Revisiting Multiple Inheritance for Modularity and Reuse

by

Yanlin Wang

A thesis submitted in partial fulfillment of the requirements for
the degree of Doctor of Philosophy
at The University of Hong Kong

July 2019

Abstract of thesis entitled

"Revisiting Multiple Inheritance for Modularity and Reuse"

Submitted by

Yanlin Wang

for the degree of Doctor of Philosophy
at The University of Hong Kong
in July 2019

Since software modularity makes software systems controllable, maintainable and extensible, improving software modularity has been a hot topic in the programming community for many years since the beginning of programs. Researchers and software developers often apply different modularity techniques to reduce the complexity of software systems and make them more modular. These techniques include new design patterns, new programming languages, new type systems and tool support. Object-Oriented Programming (OOP) is a programming paradigm, which has been investigated and used for a long time. It improves modularity by modeling the world as objects and providing abstractions by wrapping data and functions into a single unit. Later, the idea of multiple inheritance is proposed to further improve modularity in OOP. However, there are still unsolved (or not elegantly solved) problems such as the Expression Problem and issues in supporting multiple inheritance.

This thesis investigates various software modularity and multiple inheritance related issues in the context of OOP. In particular, we divide the thesis into two parts: (I) multiple inheritance for modularity and reuse, and (II) revisiting models of multiple inheritance. In part I, we explore the canonical modularity problem (the Expression Problem) and discuss various existing solutions. These solutions are not satisfactory because they are complex and require either advanced type systems or complex language features. Our proposed solution in both Scala and Java uses simple subtyping feature and it shows that only subtyping is enough to solve the EP. Then we propose a new programming paradigm called interface-based programming, as opposed to class-based programming and prototype-based programming. The purpose is to lower coupling among software components. We implement this programming style into Java and it is called Classless Java, where OO programs and reusable libraries can be defined and used without defining a single class.

In part II, we explore the conflicts issues in multiple inheritance. In particular, we explore the unintentional conflict problems, which previous approaches/models either do not support or cannot provide satisfactory solutions. The reason is that they have to compromise between code reuse and type safety. Moreover, they do not support hierarchical overriding in the presence of unintentional conflicts. We propose the formal multiple inheritance model **FHJ**, solving the unintentional method conflicts issue by supporting *hierarchical dispatching* and *hierarchical overriding* algorithms. To further solve the unintentional state conflicts problem and provide a general treatment to the diamond problem, we propose **FHJ+** as an extension to **FHJ**. Both models are formalized following the style of Featherweight Java. We also propose many properties of the two models to prove the models to be sound.

In summary, this thesis investigates and solves the interesting problems around modularity and multiple inheritance, providing insights for future researchers to improve software modularity further.

An abstract of exactly 294 words

Declaration

I declare that this thesis represents my own work, except where due acknowledgment is made, and that it has not been previously included in a thesis, dissertation or report submitted to this University or to any other institution for a degree, diploma or other qualifications.

. . . .

Yanlin Wang

July 2019

Acknowledgments

I would like to express my special thanks to my PhD supervisor Prof. Bruno C. d. S. Oliveira for your valuable guidance and insightful discussions about the research. I still remember in my first year (after my PhD to MPhil transfer), at the time when I need to find a new supervisor, Jeremy told me that his supervisor is smart, handsome, and doing interesting research on PL. The timing is perfect, we had our first meeting in your office. And after the meeting, I told myself: "it would be incredibly great if he is willing to take me as his student". Then soon the dream come true and I joined your PL research group. Later I transferred back to the PhD program because you gave me enough faith to pursue a PhD degree. For me, you are the best role model for a researcher, mentor, and teacher. Your support, advise, and enthusiasm for research will keep encouraging me in the future.

I would also like to thank Prof. Marco Servetto from Victoria University of Wellington and my excellent colleague Haoyuan Zhang. You two are the most important coauthors of around half my PhD publications and the primary source of getting my research questions answered. I cherish the time of the whole discussion, brainstorming, experimenting, and paper writing process with you guys.

I thank all the members of the PL group: Xuan Bi, Tomas Tauber, Weixin Zhang, Yanpeng Yang, Huang Li, Ningning Xie, Jinxu Zhao, Yaoda Zhou, and Xuejing Huang. We cooperated and supported each other, and I will not forget the excellent lab life we had together and best wishes to all of you.

I also have to thank the members of my PhD committee, Professors Atsushi Igarashi, Giulio Chiribella and K.P. Chow for your helpful suggestions in general. Especially, I want to thank Professor Igarashi for the discussion and suggestions on the model and proof of FHJ+. Also I would like to thank my first PhD supervisor Prof. Nikos Mamoulis for giving me the chance to become a PhD student in HKU. Although we could not do database research together, I enjoyed the dinners every time you come back to Hong Kong. Your humor sense, wisdom and caring for students always inspire me to be an interesting person.

I especially thank my friends (too many to list here) for listening, encouraging and supporting me through the entire PhD process. This is definitely not an easy journey. I would not have had a beautiful PhD life without you guys.

I thank my family members (especially my parents) for providing unconditional love and constant support. Thank you for having faith in me and for giving me freedom when making decisions. Thank you for always being there for me. I would not have been the current me without you.

Last but not least, I thank God Almighty for accompanying me, not giving me up but giving me the strength in my darkest time. Glory to my Lord my God for Your endless grace and mercy.

Contents

List of Figures

List of Tables

1 Introduction

In the Object-Oriented Programming community, people strive to improve software modularity. Two main OO models emerged to this end: prototype-based languages (PB) such as Self [Ungar and Smith, 1987a] and class-based languages (CB) such as Java, C# or Scala. In prototype-based languages, objects inherit from other objects. Thus objects own both behavior and state (and objects are all you have). In class-based languages, an object is an instance of a specific class, and classes inherit from other classes. Objects own state, while classes contain behavior and the structure of the state. Concepts such as multiple inheritance have been proposed to improve modularity. However, people also suffer from the problems accompanying multiple inheritance. Such problems include member conflicts, the Expression Problem [Wadler, 1998], state issues, etc.

In this thesis, we will investigate various modularity and multiple inheritance-related issues in the context of Object-Oriented Programming. We will discuss the Expression Problem and provide a simple solution (as a new design pattern) to it. We will study various multiple inheritance models, such as the trait model, the mixin model, and the C++ model. Also, we will analyze their advantages and disadvantages. Meanwhile, we will discuss the multiple inheritance related issues in these languages and provide our solutions to them. The programming language techniques we use include but are not restricted to design patterns, language features, and formal calculi.

1.1 Modularity

Modularity techniques, in simple words, are techniques to divide systems into multiple components. In principle, each of these components should implement one functionality and be relatively independent. They can communicate with each other via interface signatures. Software systems are getting more and more complicated as they are proliferating as time goes. Software designers and developers often apply different modularity techniques, which are essential to alleviate the complexity of software systems and make them controllable, maintainable and extensible.

1.1.1 Modularity Benefits

Modularity of software systems brings many benefits to software development. Firstly, modularity improves work efficiency. For example, a team has developed a module in project A. After engineers finish project A, they may start another project B. In project B, they can reuse the modules in project A directly. Reusable software will cut the cost of future development. The more frequently the modules get reused, the lower the amortized cost of the initial development of the modules will be. Secondly, modularity improves software quality. Reusable software systems with modularity are usually more reliable than non-reusable software systems in terms of system maintainability. The reason is that in the process of reuse, bugs and defects are getting fixed and the reusable software becomes more and more stable and robust. Last but not least, human resources can be better managed with modularity. The more experienced engineers can be responsible for the definition of module interfaces, while the less experienced engineers can work on concrete module development.

1.1.2 How to Achieve Modularity

What are the modules? As we mentioned above, modularity is about dividing software systems into small components, based on the divide-and-conquer idea. As a result, the divided modules should have clear boundaries. Different languages treat modules differently. For instance, in Smalltalk [Goldberg and Robson, 1983], which is one of the pioneers as an Object-Oriented Programming language, there is no concept of modules and the only way to divide modules is using classes. As a comparison, the concept of packages is supported in Java, and both classes and modules can be used to divide modules. Another example is JavaScript, which is a prototype-based programming language. There is no need to declare a class to create an object in JavaScript and objects are naturally used to divide modules.

In addition to language features, various design patterns are also necessary for achieving software modularity. Design patterns are the common practice of solutions to various issues in software design. The principles of design patterns can be concluded as *SOLID* [Fenton et al., 2017], which stands for:

- Single Responsibility Principle
- Open/Close Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

The detailed meaning of these principles can be found at [Gamma et al., 1995]. On the whole, all the principles aim at avoiding tight coupling, which means a group of classes/modules are highly dependent on one another.

1.2 Object-Oriented Programming and Modularity

Originated from Simula [Dahl and Nygaard, 1966], a programming language for modeling real-world phenomena, Object-Oriented Programming has been investigated and used for a long time. Object-Oriented Programming is a programming paradigm which attempts to model the world as objects and the interactions among the objects as methods. It modularizes programs by wrapping data and functions into a single unit that can be used as templates for duplicating such modules when needed. The outstanding characteristics such as data abstraction, encapsulation, inheritance and polymorphism make Object-Oriented Programming attractive for *modular* software development. Modularity enables reusability and minimizes code duplication. Moreover, modularity makes it easier to find and fix problems as they can be traced to specific system models, thus limiting the scope of particular error searching.

1.3 Necessity of Multiple Inheritance

Many OOP languages are restricted to single inheritance, which is less expressive and flexible than multiple inheritance, which is one way to increase software modularity in the context of OOP. Multiple inheritance describes the case where one class can have multiple superclasses and inherit features from all of these superclasses. Different flavours of multiple inheritance have been adopted in some popular OOP languages. C++ has had multiple inheritance from the start. Scala adapts the ideas from traits [Schärli et al., 2003, Ducasse et al., 2006, Liquori and Spiwack, 2008] and mixins [Bracha and Cook, 1990, Flatt et al., 1998, Limberghen and Mens, 1996, Ancona et al., 2003, Hendler, 1986] to offer a disciplined form of multiple inheritance. Java 8 provides a simple variant of traits, disguised as interfaces with default methods [Goetz and Field, 2012].

Multiple inheritance is necessary because it is a common need to combine multiple components to create a new component/class. For example, we want to create a bunch of objects of colored shapes, red rectangular, red triangle, white triangle, blue circle, etc. Thus, we need to create the corresponding classes, `RedRectangular`, `RedTriangle`, `WhiteTriangle`, `BlueCircle`, etc. As we noticed, naturally, these classes should inherit from common superclasses `Red`, `Blue`, `Triangle`, `Circle`, etc.

