CZ: Multiple Inheritance Without Diamonds

Donna Malayeri and Jonathan Aldrich

December 2008
CMU-CS-08-169

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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 important 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 [29]), 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.

This research was supported in part by the U.S. Department of Defense, Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems," and NSF CAREER award CCF-0546550.

Keywords: Multiple inheritance, multiple dispatch, diamond inheritance

1 Introduction

Single inheritance, mixins [11, 19], and traits [16, 29] 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 well-known 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) [30, 32]. The first, the conflicting-features problem, can be solved by allowing renaming (e.g., Eiffel [24]) or by linearizing the class hierarchy [33, 32]. 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 [18]. 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.

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 [13, 15]; for these reasons, it has been adopted by designers of new programming languages, such as Fortress [2]. Unfortunately however, problems arise when integrating modular multimethods even with restricted forms of multiple inheritance, such as traits or Java 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 [28]) and actual inheritance. Through examples, we illustrate that programs that require diamond inheritance can be translated to a hierarchy that uses a combination of `requires` and multiple inheritance, 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 [16]. When a call is made to $A.\texttt{super}.f()$ 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) [22] (Sect. 8) and have proved it sound (Appendix A).

Contributions:

- The design of a novel multiple inheritance scheme that solves (1) the object initialization problem and (2) the modular typechecking of external methods, by forbidding diamond inheritance (Sect. 6).

[1] Without loss of generality, our formal system includes external methods (also known as open classes) rather than full multimethods; see Sect. 2.

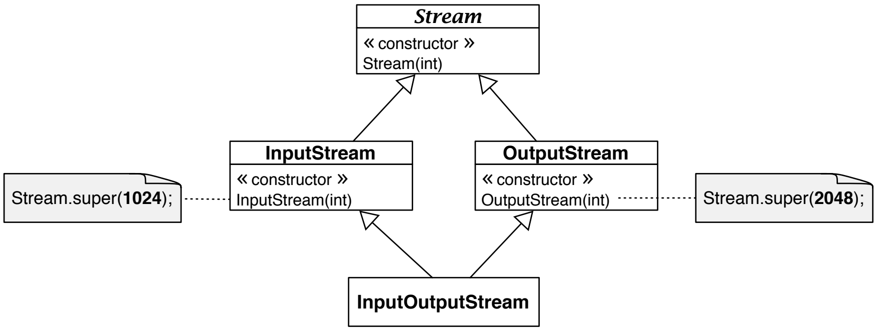

Figure 1: An inheritance diamond. Italicized class names indicate abstract classes.

- 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 4 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 (Sect. 8) and proof of type safety.
- An implementation of a typechecker for the language (using JastAdd [17]).

2 The Problem

To start with, the diamond problem 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" [12]? 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) [28, 24, 18].

Next, diamond inheritance leads to (at least) two major problems that have not been adequately solved: (1) determining how and when the superclass constructor/initializer should be called [33, 32], and (2) how to ensure non-ambiguity of multimethods in a modular fashion [26, 20, 3]. The first problem arises when the graph inheritance semantics is chosen, while the second appears with either tree or graph semantics.

Object initialization. To illustrate the first 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.

Modular multiple dispatch. The second problem regards multiple dispatch, which has been argued to be more natural and expressive than single dispatch [13, 15, 14]. However, typechecking multiple dispatch in a modular fashion becomes very difficult in the presence of multiple inheritance—precisely because of the diamond problem.

To simplify the discussion in this paper, we focus on external methods (also known as open classes), which are essentially multimethods that dispatch on the first argument only (corresponding to the receiver of an ordinary method). Multimethods with asymmetric dispatching semantics (i.e., the order of arguments affects dispatch) can be translated to external methods in a straightforward manner.[2] External methods essentially allow programmers to add methods to a class outside of its definition.

To see why diamond inheritance causes problems, consider the following definition of the external method[3] `seek`:

```
method Stream.seek {
  void Stream.seek(long pos) { } // default implementation: do nothing
  void InputStream.seek(long pos) { ... } // seek if pos <= eofPos
  void OutputStream.seek(long pos) { ... } // if pos > eofPos, fill with zeros
}
```

In the context of our diamond hierarchy, this method definition is ambiguous—what if `seek()` is called on an object of type `InputOutputStream`? 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 external methods 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 (or tags) on which dispatch is performed. For instance, the problem appears if dispatch is permitted on Java interfaces, as in JPred [20].

3 Previous Solutions

Object initialization. Languages that attempt to solve the object initialization problem include Eiffel [24], C++ [18], Scala [28] 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.[4] Though the `Stream` constructor is called only once, this awkward design has the problem that constructor calls are ignored. The semantics of `InputStream` may require that a particular `Stream` constructor be called, but the language semantics would ignore this dependency.

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.

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].

Finally, we note that although (stateless) traits and mixins do not suffer from the object initialization problem, they are less expressive than multiple inheritance. In particular, non-private accessors in a trait negatively impact information hiding, and introducing new "state" in a trait (through accessors) results in client classes having to implement these accessors [8]. On the other hand, though mixins do contain state,

[2] An asymmetric, or *encapsulated* multimethod dispatching on classes $A_1, \ldots, A_n$ can be translated to external methods defined on each A_i, where each method calls the method in class A_{i+1}, with the actual code defined in the method on A_n. Symmetric multiple dispatch cannot be encoded using external methods; this semantics adds a few orthogonal typechecking issues.

[3] We have defined this method externally for illustrative purposes.

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

they must be linearly applied and mixins cannot inherit from one another [11, 5]. If the latter were allowed, this would be essentially equivalent to Scala traits, which *do* have the object initialization problem.

Modular multiple dispatch. There are two main solutions to the problem of modular typechecking of multiple dispatch (or external methods) in the presence of multiple inheritance. The first solution is simply to restrict expressiveness and disallow multiple inheritance across module boundaries; this is the approach taken by the "System M" variant of Dubious [26].

JPred [20] 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 external methods 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 JPred syntax):

```
void f(Stream s) 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` is ever defined. Even if two types will *never* have a common subtype, the programmer must specify a disambiguating method, one that perhaps throws an exception.[5] 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.

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.

4 An Overview of CZ

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) [31]. In the first role, inheritance *is* necessary—it is the implementation strategy. 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. This notion of inheritance dependency is of key importance in CZ, because while multiple inheritance is permitted, inheritance diamonds are forbidden.

Consider the inheritance diamond of Fig. 1. To translate this hierarchy to CZ, `InputStream`'s relationship with `Stream` would be changed from inheritance to an inheritance *dependency*, requiring that 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. If we make `InputStream` an abstract class (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 [29, 28].[6]

[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.

[6] 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 [35].

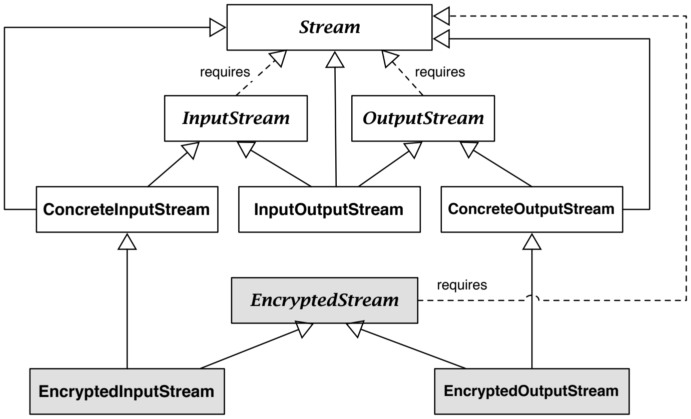

Figure 2: The stream hierarchy of Fig. 1, translated to CZ, with an encryption extension in gray. Italicized class names indicate abstract classes, solid lines indicate `extends`, and dashed lines indicate `requires`.

When a class *C* `requires` a class *B*:

- *C* is abstract, and
- *C* is a subtype of *B* (but not a subclass), and
- Subclasses of *C* must either `require` *B* themselves (making them abstract) or `extend` *B* (allowing them to be concrete).

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.)

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. This allows programmers to express inheritance relationships that would otherwise require diamond inheritance, without those associated problems.

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.

Modular multiple dispatch. A similar principle solves the problem of modular multiple dispatch. In CZ, an external method may only override a method in a superclass, not a required class. (This restriction does not apply to internal methods, as this scenario does not cause problems for modular typechecking.) So, the definitions of InputStream.seek and OutputStream.seek do not override Stream.seek—such a definition would essentially create two unrelated overloads of a method named seek.

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:

