Forsaking Inheritance: Supercharged Delegation in DelphJ

Prodromos Gerakios Aggelos Biboudis Yannis Smaragdakis

Department of Informatics
University of Athens, 15784, Greece
{pgerakios,biboudis,smaragd}@di.uoa.gr

Abstract

We propose DelphJ: a Java-based OO language that eschews inheritance completely, in favor of a combination of class morphing and (deep) delegation. Compared to past delegation approaches, the novel aspect of our design is the ability to emulate the best aspects of inheritance while retaining maximum flexibility: using morphing, a class can select any of the methods of its delegatee and export them (if desired) or transform them (e.g., to add extra arguments or modify type signatures), yet without needing to name these methods explicitly and handle them one-by-one. Compared to past work on morphing, our approach adopts and adapts advanced delegation mechanisms, in order to add late binding capabilities and, thus, provide a full substitute of inheritance. Additionally, we explore complex semantic issues in the interaction of delegation with late binding. We present our language design both informally, with numerous examples, and formally in a core calculus.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features; D.2.13 [*Software Engineering*]: Reusable Software

General Terms Languages

Keywords meta-programming; language extensions; morphing; object composition; delegation; static reflection

1. Introduction

Inheritance is the major feature that Object-Oriented language designers love to hate. It is the main mechanism for code reuse in most OO languages: a class `C` inheriting from another class `S` means that `C` can simultaneously both see all the members of `S` and export them to the outside world (filtered by preset access policies). At the same time, inheritance has often been criticized in the research literature (e.g., [5, 6]), and specialty press (e.g., [9]). Briefly:

- Inheritance confuses the role of a class as a model for object behavior and its role as an organizational unit of code. When `C` wants to reuse code from `S` it does not necessarily need to play the role of `S` to the outside world. The result is either chaotic automatic reuse of members or limiting the ability to reuse code. Languages such as C++ distinguish "inheritance for code reuse" (or plain subclassing) and "inheritance for subtyping" (or subclassing combined with subtyping). Multiple style guides for C++ programming explicitly warn programmers to always use "inheritance for subtyping". It is one of the main OO principles (effectively a corollary of the Liskov substitutability principle) that inheritance without subtyping is brittle, low-level, and generally dangerous. Accordingly, languages like Java only support inheritance for subtyping, from the perspective of the type system. Of course, no type system can ensure that programmers define proper subtypes, in a behavioral sense. OO programmers are taught to never reuse code for convenience when the subclass is not truly usable wherever the superclass is (i.e., when the subclass is not a true subtype).
- Inheritance is a coarse-grained mechanism for code reuse: `C` may need to reuse only a subset of the members of `S`, yet is forced to inherit all of them. This aspect of inheritance also exhibits itself as an annoying resistance to composition. If inheritance is primarily a code reuse mechanism, it seems that the first requirement would be for the ability to reuse code from multiple places. Nevertheless, multiple inheritance is a standard thorny feature of OO languages: naming conflicts, constructor ambiguities, and inadvertent double-state problems arise. Modern OO languages such as Java, C#, and Scala have explicitly abandoned multiple inheritance.
- Inheritance is rigid: in its base form, the relationship between a class and its superclass is determined at the time of writing the code for the subclass. Mechanisms

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.

http://dx.doi.org/10.1145/2509136.2509535

such as mixins and traits [3, 6, 28] attempt to remedy this shortcoming by adding much needed flexibility.

The mechanism of method delegation is often used as a more disciplined substitute of inheritance: the programmer explicitly creates a reference to a delegatee object and methods that forward their arguments to the appropriate method of the delegatee object. In this way, the programmer is in full control of the code reuse mechanisms, circumventing many of the surprising aspects of (single or multiple) inheritance. Nevertheless, delegation is a primitive mechanism, affording none of the convenience and automatic code reuse of inheritance. More advanced delegation mechanisms have been proposed [19, 26] but either suffer from the coarse-grained reuse problem of inheritance or do not support automatic reuse of code without naming reused members explicitly.

In past work we have proposed the idea of *class morphing* [10–12], which provides a glimpse at a possible full substitute of inheritance. Morphing is a programming construct that allows a program class or module to have non-fixed contents. E.g., instead of a class declaring a specific set of methods, it may declare that its methods match one-to-one (with appropriate changes based on a fixed pattern) those of another class. For instance, a morphed class `Listify` may statically iterate over all the methods of another class, `Subj`, pick those that have a single argument, and offer isomorphic methods: whenever `Subj` has a method with argument `A`, `Listify` accepts a `List<A>`. (The implementation of every method in `Listify` can then, e.g., iterate over all list elements, and manipulate them using `Subj`'s methods.) The MorphJ programming language is an extension of Java that serves as the reference embodiment of morphing principles. The described `Listify` functionality in MorphJ is shown below:

```
1 class Listify {
2   Subj ref;
3   Listify(Subj s) {ref = s;}
4
5   <R,A>[m] for (public R m(A): Subj.methods)
6   public R m (List<A> a) {
7     ... /* e.g., call m for all elements */
8   }
9 }
```

Everything but line 5 looks like plain Java code, but it is line 5 that determines what lines 6-8 really mean. Lines 5-8 form a *reflective declaration block*: line 5 defines the range of elements being iterated over; lines 6-8 are the method being declared once *for each* element in the iteration range. *The iteration happens statically.* Line 5 says that we want to iterate over all methods of `Subj` that match the pattern "`public R m (A)`", where `R`, `A`, and `m` are pattern variables (declared before the keyword `for`). `R` and `A` are pattern type variables, where `R` and `A` match any type except `void`; `m` is a name variable, and matches any identifier. Thus, this block iterates over all `public` methods of `Subj` that take one argument of any type and have non-`void` returns. For each such method, lines 6-8 declare a method with the same return type, name, and argument types equal to a list of the original argument type.

Morphing can be used to transform delegation into a mechanism that is as transparent and convenient as inheritance, without sacrificing programmer control. A client class, `C`, may want to transparently support all methods from class `S` and also supply methods `foo` and `bar`. The straightforward way to do this is to just use a static iteration over `S`'s methods and give implementations for `foo` and `bar`:

```
1  class C {
2    S ref;
3    C(S s){ref = s;} // to initialize ref
4
5    <R,A*>[m] for (public R m(A) : S.methods)
6    public R m (A a) {
7      ...
8      return ref.m(a);
9    }
10   void foo(...) {...}  // anything
11   void bar(...) {...}
12 }
```

(We use a new primitive in this example: the star after type parameter `A` means that it can match any number of arguments.)

Morphing-plus-delegation still does not equal inheritance, however! The problem is that the above combination does not support the *late binding* properties of inheritance. What if a method from `S` was also defined in `C`, i.e., if `S` defined (and used) methods `foo` or `bar`? In an inheritance setting, the method is overridden: method calls inside `S` are late-bound, so that they may dispatch to methods in `C` instead. Importantly, this can be performed in already compiled code. Yet no such late binding occurs when morphed class `C` uses `S` as a mere external client.

The facility of late binding is essential in the context of morphing. As a result, it requires us to introduce the concept of a strong bond between a morphed class and its delegatee. We use a special keyword, `subobject`, to signify such a bond. As we show, this is similar to past work on delegation mechanisms [19], yet with the subtlety that morphing allows strong bonds between objects and their subobjects without any obligation for automatically inheriting code or exporting members to clients—any automation desired can be selectively introduced by the programmer using a static iteration (a morphing `for` loop) instead of implicitly.

The result of this work is a Java-like language design that presents a full substitute of inheritance through delegation and morphing. At the same time, the inflexibility, coarse-granularity, etc. problems of inheritance are addressed directly. The contributions of our work are as follows:

- We show mechanisms that address the traditional problems of inheritance without sacrificing automation. Compared to past work on delegation mechanisms, we dissociate the concept of late binding from the concept of reusing members of a delegatee, resulting in a clean sep-

aration with many flexibility mechanisms. Compared to past work on morphing mechanisms, we add the essential capability of late binding, which allows delegation to be a full substitute of inheritance.

- Our language, DelphJ,[1] supports a unique combination of features yielding great power: subtyping (i.e., interface conformance and dynamic substitutability of subtype objects for supertype objects) but not inheritance; delegation with late binding; programmer control over all reuse without sacrificing automation; and modular type checking in the case of generic classes.
- We present our design both through informal examples (which elucidate some of the difficulties of late binding for delegation—e.g., versions of the fragile base class problem) and through a formal model based on Featherweight GJ (which makes our design decisions fully precise).

In the rest of the paper, we present our language design informally, via examples (Section 2), discuss the deep issues concerning the addition of delegation (Section 3), detail a formalism that captures the essence of our approach (Section 4), compare to related work (Section 5), and conclude (Section 6).

2. Morphing and Late Binding

Class-based inheritance provided by popular languages such as Java is often not flexible enough to support agile behavior sharing mechanisms between objects and to represent dynamic object evolution. It can also introduce problems that arise either accidentally, as in the case of the *accidental method overriding* problem, or due to poor design decisions and documentation, as in the case of the *fragile base class problem* [24] or the circle-ellipse problem [16].

Our work avoids inheritance-related shortcomings, by eliminating class-based inheritance in favor of a more flexible model, where code reuse is separated from subtyping: conformance to an externally visible interface is completely dissociated from the ability to reuse implementations. In this section, we show the design of our language, DelphJ, via informal examples. These will show how object composition via delegation, late binding, interface subtyping and morphing can be combined. The presentation emphasizes the points of difference between DelphJ and Java. That is, for features not mentioned it is typically safe to assume that they are handled as in Java.

Example 1 (**Consultation**). The combination of morphing ideas with traditional delegation in mainstream OO languages can express a pattern often called *consultation* or *forwarding* [19, 21]. In DelphJ, consultation is expressed (and has been since the original MorphJ work) via static-for loops as shown in the class below that logs method invocations before performing them:

```
1  class Logger {
2    Subj ref;
3    Logger(Subj s) {ref = s;} // initialize
4
5    <R,A*>[m] for (public R m(A) : Subj.methods)
6    public R m (A a) {
7      System.out.println("method " + m.name +
8                         " called with arg " + a);
9      return ref.m(a);
10   }
11 }
```

In this example, a client of `Logger` calls methods that are eventually forwarded to `Subj`.

Consultation is an automatic mechanism to forward method calls and morphing allows its flexible use. We do not sacrifice automation (since a single static-for use can implement forwarding for any number of methods) yet the user also has complete control over which method calls get forwarded (via the pattern employed in the static-for). This is in contrast to past language constructs for consultation, where automatic forwarding applied indiscriminately to all methods.

Example 2 (**Late binding**). As already mentioned in the Introduction, consultation is not always the mechanism one may want to employ, because it lacks *late binding* capabilities. In our `Logger` example, if class `Subj` has two methods `foo` and `bar`, such that `foo` invokes `bar`, both methods will be logged, but the call of `bar` inside `foo` will not.

The addition of a late binding facility is a major one in the DelphJ language and has significant implications, as we also discuss in detail in Section 3. Automatic mechanisms for forwarding method calls together with late binding are often called just *delegation* [22] in the research literature, but we will use the term *deep delegation* to avoid confusion with existing mainstream facilities. Deep delegation consists of the automatic method call forwarding from a child to a parent object, when the *self* reference is bound to the method receiver (the child). Consultation, on the other hand, is also an automatic method forwarding mechanism, however *self* is bound to the method holder (the parent). Informally speaking, deep delegation can emulate inheritance, while consultation is merely an automation mechanism.

Our late-binding facility applies on a per-field basis. We introduce a new modifier, `subobject`, to indicate fields with implicit support for late binding. Our earlier `Logger` class only needs to declare that its `ref` field refers to a subobject.

```
1  class Logger {
2    subobject Subj ref;
3    ... // as before
4  }
```

Methods of `Subj` are *overridden* if they are designated as public and not final, and are also declared in the `Logger` class. Therefore, `Logger` can now properly intercept and

[1] DelphJ is just MorphJ augmented with delegation features (and eliminating inheritance), as described in this paper. In all ways not relating to delegation (esp. morphing features) the languages are identical.

log *all* public method invocations, including those internal to `Subj`.