For example, without multiple inheritance, the following code would be impossible:

```
class Red { ... }
class White { ... }
class Blue { ... }

class Triangle { ... }
class Circle { ... }
class Rectangular { ... }

class RedTriangle extends Red, Triangle { ... }
class WhiteTriangle extends White, Triangle { ... }
class BlueCircle extends Blue, Circle { ... }
...
```

This kind of scenarios is ubiquitous. In languages that do not support multiple inheritance, programmers may use some design patterns (such as the Composite pattern [Gamma et al., 1995]) as workarounds to mimic multiple inheritance. However, these workarounds have their limitations; for example, they cannot express the subtyping relationships well. In the composition way, `RedTriangle` is not a subclass of `Triangle`, which is not desirable. Therefore, multiple inheritance is needed for true modularity.

1.4 Difficulty and Issues of Multiple Inheritance

Although multiple inheritance seems to be natural and simple, the reality is that it is recognized as a technique that is hard to implement. Actually, before Bjarne Stroustrup, many people thought adding multiple inheritance to C++ was impossible. Nevertheless, Bjarne Stroustrup found an implementation technique of multiple inheritance in C++ in 1984 [Stroustrup, 1989] and Java supports a restricted form of multiple inheritance since Java 8 in 2014. However, the implementations are not perfect, and we will discuss in more detail in Section 2.2. In the literature [Schärli et al., 2003, Sakkinen, 1989, Bracha and Cook, 1990, Malayeri and Aldrich, 2009], multiple inheritance, especially when combined with state, has several tricky issues including several variants of the famous *diamond problem* [Bracha and Cook, 1990, Sakkinen, 1989], state-related issues, initialization, etc. We list the details of difficulty below:

1. In single inheritance, it is obvious which class is the father/grandfather of a class because the inheritance relationship is simple. However, in the case of multiple inheri-

tance, one class has multiple fathers, and these fathers have their own fathers, then the relationships among all these classes are complex.

2. When one class inherits two parent classes that have data members or methods with the same name, it is difficult to resolve which one is being referenced by the subclass.

3. When two parent classes `A` and `B` inherit from the same base class `T`, forming a *diamond* pattern in the inheritance hierarchy, for the subclass `C` that inherits from both `A` and `B`, conflicts will occur. In the inheritance chain, there are two paths from class `T` to class `C`. It becomes problematic to determine whether a member from `T` is in conflict with another member or not and it would be even harder when the methods from `T` are updated in class `A` or `B`.

4. The order that the initialization/elaboration of the parent classes needs to be specified. It can sometimes lead to behaviour changes when the order of the inheritance changes.

5. Some languages support a reference (e.g., `super`) to an attribute of the base class for this object. However, it is difficult to support such functionality in a language with multiple inheritance.

1.5 Multiple Inheritance Related Problems

As stated before, there are many issues in implementing multiple inheritance. In this thesis, we cannot explore all of these issues. Instead, we will focus only on part of these issues, investigate these canonical problems and propose solutions to them. By providing these solutions, many modularity related problems can be solved. Our final goal is to make software development more modular.

1.5.1 The Expression Problem

Multiple inheritance is often used to realize software modularity and extensibility, because when an extended system is normally developed by combining the existing components and new components/features. The Expression Problem (EP) [Wadler, 1998] (will be explained in detail later in Section 2.1) is one of the most typical problems in this extension process. It describes the problem of representing a system which can be extended in both dimensions (i.e., both the data structure dimension and the behaviour dimension). It has also been a hot topic in programming languages for almost twenty years.

Solving the Expression Problem is of great importance because it means solving a large set of modularity and expressiveness problems represented by the Expression Problem. However, solutions that work in existing languages (such as Java, Scala or Haskell) have employed various complex techniques and a combination of two different mechanisms: *type-parametrization* and *subtyping*. Object algebras [Oliveira and Cook, 2012] are a simple solution to solve the EP in Java-like languages. While Object algebras do not require F-bounds or wildcards, they still require generics. An open problem so far has been whether it is possible to solve the EP just with simple techniques and type systems.

1.5.2 Interface-based Programming (with State) in Multiple Inheritance

In terms of modularity and code reuse, inheritance is a key mechanism in Object-Oriented languages. Inheritance provides a modularization mechanism, which is used to reuse the implementations from inherited classes/objects. Different flavors of multiple inheritance have been adopted in some popular Object-Oriented Programming languages. C++ has had multiple inheritance from the start. Scala adapts the ideas from traits [Schärli et al., 2003, Ducasse et al., 2006, Liquori and Spiwack, 2008] and mixins [Bracha and Cook, 1990, Flatt et al., 1998, Limberghen and Mens, 1996, Ancona et al., 2003, Hendler, 1986] to offer a disciplined form of multiple inheritance. Java 8 offers a simple variant of traits, disguised as interfaces with default methods [Goetz and Field, 2012].

Unfortunately, as widely acknowledged in the literature [Schärli et al., 2003, Sakkinen, 1989, Bracha and Cook, 1990, Malayeri and Aldrich, 2009], multiple inheritance (especially when combined with state) has several tricky issues, including several variants of the famous *diamond problem* [Bracha and Cook, 1990, Sakkinen, 1989]. Many of the problems related to inheritance arise from the direct use of *fields* to model state. Inheriting two fields with the same name raises the question of whether the two fields should be kept, or only one field should exist. Initialization of the fields is also problematic, since initialization code may be inherited from multiple parents. Finally, an additional problem with mutable fields is that their type cannot be type-refined in extensions, which can cause modularity problems [Wadler, 1998, Wang and Oliveira, 2016]. Is it possible to find a solution that handles state (even mutable state) in multiple inheritance properly and in a modular way?

1.5.3 Unintentional Method Conflicts

Other issues with multiple inheritance are conflicts. One of the most sensitive and critical issues is perhaps the ambiguity introduced by multiple inheritance. One case is the famous *diamond problem* [Sakkinen, 1989, Singh, 1995] (also known as the *fork-join inheritance* [Sakkinen, 1989]). In the diamond problem, inheritance allows one feature to be inherited from multiple parent classes that share a common ancestor. Hence conflicts arise. The variety of strategies for resolving such conflicts leads to the occurrence of different multiple inheritance models, including traits, mixins, CZ [Malayeri and Aldrich, 2009], and many others. Existing languages and research have addressed the issue of diamond inheritance extensively.

In contrast to diamond inheritance, the second case of ambiguity is *unintentional method conflicts* [Schärli et al., 2003]. In this case, conflicting methods do not actually refer to the same feature. In a nominal system, methods can be designed for different functionality but happen to have the same names (and signatures). We call such a case *fork inheritance*, in analogy to diamond inheritance. When this kind of unintentional method conflicts happens, they can have severe effects in practice if no appropriate mechanisms to deal with them are available. In practice, existing languages only provide limited support for the issue. In most languages, the mechanisms available to deal with this problem are the same as the diamond inheritance. However, this is often inadequate and can lead to tricky problems in practice. It is especially the case when it is necessary to combine two large modules and their features, but the inheritance is simply prohibited by a small conflict. As a workaround from the diamond inheritance side, it is possible to define a new method in the child class to override those conflicting methods. However, using one method to fuse two unrelated features is clearly unsatisfactory. For example, C++ supports defining two methods with the same name in two classes and a new class can extend both of them without method conflicts. However, these methods can only be overridden simultaneously, but they cannot be updated separately in a flexible way. An open question has been whether it is possible to find a general solution to treat unintentional method conflicts properly and formalize the solution.

1.5.4 State Conflicts

When dealing with multiple inheritance, as mentioned before, one issue is the unintentional *method* conflicts. However, when it comes to multiple inheritance with *state*, another similar but different issue arises, which is dealing with the state conflicts (both intentional and unintentional) scenarios. The concrete questions include, for example:

- How to represent state?

- When dealing with unintentional state conflicts, how to keep both fields from different parents without conflicts?
- Is is possible to refine the type of fields without duplication?
- How to merge the conflicted fields flexibly?
- How to deal with the more complicated diamond case?

The existing language solutions/models are not satisfying or cannot completely solve the problem of multiple inheritance with state. For example, in research area, the trait model is a popular language model to deal with multiple inheritance and in practice it has also been applied to various languages such as Java. However, the classical model does not support state. So we cannot rely on traits to solve the above mentioned issues.

The mixin model is another multiple inheritance model. It supports state but suffers from the linearization problem (see details in Section 7.3.3). C++ is capable of keeping the conflicted fields but still problematic, especially in the diamond problem (see details in Chapter 6). Various other models have their own problems. Can we create a language or a model that can represent state conflicts properly and provide a satisfying approach to treat them so that they can be flexibly inherited, merged and refined? Can we also handle the diamond case of state conflicts properly?

1.6 Solutions and Main Contributions

In this thesis, starting from the expression problem [Wadler, 1998], we propose new design patterns, new programming paradigms, new programming models to solve various multiple inheritance related problems. In this section, we summarize the key contributions.

1.6.1 The Expression Problem, Trivially!

Our first contribution is to solve the Expression Problem using simple language features that common Object-Oriented Programming languages support and use advanced features as little as possible. We propose a variant of the Composite pattern that addresses the EP.

In Section 4.2, we will show that conventional subtyping is enough to solve Wadler's Expression Problem and present a concise solution in Scala, which is essentially the same code that programmers usually write in a typical (failed) attempt to solve the Expression Problem. The only difference is a simple type annotation. We also provide a Java solution, which is slightly more cumbersome due to the use of *covariant return types* to simulate covariant type-refinement of fields. Nevertheless, the Java solution is still quite simple and uses no generics either. The key idea is to make use of covariant type-refinement of immutable fields (in Scala) and covariant method return types (in Java).

1.6.2 Classless Java

In Chapter 4, we present a novel Object-Oriented model called *interface-based* object-oriented programming (IB), where objects implement interfaces directly and fields are not directly supported. Objects are instantiated by static factory methods in object interfaces. To model state, we mimic fields in an object interface with *abstract state operations*. The abstract state operations include various common utility methods (such as getters and setters, or clone-like methods). Objects are only responsible for defining the ultimate behavior of a method.

Object interfaces also provide support for automatic type-refinement. One of our works, *The Expression Problem, Trivially* [Wang and Oliveira, 2016] in Java-like languages shows how easy it is to solve the problem using only type-refinement. In IB, due to its emphasis on type-refinement, the solution would be clearer and more concise. To summarize, the main contributions are:

- **IB and Object Interfaces.** The concept of interface-based programming enables powerful programming idioms using multiple-inheritance, type-refinement, and abstract state operations.

- **Classless Java.** IB is a general concept and **Classless Java** is a practical realization of IB in Java. Classless Java is implemented using annotation processing techniques, allowing most tools to work transparently with our approach. Existing Java projects can use our approach and still be backward compatible with their clients, in a way that is specified by our safety properties.

- **Type-safe covariant mutable state.** We show how the combination of abstract state operations and type-refinement enables a form of mutable state that can be covariantly refined in a type-safe way.

- **Applications and case studies:** we illustrate the usefulness of IB through various examples and case studies[1]. An extended version with a formal translation to Java can be found in Appendix A.1.

