and traits do not allow state. Multiple inheritance is one solution to these problems, as it allows code to be reused along multiple dimensions. Unfortunately however, multiple inheritance poses challenges itself.

There are two types of problems with multiple inheritance: (a) a class can inherit multiple features with the same name, and (b) a class can have more than one path to a given ancestor (i.e., the "diamond problem", also known as "fork-join" inheritance) [43, 46]. The first, the conflicting-features problem, can be solved by allowing renaming (e.g., Eiffel [32]) or by linearizing the class hierarchy [47, 46]. However, there is still no satisfactory solution to the diamond problem.

The diamond problem arises when a class C inherits an ancestor A through more than one path. This is particularly problematic when A has fields—should C inherit multiple copies of the fields or just one? Virtual inheritance in C++ is designed as one solution for C to inherit only one copy of A's fields [21]. But with only one copy of A's fields, object initializers are a problem: if C transitively calls A's initializer, how can we ensure that it is called only once? Existing solutions either restrict the form of constructor definitions, or ignore some constructor calls [39, 21].

There is another consequence of the diamond problem: it causes multiple inheritance to interact poorly with modular typechecking of multiple dispatch. Multiple dispatch is a very powerful language mechanism that provides direct support for extensibility and software evolution [14, 16]; for these reasons, it has been adopted by designers of new programming languages, such as Fortress [2]. Unfortunately however, modular multimethods are difficult to combine with any form of multiple inheritance—even restricted forms, such as traits or Java-style multiple interface inheritance. Previous work either disallows multiple inheritance across module boundaries, or burdens programmers by requiring that they always provide (possibly numerous) disambiguating methods.

To solve these problems, we take a different approach: while permitting multiple inheritance, we disallow inheritance diamonds entirely. So that there is no loss of expressiveness, we divide the notion of inheritance into two concepts: an *inheritance dependency* (expressed using a requires clause, an extension of a Scala construct [39]) and implementation inheritance (expressed through extends). Through examples, we illustrate that programs that are expressed using diamond inheritance can be translated to a

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.
Copyright © 2009 ACM 978-1-60558-734-9/09/10...$10.00

hierarchy that uses a combination of requires and extends, without the presence of diamonds. As a result, our language, CZ—for cubic zirconia—retains the expressiveness of diamond inheritance.

We argue that a hierarchy with multiple inheritance is conceptually two or more separate hierarchies. These hierarchies represent different "dimensions" of the class that is multiply inherited. We express dependencies between these dimensions using requires, and give an extended example of its use in Sect. 5.

Our solution has two advantages: fields and multiple inheritance (including initializers) can gracefully co-exist, and multiple dispatch and multiple inheritance can be combined. To achieve the latter, we make an incremental extension to existing techniques for modular typechecking of multiple dispatch.[1]

An additional feature of our language is a dynamically-dispatched super call, modelled after trait super calls [19]. When a call is made to A.super.$f()$ on an object with dynamic type D, the call proceeds to f defined within D's immediate superclass along the A path. With dynamically-dispatched super calls and requires, our language attains the expressiveness of traits while still allowing classes to inherit state.

We have formalized our system as an extension of Featherweight Java (FJ) [28] (Sect. 8) and have proved it sound [31].

Contributions:

- The design of a novel multiple inheritance scheme[2] that solves (1) the object initialization problem and (2) the modular typechecking of multimethods, by forbidding diamond inheritance (Sections 2 and 4).
- Generalization of the requires construct and integration with dynamically-dispatched super calls (Sect. 6).
- Examples that illustrate how a diamond inheritance scheme can be converted to one without diamonds (Sections 3 and 5).
- Examples from actual C++ and Java programs, illustrating the utility of multiple inheritance and inheritance diamonds (Sect. 7).
- A formalization of the language, detailed argument of modularity (Sect. 8), and proof of type safety.
- An implementation of a typechecker for the language, as an extension of the JastAddJ Java compiler [20].

[1] For simplicity and ease of understanding, our formal system only allows dispatch on a method's receiver and its first argument, corresponding to double dispatch. The system can be easily generalized to n-argument multimethods, as all interesting typechecking issues also arise in the two-argument case. See Sect. 8 for further details.

[2] This is a revised and expanded version of a paper presented at the FOOL '09 workshop [30]. (FOOL has no archival proceedings and is not intended to preclude later publication.) The main changes are the inclusion of multimethods rather than external methods and a new section on real-world examples (Sect 7).

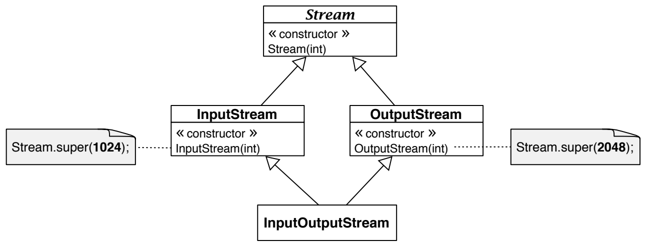

Figure 1. An inheritance diamond. Italicized class names indicate abstract classes.

2. Object Initialization

To start with, diamond inheritance raises a question: should class C with a repeated ancestor A have two copies of A's instance variables or just one—i.e., should inheritance be "tree inheritance" or "graph inheritance" [13]? As the former may be modelled using composition, the latter is the desirable semantics; it is supported in languages such as Scala, Eiffel, and C++ (the last through virtual inheritance) [39, 32, 21]. Unfortunately, the object initialization problem occurs in this semantics, depending how and when the superclass constructor or initializer is called [47, 46].

The Problem. To illustrate the problem, consider Figure 1, which shows a class hierarchy containing a diamond. Suppose that the Stream superclass has a constructor taking an integer, to set the size of a buffer. InputStream and OutputStream call this constructor with different values (1024 and 2048, respectively). But, when creating an InputOutputStream, with which value should the Stream constructor be called? Moreover, InputStream and OutputStream could even call different constructors with differing parameter types, making the situation even more uncertain.

Previous Solutions. Languages that directly attempt to solve the object initialization problem include Eiffel [32], C++ [21], Scala [39] and Smalltalk with stateful traits [8].

In Eiffel, even though (by default) only one instance of the repeatedly inherited class is included (e.g., Stream), when constructing an InputOutputStream, the Stream constructor is called twice. This has the advantage of simplicity, but unfortunately it does not provide the proper semantics; Stream's constructor may perform a stateful operation (e.g., allocating a buffer), and this operation would occur twice.

In C++, if virtual inheritance is used (so that there is only one copy of Stream), the constructor problem is solved as follows: the calls to the Stream constructor from InputStream and OutputStream are ignored, and InputOutputStream must call the Stream constructor explicitly.[3] Though the Stream constructor is called only once, this awkward design has the problem that constructor calls are

[3] Since there is no default Stream constructor, this call cannot be automatically generated.

ignored. The semantics of InputStream may require that a particular Stream constructor be called, but the language semantics would ignore this dependency by bypassing the constructor call.

Scala provides a different solution: trait constructors may not take arguments. (Scala traits are abstract classes that may contain state and may be multiply inherited.) This ensures that InputStream and OutputStream call the same super-trait constructor, causing no ambiguity for InputOutputStream. Though this design is simple and elegant, it restricts expressiveness (in fact, the Scala team is currently seeking a workaround to this problem [50]).

Smalltalk with stateful traits [8] does not contain constructors, but by convention, objects are initialized using an initialize message. Unfortunately, this results in the same semantics as Eiffel; here, the Stream constructor would be called twice [7]. The only way to avoid this problem would be to always define a special initializer that does not call the superclass initializer. Requiring that the programmer define such a method essentially means that the C++ solution must be hand-coded. Aside from being tedious and error-prone, this has the same drawbacks as the C++ semantics.

Mixins and traits do not address the object initialization problem directly, but instead restrict the language so that the problem does not arise in the first place. We compare CZ to each of these designs in Sect. 3.2.

3. An Overview of CZ

This section describes the CZ language design at a high level, including a description of how CZ addresses the object initialization problem and a comparison to related language designs.

3.1 CZ Design

CZ's design is based on the intuition that there are relationships between classes that are not captured by inheritance, and that if class hierarchies could express richer interconnections, inheritance diamonds need not exist. Suppose the concrete class C extends A. As noted by Schärli et al., it is beneficial to recognize that C serves two roles: (1) it is a generator of instances, and (2) it is a unit of reuse (through subclassing) [44]. In the first role, inheritance is the implementation strategy and may not be omitted. In the second role, however, it is possible to transform the class hierarchy to one where an inheritance *dependency* between C and A is stated and where *subclasses* of C inherit from both C and A. The key distinguishing feature of CZ is this notion of inheritance dependency, because while multiple inheritance is permitted, inheritance diamonds are forbidden.

Consider the inheritance diamond of Fig. 1. To translate this hierarchy to CZ, InputStream would be made abstract and its relationship to Stream would be changed from inheritance to an inheritance *dependency*, requiring that (concrete) subclasses of InputStream also inherit from Stream. In other words, InputStream *requires the presence of* Stream *in the* extends *clause of concrete subclasses*, but it need not extend Stream itself. Since InputStream is now abstract (making it serve only as a unit of reuse), it can be safely treated as a subtype of Stream. However, any concrete subclasses of InputStream (generators of instances), must also inherit from Stream. Accordingly, InputOutputStream must inherit from Stream directly.

We have reified this notion of an inheritance dependency using the requires keyword, a generalized form of a similar construct in Scala [40, 39].[4]

Definition 3.1 (Subclassing)**.** The subclass relation is defined as the reflexive, transitive closure of the extends relationship.

Definition 3.2 (Requires)**.**
When a class C requires a class B, we have the following:

- C is abstract
- C is a subtype of B (but not a subclass)
- Subclasses of C must either require B themselves (making them abstract) or extend B (allowing them to be concrete). This is achieved by including a requires B' or extends B' clause, where B' is a subclass of B.

In essence, C requires B is *a contract that C's concrete subclasses will extend B.*

The revised stream hierarchy is displayed in Fig. 2. In the original hierarchy, InputStream served as both generator of instances and a unit of reuse. In the revised hierarchy, we divide the class in two—one for each role. The class ConcreteInputStream is the generator of instances, and the abstract class InputStream is the unit of reuse. Accordingly, InputStream requires Stream, and ConcreteInputStream extends both InputStream and Stream. The concrete class InputOutputStream extends each of Stream, InputStream, and OutputStream, creating a sub*typing* diamond, but not a sub*classing* diamond, as requires does not create a subclass relationship.