```
// unrelated methods that share the same name
void InputStream.doSeek(long pos) { ... }
void OutputStream.doSeek(long pos) { ... }

method Stream.seek {
  void Stream.seek(long pos) { ... } // default implementation: do nothing
  void ConcreteInputStream.seek(long pos) {
    InputStream.super.doSeek();
  }
  void ConcreteOutputStream.seek(int pos) {
    OutputStream.super.doSeek();
  }
}
```

(Though these definitions are slightly more verbose than before, syntactic sugar could be provided.)

Note that the typechecker does *not* require that a disambiguating method be provided for "InputStream && OutputStream", unlike JPred and Fortress. If a programmer later defines InputOutputStream, but does not re-define seek, the default implementation of Stream.seek will be inherited. An external or internal method for InputOutputStream can then be implemented, perhaps one that calls OutputStream.doSeek().

Here, it is of key importance that subclass diamonds are disallowed; because they cannot occur, external methods can be easily checked for ambiguities. Sub*typing* diamonds do not cause problems, as external method overriding is based on sub*classing*.

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.

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.

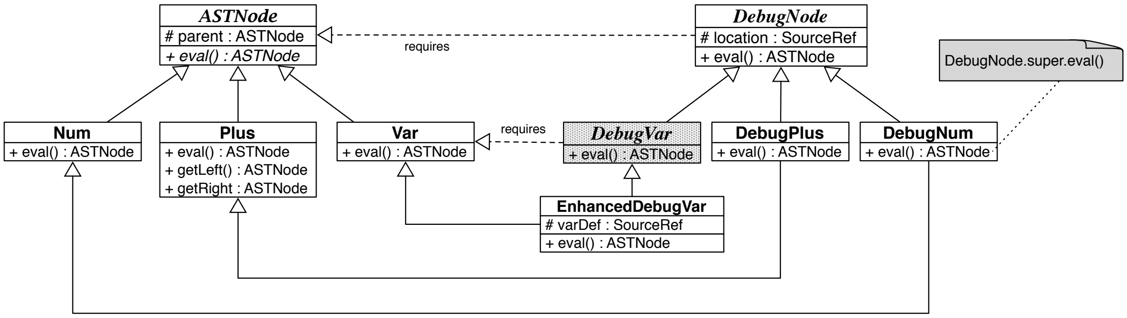

Figure 3: The AST node example in CZ. Abstract classes and abstract methods are set in italic.

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. [21, 34, 14]). There is an important difference, however, regarding information hiding. In a language that separates subclassing and subtyping, an interface type (used in a non-Java sense) must necessarily contain only "public" members; otherwise an arbitrary class would be able to access another class's private or protected state. For this reason, the `requires` relationship establishes a stronger relationship than simply subtyping; for example, a member in `Stream` may be declared "protected" and may then be accessed by `InputStream`. This does not violate information hiding, however, as we are guaranteed that concrete subclasses of `InputStream` will extend `Stream`, fulfilling the intent that only extenders have access to protected state. For a more detailed discussion, see Appendix B.

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 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 supports external methods, a method `defCheck()` could be added externally as follows:

```
method ASTNode.defCheck {  // external method
    void ASTNode.defCheck() { ... }
    void Var.defCheck() { ... }
    void Plus.defCheck() { ... }
    void Num.defCheck() { ... }
}
```

(We could also use the Visitor pattern, but the need for this must be anticipated, and the double dispatch code it requires is tedious and error-prone [15].) Note that the programmer would *only* have to define cases for `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).

Now, 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, which we express using `requires`. (For the moment, suppose that `EnhancedDebugVar` inherits directly from `DebugNode` and ignore `DebugVar`. We will come back to this when comparing to mixins.) Now, classes like `DebugPlus` can multiply inherit from `ASTNode` and `DebugNode` without creating a subclassing diamond. In particular, `DebugPlus` does *not*

```
class ASTNode {
    abstract ASTNode eval();
}
class Plus extends ASTNode {
    ASTNode eval() { ... }
    String toString() { return "+"; }
}
     ...
class DebugNode requires ASTNode {
    ASTNode eval() {
        print(this.toString());
        return ASTNode.super.eval(); // dynamic super call
    }
}
class DebugPlus extends DebugNode, Plus {
    ASTNode eval() {
        return DebugNode.super.eval(); // ordinary super call
    }
}
```

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

inherit two copies of the `parent` field, because `DebugNode` does not inherit from (i.e, is 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 "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()` → `DebugNode.eval()` → `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.

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, external dispatch. In other words, external dispatch in our system is analogous to case-analyzing datatypes in functional programming.

Single Inheritance. This example would be more difficult to express in a language with single inheritance. One straightforward design in a Java-like language is presented in Fig. 5. Multiple inheritance is simulated using interfaces for subtyping, and composition for dispatch. For instance, calls to `DebugPlus.getLeft()` are delegated to the wrapped `IPlus` object. The template method design pattern is used by `DebugNode` to implement `eval()` (subclasses override `getWrapped()`).

Note the addition of 4 new interfaces and the boilerplate code needed to implement getters, setters and delegation. The problem would be even worse if another dimension of behavior were to be added. Furthermore, the design has the problem that the getters and setters have to be public, since they are defined

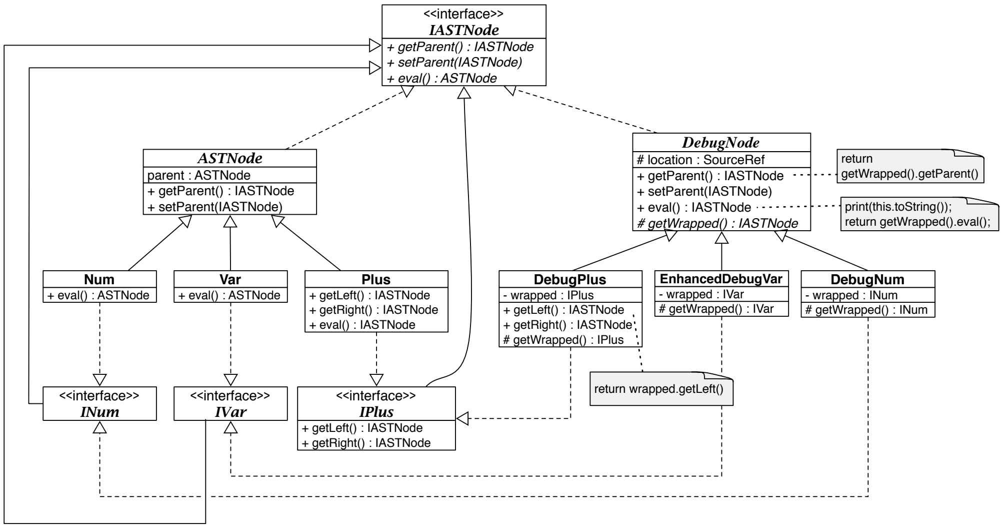

Figure 5: The example of Fig. 3 expressed in a Java-like language, resulting in a proliferation of interfaces and boilerplate code. The visibility modifiers '+', '-' and '#' indicate public, private and protected, respectively. Dashed lines represent `extends`; solid lines represent `implements`.

in an interface. For instance, the "parent" field in `ASTNode` is effectively fully visible, adversely affecting information hiding. Additionally, one would have to implement the visitor design pattern (not shown) to allow external traversal of the AST.

Traits. Traits could be used to express this example, but they lack state, resulting in an information-hiding problem with accessors (see Sections 3 and 9) similar to that of the single inheritance design. Stateful traits do not address the object-initialization problem, as previously mentioned.

Using Mixins. This example would be difficult to express using mixins. Aside from the limitation that a total ordering must be specified during mixin composition [16], other issues arise. Suppose that `DebugVar.eval()` prints the variable name, while `EnhancedDebugVar.eval()` prints the variable name and the location in the source program where it is defined. Since `DebugVar` is intended to be reused, it `requires` rather than `extends Var`.

To translate this example, both `DebugNode` and `DebugVar` must become mixins (which we will prefix with `M`). Since mixins cannot inherit from one another, `MDebugVar` would not be able to express an explicit relationship with `MDebugNode`, but would instead have to declare `location` and `eval()` as required members (e.g., Jam, Strongtalk [4, 5]). This, in turn, leads to two problems. First, `MDebugVar` cannot be treated as a subtype of `MDebugNode`, greatly reducing expressiveness. Second (and more significantly), supposing that external methods were to be integrated with mixins, it would be impossible to write an external method for `MDebugNode` and override it for `MDebugVar`—there is no relationship between the two mixins. That is, we may wish to write methods `MDebugNode`.f and `MDebugVar`.f (its override), and have `EnhancedDebugVar` inherit this latter definition. Instead, the definition of f must be pushed down to `EnhancedDebugVar`, which creates problems for code reuse. In particular, suppose that method f and `EnhancedDebugVar` are independent extensions that have no knowledge of each other. In a mixin world, external method definitions cannot be truly modular extensions.

The heart of the problem is that mixins are defined in isolation—though they can be composed, they cannot be subclasses (or even subtypes) of one another. Our solution could be viewed as similar to mixins, with the addition of subtyping and design intent (through `requires`) and (no-diamond) multiple inheritance.

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

The properties of classes and internal methods in CZ are the following:

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. Each method name has 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.

C3. If a class C extends D_1 and D_2 and method m is defined on both D_1 and D_2 (internally or externally), then C must also define m.

We have already described the reason for the no-diamond property, condition *C1*. 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.) 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* is imposed so that one kind of ambiguity can be checked locally. It prevents a name clash if two methods (internal or external) in unrelated classes A and B coincidentally have the same name and a third class inherits from both A and B.[7] The condition can be easily implemented in the compiler by appending the class name to a method name at the point in which it is introduced (i.e., method m first introduced in class A becomes m_A). For example, if the classes `Circle` and `Cowboy` both have a method `draw`, in the calculus the methods would be named `draw_Circle` and `draw_Cowboy`. 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 [24]) or by using qualified names (e.g., C# interfaces and C++).

Note that if C `requires` B and it defines an *internal* method m, then $C.m$ overrides $B.m$ and is considered part of the same method family (and therefore has the same name).

Condition *C3* ensures that diamond subtyping does not lead to problems. If two classes D_1 and D_2 have a common method m, then m must be contained in some common required class (as otherwise the two m's would have different qualified names). Since m is contained in two superclasses of C, this is an ambiguity that must be resolved by C.

6.2 External Methods

CZ includes *external methods*; methods can be added to a class outside of its definition. Such methods may be overridden by other methods, either internal or external. Typechecking an external method 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, only one method is applicable). As previously mentioned, asymmetric multimethods can be encoded with external methods; accordingly, the same typechecking issues apply to both.

In CZ's formal system, exhaustiveness of external methods is ensured because there are no abstract methods; if the language included abstract methods, external method definitions would *not* be permitted to be abstract. This, and as the restrictions below, are enforced in Millstein and Chambers's "System M" variant of the Dubious calculus [26], and in later extensions such as MultiJava [15]. Of these, only System M includes multiple inheritance.

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

To allow *modular* ambiguity checking, CZ methods must obey the following rules:

E1. All external method definitions of a method *m* must appear in the method block where the method family *m* is *introduced* (using the `method` declaration).

E2. When an external method family *m* is introduced, it must declare an *owner class C*: this specifies that the method family is rooted at *C*. *C* must be a proper subtype of `Object`, the root of the inheritance hierarchy. An external method definition *m* for class *D* is valid only if *D* is a sub*class* of *C*.

E3. An external method *must not* override an internal one (though an internal method *may* override an external one).

While all three conditions are the same as those in System M, that language did not allow multiple inheritance across module boundaries. In CZ we can remove this restriction by ensuring that diamond inheritance does not occur—condition *C1*. (Note that in CZ, each class and each top-level method declaration is in its own module.)

Condition *E1* is necessary because otherwise there could be two external method definitions *m* defined for the same class *C*, leading to an ambiguity.

Condition *E2* ensures that diamonds with `Object` at the top (permitted by condition *C1*) do not cause an ambiguity. Concretely, consider the following method definition:

```
method Object.g { // illegal CZ definition—owner cannot be Object
    void Stream.g() { ... }
    void Foo.g() { ... }
}
class Bar extends Stream, Foo { ... } // problem! two versions of g()!
```

If this were valid code, there would exist a method definition `g()` for each of `Stream` and `Foo`. In this case, `Bar` would inherit two equally valid definitions of `g()`. For typechecking to be modular, when checking `Bar`, we should not have to check all definitions of external methods, including `g()`. Note that *not* specifying an owner class has the same effect as using `Object` as an owner.

Additionally, condition *E2* ensures that diamond sub*typing* (as opposed to subclassing) does not result in a class inheriting the same external method through more than one path. If overriding were permitted based on subtyping, the problem described with diamond inheritance (Sect. 2) would re-appear.

The owner class is also important for implementing condition *C2* from the previous section—the owner class name must be part of the method name used by the internal language. (A related issue, defining two external methods with the same name *m*, can be resolved by using a naming convention for compilation units.)

Condition *E3* is required to avoid a situation where an external method *m* is defined on class *C* and *C* *also* defines an internal method *m*, causing an ambiguity.[8]

6.3 Discussion

Extensions. It would 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 *C* is the owner of a method family *m*, the typechecker would ensure that *m* contained disambiguating definitions for the case of a diamond—the JPred and Fortress solutions.

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 [33].

[8] This restriction unfortunately prevents a class *C* from declaring an abstract method *m* for the purpose of allowing clients to "plug in" different versions of *m* (in different namespaces) into the context where *m* is called. There is a solution to this, however: one can use structural subtyping instead of abstract methods to define an interface for *C*.*m* [23].

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., `Vector` extending `Stack` in the Java standard library), then it is very likely that *composition* is a more appropriate design [9]. 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. This principle is in accordance with the design of classes in Sather [34], traits in Scala and Fortress [28, 2, 3], and the advice that "non-leaf" classes in C++ be abstract [25]. In Sather, for example, only abstract classes may have descendants; concrete classes form the leaves of the inheritance hierarchy [34].

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[9] and Guikachu.[10]

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. 6 (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. 6 (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`,

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

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

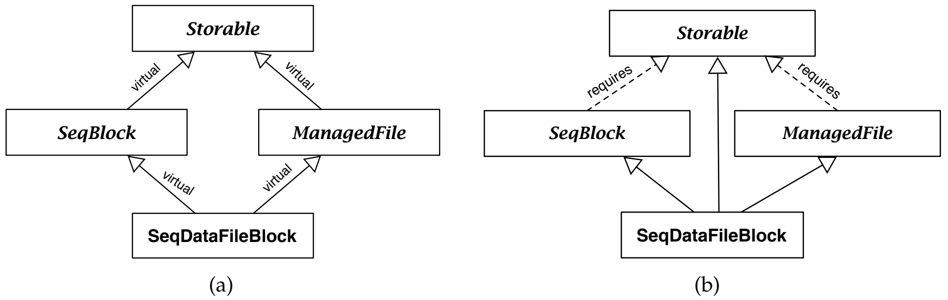

Figure 6: An inheritance diamond (a) in the Audacity application, and (b) the re-written class hierarchy in CZ. Abstract classes are set in italic.

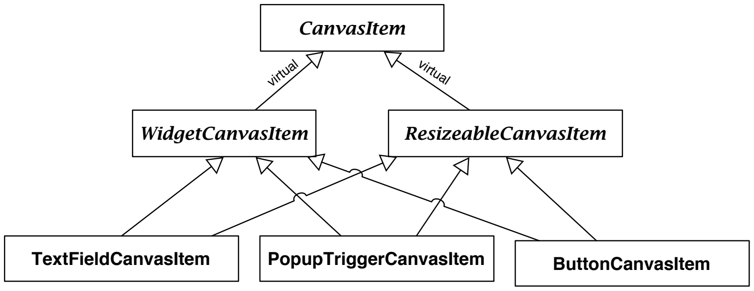

Figure 7: Three inheritance diamonds in the Guikachu application. Abstract classes are set in italic.

WidgetCanvasItem, and ResizeableCanvasItem. Figure 7 shows three of these 10 diamonds, formed by TextFieldCanvasItem, PopupTriggerCanvasItem and ButtonCanvasItem, 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 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, 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.

Due to space considerations, we have not shown the corresponding CZ class hierarchy; it would be very similar to that of Fig. 6 (b). Essentially, the virtual inheritance would be replaced with requires and each of the classes at the bottom of the diamond would 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.

7.2 Java Example: Eclipse JDT

The Eclipse JDT (Java Development Tools) provides 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 property 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 (for instance, `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);
```

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.

8 Formal System

In this section, we describe the formalization of CZ, which is based on Featherweight Java (FJ) [22]. 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. We use the same metavariables as FJ, with the addition that m and n range over internal and external method names, respectively; $\overline{M}$ and $\overline{N}$ range over internal and external method declarations, respectively.

The grammar of CZ is presented in Fig. 8. Modifications to FJ are highlighted. Class declarations may `extend` or `require` a *list* of classes. There is also a new type of declaration: top-level methods. The declaration `method` $C.m\{\overline{N}\}$ introduces an external method family with the owner class C. (Owner classes were described in Sect. 6.2.) The syntax requires that each external method be defined within the `method` block; this effectively enforces condition *E1* of Sect. 6.2.

Aside from virtual 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., [19, 27]) 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 subtype judgement (Fig. 9), denoted by '$<:$', which handles the `requires` relationship. Subclassing ('$\preceq$') 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. This is enforced by the typechecking rules.

$$
\begin{array}{lrl}
\text{Declarations} & L & ::= \textsf{class}\ C\ \textsf{extends}\ \overline{C}\ \textsf{requires}\ \overline{C}\ \{\overline{C}\,\overline{f};\ K\ \overline{M}\}\ \mid\ \textsf{method}\ C.m\ \{\ \overline{N}\ \} \\
\text{Constructors} & K & ::= C(\overline{C}\,\overline{f})\ \{\ \textsf{this}.\overline{f} = \overline{f};\}\\
\text{Methods} & M & ::= C\ m(\overline{C}\,\overline{x})\ \{\ \textsf{return}\ e;\}\\
\text{External Methods} & N & ::= C\ C.m(\overline{C}\,\overline{x})\{\ \textsf{return}\ e;\}\\
\text{Expressions} & e & ::= x \mid e.f \mid e.m(\overline{e}) \mid e.C.\textsf{super}.m(\overline{e}) \mid \textsf{new}\ C(\overline{e})
\end{array}
$$

Figure 8: CZ grammar

Subclassing $\boxed{C \preceq D}$

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

Subtyping $\boxed{C <: D}$

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

Figure 9: Subclassing ($\preceq$) and subtyping ($<:$) judgement

The auxiliary judgements for typechecking and evaluation appear after the typechecking and evaluation rules, in Fig. 13. We will describe each of these when describing the rules that use them.

Static Semantics. The rules for typechecking expressions are in Fig. 10. 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. 13) has two additional rules as compared to FJ: one for external method definitions, and one for methods received from a required class. The judgement first looks for the method *m* in the class itself, if it is not there, then it looks for an external method family with that name. If neither of those two cases applies, superclasses are recursively searched; otherwise required classes are searched.