[1]https://github.com/YanlinWang/classless-java

1.6.3 FHJ

In Chapter 5, we will propose two mechanisms to deal with unintentional method conflicts: *hierarchical dispatching* and *hierarchical overriding*. Hierarchical dispatching is inspired by the mechanisms in C++ and provides an approach to method dispatching, which combines static and dynamic information. Using hierarchical dispatching, the method binder will look at both the *static type* and the *dynamic type* of the receiver during runtime. When there are multiple branches that cause unintentional conflicts, the static type can specify one branch among them for unambiguity, and the dynamic type helps to find the most specific implementation. In that case, both unambiguity and extensibility are preserved. The main novelty over existing work is the formalization of the essence of the hierarchical dispatching algorithm, which (as far as we know) has not been proposed before.

Hierarchical overriding is a novel language mechanism that allows method overriding to be applied only to one branch of the class hierarchy. Hierarchical overriding adds the expressive power that is not available in languages such as C++ or C#. In particular, it allows overriding to work for classes with multiple (conflicting) methods sharing the same names and signatures.

To present hierarchical overriding and dispatching, we introduce a formalized model **FHJ** in Section 5.3 based on Featherweight Java [Igarashi et al., 2001], together with the theorems and proofs for type soundness. To summarize, the main contributions are:

- **A formalization of the hierarchical dispatching algorithm** that integrates both the static type and dynamic type for method dispatch, and ensures unambiguity as well as extensibility in the presence of unintentional method conflicts.

- **Hierarchical overriding:** a novel notion that allows methods to override individual branches of the class hierarchy.

- **FHJ:** a formalized model based on Featherweight Java, supporting the above features. We provide the static and dynamic semantics and prove the type soundness of the model.

- **Prototype implementation**[2]**:** a simple implementation of **FHJ** interpreter in Scala. The implementation can type-check and run variants of all the examples shown in this chapter.

[2]The implementation is available at `https://github.com/YanlinWang/MIM/tree/master/Calculus`

1.6.4 FHJ+

In Chapter 6, combining the ideas from of **FHJ** and **Classless Java**, we have a clue to solve the problem of state conflicts and also provide a general solution of treating state in the diamond problem. We make use of abstract state operations in **Classless Java** to mimic state and apply the hierarchical dispatching and overriding mechanism in **FHJ**. Based on **FHJ**, we provide a formal model called **FHJ+**. **FHJ+** makes use of the hierarchical dispatching algorithm and adds the flavor of supporting state conflicts, which integrates the ideas from **Classless Java** to support state with abstract state operations. More details are explained in Section 6.3. The main contributions are:

- **Representation of conflicted fields.** Inspired by **Classless Java**, **FHJ+** provides a new way to represents state where these conflicted fields can be represented smoothly with abstract state operations.
- **Fields deferment.** In **FHJ+**, fields are declared and manipulated by abstract state operations, just like the way in **Classless Java**. However, different from **Classless Java**, declaration of a field needs not only to be declared as a getter (and a setter, if needed) but also to be declared in the object constructor. Therefore, programmers may declare an abstract method in a class, but defer the decision of making it a field later.
- **FHJ+ also provides a general solution to the diamond problem (with state)**. Facing the diamond problem with state conflicts, programmers have the freedom to decide how to merge fields, report conflicts, etc.
- **A formalization of FHJ+, which is an extension of FHJ with the field conflicts support**. The formalization differs with **FHJ** in the way that a new form of state conflicts is supported. The model has the expressive power of field conflicts and chooses to merge them or not.
- **Prototype implementation[3]:** a simple implementation of **FHJ+** interpreter in Scala.

1.7 Thesis Organization

The rest of this thesis is structured as follows. We begin with some background about the main topics of this thesis in Chapter 2, in order to keep this thesis self-contained. Then, we will divide the technical content into two parts:

[3]The implementation is available at `https://github.com/YanlinWang/MIM/tree/master/StateMIM-small`

Part I: In Part I, we exploit multiple inheritance in existing languages, mostly in Java and Scala. We leverage on the covariant return types feature in Java, which is used in both works.

Chapter 3 exploits the Expression Problem. We first review the definition of the Expression Problem and know the difficulty. Then we will directly give the specification of our solution in Scala. Also, we will discuss the independent extensibility requirement and how our Scala solution achieves this requirement. After that, we will present a parallel Java solution, which is more cumbersome than the Scala solution, but still concise. Finally, we will have some discussion on this solution and talk about limitations and related work.

In Chapter 4 we first propose the paradigm of interface-based programming. We then define the calculus Classless Java to achieve the paradigm in Java. The prototype is also implemented in Java on top of the project Lombok [Neildo, 2011]. The annotation processing technique is used to transform programs from Classless Java to standard Java. We prove that the transformation is safe so that the whole transformation reductions are also safe. We also illustrate the usefulness of IB through various examples and case studies.

Part II: In this part, we explore more issues related to conflicts in multiple inheritance. We explain how the current solutions/languages are not enough to solve a specific kind of conflict problems: unintentional conflicts. We propose new language features and provide formal models to improve multiple inheritance for Java-like languages.

In Chapter 5, we first propose the issue of unintentional method conflicts and how it affects software development semantics. We provide a formalization of the hierarchical dispatching algorithm which has not been proposed before. We then provide a formal model called **FHJ**, a formalized calculus based on Featherweight Java, supporting hierarchical overriding. We also provide prototype implementation. At the end of this part, we also propose several theorems of the model and provide the full proof.

As an extension of **FHJ**, in Chapter 6 we exploit the problem of unintentional state conflicts. We motivate the problem with a bank payment example. We then give the syntax, typing rules and semantics of the new calculus **FHJ+**. We also provide the type soundness property of the calculus and give the full proof.

Chapter 7 reviews related work and Chapter 8 summarizes and concludes the whole thesis. This thesis is mostly based on three publications of the author and the work on **FHJ+** is based on an on-going draft (Chapter 6).

The materials covered in this thesis have been published in the following papers:

Chapter 3: Yanlin Wang, Bruno C. d. S. Oliveira, “The Expression Problem, Trivially!” In *15th International Conference on Modularity* (Modularity 2016).

Chapter 4: Yanlin Wang, Haoyuan Zhang, Marco Servetto and Bruno C. d. S. Oliveira, “Classless Java” In *International Conference on Generative Programming: Concepts and Experiences* (GPCE 2016).

Chapter 5: Yanlin Wang, Haoyuan Zhang, Bruno C. d. S. Oliveira and Marco Servetto, “FHJ: A Formal Model for Hierarchical Dispatching and Overriding.” In *32nd European Conference on Object-Oriented Programming* (ECOOP 2018).

2 Background

In this chapter, we will elaborate on the background of the topics (i.e., the Expression Problem, multiple inheritance and Featherweight Java) which we are going to discuss in the remaining chapters.

2.1 The Expression Problem

In this section, we will review Wadler's formulation of the Expression Problem (EP) and present the naive non-solution which is often implemented by programmers, and then point out the problems of this non-solution.

2.1.1 Problem Statement

Software nowadays is getting more and more complex to maintain. Programmers and researchers in both industry and academia have learned that software should be constructed in a modular way. However, modularity is hard to achieve due to both theoretical and practical reasons. As we discussed in Chapter 1, solving the Expression Problem is beneficial to improve modularity. Therefore, in this section, let us review the concept of the Expression Problem.

The EP has been widely discussed in the literature [Reynolds, 1975, Cook, 1990, Krishnamurthi et al., 1998]. It was coined by Wadler [Wadler, 1998] to illustrate modular extensibility issues in software evolution, especially when involving recursive data structures. Wadler set a simple programming exercise: implementing a language for a straightforward form of arithmetic expressions (for example: $1 + 2$ or 3). There is an initial set of features consisting of two types of expressions (integer *literals* and *addition*); and one operation (*evaluation* of expressions). Later, two additional features, which represent possible evolutions of the original system, are added:

- **New variant:** a new type of expressions, e.g., *subtraction*.
- **New operation:** a new method, e.g., *pretty printing*.

It is usually easy to extend the system in one dimension (either new variants or new operations). In particular, extending the system with new operations in functional languages is easy, but adding new data variants is difficult. While in common object-oriented languages, the dual problem appears: adding new data variants is easy, but adding new operations is more difficult. Although design patterns like the Visitor pattern [Gamma et al., 1995] allow operations to be added easily, it will make adding new data variants more difficult. Therefore, the traditional Visitor pattern does not solve the expression problem: it merely swaps the dimension of extensibility. The challenge is how to design a programming technique that supports software evolution in both dimensions in a modular way, without modifying the previously written code. The requirements of the solutions to the Expression Problem can be stated more precisely as follows:

- *Extensibility in both dimensions*: A solution must allow the addition of new data variants and new operations and support extending existing operations.
- *Strong static type safety*: A solution must prevent applying an operation to a data variant which it cannot handle using static checks.
- *No modification or duplication*: Existing code must not be modified nor duplicated.
- *Separate compilation and type-checking*: Safety checks or compilation steps must not be deferred until link or runtime.

There is also a common fifth requirement which is proposed by Zenger and Odersky [Zenger and Odersky, 2005]:

- *Independent extensibility*: It should be possible to combine independently developed extensions so that they can be used jointly.

2.1.2 A Non-Solution to Wadler's Expression Problem

Figure 2.1 shows a naive attempt that many programmers try when first faced with the Expression Problem. The basic idea is to define an interface `Exp` for expressions with an evaluation operation `eval()` in it, and then define concrete implementations (i.e., data variants) such as `Lit` and `Add` of that interface for particular types of expressions.

Adding a New Class (Variant): It is easy to add new data variants to the code in Figure 2.1 while satisfying all the requirements for a solution. For example, Figure 2.2 shows the code for adding the subtraction variant.

```
interface Exp { int eval(); }
class Lit implements Exp {
    int x;
    Lit(int x) { this.x = x; }
    public int eval() { return x; }
}
class Add implements Exp {
    Exp e1, e2;
    Add(Exp e1, Exp e2) {
        this.e1 = e1;
        this.e2 = e2;
    }
    public int eval() {
        return e1.eval() + e2.eval();
    }
}
```

Figure 2.1: An object-oriented encoding of arithmetic expressions.

```
class Sub implements Exp {
    Exp e1, e2;
    Sub(Exp e1, Exp e2) {
        this.e1 = e1;
        this.e1 = e2;
    }
    public int eval() {
        return e1.eval() - e2.eval();
    }
}
```

Figure 2.2: An object-oriented approach for adding a new variant Sub.

Adding a New Method (Operation): Figure 2.3 illustrates an example of adding a new operation `print`. The basic idea is to define interfaces for extending old interfaces and to define classes for extending old classes. This technique is needed in order to meet the "*no modification*" requirement and is what many programmers usually implement when facing the Expression Problem. The interface `ExpP` extending interface `Exp` is defined with the extra function `print()`. Classes `LitP` and `AddP` are defined with concrete implementations of operation `print()`, extending base classes `Lit` and `Add`, respectively.