The following example illustrates a multilevel wrapper case. More specifically, we have two levels of wrappers, but this can be generalized to any number of levels.

```
class Wrapper1 {
  subobject Subj ref = ...

  <R,A*>[m]
  for (public R m(A): Subj.methods)
  public R m (A a) {
    ...
    return ref.m(a);
  }
}
class Wrapper2 {
  subobject Wrapper1 ref = ...

  <R,A*>[m]
  for (public R m(A):Wrapper1.methods)
  public R m (A b) {
    ...
    return ref.m(b);
  }
}
```

Assuming that we invoke a method on an instance of `Wrapper2`, `ref.m(b)` will be called, which in turn calls `ref.m(a)` within the body of the first wrapper. Eventually, the actual implementation of method `m` will be invoked. The object path through which the actual implementation of `m` is reached is termed *access path* and has a crucial role in DelphJ. In the example above, the path that `m(b)` (inside `Subj`) is accessed from is `Wrapper2` $\rightarrow$ `Wrapper1` $\rightarrow$ `Subj` (in a simplified form, showing only the types of each object on the path) . The dynamic dispatch algorithm utilizes the access path in order to locate the appropriate method to dispatch to. Intuitively, when an object *o* is accessed through `subobject` fields, the reference to *o* is not merely a pointer to it, but the full list of object pointers and `subobject` field identifiers that has led to *o*. Section 3 discusses in more detail the access path and how it is formed during execution.

Deep delegation and subclassing discussion. Several language and software designers have criticized the use of inheritance and have instead promoted delegation, which, however, lacks much of the inheritance convenience—delegated methods need to be explicitly named one-by-one, resulting in brittleness and verbosity. At the opposite end, past mechanisms for deep delegation support both late binding and automatic forwarding of method calls using a single keyword (e.g., `delegatee` [19, 21]).

In DelphJ we dissociate the two concepts completely. Our `subobject` annotation does not imply any automatic forwarding of method calls to the subobject reference. Instead, the user of the mechanism is responsible for establishing forwarding as desired using morphing capabilities. This approach achieves automation—e.g., it reduces the impact of the fragile base class problem, as modifications to a "base" class are automatically propagated to the "child" classes via morphing. At the same time it gives the programmer full control over which method calls will be forwarded.

Example 3 (Multiple parents). Delegation can be combined with late binding in DelphJ to emulate multiple inheritance, but without any loss of control. As a standard example of multiple code reuse, the following `GradStudent` class has subobjects of type both `Employee` and `Student`, thus reusing the functionality of both.

```
class GradStudent {
  subobject Student sref;
  subobject Employee eref;

  GradStudent(Student s, Employee e)
  { sref = s; eref = e; } // initialize

  <R,A*>[m]
  for (public R m(A): Student.methods;
       no R m(A): Employee.methods)
  public R m (A a) { ... }
  // handle methods only in Student

  <R,A*>[m]
  for (public R m(A): Employee.methods;
       no R m(A): Student.methods)
  public R m (A a) { ... }
  // handle methods only in Employee

  <R,A*>[m]
  for (public R m(A): Employee.methods;
       some public R m(A): Student.methods)
  public R m (A a) { ... }
  // handle methods in both
}
```

The example defines three static-for loops. The first includes methods from `Student` but not `Employee`, the second does the converse, and the third includes methods common to both. The listing showcases an important feature of DelphJ (inherited transparently from MorphJ [10]): *nested patterns* (via the `no` and `some` keywords). We use them above to select methods from one subobject that are/are not present in the other. The DelphJ type system ensures that the reflective blocks produce non-conflicting methods, i.e., the user receives a type error if overlapping cases exist. (Intuitively, conflicts are detected in two steps: first by trying to unify the pattern describing methods generated by each static-for loop, and, second, by checking whether the primary and nested patterns are enough to ensure that seemingly conflicting methods will never have the same name or argument types.)

We, thus, see that morphing ensures convenient code reuse but without relinquishing control over method forwarding. Also, although our examples focus on methods, the same handling applies to fields (e.g., the code could specify a pattern over `Student.fields` instead of `Student.methods`).

Example 4 (Dynamic object evolution). A common issue with class-based inheritance is that it fails to capture *dynamic object evolution*. For instance, it is not possible to

change the functionality of base classes at run-time (*hot-swapping*) [23]. DelphJ permits *dynamic delegation* [20], where `subobject` fields may be safely mutated without having to re-compile the original code. This feature also raises semantic complexities, which we defer discussing until Section 3. The example below modifies the `subobject` field, `ref`, dynamically. Method `Window.draw` overrides the `draw` method of its `Rendering` subobject. The invocation of `w.setWidget` alters the behavior of object `w` dynamically and from this point on the `draw` implementation of `SmoothDrawing` is used.

```
interface Rendering {
  public void draw();
}

class Window {
  subobject Rendering ref = new DefaultRenderer();

  public void draw() { return ref.draw();}

  void setWidget(Rendering newRef)
  { ref = newRef; }
}
...
Window w = new Window();
w.draw();  //Dispatching to ref's draw.
w.setWidget(new SmoothDrawing());
w.draw();  //Altered behavior.
...
```

Example 5 (Subtyping). Our language design eliminates inheritance and subclassing, but fully supports subtyping. This is effected through interfaces. DelphJ supports nominal subtyping and a class/interface can be a subtype of multiple interfaces. As in Java, interfaces are organized hierarchically and conformance of a class to an interface is designated by the keyword `implements`. An object of class `C` implementing an interface `I` can be used where a value of type `I` is expected, satisfying the Liskov substitution principle. Interfaces can also be used as types of `subobject` fields.

Since we have dissociated reuse of code from subtyping, there is no requirement that method signatures of `subobject` fields of `C` also exist in the interface implemented by `C`, but the programmer can easily effect this if desired by using a static-for construct. In the example below, the interface, `I`, of the subobject of type `Subj` is also supported by the `Wrapper` class, and even extended with extra members (which are in interface `IPlus`). Class `Wrapper` showcases that a) a class can use a pattern to delegate to *some* methods of a subobject; b) a class can override other methods of the subobject with explicit definitions; c) a class can provide extra methods, thus extending its interface.

```
interface I {
  void foo();
  void bar();
  void baz();
}
interface IBasic {
  void foo();
}
interface IPlus extends I {
  void foobaz();
}
class Subj implements I {...}

class Wrapper implements IPlus {
  subobject Subj ref;
  Wrapper(Subj s) {ref = s;} // initialize

  <R,A*>[m] for (public R m(A) : I.methods;
                 no R m(A) : IBasic.methods)
  public R m (A a) { return ref.m(a) }
  // delegate methods in Subj but not in IBasic

  void foo() {...} // handle the IBasic method
  void foobaz() {...} // handle any extra methods
}
```

Example 6 (Generics and modular type safety). All our examples so far contained known types. Static-for loops were thus merely a syntactic convenience. The hallmark feature of morphing is that the same capabilities exist even over type parameters. That is, we can define generic classes that take other types as yet-to-be-determined parameters. Our examples would all work even when the types involved (e.g., the type `Subj` of the subobject reference) were unknown. For instance, our earlier `Logger` class would most likely be written in a fully generic form:

```
class Logger<X> {
  X ref;
  Logger(X x) {ref = x;} // initialize

  <R,A*>[m] for (public R m(A) : X.methods)
  public R m (A a) {
    System.out.println("method " + m.name +
                       " called with arg " + a);
    return ref.m(a);
  }
}
```

The ability to have generic types offers great power in DelphJ. For instance, we can define as a single, reusable generic type a static computation yielding the public interface of *any* class:

```
interface PublicInterface<X> {
  <R,A*>[m] for (public R m(A) : X.methods)
  public R m (A);
}
```

For any class `C`, `PublicInterface<C>` yields a new interface containing `C`'s public members.

The interesting feature of DelphJ (and MorphJ before it) with respect to generics is that it maintains *modular type safety*. This means that a generic class is type-checked once-and-for-all and, if it passes type-checking, it will yield type-safe code for *any* type parameter passed to it that respects the stated obligations of type parameters. For instance, in our earlier generic `Logger<X>` class, the type system verifies that the call `ref.m(a)` is always possible, regardless of the type of `ref`, i.e., regardless of what `X` has been passed as a type parameter. In cases of complex patterns and mixes

of pattern-based and hand-coded methods, this capability is invaluable for the debugging of generic code.

An even more interesting example comes from generalizing our earlier scenario of a class that has two subobjects ("multiple parents") and overrides methods from either of them using different strategies. The generic class below captures such a general skeleton. For convenience, it first defines a generic interface that computes the difference of the interfaces of two given types. It then (for the sake of the example) just dispatches to the unique methods of either subobject and uses a filter to decide how to handle common methods.

```
1  interface SetMinus<X,Y> {
2    <R,A*>[m] for (public R m(A) : X.methods;
3                   no R m(A) : Y.methods)
4    public R m (A);
5  }
6
7  class C<X,Y> {
8    subobject X parent1 = ...
9    subobject Y parent2 = ...
10
11   <R,A*>[m]
12   for (public R m(A): // only in X
13        SetMinus<X,Y>.methods)
14   public R m (A a) { return parent1.m(a);}
15
16   <R,A*>[m]
17   for (public R m(A): // only in Y
18        SetMinus<Y,X>.methods)
19   public R m (A a) { return parent2.m(a);}
20
21   <R,A*>[m]
22   for (public R m(A): X.methods;
23        some public R m(A): Y.methods)
24   public R m (A a) {
25     return Coin.flip()?parent1.m(a):parent2.m(a);
26   }
27 }
```

This generic combination of functionality from two subobjects is still type checked modularly. The important check here is for the non-conflicting definitions of methods. The DelphJ type system ensures that declared methods do not conflict (i.e., there are no duplicate definitions with the same argument types), for *any* values of unknown type parameters `X` and `Y`. The type system manages to do this by leveraging the patterns (including nested patterns) in the above code.

Aspect-oriented features discussion. Morphing does not just supercharge delegation but also captures many common needs for *aspect-oriented programming* [18]. Using static-for loops, we can define common behavior for a set of methods. This behavior can play the role of *before-advice*, *after-advice*, or *around-advice* [17]. However, the emphasis of morphing is on modular reasoning and modular type-checking: as we saw in our last examples, the language is designed in such a way that a morphed generic class can be type-checked and its well-formedness can be ensured regardless of the code it is applied to. In contrast, standard aspect-oriented constructs offer no well-formedness guarantees when the code they apply to is unknown. Some research efforts in this direction have been made (e.g., [31]) though not with a formally proven type system for generic aspects.

3. Subtleties of Delegation

The previous section showcased our overall programming model, i.e., the use of morphing together with deep delegation as a full substitute of inheritance and an overall expressive, modular mechanism. We next focus more deeply on the subtleties of deep delegation and the relevant design decisions in DelphJ.

Nominal subtyping via interfaces. Interfaces are organized in a hierarchy and serve two purposes: first, they act as supertypes of classes, designated via the `implements` clause; second, interfaces can be employed as `subobject` field types to provide fine-grained method visibility and overriding for objects stored in such fields.

Our overriding strategy is that a wrapper class can *only* override non-final methods of a `subobject` field if the methods are present in the static (i.e., declared) field type. Since in DelphJ subtyping only occurs via interfaces, when we use an interface as the static type of a `subobject` field, we are limiting the potential for late binding to methods defined in the interface (and not all methods in the object that the `subobject` field currently references). This design decision has the rationale that classes can be developed separately from their `subobject` fields' classes, and a wrapper class should never be able to override a method it does not know about.

This feature prevents *accidental method overriding* by wrapper classes as in Figure 1. Let us assume that classes `Widget` and `ThirdParty` have been independently developed and only method `checkBounds` should be overridden, as indicated by `IThirdParty`. `Widget` obtains the `checkBounds` functionality by delegating `checkBounds` invocations to a `ThirdParty` object. Both `Widget` and `ThirdParty` independently implement method `checkRes`, thus `Widget` is unaware of the `checkRes` implementation in `ThirdParty`. In this code, overriding the behavior of `ThirdParty.checkRes` would result in a runtime exception.