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

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

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

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

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

Figure 10: Expression typing

$$\boxed{M\ \mathbf{ok}\ \text{in}\ C}$$

$$\frac{\begin{array}{c}\text{❶}\ \overline{x} : \overline{C}, \text{this} : C \vdash e_0 : E_0 \qquad \text{❷}\ E_0 <: C_0 \qquad \text{❸}\ \text{class}\ C\ \text{extends}\ \overline{D}\ \text{requires}\ \overline{E} \\ \text{❹}\ \mathit{override}(m, \overline{D}, \overline{C} \to C_0) \qquad \text{❺}\ \mathit{override}(m, \overline{E}, \overline{C} \to C_0)\end{array}}{C_0\ m(\overline{C}\ \overline{x})\{\ \text{return}\ e_0;\}\ \mathbf{ok}\ \text{in}\ C}\ (\text{T-METHOD})$$

$$\boxed{N\ \mathbf{ok}\ \text{in}\ C.m}$$

$$\frac{\begin{array}{c}\text{❶}\ \overline{x} : \overline{C}, \text{this} : C \vdash e_0 : E_0 \qquad \text{❷}\ E_0 <: C_0 \qquad \text{❸}\ \text{class}\ C\ \text{extends}\ \overline{D}\ \text{requires}\ \overline{E} \\ \text{❹}\ \nexists C'.\, \mathit{internalDef}(m, C) = C' \qquad \text{❺}\ C \preceq B \qquad \text{❻}\ \mathit{override}(m, \overline{D}, \overline{C} \to C_0)\end{array}}{C_0\ C.m(\overline{C}\ \overline{x})\{\ \text{return}\ e_0;\}\ \mathbf{ok}\ \text{in}\ B.m}\ (\text{T-EXT-METHOD})$$

Figure 11: Method typing

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

$$\frac{\begin{array}{c}\text{❶}\ \mathit{fields}(D_i) = \overline{D}_i\ \overline{g}_i \quad (i \in 1..n) \qquad \text{❷}\ K = C(\overline{D}_i\ \overline{g}_i, \overline{C}\ \overline{f})\{\ \text{this}.\overline{g}_i = \overline{g}_i\ ^{i \in 1..n}; \text{this}.\overline{f} = \overline{f}\} \\ \text{❸}\ \overline{M}\ \mathbf{ok}\ \text{in}\ C \qquad \text{❹}\ \text{class}\ D_i\ \text{requires}\ E', \text{implies}\ \exists k.\, D_k \preceq E'\ \text{or}\ \exists k.\, E_k \preceq E' \quad (i \in 1..n) \\ \text{❺}\ \text{class}\ E_i\ \text{requires}\ E'', \text{implies}\ \exists k.\, D_k \preceq E''\ \text{or}\ \exists k.\, E_k \preceq E'' \quad (i \in 1..n) \\ \text{❻}\ \forall i, j \in 1..n.\, i \neq j, \text{implies}\ \nexists D'.\, D_i \preceq D'\ \text{and}\ D_j \preceq D' \quad (D' \neq \text{Object}) \\ \text{❼}\ \forall m.\, \exists i, j.\, i \neq j.\, \mathit{internalDef}(m, D_i) \neq \mathit{internalDef}(m, D_j), \text{implies}\ m \in \overline{M} \\ \text{❽}\ \forall m.\, \exists i, j.\, i \neq j.\, \mathit{internalDef}(m, D_i)\ \text{and}\ \mathit{external}(m, D_i) = A\ \text{and}\ D_j \preceq A, \text{implies}\ m \in \overline{M}\end{array}}{\text{class}\ C\ \text{extends}\ D_1, \ldots, D_n\ \text{requires}\ E_1, \ldots, E_n\ \{\overline{C}\ \overline{f}; K\ \overline{M}\}\ \mathbf{ok}}\ (\text{T-CLASS})$$

$$\frac{C \neq \text{Object} \qquad \overline{N}\ \mathbf{ok}\ \text{in}\ C.m}{\text{method}\ C.m\ \{\ \overline{N}\ \}\ \mathbf{ok}}\ (\text{T-TOP-METHOD})$$

Figure 12: Class and external method typing

In the last rule, $\nexists mtype(m, D_k)$ is shorthand for $\nexists \overline{B} \to B.\, mtype(m, D_k) = \overline{B} \to B$.

Note that in our formalism, as in FJ, all definitions (including external methods) have global scope. In a real implementation, there would be specific import statements for external methods that would control their visibility.

The rule T-SUPER-INVK checks the virtual 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 the 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. 11. The rule T-METHOD checks internal methods, and uses the *override* auxiliary judgement, which is the same as that of FJ. In this rule, we check that method m is a valid override of the same method in all superclasses and required classes.

Typechecking external methods is a bit more involved than checking internal ones. The first three premises of the rule T-EXT-METHOD are the same of those in T-METHOD. Premise (4) ensures that there is no internal definition of m in C, enforcing condition *E3* of Sect. 6.2. Premise (5), $C \preceq B$, ensures that the class C on which the external method is defined is a sub*class* of the method family's owner class B, as required by condition *E2*. Finally, premise (6) ensures that the method being overridden (which will always

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

$$\frac{}{\mathit{fields}(\mathsf{Object}) = \bullet} \qquad \frac{\begin{array}{c}\mathsf{class}\ C\ \mathsf{extends}\ \overline{D}\ \mathsf{requires}\ \overline{E}\ \{\overline{C}\,\overline{f}; K\,\overline{M}\}\\ \mathit{fields}(D_i) = \overline{B_i}\,\overline{g_i}\ \ (i \in 1..n)\end{array}}{\mathit{fields}(C) = \overline{B_i}\,\overline{g_i}\,, \overline{C}\,\overline{f}}$$

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

$$\frac{\begin{array}{c}\mathsf{class}\ C\ \cdots\ \{\overline{C}\,\overline{f}; K\,\overline{M}\}\\ B\ m(\overline{B}\,\overline{x})\ \{\ \mathsf{return}\ \ e\} \in \overline{M}\end{array}}{\mathit{mtype}(m, C) = \overline{B} \to B} \qquad \frac{\begin{array}{c}\mathsf{class}\ C\ \cdots\ \{\overline{C}\,\overline{f}; K\,\overline{M}\}\\ m \notin \overline{M} \qquad CT(m) = \mathsf{method}\ \ D.m\ \{\ \overline{N}\ \}\\ B\ C.m(\overline{B}\,\overline{x})\ \{\ \mathsf{return}\ \ e\} \in \overline{N}\end{array}}{\mathit{mtype}(m, C) = \overline{B} \to B}$$

$$\frac{\begin{array}{c}\mathsf{class}\ C\ \mathsf{extends}\ \ \overline{D}\ \mathsf{requires}\ \overline{E}\ \ \{\overline{C}\,\overline{f}; K\,\overline{M}\}\\ m \notin \overline{M}\\ CT(m) = \mathsf{method}\ \ F.m\ \{\ \overline{N}\ \}\ \mathsf{implies}\ C.m \notin \overline{N}\\ \exists k.\, \mathit{mtype}(m,\ D_k\ ) = \overline{B} \to B\end{array}}{\mathit{mtype}(m, C) = \overline{B} \to B} \qquad \frac{\begin{array}{c}\mathsf{class}\ C\ \mathsf{extends}\ \ \overline{D}\ \mathsf{requires}\ \overline{E}\ \ \{\overline{C}\,\overline{f}; K\,\overline{M}\}\\ m \notin \overline{M}\\ CT(m) = \mathsf{method}\ \ F.m\ \{\ \overline{N}\ \}\ \mathsf{implies}\ C.m \notin \overline{N}\\ \forall k.\, \nexists \mathit{mtype}(m, D_k)\\ \exists k.\, \mathit{mtype}(m, E_k) = \overline{B} \to B\end{array}}{\mathit{mtype}(m, C) = \overline{B} \to B}$$

$$\boxed{\mathit{internalDef}(m, C) = D}$$

$$\frac{\begin{array}{c}\mathsf{class}\ C\ \cdots\ \{\overline{C}\,\overline{f}; K\,\overline{M}\}\\ B\ m(\overline{B}\,\overline{x})\ \{\ \mathsf{return}\ \ e\} \in \overline{M}\end{array}}{\mathit{internalDef}(m, C) = C} \qquad \frac{\begin{array}{c}\mathsf{class}\ C\ \mathsf{extends}\ \ \overline{D}\ \ \cdots\ \{\overline{C}\,\overline{f}; K\,\overline{M}\}\\ m \notin \overline{M} \qquad \exists k.\, \mathit{internalDef}(m, D_k) = D'_k\end{array}}{\mathit{internalDef}(m, C) = D'_k}$$

$$\frac{\begin{array}{c}\mathsf{class}\ C\ \mathsf{extends}\ \ \overline{D}\ \mathsf{requires}\ \overline{E}\ \ \{\overline{C}\,\overline{f}; K\,\overline{M}\} \qquad m \notin \overline{M}\\ \forall k.\, \nexists D'.\, \mathit{internalDef}(m, D_k) = D' \qquad \exists k.\, \mathit{internalDef}(m, E_k) = E'_k\end{array}}{\mathit{internalDef}(m, C) = E'_k}$$

$$\boxed{\mathit{external}(m, C) = A}$$

$$\frac{\begin{array}{c}CT(m) = \mathsf{method}\ \ F.m\ \{\ \overline{N}\ \}\\ C.m \in \overline{N}\end{array}}{\mathit{external}(m, C) = F} \qquad \frac{\begin{array}{c}\mathsf{class}\ C\ \mathsf{extends}\ \overline{D}\ \mathsf{requires}\ \ \overline{E}\\ CT(m) = \mathsf{method}\ \ F.m\ \{\ \overline{N}\ \} \qquad C.m \notin \overline{N}\\ \exists k.\, \mathit{external}(m, D_k) = F'\end{array}}{\mathit{external}(m, C) = F'}$$

$$\frac{\begin{array}{c}\mathsf{class}\ C\ \mathsf{extends}\ \overline{D}\ \ \mathsf{requires}\ \overline{E} \qquad CT(m) = \mathsf{method}\ \ F.m\ \{\ \overline{N}\ \} \qquad C.m \notin \overline{N}\\ \forall k.\, \nexists A.\, \mathit{external}(m, D_k) = A \qquad \exists k.\, \mathit{external}(m, E_k) = F'\end{array}}{\mathit{external}(m, C) = F'}$$

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

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

Figure 13: CZ typechecking and evaluation auxiliary judgements

be an external method, due to the $\nexists$*internalDef* premise) has the same type as the current method.