The system with the new print operation is used as follows:

```
ExpP p = new AddP(new LitP(4), new LitP(9));
System.out.println(p.print() + " = " + p.eval());
```

```
interface ExpP extends Exp { String print(); }
class LitP extends Lit implements ExpP {
   LitP(int x) { super(x); }
   public String print() { return "" + this.x; }
}
class AddP extends Add implements ExpP {
   AddP(Exp e1, Exp e2) {
      super(e1,e2);
   }
   public String print() {
      return  ((ExpP) e1).print()
             + " + "
             + ((ExpP) e2).print();
   }
}
```

Figure 2.3: An object-oriented approach for adding a new operation `print`.

The aforementioned straightforward approach seems to solve Wadler's Expression Problem. However, this approach is **not type-safe**: in the definition of the overriding function `print` in class `AddP`, there are two downcasts from `Exp` to `ExpP`. The following example shows that these casts violate the "*strong static type safety*" requirement. In this example, variable `a` is defined as an instance of class `AddP`, with `Lit(4)` and `Lit(9)` as its two member fields. If we try to call `a.print()`, a run-time class casting exception will occur because `Lit(4)` and `Lit(9)` cannot be cast to `ExpP`.

```
   ExpP a = new AddP(new Lit(4), new Lit(9));
   System.out.println(a.print()); // run-time error!
```

The casting problem still exists, even if we change the definition of the constructor `AddP` to be:

```
   public AddP(ExpP e1, ExpP e2) {
      super(e1,e2);
   }
```

Here is an example of the casting problem with the above new constructor `AddP`.

```
AddP a = new AddP( new LitP(1), new LitP(2));
System.out.println(a.print());
a.e1 = new Lit(3);
System.out.println(a.print()); // run-time error!
```

In this example, `a` is an instance of class `AddP`. We can call `a.print()` directly because `AddP` implements the interface `ExpP` and both member fields of `a` are of type `ExpP`. Now if we try to update field `a.e1` with a `Lit` object and then call `a.print()` again, this will pass static type checking at compile time but will generate a class casting exception at runtime. The *no type-safety* problem, which is a clear violation of the requirements for a solution to the Expression Problem, will be solved in Chapter 3.

2.2 Multiple Inheritance Models

In this section, we will review the concept of multiple inheritance and introduce the mainstream multiple inheritance models, especially ones that are related to our research work later.

Multiple inheritance is a feature of Object-Oriented Programming where a class inherits from more than one parent. This is different from single inheritance, where a class can inherit from only one parent. For example, C++ supports a form of multiple inheritance, so that a class in C++ can inherit from two or more parents. In contrast, Java only supports single (class) inheritance, so the same feature as in C++ cannot be represented by Java. In the following, we will introduce and discuss various multiple inheritance models.

2.2.1 Traits

Traits have been used to represent different things in Object-Oriented Programming. The original definition from Schärli et al. [Schärli et al., 2003] is: "traits are a simple compositional model for structuring object-oriented programs". A trait is a collection of methods that can be composed in arbitrary order. In this thesis, we follow the definition from Schärli et al. As Schärli et al. [Schärli et al., 2003] summarized, the following equation holds:

$$\mathsf{Class} = \mathsf{Superclass} + \mathsf{State} + \mathsf{Traits} + \mathsf{Glue}$$

In the trait model, a class can inherit from one and at most one superclass. A class may consist of one or multiple traits, which contains a group of methods that serve as a building block for classes and is a primitive unit of code reuse. In this model, classes are composed of a superclass, a set of traits and the glue code that connects the traits together and accesses the necessary state. Traits provide great structure, modularity, and reusability within classes. However, they can be ignored when one considers the relationships between one class and another. Traits greatly balance the reusability and understandability and enable better conceptual modeling.

Let us look at a concrete example of using traits. In this example, we developed an animal system with a toy shark, which will swim like a real shark. However, it should not extend class `Fish`, since a toy shark is not fish. On the other hand, we know that fish is not the only animal that swims, many mammals also swim. Therefore, the swimming activity should not be classified as the behavior of fish. Instead, it should be regarded as a general activity. In the following, we will demonstrate how to implement it in Scala with traits, which supports concrete implementation inside:

```
trait Swimming {
  def swim() = println("I'm swimming like a Shark")
}

trait Charging {
  def charge() = println("I'm charging my battery...")
}

class Fish
class Toy

class Shark extends Fish with Swimming
class ToyShark extends Toy with Swimming with Charging
class Dog extends Swimming {
  override def swim() = println("I'm swimming using my four legs!")
}

val shark = new Shark
val toy = new ToyShark
val dog = new Dog

shark.swim()      // a real shark can swim
//shark.charge() // but cannot charge
toy.swim()        // a toy shark can swim
dog.swim()        // a dog also swims, but in a different way as
    sharks.
```

In this way, traits improve the reusability and more importantly do not break the required subtyping relationship.

Every trait can define required methods and required fields. A trait can be defined directly by specifying its methods or by composing one or more traits. The composition of traits is performed through these operators: *sum*, *override*, *exclusion*, and *aliasing*. With these four trait operators, which are similar to the arithmetic operations in expressions, people can do many creative designs.

- The *sum* operation combines multiple existing traits into one single trait. The sum operation is symmetric, just like the sum operation in arithmetic expressions, so that `A` sum `B` is the same as `B` sum `A`. If there are no conflicts, then the sum operation succeeds. Otherwise, we have to use other operations to resolve the conflicts.

- The *overriding* operation enables programmers to add additional methods to an original trait to produce a target trait. If the method already exists in the original trait, then the old version will be overridden, and finally, the new method is the effective one in the target trait.
- The *exclusion* operation is similar to the subtraction operation in arithmetic expressions. It can take a trait (original trait) and exclude one or more methods from it and produce a new trait (target trait).
- Another trait operation is called *aliasing*. Given a trait (original trait), we can add an additional name to a method to create a new trait (target trait). It seems that this operation alone is meaningless. However, combining with the above operations, they can be very beneficial. For example, in the case of method conflicts during the sum operation, people can alias the conflicting methods and then exclude the conflicting ones.

Starting from the original trait model, different researchers have proposed many variations and they share some common properties. Below, we will list some typical properties for these trait models.

1. A trait contains a set of methods (provided methods) that implement concrete functionalities.
2. A trait can declare a set of methods (required methods) that will be used in the provided methods.
3. A trait does not contain state.
4. The composition order of classes and trait is not important.
5. Conflicting methods must be explicitly resolved.
6. Trait composition does not affect the semantics of a class: the meaning of the class is the same as all of the methods obtained from the trait(s) were defined directly in the class. It is also called the *flattening property*.
7. Similarly, trait composition does not affect the semantics of a trait itself: a composite trait is equivalent to a flattened trait containing the same methods.

With the trait flattening property, for example, if class `A` is defined using trait `T`, and `T` defines methods `a` and `b`, then the semantics of `A` is the same as it would be if `a` and `b` were defined directly in the class A. Naturally, if the glue code (code that serves solely to ”adapt” different parts of code that would otherwise be incompatible) of A defines a method b directly, then this b would override the method b obtained from T. Readers are referred to [Ducasse et al., 2006, Schärli et al., 2003, Bettini et al., 2013, Bono et al., 2014, Liquori and Spiwack, 2008, Reppy and Turon, 2006, Reppy and Turon, 2007], for more details about traits.

2.2.2 Java 8

It is widely known that Java does not support multiple (class) inheritance, and only single inheritance is supported. However, Java allows multiple interface implementation. Since Java 8, a new language construct is introduced: default methods inside interfaces. Researchers started to explore whether we can achieve a form of multiple inheritance in Java. In 2014, Bono et al. [Bono et al., 2014] show a different usage of interfaces with default methods to achieve that. They use interfaces as traits and default methods for functionality implementation, which is called *trait-oriented programming* in Java. With this idea, a restricted form of multiple inheritance becomes possible in Java and it improves code modularity a lot.

The syntax of default methods in Java is similar to ordinary methods with the difference that the keyword **default** is used as the prefix. The main characteristic of default methods is that they are virtual like all methods in Java, but they provide a default implementation within an interface. For example:

```
interface A {
  default void foo() {
    System.out.println("In interface A");
  }
}

interface B {
  default void bar() {
    System.out.println ("In interface B");
  }
}

class C implements A, B { }

public class Main {
  public static void main(String [] args) {
    C c = new C();
    c.foo();
  }
}
```

Now, class C inherits all implementations from components A and B. In this sense, Java 8 interfaces play the role of traits, with default methods as provided methods and abstract methods as required methods. Java 8 also takes the same approach for dealing with method conflicts: the programmer explicitly overrides the conflicting methods. For example:

```
interface A {
  default void foo() {
    System.out.println("In interface A");
  }
}

interface B {
  default void foo() {
    System.out.println ("In interface B");
  }
}

// class C implements A, B { } //Duplicate default methods named foo.

class C implements A, B {
  public void foo() {
    System.out.println("In interface C");
  }
}
```

In the above code, class C extends interface A and B with the conflicting method foo(), which needs to be explicitly resolved (by overriding) inside C.

As with stateless traits, required fields are encoded as required accessory (getter and setter) methods, that is, as abstract methods, whose implementation will be provided as glue code by the class implementing the traits. In order to introduce the trait-oriented programming style, various programming patterns to match the trait operators listed in Section 2.2.1 are proposed. See the mimic details in their paper [Bono et al., 2014].

There are also differences of default methods in Java from traits, for example, default methods conflict with any other default or abstract method. For example the following code is rejected due to method conflicts.

```
interface A2 { default int m() {return 1;}}
interface B2 { int m(); }
interface C2 { default int m() {return 2;}}
interface D2 extends A2,B2 {} //rejected due to conflicting methods
interface E2 extends A2,C2 {} //rejected due to conflicting methods
```

Note how this is different from what happens in most trait models, where D2 would be accepted, and the implementation in A2 would be part of the behavior of D2.

2.2.3 Mixins

Different from traits, the mixin model is another multiple inheritance model. As proposed in [Bracha and Cook, 1990] by Bracha and Cook, "a mixin is an abstract component, which is a subclass that may be applied to different superclasses to create a related family of modified classes." For example, a mixin can be defined to have the functionality of adding chocolate flavor, and then this mixin could be applied to any kind of ice cream to create a chocolate ice cream class.

Similar to the trait model, every class may have one (and at most one) superclass, but it may have multiple mixins. For example, in the Scala programming language (although the keyword for mixin is "trait"), the keyword `extends` is for class inheritance, and the keyword `with` is for mixin. When composing a new class, `extends` can be used one (and at most one) time, however, `with` can be used for multiple times. For example, we have a class `IceCream`, and now we want to create an instance of the ice-cream with both chocolate and strawberry flavor. In Scala, we may write:

```
val myIceCream = new IceCream with Chocolate with Strawberry
```

In case of conflict, mixin composition mechanism will chose one implementation (might be the left or right mixin). And it gives priority to one component depends on the implementation of mixins. See more details in the mixins original paper [Bracha and Cook, 1990].