Our policy of requiring that overridden methods in a subobject be both non-final and declared in the static type of the `subobject` field yields fine-grained control, for both parties in the subobject relationship. The wrapper class can control what it overrides, allowing for the possibility of non-interfering, unknown subobject methods. The subobject's class can also prevent overriding of a method by declaring it to be final: a method that is declared in a wrapper class, but is not a non-final method of a `subobject` field type will be ignored by the dynamic dispatch algorithm.

Access paths and late binding. The access path of an object o is an important concept in our language design and is defined as the ordered sequence of "`subobject`s" through

Figure 1 Accidental method overriding (avoided).

```
interface IThirdParty {
  public void checkBounds();
}

class ThirdParty {
  public void checkBounds(){
    if(checkRes() > 1024)
      throw new InvalidResolution();
    else ...
  }
  public int checkRes() { ... }
}

class Widget {
  subobject IThirdParty impl;
  public Widget(IThirdParty i){ impl = i; }
  public void checkBounds(){ impl.checkBounds(); }
  public int checkRes() { return 2048; }
}

new Widget(new ThirdParty()).checkBounds();
```

which o is accessed. Intuitively, the access path is the path to be searched to implement dynamic dispatch. A way of understanding access paths is to regard them as substitutes of `this`: if $(o_1 \rightarrow_{f_1} \ldots \rightarrow_{f_n} o_n).m()$ is invoked, where o_1 to o_n is the access path, then method lookup within m and for m itself will consider the entire path o_1 to o_n for finding the appropriate method. (The suffixes on the arrows in the above representation identify the field of the object through which the subobject is accessed, since its type matters for dispatch, per our preceding discussion.) The intuition is that the most recent version of m will be selected, which implies selecting the left-most element of the path that overrides m.

A key point is that access paths do not pertain to objects but to references. The same object can have multiple access paths: one for each reference (a.k.a. *alias*) to the object. We will use the following example to explicate the concept:

```
class Wrapper {
  subobject Subj ref;
  Wrapper (Subj s) { ref = s; }
  ...
}

Subj subj = new Subj();          // object s1
Wrapper w1 = new Wrapper(subj); // object o1
Wrapper w2 = new Wrapper(subj); // object o2
Subj alias = w2.ref;
```

The example gives names (`s1`, `o1`, `o2`) to the three distinct objects created. Objects `o1` and `o2` share `s1` as a subobject. This means that the value of reference `w1.ref` is *not* just a pointer to `s1` (as would typically be in Java) but a representation of the access path $(o1 \rightarrow_{\texttt{ref}} s1)$. Similarly, the value of reference `w2.ref` is $(o2 \rightarrow_{\texttt{ref}} s1)$ and not just `s1`. Additionally, the two references `subj` and `alias` are *not* equivalent, although they refer to the same object (`s1`), because they evaluate to different access paths. (In the case of `subj` the access path is *trivial*, i.e., contains a single member—`subj` is a direct reference to object `s1`, just like in Java.) Thus, the dynamic dispatch behavior is different for methods invoked via `alias` compared to methods invoked via `subj`.

Fundamentally, the reason that references hold access paths, and not merely object pointers, in our language design is that our objects *do not own* their subobjects, unlike in a language with inheritance. The concept of a subobject also exists in inheritance-based languages—the term is especially common for languages with multiple inheritance: an object of class `C` with superclasses `S` and `T` is said to contain `S` and `T` subobjects. However, an object in an inheritance-based language *owns* its superclass subobject. That is, every object of class `C` can be thought of as having a unique, non-aliasable reference to a full object of `C`'s superclass. (This also prevents circular references. In our language model it is the responsibility of the programmer to avoid circular references, or a run-time error ensues.) At the implementation level, this ownership typically translates into embedding the superclass subobject inside the object. In our design, however, since subobjects can be aliased, they need a per-alias record of how they are accessed, in order to implement dynamic dispatch properly. (In other delegation-based approaches [20], object access through a `subobject` field does not evaluate to the entire access path but to a single reference to the object, limiting the generality of dynamic dispatch.)

Our references can be passed around as arguments or return values, stored in local variables, etc. The only semantic question arising is how access paths are built. The general syntax of a field access (object field read) is "`obj = obj2.fld`". If `obj2` already encodes a non-trivial access path, then this path is augmented with the (pointer) value of field `fld` and the resulting access path is the value of `obj`. This can be viewed as *merging* access paths. That is, if we view field `fld` as storing the full access path of the last reference that was assigned to it, then during the field read operation "`obj = obj2.fld`" the access path stored in `fld` is ignored, except for the last object itself, which is appended to the access path of `obj2`. For instance, with the definition of class `Wrapper` as in our previous example, let us define references as follows:

```
Subj subj1 = new Subj();             // object s1
Subj subj2 = new Subj();             // object s2
Wrapper wrap1 = new Wrapper(subj1); // object o1
Wrapper wrap2 = new Wrapper(subj2); // object o2
Subj aliasForS2 = wrap2.ref;
wrap1.ref = aliasForS2;
Subj subj3 = wrap1.ref;
```

The access path for reference `wrap1` is $o1$ (a trivial path), while the access path for reference `aliasForS2` is $(o2 \rightarrow_{\texttt{ref}} s2)$. After the assignment of the next-to-last line, field `ref` of object $o1$ holds the access path $(o2 \rightarrow_{\texttt{ref}} s2)$ (exact copy of the value of `aliasForS2`). After the last line, however, reference `subj3` holds the access path $(o1 \rightarrow_{\texttt{ref}}$

$s2$): the only part of the access path ($o2 \rightarrow_{\texttt{ref}} s2$) kept is the object being referenced ($s2$), and that object is appended to the access path of `wrap1`.

The above is the main interesting semantic issue of our deep delegation approach. Reads from fields is both the way to form more complex access paths and the only case an access path is not copied when references are assigned. In all other cases, access paths are propagated unchanged. Our above discussion of access path merging also closely matches the main rule of our operational semantics in Section 4. As we discuss later, an actual implementation can optimize away access paths for references stored in object fields—only local variables (of a reference type) need to store access paths.

Mutability of `subobject` fields. As we saw, a significant difference between our deep delegation approach and traditional inheritance is that our objects do not own their subobjects. A further difference is that if a `subobject` field is not declared to be `final`, it can be mutated. As discussed in Section 2, this is a desirable capability since it allows dynamic object evolution. However, mutability of `subobject` fields (*dynamic delegation* [20]) raises interesting issues. Our design decisions described earlier in this section (use of static type of subobject reference in overriding, associating access path with references) address the issue: an object may change its subobject, yet a reference always holds the access path that allows it to perform the operations that its type allows, ensuring type soundness.

This aspect is worth elucidating since it is key to our approach. Access paths are *values* in DelphJ. They are immutable, although the references between objects are mutable. Consider an access path ($o1 \rightarrow_{\texttt{ref}} o2$) stored in reference variable `v`. Even if the linking of objects changes (e.g., $o1$ now points through field `ref` to $o3$ instead of $o2$) the access path inside variable `v` remains ($o1 \rightarrow_{\texttt{ref}} o2$). This does not, however, mean that variable `v` is immutable—it can be set to point to any other object, just like reference variables in Java can.

If access paths were not immutable values but `subobject` references could be mutated, several type or semantic soundness violations would arise. To see the issue, consider the example of Figure 2. (The example has been minimized yet remains involved, hence it is easier to follow it via our explanation below.)

Interface `S` defines two methods, `foo` and `baz`. Two classes, `S1` and `S2`, implement `S`. The implementation of `S1.foo` invokes `baz` and the private method `bar`. Notice that `S2` also declares `bar` with a different type signature. Finally, the wrapper class `C`, overrides the two methods of `S` and swaps the implementations of `S1` and `S2` when `baz` is invoked. The main program invokes `C.foo`, which delegates the call to `S1.foo`. The next method to be invoked is `baz`, which is overridden by `C.baz`. The execution of `C.baz` causes the mutation of the `impl` subobject reference, replacing an `S2` object in the place of an `S1`. The `C.baz` method returns to its caller (`S1.foo`) which then invokes `bar`. A naive delegation semantics would invoke `bar` on the replaced subobject (`S2.bar`) which would cause a runtime type error. Even if `S2.bar` returned an integer (i.e., was type compatible with `S1.bar`), the naive delegation semantics would cause a semantic error as there would be a partial state update of both `S1` and `S2`.

Figure 2 Mutable `subobject` fields may violate type safety.

```
1  interface S {
2    public int foo();
3    public void baz();
4  }
5
6  class S1 implements S {
7    final public int foo() {
8      this.baz();
9      return this.bar();
10   }
11   public void baz() { ... }
12   int bar() { ... }
13 }
14
15 class S2 implements S {
16   final public int foo() { ... }
17   public void baz() { ... }
18   String bar() { ... }
19 }
20
21 class C  {
22   subobject S impl;
23   public C(){impl = new S1();}
24   public void baz(){impl = new S2();}
25   public int foo(){return impl.foo();}
26 }
27 ...
28 int result = new C().foo();
29 ...
```

Our delegation semantics binds `this` of method `foo`[2] to the *immutable* access path `C` $\rightarrow_{\texttt{impl}}$ `S1`, and therefore, invokes the right version of `bar`.

Implementation considerations. We have implemented DelphJ in a prototype compiler,[3] with front-end support for the syntax and back-end translation to Java bytecode, using the JastAdd framework [7]. The back-end issues for the language are significant, however, and are clearly hinted at from our preceding discussion concerning access paths and their semantics. The interesting elements we have discussed above are:

- Access paths are immutable.
- References in DelphJ may refer to access paths and not directly to objects.

[2] The binding is done at invocation time, as in the usual handling of `this` in OO languages—e.g., implemented via passing an extra method argument for `this`.

[3] Available at `https://github.com/plast-lab/DelphJ`.

- Access paths are built/manipulated during field reads.
- Method dispatch needs to traverse an access path.

This suggests some inefficiencies of the DelphJ programming model compared to plain Java: every method invocation has to suffer extra overhead firstly for manipulating access paths (building them during field reads) and also to perform lookup in the access path data structure, instead of just making a (highly-optimized in modern VMs) virtual method call. (In contrast, there is no extra overhead in reference assignments: access paths are immutable so creating extra references to an access path does not entail copying it.)

An interesting note is that, in contrast to our earlier step-by-step description, object fields (of a reference type) do not need to store access paths: all objects except the last one on the access path will be ignored, no matter which alias is used to access the field. Consider a fragment of our earlier example:

```
...
wrap1.ref = aliasForS2;
Subj subj3 = wrap1.ref;
```

No matter what access path `aliasForS2` stores, it is unnecessary to store all of it inside field `ref` of the object referenced by `wrap1`. It is sufficient to only store the last object in the access path, which is the only part of the access path to be later used in any field dereference—e.g., appended to the access path of `wrap1` in the last line of the above example.

The consequence of the above discussion is that access paths are values for stack references only, while references inside the heap (i.e., in fields) can be direct pointers, just like in Java.

Our current implementation does not try to eliminate the above overheads. Instead, our back-end defines a plain Java library that performs the look up and access path merging operations, and our front-end translates DelphJ code to use such library operations. In the future, it is interesting to consider optimizations so that these overheads can be brough to a minimum, possibly with low-level data structure support and special-case treatment of references inside a dedicated DelphJ virtual machine.

4. Formalization

We next formalize our above informal semantic discussion to make the design decisions more precise. We capture the main features of DelphJ and sketch type soundness through a formalism, FDJ, based on FGJ [13] and adapting our earlier morphing formalism, FMJ [10, 11].

4.1 Syntax

Our formalism captures the salient features of DelphJ but eliminates unnecessary complexity: we model both classes and interfaces (as classes with no fields and with all their methods having empty bodies). All fields are implicitly `subobject` fields in FDJ.