The code for InputStream will be essentially the same as before, except for the call to its super constructor (explained further below). Because InputStream is a subtype of Stream, it may use all the fields and methods of Stream, without having to define them itself.

Programmers may add another dimension of stream behavior through additional abstract classes, for instance EncryptedStream. EncryptedStream is a type of stream, but it need not extend Stream, merely require it. Concrete subclasses, such as EncryptedInputStream must inherit from Stream, which is achieved by extending ConcreteInputStream. (It would also be possible to extend Stream and InputStream directly.)

[4] In Scala, requires is used to specify the type of a method's receiver (i.e., it is a selftype), and does not create a subtype relationship. As far as the Scala team is aware, our proposed use of requires is novel [50].

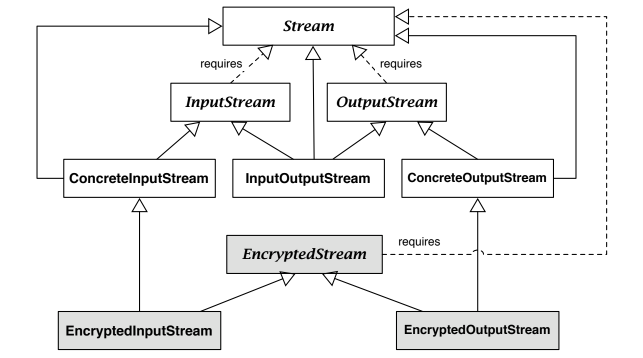

Figure 2. The stream hierarchy of Fig. 1, translated to CZ, with an encryption extension in gray. Italicized class names indicate abstract classes, solid lines indicate extends, and dashed lines indicate requires.

The requires relationship can also be viewed as declaring a semantic "mixin"—if B requires A, then B is effectively stating that it is an extension of A that can be "mixed-in" to clients. For example, EncryptedStream is enhancing Stream by adding encryption. Because the relationship is explicitly stated, it allows B to be substitutable for A.

Using requires is preferable to using extends because the two classes are more loosely coupled. For example, we could modify EncryptedInputStream to require InputStream (rather than extend ConcreteInputStream). A concrete subclass of EncryptedInputStream could then also extend a *subclass* of InputStream, such as BufferedInputStream, rather than extending InputStream directly. In this way, different pieces of functionality can be combined in a flexible manner while avoiding the complexity introduced by inheritance diamonds.

Object initialization. Because there are no inheritance diamonds, the object initialization problem is trivially solved. Note that if class C requires A, it need not (and should not) call A's constructor, since C does not inherit from A. In our example, InputStream does not call the Stream constructor, while ConcreteInputStream calls the constructors of its superclasses, InputStream and Stream. Thus, a sub*typing* diamond does not cause problems for object initialization.

This may seem similar to the C++ solution; after all, in both designs, InputOutputStream calls the Stream constructor. However, the CZ design is preferable for two reasons: a) there are no constructor calls to non-direct superclasses, and, more importantly, b) no constructor calls are ignored. In the C++ solution, InputStream may expect a particular Stream constructor to be called; as a result, it may not be properly initialized when this call is ignored. Essentially, CZ does not allow the programmer to create constructor dependencies that cannot be enforced.

Using "requires". Introducing two kinds of class relationships raises the question: when should programmers use requires, rather than extends? A rule of thumb is that requires should be used when a class is an extension of another class and is itself a unit of reuse. If necessary, a concrete class extending the required class (such as ConcreteInputStream) could also be defined to allow object creation. Note that this concrete class definition would be trivial, likely containing only a constructor. On the other hand, when a class hierarchy contains multiple disjoint alternatives (such as in the AST example in the next section), extends should be used; the no-diamond property is also a semantic property of the class hierarchy in question.

The above guideline may result in programmers defining more abstract classes (and corresponding concrete classes) than they may have otherwise used. However, some argue that it is good design to make a class abstract whenever it can be a base class. This is in accordance with the design of classes in Sather [49], traits in Scala and Fortress [39, 2, 3], and the advice that "non-leaf" classes in C++ be abstract [33]. In Sather and Fortress, for example, only abstract classes may have descendants; concrete classes (called "objects" in Fortress) form the leaves of the inheritance hierarchy [49]. Furthermore, a language could define syntactic sugar to ease the task of creating concrete class definitions; we sketch such a design in Sect. 6.4.

3.2 Related Work

Subtyping and subclassing. Since requires provides subtyping without subclassing, our design may seem to bear similarity to other work that has also separated these two concepts (e.g. [27, 18, 49, 15, 29]). There is an important difference, however, regarding information hiding. In a language that separates subclassing and subtyping, an "interface" type cannot contain private members; otherwise superclasses would be able to access private members defined in subclasses. Unfortunately, this restriction can be problematic for defining binary methods such as the equals method; its argument type must contain those private members for the method be able to access them. But, for this type to contain private members, it must be tied to a particular class implementation, as only subclasses (as opposed to subtypes) should conform to this type. See Appendix A for an example illustrating this issue.

This difficulty does not arise when using requires, as it establishes a stronger relationship than just subtyping; concrete subclasses must (directly or indirectly) inherit from the required class, as opposed to any class that provides a particular interface. Therefore, Stream may define an equals(Stream) method, and objects of type e.g. InputStream, OutputStream, or InputOutputStream may be safely passed to this method. Since the private member is defined in Stream and is only accessed by a method of Stream, this does not violate information hiding.

Mixins. Mixins, also known as abstract subclasses, provide many of the reuse benefits of multiple inheritance while fitting into a single inheritance framework [12, 5, 24, 22, 4, 9]. While mixins allow defining state, they have the drawbacks that they must be explicitly linearized by the programmer and they cannot inherit from one another (though most systems allow expressing implementation dependencies, such as abstract members). If mixin inheritance were allowed, this would be essentially equivalent to Scala traits, which *do* have the object initialization problem. Additionally, the lack of inheritance has the consequence that mixins do not integrate well with multiple dispatch; multiple dispatch requires an explicit inheritance hierarchy on which to perform the dispatch.

Traits. Traits were proposed as a mechanism for finer-grained reuse, to solve the reuse problems caused by mixins and multiple inheritance [19, 23, 40]. In particular, the linearization imposed by mixins can necessitate the definition of numerous "glue" methods [19]. This design avoids many problems caused by multiple inheritance since fields may not be defined in traits.

Unfortunately, this restriction results in other problems. In particular, non-private accessors in a trait negatively impact information hiding: if a trait needs to use state, this is encoded using abstract accessor methods, which must then be implemented by the class composed using the trait. Consequently, it is impossible to define "state" that is private to a trait—by definition, all classes reusing the trait can access this state. Additionally, introducing new accessors in a trait results in a ripple effect, as all client classes must now provide implementations for these methods [8], even if there are no other changes.

In contrast, CZ allows a class to multiply inherit other classes, which may contain state. In particular, a class may extend other concrete classes, while in trait systems, only traits may be multiply inherited.

Stateful traits. Stateful traits [8] were designed to address the aforementioned problems with stateless traits. But, as previously mentioned, this language does not address the problem of a correct semantics for object initialization in the presence of diamonds. Additionally, stateful traits do not address the information hiding problem, as they have been designed for maximal code reuse. In this design, state is hidden by default, but clients can "unhide" it, and may have to resort to merging variables that are inherited from multiple traits. While this provides a great deal of flexibility for trait clients, this design does not allow traits to define private state.

4. Modular Multiple Dispatch

CZ also supports multiple dispatch, which we and others believe is more natural and more expressive than single dispatch [14, 16, 15]. In fact, one common source of bugs in Java programs occurs when programmers expect static overloading to behave in a dynamic manner [26]. Multiple dispatch also avoids the extensibility problem inherent in the Visitor pattern, as well as the complexity introduced by manual double dispatch. However, typechecking multiple dispatch in a modular fashion is very difficult in the presence of *any* form of multiple inheritance—precisely because of the diamond problem.

4.1 The Problem

To see why diamond inheritance causes problems, suppose we have the original diamond stream hierarchy, and we now define a multimethod seek in a helper class (supposing that such functionality did not already exist in the classes in question):

```
class StreamHelper {
  void seek(Stream s, long pos) {
    // default implementation: do nothing
  }
  void seek(Stream@InputStream is, long pos) {
    // seek if pos <= eofPos
  }
  void seek(Stream@OutputStream os, long pos) {
    // if pos > eofPos, fill with zeros
  }
}
```

The declaration seek(Stream@InputStream, long) specifies that the method *specializes* seek(Stream, long) for the case that the first argument *dynamically* has type InputStream.

Unfortunately, in the context of our diamond hierarchy, this method definition is ambiguous—what if we perform the call h.seek(new InputOutputStream(), 1024)? Unfortunately, it is difficult to perform a *modular* check to determine this fact. When typechecking the definition of seek(), we cannot search for a potential subclass of both InputStream and OutputStream, as this analysis would not be modular. And, when typechecking InputOutputStream, we cannot search for multimethods defined on both of its superclasses, as that check would not be modular, either. We provide a detailed description of the conditions for modularity in Sect. 8.1.

It is important to note that this problem is *not* confined to multiple (implementation) inheritance—it arises in any scenario where an object can have multiple dynamic types on which dispatch is performed. For instance, the problem appears if dispatch is permitted on Java interfaces, as in JPred [25], or on traits, as in Fortress [3, 2]. For this reason, some languages restrict the form of dispatch to the single-inheritance case; e.g., MultiJava disallows dispatching on interfaces [16, 17].

4.2 Previous Solutions

There are two main solutions to the problem of modular typechecking of multiple dispatch in the presence of multiple inheritance. The first solution is simply to restrict expressiveness and disallow multiple inheritance across mod-

Language	Object initialization	Multimethod ambiguity
Eiffel	repeat initialization	–
C++	special constructor semantics	–
Scala	no-arg constructors	–
Fortress traits	n/a	disambiguating methods
Stateful traits	repeat initialization	–
Mixins	linearization	–
JPred	n/a	disambiguating methods
Dubious	n/a	MI restrictions

Table 1. Summary of related work and solutions to the object initialization and modular multimethod problems.

ule boundaries; this is the approach taken by the "System M" type system for Dubious [37].