In contrast to traits, mixins may contain state. Thus instance variables can be defined in mixins. In the mixin model, multiple inheritance is treated in a subtle way. Linearization is enforced in the inheritance hierarchy so that every mixin is arranged to an appropriate place.

Mixin programming takes advantage of multiple inheritance in a subtle and unintuitive way. When a mixin is defined with multiple mixins, it will choose the ordering of these parent mixins. Each mixin defines a certain ordering of its fathers, which determine a partial ordering of all the mixins components. However, the linearization of mixins is also criticized by many researchers, see details in Chapter 7.

2.3 Featherweight Java

One important topic of this thesis is Object-Oriented Programming, and to model Object-Oriented Programming there are many models/languages. One way is to encode the features of Object-Oriented Programming into a core calculus, which can be formally modeled and reasoned. Why do people investigate formal models? Firstly, formal models make precise what a language means. They will specify what programs are accepted in the language and

Programs	P	::=	$\overline{\mathrm{CL}}\ e$
Classes	CL	::=	$\texttt{class}\ \mathrm{C}\ \texttt{extends}\ \mathrm{C}\{\overline{\mathrm{C}\ \mathrm{f}};\mathrm{K}\ \overline{\mathrm{M}}\}$
Constructors	K	::=	$\mathrm{C}(\overline{\mathrm{C}\ \mathrm{f}})\{\texttt{super}(\overline{\mathrm{f}});\overline{\texttt{this}.\mathrm{f} = \mathrm{f}};\}$
Methods	M	::=	$\mathrm{C}\ \mathrm{m}(\overline{\mathrm{C}\ \mathrm{x}})\{\texttt{return}\ e;\}$
Expressions	e	::=	$\mathrm{x} \mid e.\mathrm{f} \mid e.\mathrm{m}(\overline{e}) \mid \texttt{new}\ \mathrm{C}(\overline{e}) \mid (\mathrm{C})e$
Values	v	::=	$\texttt{new}\ \mathrm{C}(\overline{v})$
Evaluation Context	E	::=	$[.] \mid \mathrm{E}.\mathrm{f} \mid \mathrm{E}.\mathrm{m}(\overline{e}) \mid v.\mathrm{m}(\overline{v},\mathrm{E},\overline{e}) \mid \texttt{new}\ \mathrm{C}(\overline{v},\mathrm{E},\overline{e}) \mid (\mathrm{C})\mathrm{E}$

Figure 2.4: Syntax of Featherweight Java

how a program executes. Secondly, formal models allow us to prove formal properties. Typically, they will reduce specific run-time errors.

In this section, we will introduce a core calculus for Java proposed by Igarashi, Pierce, and Wadler called *Featherweight Java* [Igarashi et al., 2001]. Featherweight Java is a *small* calculus and is a subset of the current Java programming language. The concise language components of Featherweight Java includes classes, constructors, fields, methods, and casts. The essential feature is class inheritance, and in Featherweight Java, only single inheritance is supported. The model does not include many popular features, such as interfaces, overloading, exceptions, or the `public`, `private` or `protect` modifiers. Because of the simplicity of the model, various reasoning about type soundness and other properties of the language is possible. It is also easy to extend; actually, it has been a classic model for other calculus, for example, FGJ(Featherweight Generic Java).

The syntax of Featherweight Java is given in Fig. 2.4. This is the standard BNF definition. Note that an over-bar represents a sequence. A program consists of a sequence of class definitions and an expression to be evaluated, which corresponds to the body of the `main` method in Java. The class definition of Featherweight Java $\texttt{class}\ \mathrm{C}\ \texttt{extends}\ \mathrm{C}\{\overline{\mathrm{C}\ \mathrm{f}};\mathrm{K}\ \overline{\mathrm{M}}\}$ contains the name of the class C, the parent class D, fields $\overline{\mathrm{f}}$ with types $\overline{\mathrm{C}}$, a single constructor K and a set of method definitions $\overline{\mathrm{M}}$. The constructor declaration $\mathrm{C}(\overline{\mathrm{C}\ \mathrm{f}})\{\texttt{super}(\overline{\mathrm{f}});\overline{\texttt{this}.\mathrm{f} = \mathrm{f}};\}$ takes a sequence of field values and use part of them to initialize the fields from the superclass and the rest to initialize the fields defined inside the current class. The method declaration $\mathrm{C}\ \mathrm{m}(\overline{\mathrm{C}\ \mathrm{x}})\{\texttt{return}\ e;\}$ introduces the method m with parameters $\overline{\mathrm{x}}$ with types $\overline{\mathrm{C}}$ and the method body e with the return type C. Expressions in Featherweight Java can be a single variable x, field lookup $e.\mathrm{f}$, method call $e.\mathrm{m}(\overline{e})$, object creation $\texttt{new}\ \mathrm{C}(\overline{e})$ or expression casting $(\mathrm{C})e$. Note that it is a strict subset of Java so that every Featherweight Java program can be executed in a standard Java compiler and JVM.

The subtyping rules of Featherweight Java are shown in Fig. 2.5 is the standard subtyping relation, which is reflexive (meaning $A <: A$ for any type A) and transitive (meaning that if $A <: B$ and $B <: C$ then $A <: C$). Also, the subclass relation declared by **extends** also immediately lead to subtyping relation.

The reduction rules of Featherweight Java are shown in Fig. 2.6. It is the standard call-by-value operational semantics. The explanation of the rules and more details (including the type checking rules) can be found in TAPL [Pierce, 2002] Chapter 19.

$$\boxed{C <: D} \qquad C <: C$$

$$\frac{C <: D \qquad D <: E}{C <: E} \qquad \frac{\texttt{class}\ C\ \texttt{extends}\ D\ \{...\}}{C <: D}$$

Figure 2.5: Subtyping rules of Featherweight Java

$$\boxed{e \to e'} \qquad (\textsc{E-ProjNew})\ \frac{\texttt{fields}(C) = \overline{C}\ \overline{f}}{\texttt{new}\ C(\overline{v}).f_i \to v_i}$$

$$(\textsc{E-InvkNew})\ \frac{\texttt{mbody}(m, C) = (\overline{x}, e_0)}{\texttt{new}\ C(\overline{v}).m(\overline{u}) \to [\overline{x} \mapsto \overline{u}, \texttt{this} \mapsto \texttt{new}\ C(\overline{v})]e_0}$$

$$(\textsc{E-CastNew})\ \frac{C <: D}{(D)(\texttt{new}\ C(\overline{v})) \to \texttt{new}\ C(\overline{v})} \qquad (\textsc{E-Field})\ \frac{e_0 \to e_0'}{e_0.f \to e_0'.f}$$

$$(\textsc{E-Invk-Recv})\ \frac{e_0 \to e_0'}{e_0.m(\overline{e}) \to e_0'.m(\overline{e})}$$

$$(\textsc{E-Invk-Arg})\ \frac{e_i \to e_i'}{v_0.m(\overline{v}, e_i, \overline{e}) \to v_0.m(\overline{v}, e_i', \overline{e})}$$

$$(\textsc{E-New-Arg})\ \frac{e_i \to e_i'}{\texttt{new}\ C(\overline{v}, e_i, \overline{e}) \to \texttt{new}\ C(\overline{v}, e_i', \overline{e})} \qquad (\textsc{E-Cast})\ \frac{e_0 \to e_0'}{(C)e_0 \to (C)e_0'}$$

Figure 2.6: Evaluation rules of Featherweight Java

Below, we will use an example from TAPL to illustrate the standard step-by-step evaluation process, which will be used later in Section 6.3:

```
class A extends Object { A() { super(); } }

class B extends Object { B() { super(); } }

class Pair extends Object {
  Object fst;
  Object snd;
  // Constructor:
  Pair(Object fst, Object snd) {
    super(); this.fst = fst; this.snd = snd;
  }
  // Method definition:
  Pair setfst(Object newfst) {
    return new Pair(newfst, this.snd);
  }
}
```

Evaluation steps:

```
((Pair)(new Pair(new Pair(new A(), new B()), new A()).fst).snd)
-> ((Pair)new Pair(new A(), new B())).snd
-> new Pair(new A(), new B()).snd
-> new B()
```

Under the standard call-by-value evaluation strategy, the expression `((Pair)(new Pair (new Pair(new A(), new B()), new A()).fst` will first be reduced at the `(new Pair(new Pair(new A(), new B()), new A()).fst).snd` part, resulting the expression on the second line above (which is `((Pair)new Pair(new A(), new B())).snd`). Then `((Pair)new Pair(new A(), new B()))` will be reduced to `new Pair(new A(), new B())`, resulting the expression on the third line, which is `new Pair(new A(), new B()).snd`. Finally, the expression will be reduced to the single value `new B()`.

2.4 Project Lombok

Java supports compilation agents, where Java libraries can interact with the Java compilation process, acting as a man in the middle between the generation of AST and bytecode.

This process is facilitated by frameworks like Lombok [Zwitserloot and Spilker, 2016]: a Java library that aims at reducing Java boilerplate code via annotations. The flow of annotation processing is illustrated as follows. First Java source code is parsed into an AST. The AST is then captured by Lombok: each annotated node is passed to the corresponding (Eclipse or Javac) handler. The handler is free to modify the information of the annotated node, or even inject new nodes (like methods, inner classes, etc.). Finally, the Java compiler works on the modified AST to generate bytecode.

There are a number of annotations provided by the original Lombok, including `@Getter`, `@Setter`, `@ToString` for generating getters, setters and `toString` methods, respectively. Furthermore, Lombok provides a number of interfaces for users to create custom transformations, as extensions to the original framework.

Advantages of Lombok The Lombok compilation agent has advantages with respect to alternatives like pre-processors, or other Java annotation processors. Lombok offers in Java an expressive power similar to that of Scala/Lisp macros; except, for the syntactic convenience of quote/unquote templating.

Direct modification of the AST Lombok alters the generation process of the class files, by directly modifying the AST. Neither the source code is modified nor new Java files are generated. Moreover and probably more importantly, Lombok supports the generation of code *inside* a class/interface, while conventional Java annotation processors (e.g., `javax.annotation`) do not support.

Modularity While general preprocessing acts across module boundaries, compilation agents act modularly on each class/compilation unit. It makes sense to apply the transformations to one class/interface at a time, and only to annotated classes/interfaces. This allows library code to be reused without being reprocessed or recompiled, making our approach 100% compatible with existing Java libraries, which can be used and extended normally.