Following FMJ, we also restrict similarly our treatment of morphing: single nested patterns are permitted as long as they do not use pattern type or name variables that are not bound by their primary pattern. There can be only one name variable in a pattern, and keyword η is used for name variables and reflective definitions. Reflecting over a statically known type, using a constant name in reflective patterns, reflectively declared fields, static name prefixes, casting expressions, polymorphic methods and method overloading are not formalized. Method overriding is invariant: covariant return types or contravariant argument types are not allowed.

The syntax of FDJ is presented in Figure 3. We adopt (or straightforwardly adapt) many of the notational conventions of FGJ: `C`,`D` denote constant class names; `X`,`Y`,`Z` denote type variables; `N`,`P`,`Q`,`R` denote non-variable types (which may be constructed from type variables); `S`,`T`,`U`,`V`,`W` denote types; `f` denotes field names; `m` denotes non-variable method names; `x`,`y` denote argument names. $\lhd$ and $\uparrow$ are shorthands for the keywords `implements` and `return`, respectively. All classes or "interfaces" must implement, or extend, respectively, an *interface*, which can be the empty interface, `Interface`. Hereon, we overload the term *class* to also mean interface. The method notation of FDJ slightly diverges from standard FGJ: methods accept a single argument, and method type signature `F` and name `n` are syntactically distinct from the method body b, which can either be empty or of the form $\{$`x`$\uparrow$`e;`$\}$. The former case permits the specification of interfaces. In the latter case variable `x` is bound inside the method body. A method can be prefixed with a standard FMJ static for loop $\mathfrak{R}$.

More specifically, notations borrowed from FMJ are: η denotes a variable method name; `n` denotes either variable or non-variable names; o denotes a nested condition operator (either + or - for the keywords `some` or `no`, respectively). We use the shorthand $\overline{\mathtt{T}}$ for a sequence of types $\mathtt{T}_0, \mathtt{T}_1, \ldots, \mathtt{T}_n$; $\overline{\mathtt{x}}$ for unique variable sequence $\mathtt{x}_0, \mathtt{x}_1, \ldots, \mathtt{x}_n$; : for sequence concatenation, e.g., $\overline{\mathtt{S}}\mathtt{:}\overline{\mathtt{T}}$ is a sequence that begins with $\overline{\mathtt{S}}$, followed by $\overline{\mathtt{T}}$; $\bullet$ and $\in$ to denote an empty sequence and sequence/set membership, respectively; _ and ... for values (one or any number, respectively) of no significance to a rule. For notational convenience, we assume that all our definitions are overloaded to apply to sequences of arguments of the original expected type, instead of just one argument. We also define Λ, the reflective iteration environment, which has the form $\langle R_p, oR_n \rangle$, where R_p is the primary pattern, and oR_n the nested pattern (o can be + or -). R_p and R_n have the form $(\mathtt{T}, \mathtt{<}\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}\mathtt{>F})$. `T` is the reflective type, over whose methods R_p iterates. $\overline{\mathtt{Y}}$ are pattern type variables, bounded by $\overline{\mathtt{P}}$, and `F` is a method pattern of the form `U` $\rightarrow \mathtt{U}_0$. $[\![\mathfrak{R}]\!]$ constructs the Λ corresponding to the reflective declaration $\mathfrak{R}$.

Access paths are represented as ordered lists of `new` `C<`$\overline{\mathtt{T}}$`>` $(\overline{v})$, whose elements are connected by $::_\imath$, with $\imath$ being the index of the `subobject` field. Access paths are val-

ues (v) but valid source programs must not contain values—they are only included in the syntax since they arise during evaluation. The *empty access path* is denoted as ϵ. Note that access paths in our formalism are shown in *inverse order* compared to those in our informal discussion of Section 3: "`new C<`$\overline{\texttt{T}}$`>(`$\overline{v}$`)` $::_i$ `new C'<`$\overline{\texttt{T}'}$`>(`$\overline{v'}$`)` $::_0$ ϵ" in the formalism corresponds to "`new C'<`$\overline{\texttt{T}'}$`>(`$\overline{v'}$`)` $\rightarrow_{f_i}$ `new C<`$\overline{\texttt{T}}$`>(`$\overline{v}$`)`" in the notation of Section 3—the formalism maximizes the ease of deconstructing lists while the informal discussion optimizes exposition.

A program in FDJ is an $(\texttt{e}, CT)$ pair, where `e` is an FDJ expression, and CT is the class table. Each class declaration has an entry in CT (except `Interface`) and the subtyping relation derived from CT must be acyclic.

4.2 Operational Semantics

Figure 4 defines the operational semantics of FDJ. All congruence rules are standard. Reduction rules introduce or transform values. As we already saw informally, access paths are ordered sequences of object values and are used by the dynamic dispatch algorithm in order to select the most recent method overriding another method. The first element of an access path can be considered as the object on which field and method accesses are performed. An access path can be constructed by either reaching an object through a sequence of `subobject` field accesses or by allocating a new object.

In the former case, the object stored in a `subobject` field is itself an access path, whose tail (i.e., its former access path) is ignored and its head is appended to the current access path. This is shown in rule R-FIELD, which reads field $\texttt{f}_i$ by reading the i-th constructor parameter v_i of the first element (`new C<`$\overline{\texttt{T}}$`>(`$\overline{v}$`)`) of the access path. Notice that the left-hand-side of a field access must be a value. v_i is then deconstructed to a head and a tail (i.e., the former access path of the head) and the head is appended to its new access path. Therefore, this implements the *path elimination* strategy discussed in the previous section. The append operator is annotated with i, denoting that the head was accessed by the next element of the access path through field $\texttt{f}_i$.

In the case of allocating a new object, reduction rule R-NEW transforms a `new C<`$\overline{\texttt{T}}$`>(`$\overline{v}$`)` expression, whose arguments have been evaluated to values, to a value: a singleton access path consisting of the `new` expression itself. The append operator is annotated with zero (an arbitrary choice) and the next element is the empty access path (ϵ).

Rule R-INVK expects an access path, performs method lookup (`mbody`) using the method name `m` and the access path v. The lookup operation returns the formal parameter list ($\overline{\texttt{x}}$), the method body `e` and a suffix of the input access path v' of v such that `m` belongs in the head of v' and there is no other object after the head element that overrides `m`.

Figure 5 defines the method body lookup rules necessary for the operational semantics, which are only defined for constant method names (`m`). It is not meaningful to define method body lookup for variable method names (η)—we are looking up the definition by the name of an actual method being called, although the lookup may need to consult a reflective iteration block to find the method body.

Function `method` performs method lookup in a single class. It takes two arguments, the method name `m` and the class type `C<`$\overline{\texttt{T}}$`>`, looks up `m` only in `C<`$\overline{\texttt{T}}$`>` and returns the entire method definition `M` having substituted type and pattern variables for concrete types. A detailed discussion regarding the semantics of `method` is deferred until the following section.

Predicate `validOverride` satisfies the condition described in Section 3: a method call over a subobject can be dynamically dispatched to the holder of the `subobject` reference only if it belongs in the *static* type of the `subobject` field—programs cannot override methods they cannot see. More specifically, it establishes that a method of an object (i.e. last argument) is *legally* overridden by a method of another object, when the latter object (a) has the same method name and type signature as the overridden method (i.e. first premise) and (b) accesses the former object via a field whose type contains a method with the same name and type signature as the overridden method (i.e. second premise). Notice that the reflective definitions and the method bodies in `validOverride` premises are irrelevant for determining whether a method is legally overridden.

Finally, function `mbody` performs the lookup, by traversing the input access path, employs `method` for extracting the definition of `m`, uses `validOverride` to establish legal overrides and returns the method formal parameter, method body and a suffix of the access path, such that the head of the returned access path and the returned method body belong to the same class. The first two rules MB-CLASS-S1 and MB-CLASS-S2, represent the case where the head is not followed by any other object that legally overrides `m`. Therefore, the *input* access path is returned along with the method parameter and body. The last rule MB-CLASS-S3, detects that the successor of the access path head legally overrides `m` so it removes the head from the access path and continues the search for other successors that may legally override `m`. Notice the last two rules of `mbody` get stuck when field $\texttt{T}''_i$ (specified by operator $::_i$) does not exist in `C'`, the second element of the access path.

4.3 Static Semantics

The main typing rules of FDJ are presented in Figure 6. There are three environments used in typing judgments: Λ, Δ and Γ. The latter two are standard, mapping type variables to upper bounds and variables to types respectively. Λ serves a twofold purpose. Firstly, it acts as a distinct typing environment, mapping pattern variables to their bounds. Pattern variables can never be instantiated explicitly, as opposed to type variables declared in class definitions. Therefore, Δ does not contain pattern variables. Secondly, it maintains patterns (i.e., structural constraints) for types in R_p

Figure 3 FDJ: Syntax

$$
\begin{array}{lll}
\mathtt{T} & ::= & \mathtt{X} \mid \mathtt{N} \\
\mathtt{N} & ::= & \mathtt{C}\langle\overline{\mathtt{T}}\rangle \mid \mathtt{Interface} \\
\mathtt{CL} & ::= & \mathtt{class\ C}\langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\rangle\triangleleft\mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\} \\
\mathfrak{R} & ::= & \langle\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\rangle\mathtt{for}(\mathbb{M}_p; o\mathbb{M}_n) \mid \bullet \\
\mathtt{F} & ::= & \mathtt{T}\rightarrow\mathtt{T} \\
\mathtt{M} & ::= & \mathfrak{R}\ \mathtt{n{:}F}\ b \\
o & ::= & + \mid - \\
\mathbb{M} & ::= & \mathtt{n{:}F\ in\ T.methods} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{e.f} \mid \mathtt{e.n(e)} \mid \mathtt{new\ C}\langle\overline{\mathtt{T}}\rangle(\overline{\mathtt{e}}) \mid v \\
b & ::= & \{\mathtt{x}\uparrow\mathtt{e;}\} \mid \bullet \\
v & ::= & \epsilon \mid \mathtt{new\ C}\langle\overline{\mathtt{T}}\rangle(\overline{v}) ::_\iota v \\
\mathtt{n} & ::= & \mathtt{m} \mid \eta
\end{array}
$$

Figure 4 FDJ: Reduction Rules

Reduction Rules:

$$\frac{v_i\ =\ \mathtt{new\ C'}\langle\overline{\mathtt{T}}'\rangle(\overline{v}') ::_{\jmath'} v'}{(\mathtt{new\ C}\langle\overline{\mathtt{T}}\rangle(\overline{v}) ::_\jmath v)\,.\,\mathtt{f}_i \longrightarrow \mathtt{new\ C'}\langle\overline{\mathtt{T}}'\rangle(\overline{v}') ::_\iota (\mathtt{new\ C}\langle\overline{\mathtt{T}}\rangle(\overline{v}) ::_\jmath v)} \quad \text{(R-FIELD)}$$

$$\frac{}{\mathtt{new\ C}\langle\overline{\mathtt{T}}\rangle(\overline{v}) \longrightarrow \mathtt{new\ C}\langle\overline{\mathtt{T}}\rangle(\overline{v}) ::_0 \epsilon} \quad \text{(R-NEW)}$$

$$\frac{mbody(\mathtt{m},\ v)=(\mathtt{x},\mathtt{e},v'')}{v\,.\,\mathtt{m}(v') \longrightarrow [v'/\mathtt{x},\ v''/\mathtt{this}]\mathtt{e}} \quad \text{(R-INVK)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0'}{\mathtt{e}_0\,.\,\mathtt{f} \longrightarrow \mathtt{e}_0'\,.\,\mathtt{f}} \quad \text{(RC-FIELD)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0'}{\mathtt{e}_0\,.\,\mathtt{m(e)} \longrightarrow \mathtt{e}_0'\,.\,\mathtt{m(e)}} \quad \text{(RC-INV-RECV)}$$

$$\frac{\mathtt{e} \longrightarrow \mathtt{e}'}{v\,.\,\mathtt{m(e)} \longrightarrow v\,.\,\mathtt{m(e')}} \quad \text{(RC-INV-ARG)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{new\ C}\langle\overline{\mathtt{T}}\rangle(v_1,\ldots,v_{i-1},\mathtt{e}_i,\ldots) \longrightarrow \mathtt{new\ C}\langle\overline{\mathtt{T}}\rangle(v_1,\ldots,v_{i-1},\mathtt{e}_i',\ldots)} \quad \text{(RC-NEW-ARG)}$$

and R_n that must hold over a reflective block. Every type variable must be bounded by an interface type. Function $bound_{\Delta;\Lambda}(\mathtt{T})$ returns the upper bound of T, when T is a type variable, otherwise it returns T. It must be noted that, for notational convenience, we assume pattern and ordinary type variables have globally unique names and that pattern variables are only used in the patterns of the reflective block that introduced them. Furthermore, we assume that logical connective operands that are undefined (e.g., when a function subexpression is undefined) implicitly evaluate to *false*. Similarly, premises whose value depends on distinct conditions are grouped in a case statement (tall left brace). Each line represents a case with some side conditions. When a left brace evaluates to a particular case then the following hold: (a) the side condition of the particular case is satisfied and (b) for each of the previous cases either their side condition does not hold or the actual result is undefined.