JPred [25] and Fortress [3] take a different approach. The diamond problem arises in these languages due to multiple interface inheritance and multiple trait inheritance, respectively. In these languages, the typechecker ensures that multimethods are unambiguous by requiring that the programmer always specify a method for the case that an object is a subtype of two or more incomparable interfaces (or traits). In our streams example, the programmer would have to provide a method like the following in the StreamHelper class (in JPred syntax):

```
void seek(Stream s, long pos) when
    s@InputStream && s@OutputStream
```

(In Fortress, the method would be specified using intersection types.) Note that in both languages, this method would have to be defined for *every* subset of incomparable types (that contains at least 2 members), regardless of whether a type like InputOutputStream will ever be defined. Even if two types will *never* have a common subtype,[5] the programmer must specify a disambiguating method, one that perhaps throws an exception. Thus, the problem with this approach is that the programmer is required to write numerous additional methods—exponential in the number of incomparable types—some of which may never be called. JPred alleviates the problem somewhat by providing syntax to specify that a particular branch should be preferred in the case of an ambiguity, but it may not always be possible for programmers to know in advance which method to mark as preferred.

Note that neither JPred interfaces nor Fortress traits may contain state and thus the languages do not provide a solution to the object initialization problem; neither does Dubious, since it does not contain constructors.

These solutions and the previously described related work are summarized in Table 1.

[5] In Fortress, the programmer may specify that two traits are disjoint, meaning that there will never be a subtype of both. To allow modular typechecking, this disjoint specification must appear on one of the two trait definitions, which means that one must have knowledge of the other; consequently this is not an extensible solution.

4.3 Multimethods in CZ

To solve the problem of modular multiple dispatch, we use the same solution as for the object initialization problem: inheritance diamonds are forbidden, and requires is used as a substitute. An additional constraint is that a multimethod may only specialize a method in a superclass, not a required class (i.e., specialization is based on subclassing, not subtyping). So, in the CZ hierarchy of Fig. 2, the typechecker will signal an error, since the definitions seek(Stream@InputStream, long) and seek(Stream@OutputStream, long) are not valid specializations of seek(Stream, long).

Let us suppose for a moment that all classes in Fig. 2 have been defined, except InputOutputStream. Accordingly, we would re-write the seek methods as follows:

```
class StreamHelper {
    // helper methods
    void seekInput(InputStream s, long pos) { ... }
    void seekOutput(OutputStream s, long pos) { ... }

    // multimethods
    void seek(Stream s, long pos) { }
    void seek(Stream@ConcreteInputStream is, long pos) {
        seekInput(is, pos);
    }
    void seek(Stream@ConcreteOutputStream os, long pos) {
        seekOutput(os, pos);
    }
}
```

(Though these definitions are slightly more verbose than before, syntactic sugar could be provided, particularly for mapping multimethods to helper methods.)

Note that the typechecker does *not* require that a method be provided for "InputStream && OutputStream," unlike JPred and Fortress. If a programmer now defines InputOutputStream, but does not provide a new specialization for seek, the default implementation of seek(Stream) will be inherited. An specialization for InputOutputStream can then be implemented, perhaps one that calls seekOutput(). Note that this override need not be defined in StreamHelper directly; the method may be defined in one of its subclasses.

Here, it is of key importance that subclassing diamonds are disallowed; because they cannot occur, multimethods can be easily checked for ambiguities. Sub*typing* diamonds do not cause problems, as multimethod specialization is based on sub*classing*.

Dispatch semantics. There are two dispatch semantics that can be used for multimethods: asymmetric or symmetric. In asymmetric dispatching, the order of arguments affects dispatch. In particular, earlier arguments are treated as more important when selecting between equally specific methods. This semantics is used in a number of languages, such as Common Lisp and parasitic methods, among others [48, 41, 1, 11, 6].

Other languages employ the symmetric dispatch semantics, where all arguments have equal priority in determining method lookup [15, 45, 17, 36]. Some argue that symmetric dispatch is more intuitive and less error-prone than asymmetric dispatch [17, 36], though this form of dispatch adversely affects information hiding. In particular, a class may not hide the existence of a particular method specialization; this information is needed to correctly perform ambiguity checking of subclasses [36]. For this reason, and to simplify the type system and method lookup rules, CZ multimethod dispatch is asymmetric. However, CZ is compatible with symmetric dispatch; a symmetric-dispatch version of CZ would simply require additional (modular) checks on multimethod definitions. Incidentally, method lookup need not change, as these new ambiguity checks would ensure the same result, regardless of whether asymmetric lookup or symmetric lookup were used. Section 8 describes these issues in more detail.

Fragments of CZ. Note that it would be possible to omit multimethods from the language and use the CZ design (as is) for only the object initialization problem. That is, our solution can be used to solve either the object initialization problem, the modular multimethod problem, or both.

5. Example: Abstract Syntax Trees

Consider a simple class hierarchy for manipulating abstract syntax trees (ASTs), such as the one in Fig. 3. The original hierarchy is the one on the left, which consists of ASTNode, Num, Var, and Plus. An ASTNode contains a reference pointing to its parent node, as indicated in the figure. Each of the concrete subclasses of ASTNode implements its own version of the abstract ASTNode.eval() method.

Suppose we wish to add debugging support to our AST, after the original hierarchy is defined. Each node now additionally has a source location field, DebugNode.location. Debugging support, on the right side of the figure, is essentially a new dimension of AST nodes that has a dependency on ASTNode. We express this using requires. Now, classes like DebugPlus can multiply inherit from ASTNode and DebugNode without creating a subclassing diamond. In particular, DebugPlus does *not* inherit two copies of the parent field, because DebugNode is a subtype, but not a subclass, of ASTNode. Thus, the no-diamond property allows fields and multiple inheritance to co-exist gracefully.

In this example, each of these classes has a method eval() which evaluates that node of the AST, as in the code in Fig. 4. Suppose we intend DebugNode to act as a generic wrapper class for each of the subclasses of ASTNode. This can be implemented by using a dynamically-dispatched super call of the form ASTNode.super.eval() after performing the debug-specific functionality (in this case, printing the node's string representation). The prefix ASTNode.super means

```
class DebugNode requires ASTNode {
   ASTNode eval() {
      print(this.toString());

      // dynamic super call
      return ASTNode.super.eval();
   }
}
class DebugPlus extends DebugNode, Plus {
   ASTNode eval() {
      // ordinary super call
      return DebugNode.super.eval();
   }
}
```

Figure 4. Implementing a mixin-like debug class using dynamically-dispatched super calls, and performing multimethod dispatch on the ASTNode hierarchy.

"find the parent class of the dynamic class of this along the ASTNode path." At runtime, when eval() is called on an instance of DebugPlus, the chain of calls proceeds as follows: DebugPlus.eval() $\rightarrow$ DebugNode.eval() $\rightarrow$ Plus.eval(). If the dynamically-dispatched super call behaved as an ordinary super call, it would fail, because DebugNode has no superclass.

Each of the DebugNode subclasses implements its own eval() method that calls DebugNode.eval() with an ordinary super call. (This could be omitted if the language linearized method overriding based on the order of inheritance declarations, such as in Scala traits.) Dynamic super calls are a generalization of ordinary super calls, when the qualifier class is a required class.

Adding multimethods. Suppose that after we have defined these classes, we wish to add a new method that operates over the AST. For instance, we may want to check that variables are declared before they are used (assuming a variable declaration statement). Since CZ has multimethods, such a method defCheck() could be defined in a helper class, rather than in the classes of the original hierarchy:

```
class DefChecker {
   void defCheck(ASTNode n) { ... }
   void defCheck(ASTNode@Var v) { ... }
   void defCheck(ASTNode@Plus p) { ... }
   void defCheck(ASTNode@Num n) { ... }
}
```

Note that the programmer would *only* have to define cases for ASTNode, Num, Var and Plus; she need not specify what method should be called when an object has a combination of these types—such a situation cannot occur (as there are no diamonds).

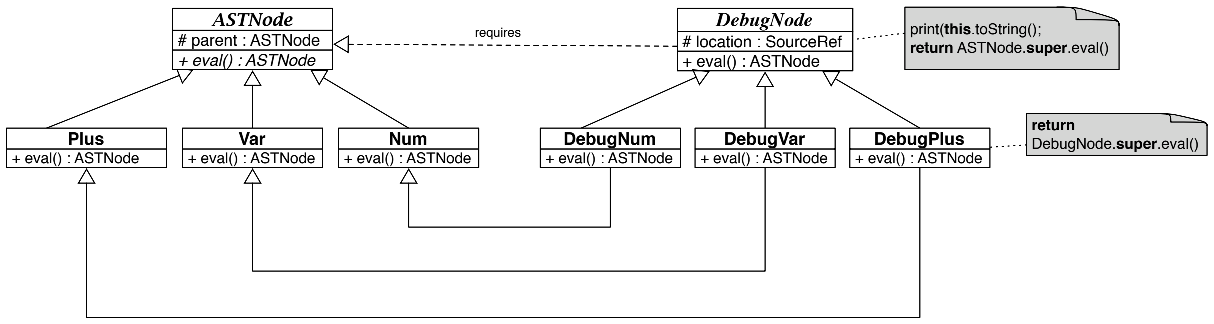

Figure 3. The AST node example in CZ. Abstract classes and abstract methods are set in italic.

Discussion. The examples illustrate that subtyping allows substitutability; subclassing, in addition to providing inheritance, defines semantic alternatives that may not overlap (such as Num, Var and Plus in the example above). Because they do not overlap, we can safely perform an unambiguous "case" analysis on them—that is, multimethod dispatch. In other words, dispatch in our system is analogous to case-analyzing datatypes in functional programming (e.g. ML, Haskell).

Alternative designs. Traits could be used to express this example, but as previously mentioned (Sect. 3.2), the lack of state results in an information hiding problem with accessors. Also, as we have noted, stateful traits do not address the object initialization problem.

Mixins could express some aspects of the class hierarchy for this example, but as previously mentioned, subclassing—or even subtyping—cannot be specified among (standard-style) mixins [12, 24, 4]. For this reason, mixins do not integrate well with multimethods. For details on this issue, see [30].

Using Java-style single inheritance would be unwieldy. A forwarding pattern would have to be used, along with the definition of at least four new interfaces [30]. Additionally, accessors would have to be public, since they would have to be defined in an interface (the only way to achieve any form of multiple inheritance in Java-like languages). Finally, the Visitor design pattern would have to be used in order to allow new operations to be defined.

6. CZ Design

In this section, we give informal details of the typechecking rules in CZ, and provide an intuition as to why typechecking is modular. In Sect. 8 we formalize CZ and provide a detailed argument showing its modularity.

6.1 Multiple Inheritance

CZ places the following constraints on class definitions:

C1. If a class C extends D_1 and D_2 then there *must not* exist some E, other than Object, such that both D_1 and D_2 are subclasses of E (the no-diamond property).