The T-CLASS rule (Fig. 12) checks class definitions. Premise (4) ensures that `requires` is propagated down each level of the inheritance hierarchy; the extending class must either `extend` or `require` its parents'

Auxilliary Judgements

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

$$\frac{\begin{array}{c}\mathsf{class}\ C\ \cdots\ \{\,\overline{C}\ \overline{f};K\ \overline{M}\,\}\\ B\ m(\overline{B}\ \overline{x})\ \{\ \mathsf{return}\ \ e\}\in\overline{M}\end{array}}{mbody(m,C)=\overline{x}.e}
\qquad
\frac{\begin{array}{c}\mathsf{class}\ C\ \cdots\ \{\,\overline{C}\ \overline{f};K\ \overline{M}\,\}\\ m\notin\overline{M}\qquad CT(m)=\mathsf{method}\ D.m\ \{\,\overline{N}\,\}\\ B\ C.m(\overline{B}\ \overline{x})\ \{\ \mathsf{return}\ \ e\}\in\overline{N}\end{array}}{mbody(m,C)=\overline{x}.e}$$

$$\frac{\begin{array}{c}\mathsf{class}\ C\ \mathsf{extends}\ \ \overline{D}\ \mathsf{requires}\ \overline{E}\ \ \{\,\overline{C}\ \overline{f};K\ \overline{M}\,\}\qquad m\notin\overline{M}\\ CT(m)=\mathsf{method}\ \ F.m\ \{\,\overline{N}\,\}\ \mathsf{implies}\ C.m\notin\overline{N}\qquad \exists\ \mathsf{unique}\ k\ .\ mbody(m,\ D_k\ )=x.e\end{array}}{mbody(m,C)=\overline{x}.e}$$

$$\boxed{super(C, D) = E}$$

$$\frac{\mathsf{class}\ C\ \mathsf{extends}\ E\qquad E\preceq D}{super(C,D)=E}$$

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

$$\frac{fields(C)=\overline{\tau}\ \overline{f}}{(\mathsf{new}\ C(\overline{e})).f_i\ \longmapsto\ e_i}
\qquad
\frac{mbody(m,C)=\overline{x}.e_0}{(\mathsf{new}\ C(\overline{e})).m(\overline{d})\ \longmapsto\ [\overline{d}/\overline{x},(\mathsf{new}\ C(\overline{e}))/\mathsf{this}]\,e_0}\ \text{(E-INVK)}$$

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

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

$$\frac{e_i\ \longmapsto\ e_i'}{\begin{array}{c}e_0.m(\ldots,e_i,\ldots)\ \longmapsto\\ e_0.m(\ldots,e_i',\ldots)\end{array}}
\qquad
\frac{e_i\ \longmapsto\ e_i'}{\begin{array}{c}e_0.C.\mathsf{super}.m(\ldots,e_i,\ldots)\ \longmapsto\\ e_0.C.\mathsf{super}.m(\ldots,e_i',\ldots)\end{array}}
\qquad
\frac{e_i\ \longmapsto\ e_i'}{\begin{array}{c}\mathsf{new}\ C(\ldots,e_i,\ldots)\ \longmapsto\\ \mathsf{new}\ C(\ldots,e_i',\ldots)\end{array}}$$

Figure 14: Evaluation rules

required classes. Premise (5) ensures that `requires` is copied at each level of the hierarchy. Premise (6) specifies that a subclassing diamond cannot occur, except for the case of `Object`. Finally, premises (7) and (8) enforce condition *C3*, ensuring that subtyping diamonds do not cause problems. The $external(m, D_i)$ judgement returns the first superclass of D_i for which an external method m is defined. Note that at most one of C's superclasses can have an external method m, as otherwise a diamond would occur.

The rule T-TOP-METHOD requires that the method owner not be `Object`, as required by condition *E2*.

Dynamic Semantics. The evaluation rules are presented in Fig. 14. Most of the rules are similar to FJ, with the notable exception of E-SUPER-INVK. This rule uses the auxiliary judgement $super(C, D)$, which finds the immediate superclass of the class C along the path D. Then, *mbody* is called on the result of the *super* call.

The *mbody* judgement (Fig. 13) mirrors *mtype* with two differences: there is no `requires` rule, and there must be a unique superclass that has a particular method body. The type safety theorems show that there is a correspondence between these two judgements, based on the class and method typechecking rules.

The remaining new rules are straightforward congruence rules.

8.1 Modularity

Here, we describe the conditions under which a class-based system with external methods 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.

Conditions for modular typechecking.

1. Checking a class definition C with methods $\overline{M}$ should only require examining: (a) signatures of methods transitively overridden in $\overline{M}$, (b) signatures of methods transitively overridden by C's inherited methods, (c) class declarations of C's supertypes, and (d) signatures of methods called by $\overline{M}$.
2. Checking the definition of a particular external method $C.m$ should only require examining: (a) the declarations of C and its supertypes, (b) the signature of the external method that $C.m$ overrides, and (c) the signatures of methods that $C.m$ calls. In particular, the typechecker may not search for subclasses of C.

We show that typechecking in CZ obeys these rules. We must consider all direct or indirect uses of *mtype*, because that judgement examines both internal and external methods. This includes uses of *override*, which calls *mtype*. We must also examine uses of the auxiliary judgement *external*. Uses of $\mathit{internalDef}(m, C)$ are permitted, as this judgement finds the signature of m in C or its supertypes. This is permitted under modularity condition 2(a).

Note that since our system does not have explicit modules, external and internal method names must contain the "module" they are defined in. This convention can be used to simulate module import statements.

In the rule for checking a class definition, T-CLASS, all premises but (3) and (8) are modular by inspection. Premise (3) uses the T-METHOD rule, which we will consider shortly, and premise (8) checks external methods overridden by some D_i. We observe that if there is a method m is defined or inherited by D_i, we are permitted to examine signatures of methods (internal or external) that it overrides. The *external* judgement searches for such an external method. As mentioned, in a complete implementation of CZ, external methods would be explicitly imported; this judgement would only examine those methods imported by supertypes.

The T-METHOD rule has three premises for which we need to demonstrate modularity: typechecking e_0 (premise 1) and the two *override* checks (premises 4 and 5). However, note that when typechecking e_0, *uses* of any external methods may indeed use T-INVK or T-SUPER-INVK (both of which use *mtype*), but this is effectively a case of client-side typechecking, rather than implementation-side typechecking. In other words, this is an instance of case 1(c).

The two *override* checks are not problematic, either. In each case, *mtype* searches for an internal or external method m in the superclasses and required classes of the class C. It does not examine all external methods or examine subclasses of C. Note that since an internal method m must include a module name, if it overrides an external method m it is implicitly "importing" the external method's module. Therefore, typechecking class definitions in CZ is modular.

Checking external methods is also modular. In the rule T-EXT-METHOD, premise (1) checks the method body as with T-METHOD; the same reasoning as above applies here. We should therefore consider premise (6). We observe that $\mathit{override}(m, \overline{D}, \cdots)$ will consider internal or external methods in superclasses of C. Since we have $\nexists \mathit{internalDef}$, internal methods are ruled out. Accordingly, *override* must be considering the overridden method of the *same* method we are already checking, due to the syntactic restriction that all cases of an external method are defined together (condition *E1*). Therefore, all the checks are modular; no other external methods are being examined.

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 method calls are always unambiguous, as the *mbody* judgement requires that there be a unique applicable method. We refer the reader to Appendix A for the proof of type safety; we give a brief outline here.

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.

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 $\mathit{mtype}(m, C) = \overline{D} \rightarrow D$ and $\Gamma \vdash \mathsf{new}\ C(\overline{e}) : C$ then $\mathit{mbody}(m, C) = \overline{x}.e_0$ for some $\overline{x}$ and e_0.

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 $\Gamma \vdash \mathsf{new}\ D_k(\overline{e}) : D_k$. Instead, we make use of two auxiliary lemmas:

Lemma 8.2. If $\mathcal{D} :: \mathit{mtype}(m, D) = \overline{B} \rightarrow B$ and $C <: D$ and $\Gamma \vdash \mathsf{new}\ C(\overline{e}) : C$, then there exist D' and $\mathcal{D}'$ such that $C \preceq D'$ and $\mathcal{D}' :: \mathit{mtype}(m, D') = \overline{B} \rightarrow B$ does not contain the rule MTYPE4.

Lemma 8.3. If $\mathcal{D} :: \mathit{mtype}(m, C)$ and $\mathcal{D}$ does not contain the rule MTYPE4, then $\mathit{mbody}(m, C) = \overline{x}.e$, for some $\overline{x}$ and e.