In the following sections we discuss key aspects of our type system: access path typing, reflective range containment-disjointness, interface and class typing, and method invocation typing.

4.3.1 Access path typing

Access paths v cannot exist in the original source code as they are intermediate values, but they must be typed for proving metatheorems. Access paths are typed via rule T-VAL. Empty access paths ϵ are *not* typable. Non-empty access paths are of the form $\mathtt{new\ C}\langle\overline{\mathtt{T}}\rangle(\overline{v}) ::_\iota v$. The first premise of T-VAL says that $\mathtt{new\ C}\langle\overline{\mathtt{T}}\rangle(\overline{v})$ must be well-typed (and thus *concrete*). It also says that the head of v_ι', where v_ι' is the ι_{th} constructor argument of the head of v,

Figure 5 FDJ: Method body lookup rules.

Valid override:

$$\frac{\emptyset;\emptyset\vdash \mathit{method}(\mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = \_\ \mathtt{m}\!:\!\mathtt{F}\ \_ \qquad \emptyset;\emptyset\vdash \mathit{method}(\mathtt{m},\ \mathtt{T}_i) = \_\ \mathtt{m}\!:\!\mathtt{F}\ \_}{\mathit{validOverride}(\mathtt{C}\langle\overline{\mathtt{T}}\rangle,\ \mathtt{T}_i,\ \_\ \mathtt{m}\!:\!\mathtt{F}\ \_)} \quad \text{(V-CLASS)}$$

Method body lookup:

$$\frac{v\ =\ \mathtt{new}\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle(\overline{v}) ::_i \epsilon \qquad \emptyset;\emptyset\vdash \mathit{method}(\mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = \mathfrak{R}\ \mathtt{m}\!:\!\mathtt{F}\ \{\mathtt{x}\uparrow\mathtt{e};\}}{\mathit{mbody}(\mathtt{m},\ v)=(\mathtt{x},\mathtt{e},v)} \quad \text{(MB-CLASS-S1)}$$

$$\frac{v\ =\ \mathtt{new}\ \mathtt{C}'\langle\overline{\mathtt{T}}'\rangle(\overline{v}') ::_j v' \qquad \emptyset;\emptyset\vdash \mathit{fields}(\mathtt{C}'\langle\overline{\mathtt{T}}'\rangle)=\overline{\mathtt{T}''}\ \overline{\mathtt{f}} \qquad \emptyset;\emptyset\vdash \mathit{method}(\mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle) = \mathtt{M} = \mathfrak{R}\ \mathtt{m}\!:\!\mathtt{F}\ \{\mathtt{x}\uparrow\mathtt{e};\} \qquad \neg\mathit{validOverride}(\mathtt{C}'\langle\overline{\mathtt{T}}'\rangle,\ \mathtt{T}''_i,\ \mathtt{M})}{\mathit{mbody}(\mathtt{m},\ \mathtt{new}\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle(\overline{v}) ::_i v)=(\mathtt{x},\mathtt{e},\mathtt{new}\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle(\overline{v}) ::_i v)} \quad \text{(MB-CLASS-S2)}$$

$$\frac{v\ =\ \mathtt{new}\ \mathtt{C}'\langle\overline{\mathtt{T}}'\rangle(\overline{v}') ::_j v' \qquad \emptyset;\emptyset\vdash \mathit{fields}(\mathtt{C}'\langle\overline{\mathtt{T}}'\rangle)=\overline{\mathtt{T}''}\ \overline{\mathtt{f}} \qquad \emptyset;\emptyset\vdash \mathit{method}(\mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)=\ \mathtt{M} \qquad \mathit{validOverride}(\mathtt{C}'\langle\overline{\mathtt{T}}'\rangle,\ \mathtt{T}''_i,\ \mathtt{M})}{\mathit{mbody}(\mathtt{m},\ \mathtt{new}\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle(\overline{v}) ::_i v)=\mathit{mbody}(\mathtt{m},\ v)} \quad \text{(MB-CLASS-S3)}$$

Figure 6 FDJ: Typing Rules

Expression typing:

$$\frac{\Delta;\Lambda;\Gamma\ \ ok}{\Delta;\Lambda;\Gamma\vdash \mathtt{x}\ \in\Gamma(\mathtt{x})} \quad \text{(T-VAR)}$$

$$\frac{\Delta;\Lambda;\Gamma\vdash \mathtt{e}_0\in\mathtt{T}_0 \qquad \Delta;\Lambda\vdash \mathit{fields}(\mathit{bound}_{\Delta;\Lambda}(\mathtt{T}_0))=\overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Delta;\Lambda;\Gamma\vdash \mathtt{e}_0\,.\,\mathtt{f}_i\in\mathtt{T}_i} \quad \text{(T-FIELD)}$$

$$\frac{\Delta;\Lambda;\Gamma\vdash \mathtt{e}\ \in\ \mathtt{T} \qquad \Delta;\Lambda;\Gamma\vdash \mathtt{e}'\in\ \mathtt{T}''' \qquad \Delta;\Lambda\vdash \mathtt{T}'\to\mathtt{T}''\ \ ok \qquad \Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathtt{T})=\mathtt{T}'\to\mathtt{T}'' \qquad \Delta;\Lambda\vdash \mathtt{T}'''<:\mathtt{T}'}{\Delta;\Lambda;\Gamma\vdash \mathtt{e}\,.\,\mathtt{n}(\mathtt{e}')\in\mathtt{T}''} \quad \text{(T-INVK)}$$

$$\frac{\Delta;\Lambda\vdash \mathtt{C}\langle\overline{\mathtt{T}}\rangle\ \ ok \qquad \Delta;\Lambda;\Gamma\ \ ok \qquad \Delta;\Lambda\vdash \mathit{fields}(\mathtt{C}\langle\overline{\mathtt{T}}\rangle)=\overline{\mathtt{T}'}\ \overline{\mathtt{f}} \qquad \Delta;\Lambda;\Gamma\vdash \overline{\mathtt{e}}\in\overline{\mathtt{T}''} \qquad \Delta;\Lambda\vdash \overline{\mathtt{T}''}<:\overline{\mathtt{T}'} \qquad \Delta;\Lambda\vdash \mathit{concrete}(\mathtt{C}\langle\overline{\mathtt{T}}\rangle,\mathtt{true})}{\Delta;\Lambda;\Gamma\vdash \mathtt{new}\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle(\overline{\mathtt{e}})\in\mathtt{C}\langle\overline{\mathtt{T}}\rangle} \quad \text{(T-NEW)}$$

$$\frac{\Delta;\Lambda;\Gamma\vdash \mathtt{new}\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle(\overline{v})\in\mathtt{C}\langle\overline{\mathtt{T}}\rangle \qquad v=\mathtt{new}\ \mathtt{C}'\langle\overline{\mathtt{T}}'\rangle(\overline{v}') ::_j v'\ \text{ implies }\ (v'_i=\mathtt{new}\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle(\overline{v}) ::_k v''\ \text{ and }\ \Delta;\Lambda;\Gamma\vdash v\in C'\langle\overline{\mathtt{T}}'\rangle)}{\Delta;\Lambda;\Gamma\vdash \mathtt{new}\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle(\overline{v}) ::_i v\in\mathtt{C}\langle\overline{\mathtt{T}}\rangle} \quad \text{(T-VAL)}$$

Method typing:

$$\frac{\mathtt{n}=\mathtt{m}\ \Leftrightarrow\ \mathfrak{R}=\bullet \qquad [\![\mathfrak{R}]\!]=\Lambda \qquad \Delta;\Lambda;\Gamma\ \ ok \qquad \Delta;\Lambda\vdash \mathtt{T},\mathtt{T}'\ \ ok \qquad b=\{\mathtt{x}\uparrow\mathtt{e};\}\ \text{ implies }\ (\Delta;\Lambda;\Gamma,\mathtt{x}\mapsto\mathtt{T}'\vdash \mathtt{e}\ \in\mathtt{T}''\ \text{ and }\ \Delta;\Lambda\vdash \mathtt{T}''<:\mathtt{T})}{\Delta;\Gamma\vdash\mathfrak{R}\ \mathtt{n}\!:\!\mathtt{T}'\to\mathtt{T}\ \ b\ \ OK} \quad \text{(T-METH)}$$

Class typing:

$$\frac{\Delta=\overline{\mathtt{X}}<:\overline{\mathtt{N}} \qquad \Gamma=\mathtt{this}\mapsto\mathtt{C}\langle\overline{\mathtt{X}}\rangle \qquad \Delta;\emptyset\vdash\mathtt{C}\langle\overline{\mathtt{X}}\rangle\!:\!\overline{\mathtt{T}}\ \ ok \qquad \text{for all } \mathfrak{R}_i\ \mathtt{n}_i\!:\!\mathtt{F}_i\ \ b_i,\ \mathfrak{R}_j\ \mathtt{n}_j\!:\!\mathtt{F}_j\ \ b_j\in\overline{\mathtt{M}},\ \Delta;\Gamma\vdash\mathfrak{R}_i\ \mathtt{n}_i\!:\!\mathtt{F}_i\ \ b_i\ \ OK \qquad \Delta\vdash \mathit{validRange}([\![\mathfrak{R}_i]\!],\ \mathtt{N}) \qquad \text{and } i\neq j \text{ implies } (\Delta\vdash \mathit{disjoint}([\![\mathfrak{R}_i]\!],\ [\![\mathfrak{R}_j]\!]) \text{ and } \mathtt{n}_i=\mathtt{n}_j=\eta \text{ or } \mathfrak{R}_i=\mathfrak{R}_j=\bullet \text{ and } \mathtt{n}_i=\mathtt{m} \text{ and } \mathtt{n}_j=\mathtt{m}')}{\mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle\lhd\mathtt{N}\ \{\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}\ \ OK} \quad \text{(T-CLASS)}$$

Figure 7 FDJ: Method type lookup.