C2. If class C extends D_1 and D_2 and the unspecialized method m is defined or inherited by both D_1 and D_2 then C must also define the unspecialized method m. Also, if method m with specializer B is defined or inherited by both D_1 and D_2, then C must also define either (1) m with specializer B', where B is a subclass of B', or (2) an unspecialized method m.

Additionally, the calculus assumes an elaboration phase that translates method names to qualified names, using the name of the class where the method was first defined; consequently, methods have a unique point of introduction. That is, in the calculus, two classes only share a method name if it exists in a common superclass or common required class. This convention prevents a name clash if two methods in unrelated classes A and B coincidentally have the same name and a third class inherits from both A and B.[6] (Of course, an implementation of the language would have to provide a syntactic way for disambiguating methods that accidentally have the same name; this could be achieved through rename directives, e.g., Eiffel [32], or by using qualified names, e.g., C# interfaces and C++.)

We have already described the reason for condition *C1*, the no-diamond property. We make a special case for the class Object—the root of the inheritance hierarchy, since every class automatically extends it. (Otherwise, a class could never extend two unrelated classes—the existence of Object would create a diamond.[7]) Note that this does not result in the object initialization problem, because Object has only a no-argument constructor. Also, this condition does not preclude a class from inheriting from two concrete classes if this does not form a diamond.

Condition *C2* ensures that if a class C inherits two identical method definitions, either specialized or unspecialized, this will not lead to an ambiguity; in such a case, C must provide an overriding definition.

[6] Incidentally, this is not the convention used in Java interfaces, but is that of C#.

[7] An alternative design would be to make every abstract class implicitly require Object and every concrete class implicitly extend Object. The problem with this design is that it would prevent a class from extending two concrete classes, as a diamond with Object at the root would result.

6.2 Multiple Dispatch

CZ allows methods to be specialized on a subclass of the first argument's class type. Any unspecialized method (i.e., an ordinary method) defined or inherited by a class C may be specialized within C, provided the method's first argument type is not Object. In general, typechecking multimethods has two components: *exhaustiveness checking* (i.e., the provided cases provide full coverage of the dispatch hierarchy) and *ambiguity checking* (i.e., when executing a given method call, there is a unique most specific applicable method).

Since the core calculus of CZ does not include abstract methods, exhaustiveness is automatically handled; we need only ensure there are no ambiguities. (Abstract methods are orthogonal to our considerations, as they are adequately handled by previous work [37, 16, 35].) We adapt previous techniques for ambiguity checking [37, 16, 35]:

M1. A method $D\ m(A@B\ x, \overline{D}\ \overline{x})$ may only be defined if in class C if all of the following hold:

1. $A \neq$ Object
2. $B \neq A$
3. B is a subclass of A
4. a method $D\ m(A, \overline{D})$ is defined or inherited by C.

These conditions, together with with *C1* and *C2*, ensure the absence of ambiguity. In particular, since B must be a strict subclass of A, condition *C1* ensures that if method $m(A@B')$ is defined or inherited by C, then either $B \preceq B'$ or $B' \preceq B$ (since $A \neq$ Object and inheritance diamonds are disallowed). Condition *C2* ensures that if $B = B'$, there exists a disambiguating definition $m(A@B'')$ within class C, where $B \preceq B''$. Together, these properties ensure that if a program typechecks, a unique most-specific applicable method always exists.

Previous work either disallowed inheritance across module boundaries [37] or did not permit interfaces to be specializers [16]. In CZ, we can remove each of these restrictions, due to the absence of inheritance diamonds.

In Sect. 8, we describe a generalization of multimethods to the n-argument case and describe why this generalization does not introduce new typechecking issues.

6.3 Dynamically-Dispatched Super Calls

As illustrated in Sect. 5, CZ includes dynamically-dispatched super calls. When A requires B (i.e., A is acting as a mixin extension of B), then within A, a call of the form B.super is dynamically resolved, similar to super calls in traits. Other super calls (i.e., those where the qualifier is a parent class) have the same semantics as that of Java.

6.4 Discussion

Extensions. External methods (also known as open classes), could also be added to CZ, without sacrificing modular typechecking. External methods are more general than multimethods, since they allow new classes to override an existing external method. For details on the typechecking issues that arise, see our previous work [30].

It would also be possible to combine our solution with existing techniques for dealing with the object initialization and modular multiple dispatch problems. A programmer could specify that a class C, whose constructor takes no arguments, may be the root of a diamond hierarchy. Then, we would use the Scala solution for ensuring that C's constructor is called only once. To solve the multiple dispatch problem, if methods $m(B)$ and $m(B')$ specialize $m(C)$, the typechecker would ensure that m contained a disambiguating definition for $(B \wedge B')$—the JPred and Fortress solutions.

Finally, the language could include syntactic sugar to ease the definition of concrete classes. If C requires B, and both C and B have no-argument constructors, the compiler could automatically generate a class C\$concrete that extends both C and B; programmers could then more easily define multimethods that dispatch on C\$concrete.

Encapsulation and the diamond problem. As noted by Snyder, there are two possible ways to view inheritance: as an internal design decision chosen for convenience, or as a public declaration that a subclass is specializing its superclass, thereby adhering to its semantics [47].

Though Snyder believes that it can be useful to use inheritance without it being part of the external interface of a class, we argue that the second definition of inheritance is more appropriate. In fact, if inheritance is being used merely out of convenience (e.g., Stack extending Vector in the Java standard library), then it is very likely that *composition* is a more appropriate design [10]. For similar reasons, we do not believe a language should allow inheritance without subtyping—e.g., C++ private inheritance—as this can always be implemented using a helper class whose visibility is restricted using the language's module system.

Nevertheless, if one takes the view that inheritance choices should *not* be visible to subclasses, a form of the diamond problem can arise in CZ. In particular, suppose class D extends B and C, C extends A, and B extends Object—a valid hierarchy (recall that condition *C1* makes a special exception for diamonds involving Object). Now suppose that B is changed to extend A, and the maintainer of B is unaware that class D exists. Now A, B and C typecheck, but D does not. Thus, the use of inheritance can invalidate subclasses, which violates Snyder's view of encapsulation.

This situation highlights the fact that, in general, requires should be favored over extends if a class is intended to be reused.

7. Real-World Examples

In this section, we present real-world examples (in both C++ and Java) that suggest that multiple inheritance, and diamond inheritance in particular, can be useful for code reuse. We also describe how these examples can be expressed in CZ.

7.1 C++ Examples

We examined several open-source C++ applications in a variety of domains and found many instances of virtual inheritance and inheritance diamonds. Here we describe inheritance diamonds in two applications: Audacity[8] and Guikachu.[9]

Audacity. Audacity is a cross-platform application for recording and editing sounds. One of its main storage abstractions is the class BlockedSequence (not shown), which represents an array of audio samples, supporting operations such as cut and paste. A BlockedSequence is composed of smaller chunks; these are objects of type SeqBlock, depicted in Fig. 5 (a). One subclass of SeqBlock is SeqDataFileBlock, which stores the block data on disk. One superclass of SeqDataFileBlock is ManagedFile, an abstraction for temporary files that are de-allocated based on a reference-counting scheme. Since both ManagedFile and SeqBlock inherit from Storable (to support serialization), this forms a diamond with Storable at the top.

This particular diamond can be easily re-written in CZ (Fig. 5 (b)), since the sides of the diamond (SeqBlock and ManagedFile) are already abstract classes. (Compare to the example in Fig. 2, where new concrete classes had to be defined for the sides of the diamond.) Here, we simply change the top two virtual inheritance edges to requires edges, and make SeqDataFileBlock inherit from Storable directly. This may even be a preferable abstraction; while in the original hierarchy SeqDataFileBlock is serializable by virtue of the fact that SeqBlock is serializable, in the new hierarchy we are making this relationship explicit.

Guikachu. Guikachu is a graphical resource editor for the GNU PalmOS SDK. It allows programmers to graphically manipulate GUI elements for a Palm application in the GNOME desktop environment. In this application, we found 10 examples of diamonds that included the classes CanvasItem, WidgetCanvasItem, and ResizeableCanvasItem. CanvasItem is an abstract base class that represents items that can be placed onto a canvas, while objects of type WidgetCanvasItem and ResizeableCanvasItem are a type of widget or are resizeable, respectively.

Figure 6(a) shows two of these 10 diamonds, formed by TextFieldCanvasItem and PopupTriggerCanvasItem, respectively. The hierarchy was likely designed this way because there exist GUI elements that have only one of the two properties. For instance, GraffitiCanvasItem and LabelCanvasItem (not shown) are not resizeable, but they are widgets. In contrast, the class FormCanvasItem (not shown) is resizeable, but is not a widget.

In this application, we also observed the use of the C++ virtual inheritance initializer invocation mechanism: TextFieldCanvasItem (for instance) directly calls the initializer of CanvasItem, its grandparent. As previously described, when initializing TextFieldCanvasItem, the initializer calls from WidgetCanvasItem and ResizeableCanvasItem to CanvasItem are ignored. In this application, the initializers happen to all perform the same operation, but this invocation semantics could introduce subtle bugs as the application evolves.

The corresponding CZ class hierarchy is displayed in Fig. 6 (b); note its similarity to that of Fig. 5 (b). Essentially, the virtual inheritance is replaced with requires and each of the classes at the bottom of the diamond inherit from all three of WidgetCanvasItem, ResizeableCanvasItem, *and* CanvasItem. The CZ design has the advantage that constructor calls do not occur more than one level up the hierarchy, and no constructor calls are ignored.

This example illustrates how a program could be translated from C++-style multiple inheritance to CZ-style. In particular, virtual inheritance would be replaced by requires, and new concrete classes would be defined as necessary (changing instantiations of the now-abstract class to instantiations of the new concrete class). Note that constructor calls can be easily generated for the new concrete classes, as C++ requires a call from the bottom of the diamond to the top of the diamond when virtual inheritance is used (such a constructor call would be necessary for the new concrete class, as it would directly extend the class at the top of the diamond).

Discussion. It would be interesting to extend the C++ study and perform a more systematic study of the nature of inheritance diamonds, quantifying how often new abstract classes would have to be defined (i.e., how often concrete classes appear on the "sides" of the diamond). One could also determine how often the initializer problem occurs in real code.

However, note that the multimethod problem will always arise in a multiple inheritance situation, even if a programmer never actually creates an inheritance diamond, and (as noted in Section 4) even if a language includes the more benign feature of multiple interface inheritance (e.g., Java-like languages).