Lemma 8.2 is needed because it is the rule MTYPE4 that could result in *mbody* not being defined—it is the only rule that has no *mbody* counterpart. We make use of this lemma in the inductive step of the Lemma 8.3, as it is straightforward to show that *mbody* is defined, but additional reasoning is needed to show that its value is unique.

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

9 Related Work

10 Related work

Here we describe related work that was not previously discussed in Sect. 3.

As mentioned in Sect. 3, traits [16, 2] cause problems for information hiding—they essentially make it impossible to have private or protected "state" that is not accessible by objects that reuse the trait, as such "state" can only be implemented using accessors. Stateful traits [8] also do not help in this regard, as they have been designed for maximal code reuse, rather than information hiding. 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 comes at the cost of information hiding. Also, as previously mentioned, this design does not address the problem of a correct semantics for object initialization in the presence of diamonds.

As mentioned in Sect. 3, JPred [20] and Fortress [3] perform modular multimethod typechecking by requiring that programmers provide disambiguating methods, some of which may never be called. Neither language solves the problem of multiple inheritance with state.

On the other hand, 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 override a `Stream` method externally (though it may override it internally), because it is not a subclass of `Stream`. In contrast, if this hierarchy were expressed in e.g. JPred (using interfaces in place of abstract classes), a predicate method defined on `Stream` could be overridden by either `InputStream` or `OutputStream`. Note, however, that programmers can achieve a similar effect in CZ by having concrete classes call helper methods (which can be defined externally) in the abstract classes.

Cecil [13, 14] 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 of multimethods.

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

It is possible to modify the semantics of multimethod dispatch so that by definition ambiguities do not arise in the presence of multiple inheritance. A language may linearize the class hierarchy [1] or choose the appropriate method based on their textual ordering [10]. However, such semantics can be fragile and confusing for programmers.

11 Conclusions

We have presented a language that solves two major problems caused by inheritance diamonds: object initialization and external method 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`.

Acknowledgements

We would like to thank Neelakatan Krishnaswami, William Lovas, Steven Matuszek, Gilad Bracha, and the anonymous reviewers of FOOL and FTfJP for feedback on an earlier version of this paper.

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.

[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] L. Bak, G. Bracha, S. Grarup, R. Griesemer, D. Griswold, and U. Hölzle. Mixins in Strongtalk. In *ECOOP 2002 Workshop on Inheritance*, 2002.

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

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

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

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

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

[14] 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.

[15] 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.

[16] 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.

[17] Torbjörn Ekman and Görel Hedin. JastAdd. `http://www.jastadd.org`, 2008.

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

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

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

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

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

[23] D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. In *ECOOP 2008*, July 2008.

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

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

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

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

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

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

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

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

[32] G. Singh. Single versus multiple inheritance in object oriented programming. *SIGPLAN OOPS Mess.*, 5(1):34–43, 1994.

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

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

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

A Type Safety Proof

A.1 Preservation Lemmas and Proof

Lemma A.1 (Reflexivity of subtyping)**.** The following rule is admissible:

$$\frac{}{C <: C}$$

Proof. Immediate. □

Lemma A.2. If $C <: D$ and $\mathsf{class}\ D\ \cdots\ \{\,\overline{C}\ \overline{f};\ K\ \overline{M}\,\}$ and $m \in \overline{M}$ then *internalDef*(m, C).

Proof. Straightforward induction on the derivation of $C <: D$. □

Lemma A.3. If $\mathit{mtype}(m, D) = \overline{C} \rightarrow C_0$, then for $C \preceq D$, $\mathit{mtype}(m, C) = \overline{C} \rightarrow C_0$.

Proof. By induction on $C \preceq D$.

case SUBC-REFL. Immediate.

case SUBC-TRANS. We have $C \preceq D$ and $D \preceq E$. By the induction hypothesis, $\mathit{mtype}(m, D) = \overline{C} \rightarrow C_0$. Applying the induction hypothesis again gives the required result.

case SUBC-CLASS. There are three cases: m is defined in C, m is defined externally on C, or m is not defined on C.

In the first case, by inversion on *override* and T-METHOD, m must be a valid override and must have type $\overline{C} \rightarrow C_0$. By rule MTYPE-1, $\mathit{mtype}(m, C) = \overline{C} \rightarrow C_0$.

In the second case, by inversion on *override* and T-EXT-METHOD, m must be a valid override and therefore has type $\overline{C} \rightarrow C_0$. By rule MTYPE-2, $\mathit{mtype}(m, C) = \overline{C} \rightarrow C_0$.

In the third case, by rule MTYPE-3, $\mathit{mtype}(m, C) = \mathit{mtype}(m, D)$. By Lemma A.6, if there is more than one D_k such that $\mathit{mtype}(m, D_k)$ is defined, they all have the same result, namely $\overline{C} \rightarrow C_0$.

□

Lemma A.4. If $C <: E$ and $\mathit{mtype}(m, E) = \overline{B} \rightarrow B$, then $\mathit{mtype}(m, C) = \overline{B} \rightarrow B$.

Proof. By case analysis of $C <: D$.

case SUB-SUBCLASS. Follows from Lemma A.3.

case SUB-TRANS. $C <: D$ and $D <: E$.
By the induction hypothesis on $D <: E$, $\mathit{mtype}(m, D) = \overline{B} \rightarrow B$. The result then follows from the induction hypothesis on $C <: D$.

case SUB-REQUIRES. There are three cases:

C defines m internally. Similar to the same case in Lemma A.3

C defines m externally. By Lemma A.2, m must be defined externally on E. We have that $CT(m) = \mathsf{method}\ B.m\{\,\overline{N}\,\}$, and by T-EXT-METHOD all external cases of m for class D must be a subclass of B, i.e., $D \preceq B$. Therefore, $E \preceq B$ and the result follows from Lemma A.3.

C does not define m. By rule MTYPE-4, $\mathit{mtype}(m, C) = \mathit{mtype}(m, E)$.

□

Lemma A.5 (Methods have a unique point of introduction)**.** If $\mathit{mtype}(m, D_1) = \overline{B} \rightarrow B$ and $\mathit{mtype}(m, D_2) = \overline{B'} \rightarrow B'$ then there exists a D' where $D_1 <: D'$ and $D_2 <: D'$ and $\mathit{mtype}(m, D') = \overline{B''} \rightarrow B''$.

Proof. Straightforward simultaneous induction on the *mtype* derivations, making use of the convention that distinct method introductions result in distinct method names. □

Lemma A.6 (*mtype* has a unique value)**.**

1. If $\mathit{mtype}(m, C) = \overline{B} \rightarrow B$ and $\mathit{mtype}(m, C) = \overline{B'} \rightarrow B$, then $\overline{B} = \overline{B'}$ and $B = B'$.
2. If $C <: D_1$ and $C <: D_2$ and $m \notin C$
   $\mathit{mtype}(m, D_1) = \overline{B} \rightarrow B$ and $\mathit{mtype}(m, D_2) = \overline{B'} \rightarrow B'$
   then $\overline{B} = \overline{B'}$ and $B = B'$.

Proof. By mutual lexicographic induction on the *mtype* derivations and the lemma clause number (clause #2 may refer to clause #1 at the same *mtype* derivation, but clause #1 may only refer to clause #2 at a smaller *mtype* derivation.)

1. We proceed by induction on the first *mtype* derivation and inversion on the second derivation. Based on the structure of the judgement, the second derivation must end in the same rule as the first; each rule excludes all other rules. Therefore, there are 4 cases to consider:

   case MType1, MType1. Immediate.

   case MType2, MType2. Immediate.

   case MType3, MType3. By the induction hypothesis on clause #2, the result follows.

   case MType4, MType4. Similar to above.

2. We proceed by simultaneous induction on the two *mtype* derivations.

   MType1, MType1. By Lemma A.5, we have that $\mathit{mtype}(m, D') = \overline{B''} \rightarrow B''$, for some D' where $D_1 <: D'$ and $D_2 <: D'$. In such a case, by Lemma A.4, each m must be a valid override of $D'.m$ so therefore $\overline{B} = \overline{B''}$ and $B = B''$, and $\overline{B'} = \overline{B''}$ and $B' = B''$. From this, it follows that $\overline{B} = \overline{B'}$ and $B = B'$, which is the required result.

   MType1, MType2. There are 3 possibilities for the relationship between D_1 and D_2: $D_1 <: D_2$, $D_2 <: D_1$, and D_1 and D_2 are unrelated.

   $D_1 <: D_2$. In this case, $D_1.m$ overrides $D_2.m$ by Lemma A.4 and therefore $\overline{B} = \overline{B'}$ and B = B'.

   $D_2 <: D_1$. By T-Ext-Method, we have $\nexists$ *internalDef*(m, D_2). By the definition of *internalDef*, we have $\nexists$ *internalDef*(m, D_2). By Lemma A.2, this implies that $D_2 \not<: D_1$.

   D_1 and D_2 are unrelated. By Lemma A.5, we have that $\mathit{mtype}(m, D') = \overline{B''} \rightarrow B''$, for some D' where $D_1 <: D'$ and $D_2 <: D'$. The rest of the reasoning is similar to that for case MType1, MType1 above.

   MType2, MType2. By inversion on T-Ext-Method, $D_1 \preceq D$ and $D_2 \preceq D$ for some D where $\mathit{CT}(m)$ = method $D.m\{\ \overline{N}\ \}$ and $B\ D.m(\overline{B}) \in \overline{N}$. By Lemma A.3, $D_1.m$ and $D_2.m$ must be valid overrides of $D.m$ so therefore $\mathit{mtype}(D_1) = \overline{B} \rightarrow B$ and $\mathit{mtype}(D_2) = \overline{B} \rightarrow B$. By the induction hypothesis for clause 1, these values are unique.