Specializing reflective environment:

$$\frac{\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda\sqsubseteq_\Lambda\langle R_p,oR_n\rangle}{\Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash \mathit{specialize}(\eta,\ \langle R_p,oR_n\rangle)} \quad \text{(SP1)}$$

$$\frac{\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{T}_0)=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{F}_1 \quad \Delta;\Lambda\vdash\overline{\mathtt{W}}<:\overline{\mathtt{P}} \quad \Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{T}_1)=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{F}_2 \Leftrightarrow o{=}+}{\Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash \mathit{specialize}(\mathtt{m},\ \langle(\mathtt{T}_0,\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\mathtt{F}_1),o(\mathtt{T}_1,\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\mathtt{F}_2)\rangle)} \quad \text{(SP2)}$$

Method lookup:

$$\frac{[\![\mathtt{C}]\!]=\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}\mathtt{;N} \quad \mathfrak{R}\ \mathtt{n'{:}F}\ \ b\ \in CT(\mathtt{C}) \quad \Lambda_d=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\![\mathfrak{R}]\!] \quad [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]=\begin{cases}[\overline{\mathtt{W}}/\overline{\mathtt{Y}}] & \mathtt{if}\ \ \Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash \mathit{specialize}(\mathtt{n},\ \Lambda_d)\\ \bullet & \mathtt{otherwise}\ \ \mathtt{n{=}n'{=}m}\ \text{and}\ \ \Lambda_d{=}\emptyset\end{cases}}{\Delta;\Lambda\vdash \mathit{method}(\mathtt{n},\ \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>})=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\mathfrak{R}\ \mathtt{n:F}\ \ [\mathtt{n}/\mathtt{n'}]b)} \quad \text{(M-CLASS)}$$

Method type lookup:

$$\frac{\begin{cases}\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathit{bound}_{\Delta;\Lambda}(\mathtt{X}))=\mathtt{F}\\ \mathtt{otherwise}\ \ (\mathtt{+(X,\ <}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>F}),\eta)\in\Lambda\times\{\mathtt{n}\}\end{cases}}{\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathtt{X})=\mathtt{F}} \quad \text{(MT-VAR)}$$

$$\frac{\begin{cases}\Delta;\Lambda\vdash \mathit{method}(\mathtt{n},\ \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>})=\mathfrak{R}\ \mathtt{n:F}\ \ b\\ \mathtt{otherwise}\ \ \Delta;\Lambda\vdash \mathit{concrete}(\mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>},\mathtt{false})\\ \text{and}\ \ [\![\mathtt{C}]\!]=\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}\mathtt{;N}\ \text{and}\ \ \Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathtt{N})=\mathtt{F}\end{cases}}{\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>})=\mathtt{F}} \quad \text{(MT-CLASS)}$$

Figure 8 FDJ: Containment and disjointness rules.

Reflective range containment:

$$\frac{\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_p\sqsubseteq_R R'_p \qquad \begin{cases}\Delta;\bullet\vdash R_n\sqsubseteq_R[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R'_n & \mathtt{if}\ \ o=+\\ \Delta;\bullet\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R'_n\sqsubseteq_R R_n & \mathtt{otherwise}\end{cases}}{\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\langle R_p,oR_n\rangle\sqsubseteq_\Lambda\langle R'_p,oR'_n\rangle} \quad \text{(SB-}\Lambda\text{)}$$

Single range containment:

$$\frac{\begin{array}{ccc}R_1=(\mathtt{T}_1,\ \mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{Q}}\mathtt{>}\mathtt{F}_1) & R_2=(\mathtt{T}_2,\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\mathtt{F}_2) & \Delta;\emptyset\vdash\mathtt{T}_2\mathtt{<:}\mathtt{T}_1\\ \mathtt{F}_1=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{F}_2 & \overline{\mathtt{W}}\mathtt{<:}\overline{\mathtt{S}}\mathtt{\subseteq}\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{Q}} & \Delta;\emptyset\vdash\overline{\mathtt{S}}\mathtt{<:}\overline{\mathtt{P}}\end{array}}{\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_1\sqsubseteq_R R_2} \quad \text{(SB-}R\text{)}$$

Reflective range disjointness:

$$\frac{\Delta\vdash{+}R_p\otimes o'R'_n\ \text{ or }\ \Delta\vdash{+}R'_p\otimes oR_n\ \text{ or }\ \Delta\vdash oR_n\otimes o'R'_n}{\Delta\vdash \mathit{disjoint}(\langle R_p,oR_n\rangle,\langle R'_p,o'R'_n\rangle)} \quad \text{(DS-}\Lambda\text{)}$$

Mutually exclusion of range conditions:

$$\frac{o\neq o' \qquad \begin{cases}\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash R_1\sqsubseteq_R R_2 & \mathtt{if}\ \ o=+\\ \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash R_2\sqsubseteq_R R_1 & \mathtt{otherwise}\end{cases}}{\Delta\vdash oR_1\ \otimes\ o'R_2} \quad \text{(ME)}$$

Figure 9 FDJ: Well-formness rules and auxiliary definitions.

Well-formed typing context:

$$\frac{\Delta \vdash \Lambda \;\; ok \quad \Delta;\Lambda \vdash \mathit{range}(\Gamma) : \mathit{range}(\Delta) \;\; ok}{\Delta;\Lambda;\Gamma \;\; ok} \qquad \text{(WF-TYCON)}$$

Well-formed types:

$$\Delta;\Lambda \vdash \texttt{Interface} \;\; ok \qquad \text{(WF-INTER)}$$

$$\frac{\texttt{X} \in \mathit{dom}(\Delta) : \overline{\texttt{X}} \qquad \Lambda = \langle (\texttt{T}_1,\ \texttt{<}\overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}\texttt{>F}_1), (\texttt{T}_2,\ \texttt{<}\overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}\texttt{>F}_2) \rangle}{\Delta;\Lambda \vdash \texttt{X} \;\; ok} \qquad \text{(WF-VAR)}$$

$$\frac{\begin{array}{c}\Delta;\Lambda \vdash \overline{\texttt{T}} \;\; ok \quad [\![\texttt{C}]\!] = \overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}};\texttt{N} \\ \Delta;\Lambda \vdash \mathit{concrete}(\texttt{N}\texttt{:}\overline{\texttt{N}}\texttt{,false}) \quad \Delta;\Lambda \vdash \overline{\texttt{T}} \texttt{<:} [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{N}} \\ \Delta \vdash \mathit{implement}(\texttt{C}\texttt{<}\overline{\texttt{T}}\texttt{>}) \qquad \overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}};\emptyset \vdash \texttt{N}\texttt{:}\overline{\texttt{N}} \;\; ok \quad \texttt{X} \text{ not in } \Delta,\Lambda \\ \Delta;\Lambda \vdash \mathit{concrete}(\texttt{C}\texttt{<}\overline{\texttt{T}}\texttt{>,true}) \text{ or } \Delta;\Lambda \vdash \mathit{concrete}(\texttt{C}\texttt{<}\overline{\texttt{T}}\texttt{>,false})\end{array}}{\Delta;\Lambda \vdash \texttt{C}\texttt{<}\overline{\texttt{T}}\texttt{>} \;\; ok} \qquad \text{(WF-CLASS)}$$

$$\frac{\Delta;\Lambda \vdash \texttt{T}_1\texttt{,}\,\texttt{T}_2 \;\; ok}{\Delta;\Lambda \vdash \texttt{T}_1 \texttt{->} \texttt{T}_2 \;\; ok} \qquad \text{(WF-FTYP)}$$

Well-formed Reflective Environments:

$$\frac{\begin{array}{c}\Lambda = \langle (\texttt{T}_1,\ \texttt{<}\overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}\texttt{>F}_1), o(\texttt{T}_2,\ \texttt{<}\overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}\texttt{>F}_2) \rangle \text{ implies } \mathit{dom}(\Delta) \cap \overline{\texttt{X}} = \emptyset \\ \Delta;\Lambda \vdash \mathit{concrete}(\overline{\texttt{N}}\texttt{,false}) \qquad \Delta;\emptyset \vdash \overline{\texttt{N}}\texttt{:}\,\texttt{T}_{1,2} \;\; ok \quad \Delta;\Lambda \vdash \texttt{F}_{1,2} \;\; ok\end{array}}{\Delta \vdash \Lambda \;\; ok} \qquad \text{(WF-}\Lambda\text{)}$$

Figure 10 FDJ: Subtyping rules.

Subtyping rules:

$$\Delta;\Lambda \vdash \texttt{T} \texttt{<:} \texttt{T} \qquad \text{(S-REFL)}$$

$$\frac{\Lambda = \langle (\texttt{T}_1,\ \texttt{<}\overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}\texttt{>F}_1), (\texttt{T}_2,\ \texttt{<}\overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}\texttt{>F}_2) \rangle \qquad \Delta' = \Delta : \overline{\texttt{X}} \texttt{<:} \overline{\texttt{N}}}{\Delta;\Lambda \vdash \texttt{X} \;\texttt{<:}\; \Delta'(\texttt{X})} \qquad \text{(S-VAR)}$$

$$\frac{\Delta;\Lambda \vdash \texttt{T}_1 \texttt{<:} \texttt{T}_2 \quad \Delta;\Lambda \vdash \texttt{T}_3 \texttt{<:} \texttt{T}_4}{\Delta \vdash \texttt{T}_1 \texttt{<:} \texttt{T}_4} \qquad \text{(S-TRANS)}$$

$$\frac{\begin{array}{c}[\![\texttt{C}]\!] = \overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}};\texttt{N} \\ \Delta;\Lambda \vdash \texttt{fields}[\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N} = \bullet\end{array}}{\Delta;\Lambda \vdash \texttt{C}\texttt{<}\overline{\texttt{T}}\texttt{>} \;\texttt{<:}\; [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}} \qquad \text{(S-CLASS)}$$

must be structurally equal to `new C<`$\overline{\texttt{T}}$`>(`$\overline{v}$`)` (i.e., v_i'=`new C<`$\overline{\texttt{T}}$`>(`$\overline{v}$`)` $::_k v''$). Therefore a relation is established via operator $::_i$ between its left and right element (i.e., the head of the access path is the i_{th} field of the head's successor). If the remaining access path v is not empty, then it must also be well-typed. Notice that the type assigned to an access path is identical to the type of its head element.

4.3.2 Disjointness and Containment

One of the core aspects of FDJ is the ability to invoke, override, and declare reflective methods. In order to safely invoke or override a method from another method, the reflective environment of the latter method must be *contained* in the reflective environment of the former method. (This is a conservative restriction, used to make reasoning over containment more manageable.) We use the predicate $\Delta;[\overline{\texttt{W}}/\overline{\texttt{Y}}] \vdash \langle R_p, oR_n \rangle \sqsubseteq_\Lambda \langle R_p', oR_n' \rangle$ (defined in Figure 8) to denote that $\langle R_p, oR_n \rangle$ is contained in $\langle R_p', oR_n' \rangle$. Notice that both environments must share the same sign o. $\overline{\texttt{W}}$ denote the pattern variables of the left operand of $\sqsubseteq_\Lambda$ that must be substituted for the pattern variables $\overline{\texttt{Y}}$ of the right operand in order for operand containment to hold (i.e., $[\overline{\texttt{W}}/\overline{\texttt{Y}}]$ can be seen as the outcome of a unification process). Reflective environment containment is expressed as range containment denoted as $\Delta;[\overline{\texttt{W}}/\overline{\texttt{Y}}] \vdash R_1 \sqsubseteq_R R_2$ between the ranges of reflective environments. When o is of the form – then the

Figure 11 FDJ: Auxiliary definitions.