7.2 Java Example: Eclipse JDT

The Eclipse JDT (Java Development Tools) is an example of where multiple inheritance could be useful for Java programs. In the JDT, every AST node contains *structural properties*. A node's structural properties allow uniform access to its components. For example, DoStatement has 2 fields of type StructuralPropertyDescriptor: EXPRESSION_PROPERTY and BODY_PROPERTY. To get the expression property of a DoStatement object, the programmer may call ds.getExpression() or ds.getStructuralProperty(DoStatement.EXPRESSION_ PROPERTY). Structural prop-

[8] http://audacity.sourceforge.net/

[9] http://cactus.rulez.org/projects/guikachu/

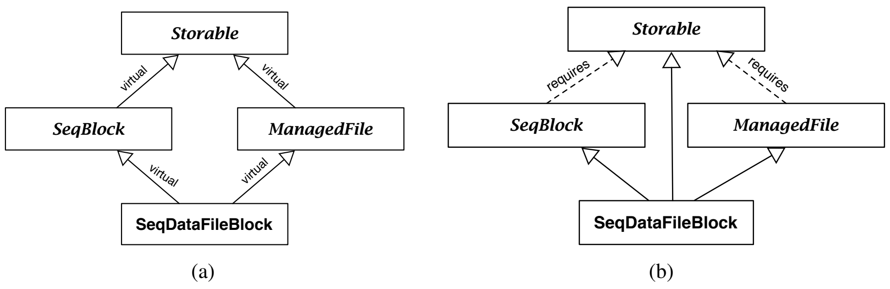

Figure 5. An inheritance diamond (a) in the Audacity application, and (b) the re-written class hierarchy in CZ. Abstract classes are set in italic.

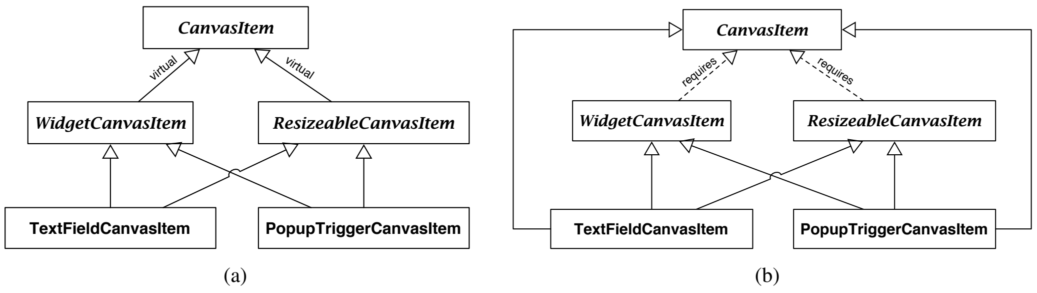

Figure 6. Two inheritance diamonds in the Guikachu application (a) and re-written in CZ (b). Abstract classes are set in italic.

erty descriptors are often used to specify how AST nodes change when a refactoring is performed.

Through inspection of the JDT code, we found that there was a great deal of duplication among the code for getting or setting a node property using the structural property descriptors. For example, 19 AST classes (e.g., AssertStatement and ForStatement) have getExpression/setExpression properties. As a result, in the method internalGetSetChildProperty (an abstract method of ASTNode), there are 19 duplications of the following code:

```
if (property == EXPRESSION_PROPERTY) {
    if (get) {
        return getExpression();
    } else {
        setExpression((Expression) child);
        return null;
    }
} else if (property == BODY_PROPERTY) {
    ... // code for body property
  }
}
```

Additionally, there are duplicate, identical definitions of the EXPRESSION_PROPERTY field. Without a form of multiple inheritance, however, it is difficult to refactor this code into a common location—DoStatement, for example, already has the superclass Statement. With multiple inheritance, the programmer could create an abstract helper class ExprPropertyHelper that requires ASTNode. This new class would contain the field definition and an override of internalGetSetChildProperty. DoStatement would then inherit from both Statement and ExprPropertyHelper and would have the following body for internalGetSetChildProperty:

```
if (property == BODY_PROPERTY) {
    ... // code for body property
} else {
    return ExprPropertyHelper.super.
            internalGetSetChildProperty(property, get, child);
}
```

Additionally, this is a scenario where multiple dispatch would be beneficial. The framework defines various visitors for traversing an AST; these could be omitted in favor of multimethods, which are more extensible.

Overall, our real-world examples suggest that multiple inheritance can be useful, and that even diamond inheritance is used in practice. We have shown that the inheritance diamonds can be easily translated to CZ and that the resulting designs offer some benefits over the original ones. In particular, CZ avoids the problem of ignored constructor calls in C++, while providing more flexible code reuse than with single inheritance.

8. Formal System

In this section, we describe the formalization of CZ, which is based on Featherweight Java (FJ) [28]. We use the same conventions as FJ; $\overline{D}$ is shorthand for the (possibly empty) list $D_1, \ldots, D_n$, which may be indexed by D_i.

The grammar of CZ is presented in Fig. 7. Modifications to FJ are highlighted. Class declarations may extend or require a *list* of classes. There is also a new syntactic form for multimethods; such methods include a specializer on the first argument type.

We relax the FJ convention that a class may not define two methods with the same name; such a case is permitted as long as one method or both methods have specializers (which must be distinct). The type of all other arguments and the return type must remain the same.

To simplify the formal system, we assume that all methods have at least one argument. A dummy object can be used to simulate a no-argument method.

To avoid syntax for resolving different superclass constructors, all fields, including those inherited from superclasses, must be initialized in the constructor.

Aside from dynamically-dispatched super calls, and the removal of casts (they are orthogonal to our goals), CZ expression forms are identical to those of FJ. For simplicity, we have not modeled ordinary super calls in our calculus, as this has been considered by others (e.g., [24, 38]) and is orthogonal to the issues we are considering. Therefore, the class qualifier of a super call must be a required class.

We have added a new subclass ('$\preceq$') judgement (Fig. 8), which is the reflexive, transitive closure of extends. The subtype judgement ($<:$) is extended to include the requires relationship. Subclassing implies subtyping, and if class A requires B then $A <: B$, but $A \not\preceq B$. In CZ, the requires relation is not transitive; subclasses must either require or extend the required class, which is enforced by the typechecking rules. Subtyping allows A be used in place of B, which is in contrast to Scala; Scala only allows such a substitution for the this reference within a class.

The auxiliary judgements for typechecking appear after the typechecking and evaluation rules, in Fig. 12. We will describe each of these when describing the rules that use them.

Static Semantics. The rules for typechecking expressions are in Fig. 9. The rule for method invocations, T-INVK, is the same as that in FJ. However, the auxiliary judgement it uses, *mtype*, is different.

The CZ judgement *mtype* (Fig. 12) has an additional rule as compared to FJ; it performs a lookup of methods from required classes, in the case that the method does not exist in the class itself or superclasses. This judgement considers only unspecialized methods.

The rule T-SUPER-INVK checks the dynamically-dispatched super call described in Sect. 6. Essentially, for a call of the form this.B.super.$m(\overline{e})$, where this : C_0, instead of looking up $mtype(m, C_0)$, we look up $mtype(m, B)$, where B is a required class of C_0.

The rule T-NEW has one additional premise as compared to FJ: the requires clause must be empty. This ensures that the class is concrete and can be instantiated, which in turn ensures the soundness of the subtyping relation induced by requires.

Rules for typechecking methods are displayed in Fig. 10. The rule T-METHOD checks unspecialized methods, and uses the *override* auxiliary judgement (which is unchanged from FJ). In this rule, we check that method m is a valid override of the same (unspecialized) method in all superclasses and required classes.

T-MULTI-METHOD checks specialized methods. The first two premises are the same as that of T-METHOD. Premises (3), (4) and (5) check conditions 1, 2, and 3, respectively, of constraint *M1*. Premise (6) checks condition 4 of *M1*; it ensures C defines or inherits an unspecialized method with type $(A, \overline{B}) \rightarrow B$, where A is the static type being specialized.

The T-CLASS rule (Fig. 11) checks class definitions. Premises (1–3) are straightforward generalizations of the corresponding premises in FJ. Premises (4) and (5) ensure that requires is propagated down each level of the inheritance hierarchy; the extending class must either extend or require its parents' required classes. Premise (6) specifies that a subclassing diamond cannot occur, except for the case of Object (condition *C1*). Finally, premise (7) enforces condition *C2*, ensuring that if C inherits two methods m with the same first argument B' (or two unspecialized methods m), then C provides an overriding definition for m. This premise uses the $methodDef(m, D_i, B')$ auxiliary judgement: a derivation of *methodDef* exists if D_i defines or inherits a method with specializer B' or first argument type B'. This premise, as well as the *methodDef* judgement, uses the notation $\langle A@\rangle$ to specify either a specialized or unspecialized method (i.e., the $A@$ part is optional).

Dynamic Semantics. The evaluation rules and auxiliary judgements are presented in Fig. 13. Most of the rules are similar to FJ, with the exception of E-INVK and E-SUPER-INVK. E-INVK passes the dynamic type of the method's first argument as an additional argument to *mbody*, which we describe below. E-SUPER-INVK uses the auxiliary judgement $super(C, D)$, which finds the first superclass of the class C that is also a subclass of D. Then, *mbody* is called on the result of the *super* call.

The main changes to the dynamic semantics are encoded in the auxiliary judgements *mbody*, *dispatch*, and *match*. The *mbody* judgement has one additional argument as compared with FJ, to (potentially) dispatch on the method's first argument. This judgement simply extracts the arguments and method body from the result of the *dispatch* judgement, which contains the actual dispatch logic.