   —, MType3. By the transitivity of subtyping, we have $C <: D_{k_1}$ and $C <: D_{k_2}$ and $\mathit{mtype}(m, D_{k_1}) = \overline{B} \rightarrow B$ and $\mathit{mtype}(m, D_{k_2}) = \overline{B'} \rightarrow B'$. By the induction hypothesis for clause #2, the result follows.

   —, MType4. Similar to above.

□

Lemma A.7 (Substitution)**.** If $\Gamma, \overline{x} : \overline{C} \vdash e : D$ and $\Gamma \vdash \overline{d} : \overline{C}'$ where $\overline{C}' <: \overline{C}$ then $\Gamma \vdash [\overline{d}/\overline{x}]\, e : D'$ for some $D' <: D$.

Proof. Similar to proof of FJ, using Lemma A.3 for the case of method invocation. □

Lemma A.8 (Weakening)**.** If $\Gamma, x : C, \Gamma' \vdash e : B$ then for $C' <: C$ and $B' <: B$, $\Gamma, x : C', \Gamma' \vdash e : B'$.

Proof. Straightforward induction on typing derivations. □

Lemma A.9. If $\mathit{mbody}(m, C) = \overline{x}.e_0$ then there exists a unique $\overline{B} \to B$ such that $\mathit{mtype}(m, C) = \overline{B} \to B$.

Proof. By induction on the derivation of *mbody*.

case MBody-1. By definition, $\mathit{mtype} = \overline{B} \to B$. By Lemma A.6 (1), this value is unique.

case MBody-2. Similar to above.

case MBody-3. We have $\mathit{mbody}(m, D_k) = x.e_0$. By the induction hypothesis, $\mathit{mtype}(m, D_k) = \overline{B} \to B$. By rule MType-3, $\mathit{mtype}(m, C) = \overline{B} \to B$. By Lemma A.6, this value is unique.

□

Lemma A.10. If $\mathit{mbody}(m, C_0) = \overline{x}.e$ and $\mathit{mtype}(m, C_0) = \overline{C} \to C$ then there exists some $D <: C$ such that $\overline{x} : \overline{C}, \mathsf{this} : C_0 \vdash e : D$.

Proof. By induction on the definition of $\mathit{mbody}(m, C_0)$.

case MBody-1. By inversion on T-Method, we have $\overline{x} : \overline{C}, \mathsf{this} : C_0 \vdash e : D$, where $D <: C$.

case MBody-2. By inversion on T-Ext-Method, we have $\overline{x} : \overline{C}, \mathsf{this} : C_0 \vdash e : D$, where $D <: C$.

case MBody-3. We have $\exists$ unique $D_k.\, \mathit{mbody}(m, D_k) = \overline{x}.e$. By Lemma A.9, there exists some unique $\overline{B} \to B$ such that $\mathit{mtype}(m, D_k) = \overline{B} \to B$. But, by the definition of *mtype*, $\mathit{mtype}(m, C) = \mathit{mtype}(m, D_k)$. Since the result of *mtype* is unique (Lemma A.6), $\overline{B} = \overline{C}$ and $C = B$. Applying the induction hypothesis to $\mathit{mbody}(m, D_k) = \overline{x}.e$ and $\mathit{mtype}(m, D_k) = \overline{C} \to C$ yields the required result.

□

Theorem A.1 (Preservation)**.** If $\Gamma \vdash e : C$ and $e \longmapsto e'$, then $\Gamma \vdash e' : C'$ for some $C' <: C$.

Proof. By induction on derivation of $e \longmapsto e'$.

case E-Field.

$e = (\mathsf{new}\ C_0(\overline{e})).f_i$
$e' = e_i$
$\mathit{fields}(C_0) = \overline{D}\,\overline{f}$
$D_i = C$

By the rule T-Field, $\Gamma \vdash \mathsf{new}\ C_0(\overline{e}) : C_0 \qquad C_0 <: C_0$.
By T-New, $\Gamma \vdash \overline{e} : \overline{C} \qquad \overline{C} <: \overline{D} \qquad C_0 = C_0$.
By transitivity of subtyping, $e_i : D_i$, which is the required result.

case E-Invk.

$e = (\mathsf{new}\ C_0(\overline{e})).m(\overline{d})$
$e' = [\overline{d}/\overline{x}, \mathsf{new}\ C_0(\overline{e})/\mathsf{this}]\, e_0$
$\mathit{mbody}(m, C_0) = \overline{x}.e_0$

By T-Invk and T-New:
$\Gamma \vdash \mathsf{new}\ C_0(\overline{e}) : C_0$

$\Gamma \vdash \overline{d} : \overline{C}$
$\overline{C} <: \overline{D}$
$mtype(m, C_0) = \overline{D} \to C$

By Lemma A.10, there exists some $D <: C$ such that $\overline{x} : \overline{D}, \mathsf{this} : C_0 \vdash e_0 : D$. By Lemma A.7, $\cdot \vdash [\overline{d}/\overline{x}, \mathsf{new}\ C_0(\overline{e})/\mathsf{this}]\, e_0 : D'$, for some $D' <: D$. By the transitivity of subtyping (Lemma A.1), $D' <: C$, which gives the required result.

case E-SUPER-INVK.
$e = (\mathsf{new}\ C_0(\overline{e})).B.\mathsf{super}.m(\overline{d})$

By T-SUPER-INVK and T-NEW:
class C_0 requires $B, \overline{E}$
class C_0 requires $\bullet$

This is a contradiction, therefore this case is vacuous. Dynamically-dispatched super calls can only be applied to classes with a non-empty `requires` clause.

The cases for the congruence rules are straightforward.

□

A.2 Progress Lemmas and Proof

Lemma A.11. If $internalDef(m, C) = D$ then $mtype(m, C) = \overline{B} \to B$, for some $\overline{B} \to B$.

Proof. Straightforward induction on the definition of $internalDef(m, C)$. □

Lemma A.12. If $external(m, C) = A$ then $C \preceq A$.

Proof. Straightforward induction on the derivation of $external(m, C)$. □

Lemma A.13. If $A \preceq B$ and B requires C, then $\exists C' \preceq C.\, A$ requires C' or $A \preceq C'$

Proof. Straightforward induction on $A \preceq B$. □

Lemma A.14. If $A <: B$ and $A \not\preceq B$ then $\exists B' \preceq B.\, A$ requires B'.

Proof. By induction on $A <: B$.

case SUB-SUBCLASS. Vacuous.

case SUB-TRANS. We have $A \preceq C$ and $C \preceq B$.
Since $A \not\preceq B$, there are three possibilities:

subcase $A \not\preceq C$, $C \not\preceq B$. By the induction hypothesis on the first derivation, we have $\exists C' \preceq C.\, A$ requires C'. By the induction hypothesis on the second derivation, $\exists B' \preceq B.\, C$ requires B'. We have $C' \preceq C$ and C requires B'. Taking these facts together, by Lemma A.13, $\exists B'' \preceq B'.\, C' requires B''$ or $C' \preceq B''$. In the first case, again by Lemma A.13, $\exists B''' \preceq B'.\, A$ requires B'''. But, since $B''' \preceq B$, this proves the required result.

subcase $A \preceq C, C \not\preceq B$. By the induction hypothesis, $\exists B' \preceq B.\, C$ requires B'. Since $A \preceq C$, by Lemma A.13, $\exists B'' \preceq B'.\, A$ requires B'' or $A \preceq B''$.

In the first case, A requires B'', the result follows from the fact that $B'' \preceq B$. In the second case, $A \preceq B''$, we have $A \preceq B$, which is a contradiction.

subcase $A \not\preceq C$, $C \preceq B$, by the induction hypothesis, $\exists C' \preceq C.\, A$ requires C'. The result follows from the fact that $C' \preceq B$.

case Sub-Requires. Immediate.

□

Lemma A.15. If $\mathcal{D} :: \mathit{mtype}(m, D) = \overline{B} \rightarrow B$ and $C <: D$ and $\Gamma \vdash \mathsf{new}\ C(\bar{e}) : C$, then there exist D' and $\mathcal{D}'$ such that $C \preceq D'$ and $\mathcal{D}' :: \mathit{mtype}(m, D') = \overline{B} \rightarrow B$ does not contain the rule MType4.

Proof. By induction on the *mtype* derivation.

case MType1, MType2. We observe that $\mathcal{D}$ does not contain the rule MType4. There are two possibilities: either $C \preceq D$, in which case let $\mathcal{D}' = \mathcal{D}$, or $C \not\preceq D$. In the latter case, by Lemma A.14, $\exists E \preceq D.\, C$ $\mathsf{requires}$ E. But, this is impossible; by inversion on T-New, C $\mathsf{requires}$ $\bullet$.

case MType3. We have D $\mathsf{extends}$ D_k where $\mathcal{D}_k :: \mathit{mtype}(m, D_k) = \overline{B} \rightarrow B$. The result follows from the induction hypothesis.

case MType4. Similar to above.

□

Lemma A.16. If $C \preceq D$ and $\mathcal{D} :: \mathit{mtype}(m, D) = \overline{B} \rightarrow B$ does not contain the rule MType4, then $\exists \mathcal{D}' :: \mathit{mtype}(m, C) = \overline{B} \rightarrow B$ that does not contain the rule MType4.