Tool support Features written in Lombok integrate and are supported directly in the language and are also supported by most tools. In Eclipse, the processing is performed transparently and the information of the interface from the compilation is captured in the "Outline" window. It contains all the methods inside the interface, including the generated ones. Moreover, as a useful IDE feature, the auto-completion also works for these newly generated methods.

Clarity against obfuscation Preprocessors bring great power, which can easily be misused producing code hard to understand. Thus code quality and maintainability are reduced. Compilation agents start from Java syntax, but they can reinterpret it. Preserving the syntax avoids syntactic conflicts, allowing many tools to work transparently.

Part I

Multiple Inheritance for Modularity and Reuse

3 The Expression Problem, Trivially!

This chapter details our solution to the Expression Problem (EP) [Wadler, 1998]. We will present the solution in Scala with covariant field refinement feature and also the solution in Java with method covariant return type feature. Then, we show a variant of our solution to the EP that can encode simple family polymorphism [Ernst, 2001]. Finally, we will have a discussion on the limitations of our approach.

3.1 Introduction

In Section 2.1, we have reviewed the concept of the EP and shown a non-solution. Solutions to the EP have been a hot topic in programming languages for almost twenty years. Today we know of various solutions to the EP that either rely on *new programming language features* [Chambers and Leavens, 1995, Clifton et al., 2000, Ernst et al., 2006, Ernst, 2001], or *design patterns* [Gamma et al., 1995] in existing languages [Torgersen, 2004, Swierstra, 2008, Oliveira, 2009, Oliveira and Cook, 2012]. The non-triviality of the EP is associated with the fact that existing solutions either require languages with specially crafted type systems or encodings in existing languages using several techniques to overcome the limitations of the type system. So far, solutions that work in existing languages (such as Java, Scala or Haskell) have employed various techniques and a combination of two different mechanisms: *type-parametrization* and *subtyping*.

This chapter shows that conventional subtyping (as found in Scala and Java) is enough to solve Wadler's EP. We present a Scala solution, which is essentially the same code that programmers usually write in a typical (failed) attempt to solve the EP. The only minimal difference is a simple type annotation. The annotation serves the purpose of covariantly refining the extended types. This shows, somewhat surprisingly, that the Wadler's EP can be (almost) trivially solved. We also present a Java solution, which is slightly more involved due to the use of *covariant return types* to simulate covariant type-refinement in fields. Nevertheless, the Java solution is still quite simple and uses no generics either. The code for the solutions presented in this chapter is online[1].

[1]https://bitbucket.org/yanlinwang/ep_trivially

The new solution also provides new insights into tackling harder extensibility challenges, such as *family polymorphism*. Using a variant of our solution we provide a technique for encoding a simple, although restricted, form of family polymorphism. The key idea is to use nested components enclosed by classes, interfaces or even packages. Such nested components allow the reuse of names of family members in extensions. Refinement of the types of family members is achieved using conventional shadowing semantics. *Default methods*, introduced by Java 8, are used for increased flexibility and improved support for multiple inheritance. The approach allows for *independent extensibility* [Zenger and Odersky, 2005] and, unlike most approaches to family polymorphism, it allows subtyping relationships across families.

Of course, having an (almost) trivial solution to the EP raises the question: are we done looking for how to improve programming languages with respect to extensibility? Unfortunately, that is not the case. Instead, we believe that what our solution shows is that Wadler's original programming challenge is too simple for exercising the difficulties encountered in the design of extensible software. Therefore we identify a set of subproblems that go beyond Wadler's Expression Problem. These subproblems are needed, for example, to solve a more fully featured form of family polymorphism, and arise more generally in approaches to extensibility. We summarize two of the main problems (together with some subproblems) faced when trying to achieve general extensibility next:

1. **Hardcoded recursive types:** When software is designed, it normally uses composite structures which model complex recursive types. However, such recursive types are used as hardcoded references. This is a problem for extensibility, since in extensions the references to recursive types should refer to the extended versions (and not to the original versions). Within this general problem there are two different subproblems:

   a) **Recursive types in fields:** This is the problem that is illustrated by the arithmetic expressions example in Wadler's Expression Problem.

   b) **Recursive types in argument/return positions:** A generalization of a) is to consider recursive types that appear in any argument or return positions of methods.

2. **Hardcoded constructors:** Another issue that arises when software is extended is that constructors are also hardcoded. So, software using constructors cannot be easily extended because the constructors keep referring to the original code, instead of the extensions.

Importantly, a solution to Wadler's original Expression Problem needs only to solve problem 1a). As we show in this chapter solving problem 1a) requires only subtyping: no type-parametrization is needed. For the more general case 1b) there are additional complications, for example, operations that *create* or *transform* expressions require the use of constructors to build values of the extended types. However, as emphasized by problem 2), the use of constructors is problematic. Also, when recursive types are used as arguments we may need to invoke methods on such parameters, but this requires more refined type information about which methods are available. For challenges 1b) and 2) we do not know of any solution in Java-like languages that does not involve type-parametrization.

To keep the spirit of Wadler's original challenge, we propose to add to that challenge a `double` operation that doubles all numbers in an expression. Unlike evaluation and pretty printing, which were the operations involved in Wadler's original challenge, `double` requires a solution to 2) and 1b), as well as 1a).

Having set the challenge, we develop a solution for it that works in Java-like languages. However, we are no longer able to achieve the solution only with subtyping: we need to resort to a range of techniques, including type-parametrization with *F-bounded quantification* [Canning et al., 1989] and *object algebras* [Oliveira and Cook, 2012]. Using those techniques, we can adapt the simple approach to family polymorphism and extend it to deal with more complicated cases.

We believe that our results present valuable insights for researchers and programming language designers interested in modularity and extensibility. Furthermore our results have immediate applicability as practical design patterns for programmers interested in improving extensibility of their programs and they have applications in the domain of *software product-lines*.

3.2 A Trivial Solution in Scala

This section presents a solution to the EP in Scala. The main Scala feature used here is the support for type refinement of (immutable) *fields*. This simple feature allows us to write the solution to the EP very directly and compactly.

Initial System The initial system shown in Figure 3.1 defines a trait `Exp` with the evaluation (`eval`) operation. Traits `Lit` and `Add` extend `Exp` with corresponding implementations of `eval`. Note that `e1` and `e2` are immutable member fields, declared as `val`s.

Adding a New Variant It is easy to add new data variants to the initial system in Figure 3.1 while satisfying all the requirements for a solution. For example, trait `Sub` illustrates the addition of new variants and is almost the same as the definition of trait `Add`.

```
trait Exp { def eval() : Int }
trait Lit extends Exp {
  val x:Int
  def eval() = x
}
trait Add extends Exp {
  val e1, e2 : Exp
  def eval() = e1.eval + e2.eval
}
```

Figure 3.1: Initial code in the Scala solution.

```
trait Sub extends Exp {
  val e1, e2:Exp
  def eval() = e1.eval - e2.eval
}
```

Adding a New Operation Figure 3.2 shows an example of extending the initial system with a pretty printing operation. The basic idea is to extend traits `Exp`, `Lit` and `Add` with traits `ExpP`, `LitP`, and `AddP`, respectively. Note that the type of member fields `e1` and `e2` in `AddP` is refined! That is, instead of keeping the type of `e1` and `e2` as `Exp`, we change it to a *subtype* (`ExpP`). Changing the type is allowed in Scala because it is just a form of covariant type refinement of types in positive positions, which is well-understood in the theory of object-oriented languages [Cardelli, 1988].

Importantly, note that it is the lack of this type-refinement that is to blame for typical naive attempts to solve the EP. In a naive attempt, the trait `AddP` would be defined as:

```
// Incorrect: typical code in naive non-solution!
trait AddP extends Add with ExpP {
  // method does not type-check!
  def print() = "("+ e1.print + "+" + e2.print +")"
}
```

The problem is that, because the type of `e1` and `e2` is not refined, the call to the method `print` fails to type-check: the trait `Exp` does not support a `print` method.

Instantiation The Scala solution is easy and concise to use:

```
    // Initial system
    val l1 = new Lit{val x=4}
    val l2 = new Lit{val x=3}
    val a = new Add{val e1=l1; val e2=l2}
    println("a.eval = " + a.eval)
```

```
trait ExpP extends Exp { def print():String}
trait LitP extends Lit with ExpP {
  def print() = "" + x
}
trait AddP extends Add with ExpP {
  val e1, e2 : ExpP // type refined!
  def print() = "("+ e1.print + "+" + e2.print +")"
}
```

Figure 3.2: Adding an operation `print` in the Scala solution.

```
    // Subtraction feature
    val s = new Sub{val e1=l1; val e2=l2}
    println("s.eval = " + s.eval)

    // Print feature
    val le1 = new LitP{val x=4}
    val le2 = new LitP{val x=3}
    val ae = new AddP{val e1=le1; val e2=le2}
    println(ae.print + " = " + ae.eval)
```

Here, various objects are created from traits in Figures 3.1 and 3.2. The first block of code illustrates how to use the initial system, building a simple expression and evaluating it. The second block shows how to use the subtraction feature. Finally, the last block shows how to build expressions and use both pretty printing and evaluation.

3.3 Independent Extensibility

Systems that satisfy independent extensibility should be able to combine multiple independently developed extensions easily. In this way, programmers can merge several extensions into a single compound one. In a trait-based language like Scala, it is easy to obtain independent extensibility by simply relying on multiple trait-inheritance [Zenger and Odersky, 2005]. To illustrate independent extensibility, we extend the initial system with a new operation `collectLit` (which collects all literal components in an expression) in Figure 3.3. The code to combine two extensions (with `print` and `collectLit` respectively) is:

```
trait ExpPC extends ExpP with ExpC
trait LitPC extends LitP with LitC with ExpPC
trait AddPC extends AddP with AddC with ExpPC {
  val e1, e2 : ExpPC
```

```
trait ExpC extends Exp {
  def collectLit(): List[Int]
}
trait LitC extends Lit with ExpC {
  def collectLit() : List[Int] = x :: List()
}
trait AddC extends Add with ExpC {
  val e1, e2 : ExpC
  def collectLit() : List[Int] = e1.collectLit ::: e2.collectLit
}
```

Figure 3.3: Adding an operation `collectLit`.

```
}
```

`ExpPC` is the new expression interface supporting `print` and `collectLit` operations; `LitPC` and `AddPC` are the extended variants. Notice that except for the routine of **extend** clauses, we only need to refine the type of `e1,e2` in `AddPC`. We will omit the instantiation code because it is essentially the same as the instantiation code presented in Section 3.2.

3.4 A Java Solution

This section presents a solution to the EP in Java. Since Java does not support type-refinement of fields, we use *covariant return types* instead to allow refinements of the types of recursive sub-expressions. Figure 3.4 shows a class diagram summarizing our Java solution.