Method dispatch is performed in two steps. The first *dispatch* rule uses the *matchArg* judgement to search for a method defined in C that is applicable for D, the dynamic

$$\begin{array}{lll}
\text{Declarations} & L ::= & \mathsf{class}\ C\ \mathsf{extends}\ \overline{C}\ \mathsf{requires}\ \overline{C}\ \{\overline{C}\,\overline{f};\ K\,\overline{M}\} \\
\text{Constructors} & K ::= & C(\overline{C}\,\overline{f})\ \{\mathsf{this}.\overline{f} = \overline{f};\} \\
\text{Methods} & M ::= & C_0\ m(\overline{C}\,\overline{x})\ \{\mathsf{return}\ e;\} \mid C_0\ m(C@C'\ x, \overline{C}\,\overline{x})\ \{\mathsf{return}\ e;\} \\
\text{Expressions} & e ::= & x \mid e.f \mid e.m(\overline{e}) \mid e.C.\mathsf{super}.m(\overline{e}) \mid \mathsf{new}\ C(\overline{e})
\end{array}$$

Figure 7. CZ grammar

Subclassing $\boxed{C \preceq D}$

$$\frac{}{C \preceq C} \qquad \frac{C \preceq D \qquad D \preceq E}{C \preceq E} \qquad \frac{CT(C) = \mathsf{class}\ C\ \mathsf{extends}\ D_1, \ldots, D_n \cdots \{\ \ldots\ \}}{C \preceq D_i}$$

Subtyping $\boxed{C <: D}$

$$\frac{C \preceq D}{C <: D} \qquad \frac{C <: D \qquad D <: E}{C <: E} \qquad \frac{CT(C) = \mathsf{class}\ C\ \mathsf{extends}\ \ \overline{D}\ \mathsf{requires}\ E_1, \ldots, E_n\ \ \{\ \ldots\ \}}{C <: E_i}$$

Figure 8. Subclassing ($\preceq$) and subtyping ($<:$) judgement

$\boxed{\Gamma \vdash e : C}$

$$\text{(T-VAR)}\ \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \text{(T-FIELD)}\ \frac{\Gamma \vdash e_0 : C_0 \qquad C_0 <: D \qquad \mathit{fields}(D) = \overline{C}\,\overline{f}}{\Gamma \vdash e_0.f_i : C_i}$$

$$\text{(T-INVK)}\ \frac{\Gamma \vdash e_0 : C_0 \qquad \mathit{mtype}(m, C_0) = \overline{D} \to C \qquad \Gamma \vdash \overline{e} : \overline{C} \qquad \overline{C} <: \overline{D}}{\Gamma \vdash e_0.m(\overline{e}) : C}$$

$$\text{(T-SUPER-INVK)}\ \frac{\Gamma \vdash e_0 : C_0 \qquad \mathsf{class}\ C_0\ \mathsf{extends}\ \overline{D}_0\ \mathsf{requires}\ B, \overline{E} \qquad \mathit{mtype}(m, B) = \overline{D} \to C \qquad \Gamma \vdash \overline{e} : \overline{C} \qquad \overline{C} <: \overline{D}}{\Gamma \vdash e_0.B.\mathsf{super}.m(\overline{e}) : C}$$

$$\text{(T-NEW)}\ \frac{\mathit{fields}(C) = \overline{D}\,\overline{f} \qquad \Gamma \vdash \overline{e} : \overline{C} \qquad \overline{C} <: \overline{D} \qquad \mathsf{class}\ C\ \mathsf{requires}\ \bullet}{\Gamma \vdash \mathsf{new}\ C(\overline{e}) : C}$$

Figure 9. Expression typing

type of the method's first argument. This latter judgement considers both specialized and unspecialized methods (via the $\langle B@\rangle$ notation). If such a definition exists, it is returned; otherwise, a set of methods $\overline{M}_E$ is composed by calling *dispatch* on each of C's superclasses. Then, the unique most specific method that is applicable for argument D is selected. Note the asymmetric dispatch semantics; if an appropriate method does not exist in C, its superclasses are searched before dispatching on the argument type.

Constructors. As in FJ, CZ does not contain state, and thus constructor definitions are trivial. However, a full implementation would have to ensure that when C extends A, B and A requires B, when creating a C object, its B-part must be initialized before its A-part. Otherwise, this could result in fields being accessed before they exist, since A is permitted to access B's fields.

Generalizations. To generalize multimethod dispatch to n arguments, in the static semantics, we would extend premise (7) of T-CLASS, which corresponds to condition *C2*. In particular, this premise would ensure that if C inherits two method definitions that have identical specializer *lists* (or argument types), then an overriding definition exists in C. In particular, the quantifier $\forall B'$ would become $\forall \overline{B}'$ in this premise.

For the dynamic semantics, we would change *dispatch* and *matchArg* to take a *list* of dynamic types of the objects passed to the method in question. The latter judgement would then select the unique most specific method such that each of its specializers (or argument types, if there is no specializer) is a supertype of the corresponding dynamic type. This could be implemented by first creating a candidate list of all applicable methods, then selecting the most specific one.

$\boxed{M\ \textbf{ok in}\ C}$

$$\frac{\begin{array}{c}❶\ \overline{x} : \overline{B}, \mathsf{this} : C \vdash e_0 : E_0 \qquad ❷\ E_0 <: B \qquad ❸\ \mathsf{class}\ C\ \mathsf{extends}\ \overline{D}\ \mathsf{requires}\ \overline{E} \\ ❹\ \forall i\,.\,override(m,\ D_i\ , \overline{B} \to B) \qquad ❺\ \forall i.\,override(m, E_i, \overline{B} \to B)\end{array}}{B\ m(\overline{B}\ \overline{x})\ \{\ \mathsf{return}\ e_0;\}\ \textbf{ok}\ \text{in}\ C}\ (\text{T-METHOD})$$

$$\frac{\begin{array}{c}❶\ x : B', \overline{x} : \overline{B}, \mathsf{this} : C \vdash e_0 : E_0 \qquad ❷\ E_0 <: B \\ ❸\ A \neq \mathsf{Object} \qquad ❹\ B_0 \neq A \qquad ❺\ B_0 \preceq A \qquad ❻\ mtype(m, C) = (A, \overline{B}) \to B\end{array}}{B\ m(A@B_0\ x, \overline{B}\ \overline{x})\ \{\ \mathsf{return}\ e_0;\}\ \textbf{ok}\ \text{in}\ C}\ (\text{T-MULTI-METHOD})$$

Figure 10. Specialized and unspecialized method typing

Declaration Typing $\boxed{\text{L}\ \textbf{ok}}$

$$\frac{\begin{array}{c}❶\ \forall i.\,fields(D_i) = \overline{F}_i\ \overline{g}_i \qquad ❷\ K = C(\ \overline{F}_i\ \overline{g}_i\ ^{i\in 1..n}\ , \overline{C}\ \overline{f})\{\ \mathsf{this}.\overline{g}_i = \overline{g}_i\ ^{i\in 1..n}\ ;\ \mathsf{this}.\overline{f} = \overline{f}\} \\ ❸\ \overline{M}\ \textbf{ok}\ \text{in}\ C \qquad ❹\ \forall i.\,\mathsf{class}\ D_i\ \mathsf{requires}\ E',\ \text{implies}\ \exists k.\,D_k \preceq E'\ \text{or}\ \exists k.\,E_k \preceq E' \\ ❺\ \forall i.\,\mathsf{class}\ E_i\ \mathsf{requires}\ E'',\ \text{implies}\ \exists k.\,D_k \preceq E''\ \text{or}\ \exists k.\,E_k \preceq E'' \\ ❻\ \forall i.\forall j \neq i.\,\nexists D' \neq \mathsf{Object}.\,D_i \preceq D'\ \text{and}\ D_j \preceq D' \\ ❼\ \forall m.\forall B'.\,\big(\exists i.\,\exists j \neq i.\,methodDef(m, D_i, B')\ \text{and}\ methodDef(m, D_j, B')\big),\ \text{implies} \\ \exists B''.\,B' \preceq B''\ \text{and}\ B_0\ m(\langle A@\rangle B''x, \overline{B}\ \overline{x}) \in \overline{M}\end{array}}{\mathsf{class}\ C\ \mathsf{extends}\ \overline{D}\ \mathsf{requires}\ \overline{E}\ \ \{\overline{C}\ \overline{f}; K\ \overline{M}\}\ \textbf{ok}}\ (\text{T-CLASS})$$

Figure 11. Class typing

We observe from the form of these judgements (*dispatch* and *matchArg*) that there could be two ways in which more than one method applies: (1) within a single argument position, more than one method applies and none is more specific than the others, (2) one method is more specific at one argument position and another method is more specific at some other argument position. Note that, in the absence of appropriate typechecking, either situation can arise, *regardless* of whether dispatch is performed on n arguments or just two arguments. Premise (6) of T-MULTI-METHOD ensures that there exists at least one method in the candidate list.

CZ's static semantics—in particular, conditions *C1* and *C2*—ensure that the first situation cannot arise. As a consequence of condition *C1* (the no-diamond restriction), for a particular argument position k, all specializer types in the candidate method list are mutually comparable (via subclassing). That is, if, at argument position k, method m_i has specializer C_i and m_j has specializer C_j, then either $C_i \preceq C_j$ or $C_j \preceq C_i$ (for $i \neq j$). This is because both types must be superclasses of D_k (the dynamic type at position k) and they also must both be subtypes of C_k, the static type at position k. Since C_k cannot be Object, C_i and C_j must be comparable types.

C2 ensures that there exists an argument position k such that $C_i \neq C_j$. We observe that two methods in the candidate list could only have identical specializer types (or argument types) if class C inherited two such methods (as there is a syntactic restriction against such a definition directly in C). But, *C2* ensures that if such a situation were to occur, that C would have an overriding definition. Therefore, the first *dispatch* rule would apply and superclasses would not be considered.

Finally, situation (2) cannot occur, due our assymmetric dispatch semantics. To change to symmetric dispatch (described in Sect. 4.3), we need only add an additional premise to T-CLASS. This new premise would ensure that there is no combination of receiver and argument tuples such that more than one method would apply, using the same modular check implemented in, for instance, MultiJava and EML [17, 35]. Note that the dynamic semantics would not need to change, since this new premise would ensure that asymmetric and symmetric dispatch produce the same result.

8.1 Modularity

Here, we describe the conditions under which a class-based system is modular when there is no explicit module system. We argue informally that typechecking in CZ is modular based on the structure of the typechecking rules. (The other languages we have mentioned also perform modular typechecking by this definition.)

Conditions for modular typechecking.