Translation Functions:

$$\frac{\mathbb{M}_p{=}\texttt{n:F}_1\ \texttt{in T.methods} \quad \mathbb{M}_f{=}\texttt{n:F}_2\ \texttt{in T}'\texttt{.methods}}{[\![\texttt{<}\overline{\texttt{Y}}\texttt{<:}\overline{\texttt{P}}\texttt{>for}\ (\mathbb{M}_p;o\mathbb{M}_f)\,]\!] = \langle(\texttt{T},\ \texttt{<}\overline{\texttt{Y}}\texttt{<:}\overline{\texttt{P}}\texttt{>F}_1), o(\texttt{T}',\ \texttt{<}\overline{\texttt{Y}}\texttt{<:}\overline{\texttt{P}}\texttt{>F}_2)\rangle}$$

$$\frac{}{[\![\bullet]\!] = \emptyset}$$

$$\frac{CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}\texttt{>}\lhd\texttt{N}\ \{\ \ldots\ \}}{[\![\texttt{C}]\!] = \overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}};\texttt{N}}$$

Subtype range validity:

$$\frac{\begin{array}{l}\text{for all } \texttt{C<}\overline{\texttt{T}}\texttt{>,n}\ \ \Delta;\emptyset\vdash\texttt{N<:C<}\overline{\texttt{T}}\texttt{>} \text{ and } \Delta;\Lambda\vdash \mathit{method}(\texttt{n},\ \texttt{C<}\overline{\texttt{T}}\texttt{>}){=}\mathfrak{R}\ \ \texttt{n:F}\ \ b \text{ implies} \\ \Lambda{=}\emptyset \Leftrightarrow \mathfrak{R}{=}\bullet \quad \text{and} \quad (\mathfrak{R}{\neq}\bullet \text{ implies } \Delta;[\overline{\texttt{W}}/\overline{\texttt{Y}}]\vdash\Lambda\sqsubseteq_\Lambda [\![\mathfrak{R}]\!] \text{ or } \Delta\vdash \mathit{disjoint}([\![\mathfrak{R}]\!],\ \Lambda))\end{array}}{\Delta \vdash \mathit{validRange}(\Lambda, \texttt{N})}$$

Concrete classes:

$$\frac{\begin{array}{c}\neg\texttt{arg} \text{ implies } \Delta;\Lambda\vdash \mathit{fields}(\texttt{N}) = \emptyset \\ \text{for all } \mathfrak{R}\ \ \texttt{n:F}\ \ b\ \in CT(\texttt{N}) \quad (b\ \neq\ \bullet \Leftrightarrow \texttt{arg})\end{array}}{\Delta;\Lambda\vdash \mathit{concrete}(\texttt{N},\texttt{arg})}$$

Field lookup:

$$\frac{\begin{array}{c}\Delta;\Lambda\vdash[\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{S}}\ \ \mathit{ok} \quad \texttt{N=Interface} \text{ and } \overline{\texttt{S}}\ \ \overline{\texttt{f}}{=}\bullet \text{ or} \\ \texttt{N=C<}\overline{\texttt{T}}\texttt{>} \text{ and } CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}}\texttt{<:}\overline{\texttt{N}}\texttt{>}\lhd\texttt{N}\ \{\overline{\texttt{S}}\ \ \overline{\texttt{f}}\texttt{;}\ \ \ldots\}\end{array}}{\Delta;\Lambda\vdash \mathit{fields}(\texttt{N}) = [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{S}}\ \ \overline{\texttt{f}}}$$

Implement relation:

$$\frac{\text{for all } \texttt{T},\Lambda,\texttt{n} \quad \Delta;\Lambda\vdash\texttt{N<:T} \text{ and } \Delta;\Lambda\vdash \mathit{mtype}(\texttt{n},\ \texttt{T}){=}\texttt{F} \text{ implies } \Delta;\Lambda\vdash \mathit{mtype}(\texttt{n},\ \texttt{N}){=}\texttt{F}}{\Delta\vdash \mathit{implement}(\texttt{N})}$$

order of nested ranges is swapped. The two candidate pairs for range containment (R_p,R'_p) and (R_n,R'_n) (or (R'_n,R_n), if swapped) must satisfy the containment condition for the same substitution map $[\overline{\texttt{W}}/\overline{\texttt{Y}}]$. Two ranges R and R' are contained when their method signatures match once pattern substitution is performed and the statically iterated type of the latter range is a subtype of the statically iterated type of the former range.

Reflective methods share the same name η. Therefore, the type system employs reflective method *disjointness* (defined in Figure 8) to guarantee absence of reflective method conflicts in class declarations. Disjointness between two reflective environments is expressed in terms of containment: two reflective environments are disjoint when there exists a pair of ranges from each reflective environment such that ranges have opposite signs and one range is contained in the other range. (This condition is conservative and can be weakened. In the full language, two positive ranges can be disjoint, e.g., by exploiting the fact that two concrete types `int` and `Object` are guaranteed distinct, hence patterns iterating over signatures that contain them will never have common members. We chose not to model this aspect in FDJ as it is orthogonal to the core reasoning.)

4.3.3 Interface and class typing

An important predicate for typing classes and "interfaces" is *concrete* (defined in Figure 11), taking two parameters, a type `T` and a boolean flag. When the flag is set to *true* the predicate holds when there exists no method with empty body in `T`. When the flag is *false*, then `T` must have the properties of an interface: it must only contain methods (no fields) with empty bodies. (Thus, $\mathit{concrete}(\texttt{C<}\overline{\texttt{T}}\texttt{>}, \mathit{false})$ is not the negation of $\mathit{concrete}(\texttt{C<}\overline{\texttt{T}}\texttt{>}, \mathit{true})$.) Therefore, premise *concrete* in rule T-NEW enforces the invariant that only concrete classes with full method implementations can be instantiated. *concrete* is also used in class well-formedness requiring that type upper bounds (including parent classes) be interfaces. Classes must implement all methods of their parent interface. This is enforced by predicate *implement*, defined in Figure 11.

T-CLASS, the rule for typing classes, enforces the following invariants: (a) classes can either contain reflective or standard method declarations but not both (b) standard method declarations have distinct names (c) reflective method declarations are *disjoint* (d) each reflective declaration is either *disjoint* or *contained* with respect to the reflective declarations of the class supertypes (using predi-

cate $validRange$) (e) each method must be well-typed using rule T-METH, which performs *conditional* type-checking to the method body b, when it is non-empty and (f) well-formedness of declared types and the typing context (defined in Figure 9).

4.3.4 Method Invocation Typing

Function invocations are typed via T-INVK, which employs function $mtype$ in order to determine the type signature F of a method n, member of some type T. The definition of function $mtype$ is provided in Figure 7.

Rule MT-VAR is applied for method signature lookup inside type or pattern variables. The lookup algorithm first visits the upper bound of the specified type variable and on failure it looks up positive structural constraints entailing the requested variable in the reflective environment. Rule MT-CLASS is applied when looking up a method signature in a class type. The lookup algorithm attempts to extract the type signature from the specified class type, using function $method$ and on failure it continues the lookup on the parent class provided that the specified class in an interface.

Function $method$, defined in rule M-CLASS, selects a method such that a pattern variable substitution map $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]$ exists for the method's reflective environment $\mathfrak{R}$. It returns the selected method having substituted class-local type and pattern variables for variables of the typing context as well as the method name. If the class contains standard methods then the substitution environment is empty and the requested method name must match the method's name. Otherwise, the class contains reflective methods (recall that a class cannot contain both standard and reflective methods) and predicate $specialize$ is employed for determining the pattern variable substitution map. The form of rule M-CLASS may at first seem strange. In order to reach the "otherwise" clause, one needs to prove that the $specialize$ predicate cannot hold. This is easy to establish via syntactic conditions, however, since, for an empty $\mathfrak{R}$ (i.e., for non-reflective methods), Λ_d will be empty, and thus can never match an outcome of a rule that establishes $specialize$.

If the requested method name is variable, i.e., of the form η, then the reflective environment of $\mathfrak{R}$ must *contain* the reflective environment of the typing context (rule SP1). Otherwise, rule SP2 applies and $mtype$ is employed on the first element of the primary range corresponding to $\mathfrak{R}$. The type signature returned by $mtype$ must be equal to the method signature specified by the second element of the primary range modulo the substitution map $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]$. If the secondary range is positive then $mtype$ must return the same substitution map. Otherwise, $mtype$ is either undefined or the returned substitution map does not match the map of the primary range.

4.4 Soundness

In this section, we state the soundness of FDJ as a result of Subject Reduction and Progress lemmas for an expression e.

Theorem 1 (Subject Reduction)**.** *If* $\emptyset;\emptyset;\emptyset\vdash \mathtt{e}\in\mathtt{T}$ *and* $\mathtt{e}\rightarrow\mathtt{e}'$*, then for some* S, $\emptyset;\emptyset;\emptyset\vdash \mathtt{e}'\in\mathtt{S}$ *and* $\emptyset;\emptyset;\emptyset\vdash \mathtt{S}<:\mathtt{T}$.

Proof sketch. By structural induction on the reduction rules. In the case of rules RC-INV-ARG and RC-NEW-ARG, their subterms make a step and are well-typed by inversion of T-INVK and T-NEW respectively. The induction hypothesis is then applied and the new subterm typing derivations are substituted in the premises of T-INVK and T-NEW, respectively. In the case of R-NEW the proof is immediate by using $\emptyset;\emptyset;\emptyset\vdash \mathtt{new\ C}\!<\!\overline{\mathtt{T}}\!>\!(\overline{v})\in\mathtt{C}\!<\!\overline{\mathtt{T}}\!>$ and T-VAL. In R-FIELD we have that v_i and $\mathtt{new\ C}\!<\!\overline{\mathtt{T}}\!>\!(\overline{v})::_j v$ are well-typed, by inversion of $\emptyset;\emptyset;\emptyset\vdash(\mathtt{new\ C}\!<\!\overline{\mathtt{T}}\!>\!(\overline{v})::_j v).\mathtt{f}_i\in\mathtt{T}$. Therefore, $\mathtt{new\ C}'\!<\!\overline{\mathtt{T}}'\!>\!(\overline{v}')::_i(\mathtt{new\ C}\!<\!\overline{\mathtt{T}}\!>\!(\overline{v})::_j v)$ is well-typed by T-VAL. The type of $\mathtt{new\ C}'\!<\!\overline{\mathtt{T}}'\!>\!(\overline{v}')$, which is identical to the type of v_i, is a subtype of T by T-NEW. In RC-FIELD, the application of the induction hypothesis implies the new subterm is well-typed with a subtype $\mathtt{S}_1$ of the original subterm type $\mathtt{T}_1$. $\mathtt{T}_1$ has at least one field, thus the *concrete* predicate holds. It suffices to show that if $\emptyset;\emptyset\vdash fields(bound_{\emptyset;\emptyset}(\mathtt{T}_1))=\overline{\mathtt{T}}\ \overline{\mathtt{f}}$, then $\emptyset;\emptyset\vdash fields(bound_{\emptyset;\emptyset}(\mathtt{S}_1))=\overline{\mathtt{T}}\ \overline{\mathtt{f}}$. $\mathtt{S}_1$ and $\mathtt{T}_1$ can only be identical as a consequence of the well-formedness relation: all supertypes of a type have no fields. In RC-INV-RECV, the new subterm is well-typed with $\mathtt{S}_1$, a well-formed subtype of the original subterm type $\mathtt{T}_1$, using the induction hypothesis. It suffices to show that if $\emptyset;\emptyset\vdash mtype(\mathtt{n},\ \mathtt{T}_1)=\mathtt{T}'\rightarrow\mathtt{T}''$, then $\emptyset;\emptyset\vdash mtype(\mathtt{n},\ \mathtt{S}_1)=\mathtt{T}'\rightarrow\mathtt{T}''$. The well-formedness of $\mathtt{S}_1$ implies $\emptyset\vdash implement(\mathtt{S}_1)$, which completes the proof. In the case of R-INVK, the typing derivation of v and $mbody(\mathtt{m},\ v)=(\mathtt{x},\mathtt{e},v'')$ imply that v'' is well-typed with $\mathtt{T}'=\mathtt{D}\!<\!\overline{\mathtt{T}}\!>$ and $\emptyset;\emptyset\vdash method(\mathtt{m},\ \mathtt{T}')=\mathtt{M}$, where M is the concrete method to be executed. T-METH implies the body e of M is well-typed: $\overline{\mathtt{X}}<:\overline{\mathtt{N}};\Lambda_d;\mathtt{this}\mapsto\mathtt{T}',\mathtt{x}\mapsto\mathtt{T}_1\vdash\mathtt{e}\in\mathtt{T}_2$, where $\mathtt{T}_2$ is a subtype of M's return type. Using the type variable substitution lemma we have that $\emptyset;\emptyset;[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\,(\mathtt{this}\mapsto\mathtt{T}',\mathtt{x}\mapsto\mathtt{T}_1)\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\mathtt{m}/\eta]\mathtt{e}\in[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_2$. Notice that method variables are also substituted in this step and Λ_d is removed. T-INVK implies v' is well-typed for some subtype of $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_1$. The proof is completed by applying the variable substitution lemma. □

Theorem 2 (Progress)**.** *If* $\emptyset;\emptyset;\emptyset\vdash \mathtt{e}\in\mathtt{T}$ *holds, then* e *is either a value* v *or it can be evaluated to another expression* $\mathtt{e}'$.