Initial System The initial system shown in Figure 3.5 is almost the same as the Scala code presented in Figure 3.1. The difference is that old member fields `e1` and `e2` in trait `Add` are now replaced by the abstract functions `getE1()` and `getE2()`. Therefore the class `Add` becomes an abstract class correspondingly. These abstract *getter* methods will enable future extensions of the initial system to covariantly refine the return types of these methods.

Adding a New Variant Extending the initial system with a new data variant `Sub` is easy, as shown here:

```
abstract class Sub implements Exp {
    abstract Exp getE1();
    abstract Exp getE2();
    public int eval() {
        return getE1().eval() - getE2().eval();
    }
```

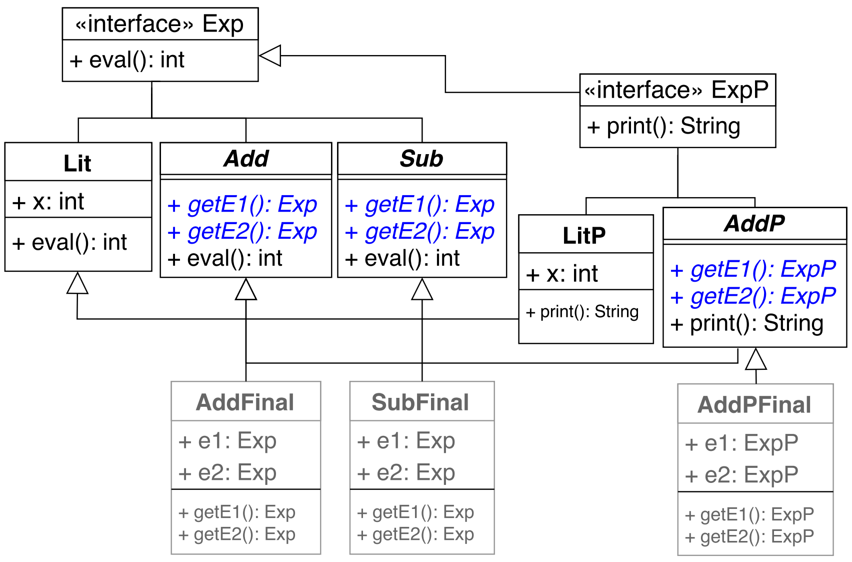


Figure 3.4: The Java solution overview.

```
}
```

Adding a New Operation Figure 3.6 shows an example of extending the initial system with a new operation `print`. Importantly, note that the definition of the `print()` method in the class `AddP` is well-typed. This is because the types of the getters `getE1()` and `getE2()` are refined, using the covariant return types feature of Java, to return `ExpP` instead of `Exp`. If the types were not refined, then there would be a type-error when using `getE1().print()` or `getE2().print()`, since method `print()` would not be defined in `Exp`.

Instantiation Note that in the initial system, the abstract class `Add` is not immediately usable: abstract classes cannot be directly instantiated. As shown in Figure 3.7, an additional class `AddFinal` is needed to extend `Add` and provide concrete implementations of abstract methods `getE1()`, `getE2()` in its superclass. With `AddFinal` we can create an expression and execute an operation on it:

```
Exp exp = new AddFinal(new Lit(7), new Lit(4));
System.out.println(exp.eval());
```

Similarly, when updating the system with new operation `print`, an additional class `AddPFinal` is defined for instantiation of the abstract class `AddP` (code for `AddPFinal` is almost the same as `AddFinal`, so we omit it here).

```
interface Exp { int eval(); }
class Lit implements Exp {
   int x;
   Lit(int x) { this.x = x; }
   public int eval() { return x; }
}
abstract class Add implements Exp {
   abstract Exp getE1(); //refinable return type!
   abstract Exp getE2(); //refinable return type!
   public int eval() {
      return getE1().eval() + getE2().eval();
   }
}
```

Figure 3.5: Initial code in the Java solution.

```
interface ExpP extends Exp { String print(); }
class LitP extends Lit implements ExpP {
   LitP(int x) { super(x); }
   public String print() { return "" + x; }
}
abstract class AddP extends Add implements ExpP {
   abstract ExpP getE1(); //return type refined!
   abstract ExpP getE2(); //return type refined!
   public String print() {
      return "(" + getE1().print() + " + " +
             getE2().print() + ")";
   }
}
```

Figure 3.6: Adding an operation `print` in the Java solution.

3.5 Simple Family Polymorphism

This section shows a variant of the solution to the EP presented in Section 3.4 that can be used to encode simple forms of *family polymorphism* [Ernst, 2001]. The key idea is to use nested components[2] in Java to group family members. An advantage compared to the solution in Section 3.4 is that the same names, as in the original system can be kept. We also discuss some differences to existing approaches to family polymorphism, including the ability of our solution to allow subtyping relations across families.

[2]By a component we mean either a class or an interface.

```
class AddFinal extends Add {
    Exp e1, e2;
    AddFinal(Exp e1, Exp e2) {
        this.e1 = e1;
        this.e2 = e2;
    }
    Exp getE1() { return e1; }
    Exp getE2() { return e2; }
}
```

Figure 3.7: An additional class for instantiation.

```
interface BaseFeature {
    interface Exp { int eval(); }
    interface Lit extends Exp {
        public int getX();
        public default int eval() { return getX(); }
    }
    interface Add extends Exp {
        public Exp getE1(); // abstract getter for e1
        public Exp getE2(); // abstract getter for e2
        public default int eval() { return getE1().eval() + getE2().
            eval(); }
    }
}
```

Figure 3.8: A base feature/family of integer expressions.

Note that we use Java instead of Scala throughout the rest of this chapter. There are significant differences between Java's semantics for nested components and Scala. Generally speaking Java's semantics for nested components is less expressive. Since one of our goals is to show the minimal set of features needed for extensibility, and Java's semantics is enough, we use Java.

3.5.1 Modelling Families as Interfaces with Nested Components

Figure 3.8 shows how to model a family of classes for integer expressions using an interface `BaseFeature` to group all members of the family. The basic idea is to use interfaces as a simple module system. Besides this difference, the code for the various (nested) interfaces and classes is almost the same as the code presented in Figure 3.5. A small difference is the use of an interface instead of an abstract class to model `Add` and `Lit`. As we shall see in Section 3.6, this will be important to support independent extensibility.

```
interface SubFeature extends BaseFeature {
    interface Sub extends Exp {
        abstract Exp getE1(); // abstract getter for e1
        abstract Exp getE2(); // abstract getter for e2
        public default int eval() {
            return getE1().eval() - getE2().eval();
        }
    }
}
```

Figure 3.9: Adding a subtraction feature to integer expressions.

Nested components in Java Many Java-like languages (including C#, C++ or Scala) support some form of nested components: components which are defined inside other components. In Java there are two forms of nested components: *inner classes* and *static nested components*. Interfaces such as `BaseFeature` can only contain static nested components. Thus `Exp` is a static nested interface and `Lit` and `Add` are static nested interfaces. The main advantage of using interfaces for grouping family members is improved support for multiple inheritance, due to Java semantics of multiple interface inheritance. Classes with nested components can also contain inner classes. The difference between an inner class and a static nested class is that instances of inner classes contain a reference to the enclosing instance of the containing class. In Section 3.7 we will see some uses of inner classes.

3.5.2 Solving the Expression Problem

The code needed to add new variants and new operations is similar to the solution presented in Section 3.4. Figure 3.9 shows the code that is required to add the subtraction variant. The interface `SubFeature` extends `BaseFeature`, thus inheriting all of its members, while at the same time adding a new interface `Sub`. The `Sub` interface is defined similarly to the `Add` interface in `BaseFeature` except for the definition of the `eval()` method.

The code to add a new pretty printing operation is shown in Figure 3.10. Similarly to adding a new variant, `PrintFeature` extends `BaseFeature` to inherit all its members. To add the new operation *locally*, the interface `Exp` in `PrintFeature` extends the corresponding interface in `BaseFeature` and adds a new method `print()`. Similarly both the `Lit` and `Add` classes need to extend the corresponding classes in `BaseFeature` and implement the new `Exp` interface. Importantly, note that we do not have to use new names for the extended classes or interfaces. Instead, the use of nested components allows us to simply shadow the old names, and reuse the same names as in `BaseFeature`.

```
interface PrintFeature extends BaseFeature {
   interface Exp extends BaseFeature.Exp { String print(); }
   interface Lit extends BaseFeature.Lit, Exp {
      public default String print() { return "" + getX(); }
   }
   interface Add extends BaseFeature.Add, Exp {
      abstract Exp getE1(); // type-refinement for e1
      abstract Exp getE2(); // type-refinement for e2
      public default String print() {
         return getE1().print() + " + " + getE2().print();
      }
   }
}
```

Figure 3.10: Adding a printing feature to integer expressions.

3.5.3 Initialization and Client Code

The code for a feature, such as `BaseFeature`, is not immediately usable for client code due to the presence of interfaces. Therefore some additional code is needed to provide the implementations for the various abstract members and allow the construction of objects. Figure 3.11 shows the required code for `BaseFeature`. The class `BaseFinal` implements `BaseFeature` and provides two *factory methods* [Gamma et al., 1995] `Lit` and `Add`. These methods create, respectively, instances of the classes `Lit` and `Add`. In the case of `Lit`, we initialize the integer field `x` and implement the method `getX`. In the case of `Add`, we create two fields `e1` and `e2` of type `Exp` and implement the getter methods in an obvious way.

With `BaseFinal` it becomes possible to write some client code. For example:

```
BaseFinal i = new BaseFinal();
BaseFeature.Exp e = i.Add(i.Lit(3), i.Lit(4));
System.out.println("Result: " + e.eval());
```

creates an instance of `BaseFinal` and uses that instance to construct a simple integer expression, whose resulting evaluation is printed on the console.

Similarly, class `SubFinal` and `PrintFinal` are also needed for implementing `SubFeature` and `PrintFeature`. The code for these final classes is essentially the same as the code in Figure 3.11, so we omit it here. Provided with `SubFinal` and `PrintFinal`, client code can be written as follows:

```
SubFinal mi = new SubFinal();
SubFeature.Exp me = mi.Sub(mi.Add(mi.Lit(3), mi.Lit(4)), mi.
   Lit(2));
System.out.println("Result: " + me.eval());

PrintFinal pi = new PrintFinal();
```

```
class BaseFinal implements BaseFeature {
   Exp Lit(final int i) {
      return new Lit() {
         int x = i;
         public int getX() { return x; }
      };
   }
   Exp Add(final Exp left, final Exp right) {
      return new Add() {
         Exp e1 = left, e2 = right;
         public Exp getE1() { return e1; }
         public Exp getE2() { return e2; }
      };
   }
   Exp exp() { return Add(Lit(3), Lit(4)); }
   void printVal(Exp e) {
      System.out.println("Value of e is:" + e.eval());
   }
}
```

Figure 3.11: Initialization code for integer expressions.

```
      PrintFeature.Exp pe = pi.Add(pi.Lit(3), pi.Lit(4));
      System.out.println("Expression: " + pe.print() + " evaluates
         to: " + pe.eval());
```

With this approach, we are able to do direct code copy-and-paste. For example, consider the `exp()` method in class `BaseFinal`. When we extend the system with pretty printing, we also want to have an `exp()` method in class `PrintFinal`. Instead of rewriting method `exp()` with new names to fit in class `PrintFinal`, we can just directly copy the `exp()` method definition and paste it into class `PrintFinal`. So an advantage with respect to the solution in Section 3.4 is that the reuse of names allows for syntactic compatibility of components.