1. Checking a class signature C with methods $\overline{M}$ should only require examining: (a) signatures of methods tran-

$$\boxed{\mathit{fields}(C) = \overline{C}\,\overline{f}}$$

$$\frac{}{\mathit{fields}(\textsf{Object}) = \bullet} \qquad \frac{\textsf{class } C \textsf{ extends } D_1, \ldots, D_n \textsf{ requires } \overline{E}\ \{\overline{C}\,\overline{f}; K\,\overline{M}\} \qquad \forall i\,.\mathit{fields}(D_i) = \overline{B}_i\,\overline{g}_i}{\mathit{fields}(C) = \overline{B}_i\,\overline{g}_i^{\;i\in 1..n}, \overline{C}\,\overline{f}}$$

$$\boxed{\mathit{mtype}(m, C) = \overline{D} \to D}$$

$$\frac{\begin{array}{c}\textsf{class } C\ \cdots\ \{\overline{C}\,\overline{f}; K\,\overline{M}\} \\ B\ m(\overline{B}\,\overline{x})\ \{\textsf{return } e;\} \in \overline{M}\end{array}}{\mathit{mtype}(m, C) = \overline{B} \to B} \qquad \frac{\begin{array}{c}\textsf{class } C \textsf{ extends } \overline{D} \textsf{ requires } \overline{E}\ \{\overline{C}\,\overline{f}; K\,\overline{M}\} \\ \exists k\,.\mathit{mtype}(m, D_k) = \overline{B} \to B\end{array}}{\mathit{mtype}(m, C) = \overline{B} \to B} \qquad \frac{\begin{array}{c}\textsf{class } C \textsf{ extends } \overline{D} \textsf{ requires } \overline{E}\ \{\overline{C}\,\overline{f}; K\,\overline{M}\} \\ \exists k.\,\mathit{mtype}(m, E_k) = \overline{B} \to B\end{array}}{\mathit{mtype}(m, C) = \overline{B} \to B}$$

$$\boxed{\mathit{methodDef}(m, C, B)}$$

$$\frac{\begin{array}{c}\textsf{class } C\ \cdots\ \{\overline{C}\,\overline{f}; K\,\overline{M}\} \\ B_0\ m(\langle B'@\rangle B\ x, \overline{B}\,\overline{x})\ \{\textsf{return } e;\} \in \overline{M}\end{array}}{\mathit{methodDef}(m, C, B)} \qquad \frac{\begin{array}{c}\textsf{class } C \textsf{ extends } \overline{D} \cdots\ \{\overline{C}\,\overline{f}; K\,\overline{M}\} \\ B_0\ m(\langle B'@\rangle B\ x, \overline{B}\,\overline{x})\ \{\textsf{return } e;\} \notin \overline{M} \\ \exists k.\,\mathit{methodDef}(m, D_k, B)\end{array}}{\mathit{methodDef}(m, C, B)}$$

$$\boxed{\mathit{override}(m, D, \overline{C} \to C_0)}$$

$$\frac{\mathit{mtype}(m, D) = \overline{D} \to D_0 \text{ implies } \overline{C} = \overline{D} \text{ and } C_0 = D_0}{\mathit{override}(m, D, \overline{C} \to C_0)}$$

Figure 12. CZ typechecking auxiliary judgements

sitively overridden or specialized in $\overline{M}$, (b) signatures of methods transitively overridden or specialized by C's inherited methods, (c) class declarations of C's supertypes.

2. Checking the definition of a particular method m (possibly specialized with class C) should only require examining: (a) the declarations of C and its supertypes, (b) the signature of the method that m specializes, and (c) the signatures of methods called by m.

By inspection, checking a class definition C obeys condition 1. Each premise examines only superclasses or required classes, and there is, for example, no search for multimethods with first argument type C.

Checking a method definition m is also modular. If m is an unspecialized method, the only generalization to the typechecking rule is additional *override* checks, which are modular. On the other hand, when a specialized method is checked, we simply ensure that the specializer has the appropriate relationship to its static type (which may not be Object), and call $\mathit{mtype}(m, C)$. Since this judgement only searches up the subtype hierarchy, it is modular.

8.2 Type Safety

We prove type safety using the standard progress and preservation theorems, with a slightly stronger progress theorem than that of FJ, due to the omission of casts. Note that in our system, type safety implies that all method calls are unambiguous, as the *dispatch* and *match* judgements require that there be a unique most-applicable method. We describe below a brief outline of the proof of type safety and refer the reader to [31] for further details.

Theorem 8.1 (Preservation). *If* $\Gamma \vdash e : C$ *and* $e \longmapsto e'$*, then* $\Gamma \vdash e' : C'$ *for some* $C' <: C$.

The proof of preservation is relatively straightforward and is similar to the proof of FJ. We make use of an auxiliary lemma (not shown) that proves that *mtype* returns a unique value. The proof of this lemma makes use of the convention that method introductions are unique.

Theorem 8.2 (Progress). *If* $\cdot \vdash e : C$ *then either* e *is a value or there is an* e' *with* $e \longmapsto e'$.

The proof of progress is slightly more complex. The proof requires the following lemma:

Lemma 8.1. *If* $\mathit{mtype}(m, C) = (B_0, \overline{B}) \to B$ *and* $\Gamma \vdash \textsf{new}\, C(\overline{e}) : C$ *and* $B' \preceq B_0$ *then* $\mathit{dispatch}(m, C, B') = M$*, for some* M.

However, unlike in FJ, we cannot prove this lemma by induction on the derivation of *mtype*, since for the inductive step, we do not have a derivation $\Gamma \vdash \textsf{new}\ D_k(\overline{e}) : D_k$. Instead, we make use of two auxiliary lemmas:

Evaluation $\boxed{e \longmapsto e'}$

(E-FIELD)
$$\frac{fields(C) = \overline{B}\,\overline{f}}{(\text{new } C(\overline{e})).f_i \longmapsto e_i}$$

(E-INVK)
$$\frac{mbody(m, C, D) = (x, \overline{x}).e_0}{(\text{new } C(\overline{e})).m(\text{new } D(\overline{e}'), \overline{d}) \longmapsto [\text{new } C(\overline{e})/\text{this}, \text{new } D(\overline{e}')/x, \overline{d}/\overline{x}]\; e_0}$$

(E-SUPER-INVK)
$$\frac{super(C, E) = C' \qquad mbody(m, C', D) = (x, \overline{x}).e_0}{(\text{new } C(\overline{e})).E.\text{super}.m(\text{new } D(\overline{e}'), \overline{d}) \longmapsto [\text{new } C(\overline{e})/\text{this}, \text{new } D(\overline{e}')/x, \overline{d}/\overline{x}]\; e_0}$$

$$\frac{e_0 \longmapsto e_0'}{e_0.f \longmapsto e_0'.f} \qquad \frac{e_0 \longmapsto e_0'}{e_0.m(\overline{e}) \longmapsto e_0'.m(\overline{e})} \qquad \frac{e_0 \longmapsto e_0'}{e_0.C.\text{super}.m(\overline{e}) \longmapsto e_0'.C.\text{super}.m(\overline{e})}$$

$$\frac{e_i \longmapsto e_i'}{e_0.m(\ldots, e_i, \ldots) \longmapsto e_0.m(\ldots, e_i', \ldots)} \qquad \frac{e_i \longmapsto e_i'}{e_0.C.\text{super}.m(\ldots, e_i, \ldots) \longmapsto e_0.C.\text{super}.m(\ldots, e_i', \ldots)} \qquad \frac{e_i \longmapsto e_i'}{\text{new } C(\ldots, e_i, \ldots) \longmapsto \text{new } C(\ldots, e_i', \ldots)}$$

Auxilliary Judgements

$\boxed{mbody(m, C, D) = \overline{x}.e}$ $\qquad$ $\boxed{super(C, D) = E}$

$$\frac{dispatch(m, C, D) = B_0\; m(\langle B@\rangle B'\; x, \overline{B}\; \overline{x})\; \{\text{return } e;\}}{mbody(m, C, D) = \overline{x}.e} \qquad \frac{\text{class } C \text{ extends } E \qquad E \preceq D}{super(C, D) = E} \qquad \frac{\nexists E' \preceq D.\, \text{class } C \text{ extends } E' \quad C \text{ extends } \overline{B} \qquad \exists k.\, super(C, B_k) = E}{super(C, D) = E}$$

$\boxed{dispatch(m, C, D) = M}$

$$\frac{\text{class } C \cdots \{\overline{C}\;\overline{f}; K\;\overline{M}\} \quad matchArg(m, D, \overline{M}) = M'}{dispatch(m, C, D) = M'} \qquad \frac{\text{class } C \text{ extends } \overline{E} \text{ requires } \overline{F}\; \{\overline{C}\;\overline{f}; K\;\overline{M}\} \qquad \nexists M'.\, matchArg(m, D, \overline{M}) = M' \quad \overline{M}_E = \{M_i \mid dispatch(m, E_i, D) = M_i\} \qquad \exists \text{ unique } M''.\, matchArg(m, D, \overline{M}_E) = M''}{dispatch(m, C, D) = M''}$$

$\boxed{matchArg(m, D, \overline{M}) = M}$

$$\frac{M' = B_0\; m(\langle B@\rangle D\; x, \overline{B}\; \overline{x})\; \{\text{return } e;\} \quad M' \in \overline{M}}{matchArg(m, D, \overline{M}) = M'} \qquad \frac{B_0\; m(\langle B@\rangle D\; x, \overline{B}\; \overline{x}) \notin \overline{M} \qquad \text{class } D \text{ extends } \overline{E} \quad \exists \text{ unique } (k, M').\, matchArg(m, E_k, \overline{M}) = M'}{matchArg(m, D, \overline{M}) = M'}$$

Figure 13. Evaluation rules and auxiliary judgements

Lemma 8.2. *If $\mathcal{D}$ :: $mtype(m, C) = (B_0, \overline{B}) \rightarrow B$ and $\mathcal{D}$ does not contain the rule* MTYPE3 *and $B' \preceq B_0$, then $dispatch(m, C, B') = M$, for some M.*

Lemma 8.3. *If $\Gamma \vdash$ new $C(\overline{e}) : C$ and $C <: D$ and $\mathcal{D}$ :: $mtype(m, D) = \overline{B} \rightarrow B$, then there exist D' and $\mathcal{D}'$ such that $C \preceq D'$ and $\mathcal{D}'$ :: $mtype(m, D') = \overline{B} \rightarrow B$ does not contain the rule* MTYPE3.

Lemma 8.2 is needed because it is the rule MTYPE3 that could result in *mbody* not being defined—it is the only rule that has no *dispatch* counterpart. We use Lemma 8.3 to produce such an *mtype* derivation.

With these lemmas, the rest of the proof of progress is straightforward.

9. Related Work

Here we describe related work that was not previously discussed in Sections 2, 3.2, and 4.2.

As mentioned in Sect. 4.2, JPred [25] and Fortress [3] perform modular multimethod typechecking by requiring that programmers provide disambiguating methods, some of which may never be called. However, we observe that the JPred and Fortress dispatch semantics may be more expressive than that of CZ. In CZ, in the class hierarchy Fig. 2, the abstract class InputStream may not be used as a specializer for Stream, because it is not a subclass of Stream. In contrast, if this hierarchy were expressed in e.g. Fortress a multimethod defined on Stream could be specialized for either InputStream or OutputStream. Note, however, that programmers can achieve a similar effect in CZ by having concrete classes call helper methods (which may themselves perform multiple dispatch) defined on the abstract classes.

Cecil [14, 15] also provides both multiple inheritance and multimethod dispatch, but it does not include constructors (and therefore provides ordinary dispatch semantics for methods acting as constructors), and it performs whole-program typechecking.