Proof. Straightforward induction on $C \preceq D$, using Lemma A.15 in the inductive step. □

Lemma A.17. If $C \preceq D_1$ and $C \preceq D_2$ and $\Gamma \vdash \mathsf{new}\ C(\bar{e}) : C$ and $m \notin C$ and $\mathit{mbody}(m, D_1) = \bar{x}_1.\bar{e}_1$ and $\mathit{mbody}(m, D_2) = \bar{x}_2.\bar{e}_2$ then either $D_1 \preceq D_2$ or $D_2 \preceq D_1$.

Proof. By simultaneous induction on the *mbody* derivations.

case MBody-1, MBody-1. We have $m \in D_1$ and $m \in D_2$. This implies that $\mathit{internalDef}(m, D_1) = D_1$ and $\mathit{internalDef}(m, D_2) = D_2$. By T-Class, this implies that $m \in C$, which is a contradiction. Therefore, $D_1 = D_2$.

case MBody-1, MBody-2. $m \in D_1 \qquad CT(m) = \mathsf{method}\ F.m\{\ \overline{N}\}, D_2.m \in \overline{N}$.
In order for the two m's to be in the same method family, $\exists B.\, D_1 <: B$ and $D_2 <: B$, where m is defined on B. m cannot be an internal method, since by inversion on T-Ext-Method, $\nexists \mathit{internalDef}(m, D_2)$. Therefore, $\mathit{external}(m, B) = F$.

Since inheritance diamonds are not permitted, either $D_1 \not\preceq B$ or $D_2 \not\preceq B$. Suppose $D_1 \not\preceq B$. By Lemma A.14, $\exists B_1.\, D_1$ $\mathsf{requires}$ B_1 and $B_1 \preceq B$. By T-Class, either C $\mathsf{extends}$ B' or C $\mathsf{requires}$ B', for some B' with $B' \preceq B_1$. We observe that the second case is impossible, by inversion on T-New.

In the first case, we have $\mathit{external}(m, B') = F$ and $B' \preceq F$ (Lemma A.12). Since D_1 $\mathsf{requires}$ B_1 and $B_1 \preceq B$, $\mathit{external}(m, D_1) = F$. By assumption, we have $\mathit{internalDef}(m, D_1) = D_1$. Taking these together, by premise (8) of T-Class, $m \in C$, which is a contradiction.

The case of $D_2 \not\preceq B$ follows the same reasoning.

case MBody1, MBody3, MBody2, MBody3. By MBody-3, we have $\exists k.\, \mathit{mbody}(m, D_k) = x_k.\bar{e}_k$, where D_2 $\mathsf{extends}$ D_k. Since $C \preceq D_k$, by the induction hypothesis, either $D_1 \preceq D_k$ or $D_k \preceq D_1$. The first case is impossible: we also have $D_2 \preceq D_k$, which results in an inheritance diamond with D_k at the top. But $D_k \neq \mathsf{Object}$, since $\forall m.\, \mathit{mbody}(m, \mathsf{Object})$ is undefined, so this case is impossible. In the second case, $D_2 \preceq D_1$, which is the required result.

case MBody2, MBody2. We have $m \notin D_1$, $m \notin D_2$, and $CT(m) = \mathsf{method}\ F.m\{\ \overline{N}\}$, where $D_1.m \in \overline{N}$ and $D_2.m \in \overline{N}$. By T-Ext-Method, $D_1 \preceq F$ and $D_2 \preceq F$. There are three possibilities: D_1 and D_2 are unrelated (by subclassing), $D_1 \preceq D_2$, or $D_2 \preceq D_1$. The first case is impossible, since $F \neq$ Object and this would mean that there is a diamond with F at the root. This proves the required result.

case MBody3, MBody3. By the premises of MBody3, we have $\exists k_1.\, mbody(m, D_{k_1}) = x_{k_1}.\bar{e}_{k_1}$, where D_1 extends D_{k_1} and $\exists k_2.\, mbody(m, D_{k_2}) = x_{k_2}.\bar{e}_{k_2}$, where D_2 extends D_{k_2}. By the induction hypothesis, either $D_1 \preceq D_{k_1}$ or $D_2 \preceq D_{k_2}$. In the first case, we have $D_1 \preceq D_{k_1}$ and $D_2 \preceq D_{k_2}$, which means there is a diamond with D_{k_2} at the top. But $D_k \neq \mathsf{Object}$, since $\forall m.\, mbody(m, \mathsf{Object})$ is undefined, so this case is impossible. The reasoning for the second case is similar.

□

Lemma A.18. If $\mathcal{D} :: mtype(m, C)$ and $\mathcal{D}$ does not contain the rule MType4, then $mbody(m, C) = \bar{x}.e$, for some $\bar{x}$ and e.

Proof. By induction on $\mathcal{D}$.

case MType1, MType2. Immediate.

case MType3. We have $\exists D_k.\, mtype(m, D_k) = \overline{B} \to B$, where C extends D_k. By the induction hypothesis, $mbody(m, D_k)$ is defined. It now suffices to show that $\nexists D_k.\, mbody(m, D_k) = \bar{x}.\, e$; the rule MBody3 then applies. Suppose $\exists D'.\, C$ extends D' and $mbody(m, D') = \bar{x}'.\, e'$. By Lemma A.17, either $D_k \preceq D'$ or $D' \preceq D_k$. If $D' \neq D_k$, then in the first case, premise (6) of T-Class (no diamond rule) is violated. Therefore, $D' = D_k$.

case MType4. Vacuous.

□

Lemma A.19. If $mtype(m, C) = \overline{D} \to D$ and $\Gamma \vdash \mathsf{new}\ C(\bar{e}) : C$ then $mbody(m, C) = \bar{x}.e_0$ for some $\bar{x}$ and e_0.

Proof. By induction on the derivation of *mtype*.

case MType1. The rule MBody-1 applies.

case MType2. The rule MBody-2 applies.

case MType3. We have $mtype(m, D_k) = \overline{D} \to D$ where C extends D_k. By Lemma A.15, $\exists D'.\, C \preceq D'$ and $\mathcal{D}' :: mtype(m, D') = \overline{B} \to B$. By Lemma A.16, $\exists D_c :: mtype(m, C) = \overline{B} \to B$ that does not contain the rule MType4. Finally, Lemma A.18 gives the required result.

case MType4. This rule cannot apply, since by inversion of T-Method, C requires $\bullet$.

□

Theorem A.2 (Progress)**.** If $\cdot \vdash e : C$ then either e is a value or there is an e' with $e \longmapsto e'$.

Proof. By induction on $e : C$.

case T-Var. Vacuous.

case T-Field. $e = e_0.f_i$
We have $fields(C_0) = \overline{C}\ \overline{f}$. By the induction hypothesis, either e_0 is a value or it evaluates to some e_0'. In the first case, the rule T-Field1 applies. In the second case, the rule E-Field2 applies.

case T-Invk. $e = e_0.m(\bar{e})$
By the induction hypothesis, either e_0 is a value or it evaluates to some e_0'. If it evaluates, then the rule E-Invk-Recv applies. If it is a value, then either the arguments $\bar{e}$ evaluate or they are values. In the first case, E-Invk-Arg applies. Otherwise, it suffices to show $mbody(m, C_0)$ is defined; the rule E-Invk then applies. We have $mtype(m, C_0) = \overline{D} \to C$ and $e_0 : C_0$. By Lemma A.19, $mbody(m, C_0)$ is defined.

case T-Super-Invk. $e = e_0.D.\mathsf{super}.m(\overline{e})$

By the induction hypothesis, either e_0 is a value or it evaluates to some e'_0. If it evaluates, then the rule E-Invk-Super-Recv applies. If it is a value, then either the arguments $\overline{e}$ evaluate or they are values. In the first case, E-Super-Invk-Arg applies. Otherwise, we have $e_0 = \mathsf{new}\ C(\overline{e})$, and it suffices to show $\mathit{mbody}(m, E)$ is defined, where $E = \mathit{super}(C, D)$; the rule E-Invk then applies.

We have $\mathit{mtype}(m, D) = \overline{D} \rightarrow C$ and $E \preceq D$ (by the definition of *super*) and $e_0 : C$. By Lemma A.3, $\mathit{mtype}(m, E) = \overline{D} \rightarrow C$. The result then follows from Lemma A.19.

case T-New. $e = \mathsf{new}\ C(\overline{e})$

By the induction hypothesis, either $\overline{e}$ evaluates or it is a value. If it evaluates, the rule E-New-Arg applies. Otherwise, the whole expression is a value.

□

B 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 both cases, there is a problem with how to handle private members. If private members are included in a subtyping relationship, this can violate encapsulation, 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 encapsulation. On the other hand, if we assume that subtyping does not include private members, then `A.equals` cannot access `other.i`.

An analogous problem occurs if structural subtyping is used. Suppose the `type` keyword defines a structural type, and we re-define `A.equals` as follows:

```
// option 1: allow access to private state, violate encapsulation
type TypeA = { private int i; boolean equals(TypeA); }

// option 2: restrict expressiveness of equals, disallow access to private state
type TypeA = { boolean equals(TypeA); }

class A {
    private int i;
    boolean equals(TypeA other) { ... }
}
```

Again we have the same problems as with nominal subtyping: if `TypeA` includes the private field, `equals` can access the private fields of other classes; if it does not, it restricts the implementation of `equals`.

A solution to this problem is to use inheritance or `requires` for calling code that uses binary methods.