Proof sketch. By structural induction on the typing derivation of e. In the case of T-VAL the proof is immediate. T-VAR does not apply as the typing context is empty. In the case of T-NEW, we perform case analysis on the shape of e. If e is of the form $\mathtt{new\ C}\!<\!\overline{\mathtt{T}}\!>\!(\overline{v})$, rule R-NEW applies. Otherwise, the induction hypothesis to the first non-value argument is applied and the proof is complete by rule RC-NEW-ARG. Similar reasoning is applied for T-FIELD using R-FIELD and RC-FIELD and T-INVK when e is not of

the form v.m(v') using RC-INV-RECV and RC-INV-ARG respectively. The last case of T-INVK is when e is of the form v.m(v'). We use the value lemma saying that when $\emptyset;\emptyset;\emptyset \vdash v \in \mathtt{T}'$ and $\emptyset;\emptyset;\emptyset \vdash mtype(\mathtt{m},\ \mathtt{T}')$ then $mbody(\mathtt{m},\ v)$ is defined and rule R-INVK is employed to perform a step. □

Theorem 3 (Type Soundness). *If* $\emptyset;\emptyset;\emptyset \vdash \mathtt{e} \in \mathtt{T}$ *and* $\mathtt{e} \longrightarrow^* \mathtt{e}'$, *then* $\mathtt{e}'$ *is either a value* v *or it can be evaluated to another expression* $\mathtt{e}''$.

Proof sketch. Conclusion follows from Theorem 1 and Theorem 2 □

5. Related Work

The aspects of our language design discussed in Sections 2 and 3 make it unique. Nevertheless, there are several approaches that relate to different facets of our design.

Delegation has been proposed for both class-based and object-based languages. Object-based languages (e.g., Javascript) do not support class-based inheritance or subtyping and behavior sharing is realized between objects by explicitly setting in runtime their parents. Self [32] is an object-based language that implements forwarding of messages. Each object can understand messages that correspond to an object's slot, or, if any slots are indicated as parents, forward the message to them. However, this approach suffers from potential runtime errors when invocations can produce missing method exceptions, due to the lack of static-typing guarantees. Furthermore, there is a fundamental difference between dispatch chaining in dynamic languages and our approach. In object-based languages, the dispatch mechanism depends on an object knowing its parents, whereas in our case the dispatch chain is different for different references to the same object.

Kniesel [19] proposed the DARWIN model using delegation as a complement of forwarding-based object composition. In this model, child objects extend the behavior of parent objects, by employing both delegation and consultation during message dispatch. The implementation of DARWIN, LAVA, supports *dynamic* delegation: two independently developed components can be composed at runtime in a type- and semantically sound manner, by restricting the declared parent type to conform to a certain interface. The DARWIN model does not provide full late binding support for methods of composed objects. Therefore it is impossible to use redefined objects that delegate method invocations with their `self` rebound.

Generic Wrappers [4] support *static* delegation between "wrappers" and "wrappees" via statically declared `wraps` clauses. Thus, the wrapper-wrappee relationship is fixed, just as in class-based inheritance.

Delegation Layers [25] combine delegation and virtual classes to provide polymorphic runtime composition and on-the-fly extensibility. Extensions have only a local effect thus both the original and the modified behaviors are accessible. The desired behavior can be selected by referring to a collaborating object via variables of different static type. In our approach, a new access path is created when accessing an object through `subobject` fields of wrapper objects. This strategy keeps the semantics simple and gives a clearer view to the programmer of when an object's behavior is extended.

The Compound Reference (CR) model proposed by [26, 27] introduces a mechanism to complement ordinary references using redirection semantics. A CR is in general a path of object references in an object tree. Once the path is constructed, it is considered immutable as a whole. The CR has a static type and a temporary type. The static type of the CR is the static type of its tail and the temporary type is updated every time a field update takes place. Thus, CRs are incrementally composed as a sequence of regular references to objects that are immutable as a whole, while the access path can change over time due to instance variable changes. Overriding in a class `C` is realized through implicit methods that are created over every field `f`. Method calls are dispatched (from `C` to `f`), based on these implicit methods. In the CR model, self messages will not be dispatched to wrapper objects; hence the model does not support late binding, in the sense of our work. The implicit methods of the CR model can be encoded in DelphJ using static-for patterns.

The FeatherWeight Wrap Java work [2], presents a formal semantics for a simple wrapper-based language. The language focuses on two aspects: wrapping precedence order and the binding of self. However, FeatherWeight Wrap Java employs "method specialization" and not complete overriding of a wrapper object as in DelphJ. Additionally, a class in [2] cannot wrap more than one object. In our work, the functionality of the "delegate after" and "delegate before" special keywords from FeatherWeight Wrap Java can be easily encoded using static-for patterns.

Bettini et al. [1] implement an object composition mechanism that uses structural subtyping, while maintaining nominal subtyping (extends relationship) among types. That work has similarities with our concept of access paths. Bettini et al.'s subobjects are shared and their "access paths" counterparts are extended using a special composition operator, which *allocates* a new object representing the new access path. In our work, when `subobject` fields are read no allocation takes place. In addition, it is possible to "re-compose" objects, by updating their fields.

Accidental method overriding can arise from poor design decisions in class hierarchies, or more complex scenarios that involve mixins. MixedJava [8], McJava [14] and improved variants [15] deal with this problem extensively through context aware conflicting method resolution meaning that run-time context information is used, called view, to determine which method should be invoked when an accidental overriding exist.

Bettini et al. [1] also propose a similar solution. In DelphJ, we adopt the strategy initially proposed

by Kniesel [20]: only non-final methods declared in `subobject` field types can be overridden.

Schippers et al. [29, 30] have developed a delegation-based machine model for aspect-oriented programming. They present semantic mappings of four high-level programs to their machine model that ensures semantic equivalence between source and the delegation-enabled translation. A `Proxy` object is the basic abstraction entity that is responsible for receiving messages directed to an object and delegating them to the actual receiver. `Proxy` objects are inserted and removed in delegation chains (a concept similar to the subobject *access path*) by static and dynamic weaving. The dispatch algorithm maintains a list of messages along with their implementations, as well as a constant function Del_i, which determines the address of the delegate of an object at some address. (The function is prepared statically and maintained dynamically.) This function is updated when the store of objects is updated. A `Look` function which recursively traverses the delegation list, uses the Del_i function.

6. Conclusions

Inheritance has primary importance in most OO languages, yet its value is routinely questioned. Despite the shortcomings of inheritance, there has not been a replacement proposed that addressed them without sacrifices. We have presented a language design that aspires to do so. Our design eliminates inheritance without sacrificing ease of code reuse. Our language supports subtyping (via interfaces), morphing (for controlled automatic forwarding of method calls) and deep delegation (for late binding semantics and refinement of existing code). Through our exploration of the combination of these features, we showed interesting subtleties in supporting flexible (per-field) late binding semantics and discussed design choices that address common problems.

We believe that our design can inspire OO languages that move beyond inheritance to achieve highly flexible, modular programming without sacrificing either power or control.

Acknowledgments

We gratefully acknowledge funding by the Greek Secretariat for Research and Technology under the "MorphPL" Excellence (Aristeia) award; and by the European Union under a Marie Curie International Reintegration Grant and a European Research Council Starting/Consolidator grant.

References

[1] Lorenzo Bettini, Viviana Bono, and Betti Venneri. Delegation by object composition. *Science of Computer Programming*, 76(11):992–1014, November 2011.

[2] Lorenzo Bettini, Sara Capecchi, and Elena Giachino. Featherweight wrap java. In *Proc. Symp. on Applied Computing (SAC)*, pages 1094–1100, Seoul, Republic of Korea, 2007.

[3] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 25, pages 303–311, 1990.

[4] Martin Büchi and Wolfgang Weck. Generic wrappers. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, pages 201–225, Sophia Antipolis and Cannes, France, 2000.

[5] William R. Cook, Walter Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proc. Symp. on Principles of Programming Languages (POPL)*, pages 125–135, San Francisco, USA, 1989.

[6] Stphane Ducasse, Oscar Nierstrasz, Nathanael Schrli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems*, 28(2):331–388, 2006.

[7] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–18, New York, NY, 2007.

[8] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proc. Symp. on Principles of Programming Languages (POPL)*, pages 171–183, San Diego, USA, 1998.

[9] Allen Holub. Why extends is evil: Improve your code by replacing concrete base classes with interfaces. http://www.javaworld.com/javaworld/jw-08-2003/jw-0801-toolbox.html, August 2003.

[10] Shan Shan Huang and Yannis Smaragdakis. Expressive and safe static reflection with MorphJ. In *Proc. Conf. on Programming Language Design and Implementation (PLDI)*, volume 43, pages 79–89, Tucson, AZ, USA, 2008.

[11] Shan Shan Huang and Yannis Smaragdakis. Morphing: Structurally shaping a class by reflecting on others. *ACM Transactions on Programming Languages and Systems*, 33(2):1–44, February 2011.

[12] Shan Shan Huang, David Zook, and Yannis Smaragdakis. Morphing: Safely shaping a class in the image of others. In Erik Ernst, editor, *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 4609 of *LNCS*, pages 303–329, July 2007.

[13] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[14] Tetsuo Kamina and Tetsuo Tamai. McJava – a design and implementation of java with mixin-types. In *Proc. of Asian Programming Languages and Systems Symp. (APLAS)*, pages 4–6, Taipei, Taiwan, 2004.

[15] Tetsuo Kamina and Tetsuo Tamai. Selective method combination in mixin-based composition. In *Proc. Symp. on Applied Computing (SAC)*, pages 1269–1273, Santa Fe, New Mexico, 2005.

[16] Majorinc Kazimir. Ellipse-circle dilemma and inverse inheritance. In *Proceedings of the 20th International Conference on Information Technology Interfaces*, Pula, Croatia, 1998.

[17] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353, London, UK, 2001.

[18] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242. Springer, Heidelberg, Germany, and New York, 1997.

[19] Günter Kniesel. Type-safe delegation for run-time component adaptation. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, pages 351–366, Lisbon, Portugal, 1999.

[20] Günter Kniesel. *Dynamic object-based inheritance with subtyping*. PhD thesis, Universität Bonn Institut für Informatik III, 2000.

[21] Günter Kniesel, Mechthild Rohen, and Armin B. Cremers. A management system for distributed knowledge base applications. In *Verteilte Künstliche Intelligenz und kooperatives Arbeiten, 4. Internationaler GI-Kongress Wissensbasierte Systeme*, pages 65–76, 1991.

[22] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 214–223, New York, NY, USA, 1986.

[23] Mira Mezini. Dynamic object evolution without name collisions. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, pages 190–219, Jyväskylä, Finland, 1997.

[24] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, pages 355–382, Brussels, Belgium, 1998.

[25] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, pages 89–110, Nantes, France, 2006.

[26] Klaus Ostermann and Mira Mezini. Object-oriented composition untangled. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 283–299, New York, NY, USA, 2001.

[27] Klaus Ostermann and Mira Mezini. Blurring the borders between object composition, inheritance, and delegation. In *Proceedings of the Inheritance Workshop at the 16th European Conference on Object-Oriented Programming*, pages 65–68, 2008.

[28] Nathanael Scharli, Stephane Ducasse, Oscar Nierstrasz, and Andrew Black. Traits: Composable units of behavior. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, Darmstadt, Germany, 2003.

[29] Hans Schippers, Michael Haupt, and Robert Hirschfeld. An implementation substrate for languages composing modularized crosscutting concerns. In *Proc. Symp. on Applied Computing (SAC)*, SAC '09, pages 1944–1951, New York, NY, USA, 2009.

[30] Hans Schippers, Dirk Janssens, Michael Haupt, and Robert Hirschfeld. Delegation-based semantics for modularizing crosscutting concerns. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 525–542, New York, NY, USA, 2008.

[31] Marcelo Sihman and Shmuel Katz. Superimpositions and aspect-oriented programming. *The Computer Journal*, 46(5):529–541, 2003.

[32] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 227–242, New York, NY, USA, 1987.