Like JPred, the language Half & Half [6] provides multimethod dispatch on Java interfaces. In this language, if there exist specialized method implementations for two incomparable interfaces A and B, the visibility of one of the two interfaces must be package-private. Like System M, this effectively disallows multiple (interface) inheritance across module boundaries (where a package is a module). Half & Half does not consider the problem of multiple inheritance with state.

Pirkelbauer et al have considered the problem of integrating multimethods into C++, which is especially difficult due to existing rules for overload resolution [42]. However, this proposal is not modular; because of the potential for inheritance diamonds, the design requires link-time typechecking.

It is worth noting that multimethods *cannot* be simulated with C# 3.0 "extension methods" or partial classes [34]. The former, extension methods, are merely syntactic sugar and cannot be overridden with a more specific type for the receiver. Partial classes, on the other hand, are simple a compile-time mechanism for splitting a class's definition across multiple compilation units. In particular, compared to multimethods, they have the following limitations: 1) they cannot span assemblies (so if the AST node classes are in a library, some other mechanism would be needed, such as the Visitor pattern); 2) partial classes may not be used to perform dispatch on interfaces, in contrast to multimethods; and 3) typechecking each part of a partial class is not modular, as all parts are composed before typechecking. This last problem can cause compilation errors if two programmers implement a partial class in incompatible ways, so it is unclear what should be the appropriate level of granularity when partial classes are used in a team environment.

10. Conclusions

We have presented a language that solves two major problems caused by inheritance diamonds: object initialization and modular typechecking of multiple dispatch. We have also shown how programs written with traditional multiple inheritance can be converted to programs in our language. We note that though diamonds can still cause encapsulation problems (depending on the definition of encapsulation), this problem can be ameliorated by preferring requires over extends.

We emphasize that although programmers may indeed have to decide ahead of time whether they want to make a class re-usable by making it abstract and by using requires instead of extends—potentially a difficult decision to make—it is a decision the class designer must already make, as a class must be designed carefully if it is to be a unit of reuse (e.g., see item 17 in [33]).

One might also raise the objection that CZ would result in a proliferation of abstract classes, for which a corresponding concrete class would have to be defined. We believe that this problem can mostly be solved through a syntactic sugar for defining concrete classes (Section 6.4). Additionally, note that our proposed solution requires just as many abstract classes as there would be mixins or traits (which also cannot be instantiated) if those solutions were to be used (Section 3.2).

Acknowledgements

We would like to thank Neelakatan Krishnaswami, Gilad Bracha, Nels Beckman, Ciera Jaspan, Kevin Bierhoff, William Lovas, and the anonymous reviewers of FTfJP, FOOL, ECOOP, and OOPSLA for helpful feedback on earlier versions of this paper. This research was supported in part by NSF CAREER award CCF-0546550, DARPA grant HR0011-0710019, and Army Research Office grant DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems."

References

[1] R. Agrawal, L. DeMichiel, and B. Lindsay. Static type checking of multi-methods. In *OOPSLA*, pages 113–128, 1991.

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J. Maessen, S. Ryu, G. Steele, Jr., and S. Tobin-Hochstadt. The Fortress Language Specification, Version 1.0. Available at http://research.sun.com/projects/plrg/Publications/fortress.1.0.pdf, 2008. Accessed 3/09.

[3] E. Allen, J. J. Hallett, V. Luchangco, S. Ryu, and G. L. Steele Jr. Modular multiple dispatch with multiple inheritance. In *SAC '07*, pages 1117–1121. ACM, 2007.

[4] D. Ancona, G. Lagorio, and E. Zucca. Jam - designing a Java extension with mixins. *ACM Trans. Program. Lang. Syst.*, 25(5):641–712, 2003.

[5] D. Ancona and E. Zucca. An algebraic approach to mixins and modularity. In *Algebraic and Logic Programming*, pages 179–193, 1996.

[6] G. Baumgartner, M. Jansche, and K. Läufer. Half & Half: Multiple dispatch and retroactive abstraction for Java. Technical Report OSU-CISRC-5/01-TR08, Dept. of Computer and Information Science, The Ohio State University, March 2002.

[7] A. Bergel. Personal communication, October 2008.

[8] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. *Computer Languages, Systems & Structures*, 34(2-3):83–108, 2008.

[9] L. Bettini, V. Bono, and S. Likavec. A core calculus of higher-order mixins and classes. In *SAC*, pages 1508–1509, 2004.

[10] J. Bloch. *Effective Java: Programming Language Guide*. Addison-Wesley, 2001.

[11] J. Boyland and G. Castagna. Parasitic methods: An implementation of multi-methods for Java. In *OOPSLA*, pages 66–76, 1997.

[12] G. Bracha and W. Cook. Mixin-based inheritance. In *ECOOP '90*, 1990.

[13] B. Carré and J. Geib. The point of view notion for multiple inheritance. In *OOPSLA/ECOOP '90*, pages 312–321. ACM, 1990.

[14] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP '92*, 1992.

[15] C. Chambers and the Cecil Group. The Cecil language: specification and rationale, Version 3.2. Available at http://www.cs.washington.edu/research/projects/cecil/, 2004. Accessed 3/09.

[16] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: modular open classes and symmetric multiple dispatch for Java. In *OOPSLA '00*, pages 130–145, 2000.

[17] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst.*, 28(3):517–575, 2006.

[18] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *POPL*, pages 125–135, 1990.

[19] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A.P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2006.

[20] T. Ekman and G. Hedin. JastAdd. http://www.jastadd.org, 2008. Accessed 3/09.

[21] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[22] R.B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. *ACM SIGPLAN Notices*, 34(1):94–104, 1999.

[23] K. Fisher and J. Reppy. A typed calculus of traits. In *Proceedings of the 11th Workshop on Foundations of Object-oriented Programming*, January 2004.

[24] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL '98*, 1998.

[25] C. Frost and T. Millstein. Modularly typesafe interface dispatch in JPred. In *FOOL/WOOD'06*, January 2006.

[26] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[27] N. C. Hutchinson. *EMERALD: An object-based language for distributed programming*. PhD thesis, University of Washington, Seattle, WA, USA, 1987.

[28] A. Igarashi, B. Pierce, and P. Wadler. Featherwieght Java: a Minimal Core Calculus for Java and GJ. In *OOPSLA '99*, November 1999.

[29] E. Johnsen, O. Owe, and I. Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci.*, 365(1):23–66, 2006.

[30] D. Malayeri. CZ: Multiple inheritance without diamonds. In *FOOL '09*, January 2009.

[31] D. Malayeri and J. Aldrich. CZ: Multimethods and multiple inheritance without diamonds. Technical Report CMU-CS-09-153, School of Computer Science, Carnegie Mellon University, August 2009.

[32] B. Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[33] S. Meyers. *Effective C++: 50 specific ways to improve your programs and designs*. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 1992.

[34] Microsoft Corporation. C# language specification, version 3.0. Available at http://download.microsoft.com/download/3/8/8/388e7205-bc10-4226-b2a8-75351c669b09/csharp%20language%20specification.doc, 2007. Accessed 8/09.

[35] T. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions. In *ICFP '02*, 2002.

[36] T. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions. *ACM Trans. Program. Lang. Syst.*, 26(5):836–889, 2004.

[37] T. Millstein and C. Chambers. Modular statically typed multimethods. *Inf. Comput.*, 175(1):76–118, 2002.

[38] N. Nystrom, S. Chong, and A. Myers. Scalable extensibility via nested inheritance. In *OOPSLA '04*, pages 99–115, 2004.

[39] M. Odersky. The Scala language specification. Available at http://www.scala-lang.org/docu/files/ScalaReference.pdf, 2007. Accessed 3/09.

[40] M. Odersky and M. Zenger. Scalable Component Abstractions. In *OOPSLA '05*, 2005.

[41] A. Paepcke. *Object-Oriented Programming: The CLOS Perspective*. The MIT Press, 1993.

[42] P. Pirkelbauer, Y. Solodkyy, and B. Stroustrup. Open multi-methods for C++. In *GPCE '07*, pages 123–134, 2007.

[43] M. Sakkinen. Disciplined inheritance. In *ECOOP*, pages 39–56, 1989.

[44] N. Schärli, S. Ducasse, O. Nierstrasz, and A.P. Black. Traits: Composable Units of Behaviour. In *ECOOP '03*. Springer, 2003.

[45] A. Shalit. *The Dylan Reference Manual: The Definitive Guide to the New Object-Oriented Dynamic Language*. Addison-Wesley, 1997.

[46] G. Singh. Single versus multiple inheritance in object

oriented programming. *SIGPLAN OOPS Mess.*, 5(1):34–43, 1994.

[47] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *OOPSLA*, pages 38–45, 1986.

[48] G. L. Steele, Jr. *Common LISP: The Language*. Digital Press, second edition, 1990.

[49] C. Szyperski, S. Omohundro, and S. Murer. Engineering a programming language: The type and class system of Sather. In J. Gutknecht, editor, *Programming Languages and System Architectures*, volume 782 of *Lecture Notes in Computer Science*. Springer, 1993.

[50] G. Washburn. Personal communication, December 2008.

A. Subtyping vs. Subclassing

In CZ, the use of requires provides subtyping without inheritance, but it also places constraints on concrete subclasses—they must inherit from their parent's required classes. This raises the question of whether simply providing subtyping without inheritance would be sufficient to encode the desired relationships.

When separating subtyping from inheritance, we may use nominal subtyping or structural subtyping. However, in either case, private members are problematic. If private members are included in a subtyping relationship, this can violate information hiding, if they are not, it can restrict expressiveness.

Concretely, consider the following program:

```
class A {
  private int i;
  boolean equals(A other) {
    ... // can access other.i?
  }
}

class B subtypes A {
  ... // declare i?
}
```

Suppose that the subtypes keyword provides nominal subtyping without inheritance (but without the additional constraints of requires). The question then arises: are private members considered when checking subtyping? If so, then B must declare a private field i. Unfortunately, this also means that A.equals can access B.i, which violates information hiding; one class should not be able to access private members defined in another class. On the other hand, if we assume that subtyping does not include private members, then A.equals cannot access other.i, which is problematic if the definition of equality depends on this field. An analogous problem occurs if structural subtyping is used.

The problem can be avoided if inheritance or requires is used for types that contain binary methods. Since requires is tied to a particular class, if we change the above code to B requires A (or B extends A), then A.equals(A other) may safely access other.i, even if an object of type B is passed to this method. Note that an information hiding problem does not arise here—the private state has not been redefined in B, but is rather (eventually) inherited from A in the concrete B implementation that was passed in.