3.5.4 Subtyping Relations

In family polymorphism, each family normally supports its own identity, and members of different families are not related by subtyping. Although in the general case disallowing subtyping is required to ensure type-safety, that approach is sometimes too conservative. Oliveira [Oliveira, 2009] has shown that in some cases subtyping across families is both type-safe and useful. Our solution also supports some cross-family subtyping.

```
interface StmtFeature extends BaseFeature {
   HashMap<String, Integer> map = new HashMap<String, Integer>();
   interface Stmt { void eval(); }
   interface Var extends Exp {
      public String getVarName();
      public default int eval() { return map.get(getVarName()); }
   }
   interface Assign extends Exp { /* code omitted */ }
   interface Expr extends Stmt {
      abstract Exp getE();
      public default void eval() { getE().eval(); }
   }
   interface Comp extends Stmt {
      abstract Stmt getS1();
      abstract Stmt getS2();
      public default void eval() {
         getS1().eval();
         getS2().eval();
      }
   }
}
```

Figure 3.12: Code for the statement feature.

Consider again the `exp()` method in class `BaseFinal`. When we extend the system with subtraction, we may want to have another method `exp2()` that can build expressions from old expressions:

```
Exp exp2() { return Sub(super.exp(), Lit(1)); }
```

Doing so is allowed in our solution because although the base family and subtraction family are in two different enclosing interfaces, they share the same interface Exp.

3.5.5 Multiple Types

In previous sections, our examples on the EP only involve single recursive type, namely Exp. However, more complex systems often need multiple (potentially mutually) recursive types. This is a typical situation in family polymorphism. We use an example by Oliveira and Cook [Oliveira and Cook, 2012] to illustrate that our solution can also support multiple types.

When we extend our language of expressions with statements, we need multiple evolving types. Figure 3.12 shows how to encode the statement feature using our solution. Now besides the recursive type Exp, there is another recursive type `Stmt`, representing statements.

In statements the evaluation method `eval()` does not return a value. The classes `Var` (variables) and `Assign` (value assignments) denote two new forms of expressions. The classes `Expr` (lifting expressions to statements) and `Comp` (statement composition) are two forms of statements. As before, a final class for `StmtFeature` is needed for instantiation. The code for `StmtFinal` is similar to the code for other final classes, so we omit it here.

3.6 More Independent Extensibility with Default Methods

Independent extensibility [Zenger and Odersky, 2005] is another requirement of solutions to the Expression Problem besides Wadler's four requirements. Systems that satisfy independent extensibility should be able to combine multiple independently developed extensions easily. In this way, programmers can merge several different extensions into a single compound extension. This section shows the first solution to the problem of independent extensibility that works in Java and uses only the built-in mechanisms for composition (inheritance).

Java 8 introduces *default methods*, allowing interfaces to add a default implementation for methods. Therefore, in combination with multiple interface inheritance, this mechanism provides a limited form of multiple (implementation) inheritance. We exploit this to improve the technique presented in Section 3.5 to be more modular and allow additional reuse and provide a solution to independent extensibility in Java.

3.6.1 Combining Features

We define a feature that collects all literals in an expression. The implementation of this feature is shown in Figure 3.13. For literals, method `collectLiterals` returns a singleton list with the literal. For addition, it merges the collected lists of literals.

Having independently defined two features (pretty printing and collecting literals), we may be interested in a system that contains both of these features, but this appears to require multiple-inheritance. The problem is that components need to be composed of two different features. Since Java only supports single (implementation) inheritance, it may look like we are in trouble.

Fortunately, this is where the choice of interfaces to encode feature modules pays off: while Java only supports single implementation inheritance, it does support multiple interface inheritance. Therefore it is possible to compose multiple feature modules using this mechanism. Figure 3.14 shows how to compose the two features. The interface `PrintCollectFeature` extends both `PrintFeature` and `CollectFeature`. Since all operations are already separately implemented in `PrintFeature` and `CollectFeature`, we only need to do type refinement for the getter methods in interface `Add`. The class

```
interface CollectFeature extends BaseFeature {
   interface Exp extends BaseFeature.Exp {
      List<Integer> collectLiterals();
   }
   interface Lit extends BaseFeature.Lit, Exp {
      public default List<Integer> collectLiterals() {
         List<Integer> lst = new ArrayList<Integer>(1);
         lst.add(getX());
         return lst;
      }
   }
   interface Add extends BaseFeature.Add, Exp {
      abstract Exp getE1(); // type-refinement for e1
      abstract Exp getE2(); // type-refinement for e2
      public default List<Integer> collectLiterals() {
         List<Integer> lst = new ArrayList<Integer>();
         lst.addAll(getE1().collectLiterals());
         lst.addAll(getE2().collectLiterals());
         return lst;
      }
   }
}
```

Figure 3.13: Adding a collecting literals feature to the base system.

`PrintCollectFinal` that implements `PrintCollectFeature` is essentially the same as the code in Figure 3.11, so we omit it here. Client code can then be defined similarly as before.

3.7 Beyond the Expression Problem

As stated in Section 3.1, we identify two main problems (with some subproblems) related to extensibility. The solutions presented so far solve problem 1a) (Hardcoded recursive types in fields) using only subtyping. However, for the more general cases 1b) and 2), we need additional techniques and type-parametrization. This section proposes to extend Wadler's challenge of arithmetic expressions with a new feature: doubling the value of expressions. The doubling feature is inspired by Zenger and Odersky's work [Zenger and Odersky, 2005] and it exercises the more general cases 1b) and 2).

```
interface PrintCollectFeature extends PrintFeature, CollectFeature {
   interface Exp extends PrintFeature.Exp, CollectFeature.Exp {}
   interface Lit extends PrintFeature.Lit, CollectFeature.Lit, Exp {}

   interface Add extends PrintFeature.Add, CollectFeature.Add, Exp {
      abstract Exp getE1(); // type-refinement for e1
      abstract Exp getE2(); // type-refinement for e2
   }
}
```

Figure 3.14: Merging the printing feature and the collecting literals feature.

```
interface DbFeature0 extends BaseFeature{
   interface Exp extends BaseFeature.Exp { Exp db(); }
   class Lit implements BaseFeature.Lit, Exp {
      int x;
      public Lit(int y) { x = y; }
      public int getX() { return x; }
      public Exp db() { return new Lit(2*x); }
   }
}
```

Figure 3.15: Implementing double operation without object algebras.

3.7.1 The Double Feature

In the original Expression Problem, only hardcoded recursive types in fields are encountered in the *evaluation* and *printing* operations. However, it is hard to encode extended operations with recursive types in parameters or return positions. An example is a *double* operation that doubles the value of an expression. The code in Figure 3.15 shows the naive approach of extending the initial system with a double operation.

Now suppose we further extend the system with a new feature, and refine the return type of `db()`, as shown in Figure 3.16. The problem is that, instead of calling `super.db()` we will have to copy the code from the method `db()` in `DbFeature0` to the method `db()` in `NewFeature0`. The expression `super.db()` is not well-typed because its type is `DbFeature0.Exp`, but the expected type is `NewFeature0.Exp`. Since `DbFeature0.Exp` is a *supertype* (and not a subtype) of `NewFeature0.Exp` the result is a type-error. Unfortunately, the code duplication can be considered as a violation of the “no duplication” requirement to the EP.

The code duplication would be solved if we had *virtual constructors*, instead of statically bound constructors. In this context virtual constructors refer to a mechanism that

```
interface NewFeature0 extends DbFeature0 {
   interface Exp extends DbFeature0.Exp { void foo(); }
   class Lit extends DbFeature0.Lit implements Exp {
      public Lit(int y) { super(y); }
      // public Exp db() { return super.db(); } //type error
      public Exp db() { return new Lit(2*x); }
      public void foo() {}
   }
}
```

Figure 3.16: Extending DbFeature with NewFeature without object algebras.

can delay constructor bindings to runtime, similarly to the use of constructors of virtual classes [Madsen and Moller-Pedersen, 1989]. Java does not support virtual constructor semantics. However, it is possible to encode a mechanism similar to virtual constructors using *object algebras* [Oliveira and Cook, 2012]. Object algebras are essentially a generalization of *abstract factories* and contain a set of factory methods. These factory methods can be used instead of constructors to create instances of objects.

3.7.2 Implementing the Double Feature

Figure 3.17 shows an example of using our solution to implement the double operation. The enclosing `DbFeature` class is itself parametrized with a type parameter `E`. This type parameter abstracts over the type of expressions. The type is bounded by `DbFeature.Exp <E>`. Note that `E` appears in the bound itself. This feature of generics is called F-bounded polymorphism [Canning et al., 1989]. The use of F-bounded polymorphism is not surprising, since various lightweight encodings of family polymorphism [Kamina and Tamai, 2007, Saito et al., 2008, Saito and Igarashi, 2008, Kamina and Tamai, 2008] with extensions of Java use it. The interface `ExpAlg<E>` is an object algebra interface with type parameter `E` and methods `lit` and `add` that return values of type E. Note that nested interfaces in Java can only be static, so the use of a type parameter of the enclosing class `DbFeature` is not allowed inside `ExpAlg`. To overcome this limitation, the interface itself has to be parametrized with `E`. The classes `Lit` and `Add` override the abstract method `db()` in class `Exp` and provide concrete implementations. To construct instances of expressions the `db` methods use an object algebra, provided by the abstract method `alg`, to construct the corresponding objects.

With object algebras, writing `NewFeature` becomes possible without code duplication. The code is shown in Figure 3.18. By dynamically deciding which constructor to call, object algebras allow us to reuse code in the superclass by calling `super.db()` in the `db()` methods of `Lit` and `Add`. This no longer breaks the “no duplication” requirement.

```
abstract class DbFeature<E extends DbFeature.Exp<E>> implements
    BaseFeature {
    interface ExpAlg<E> { // requires E parameter because it is an
        interface
        E lit(int x);
        E add(E e1, E e2);
    }
    abstract ExpAlg<E> alg();
    interface Exp<E> extends BaseFeature.Exp { abstract E db(); }
    abstract class Lit implements Exp<E>, BaseFeature.Lit {
        public E db() { return alg().lit(2 * getX()); }
    }
    abstract class Add implements Exp<E>, BaseFeature.Add {
        public abstract E getE1();
        public abstract E getE2();
        public E db() { return alg().add( getE1().db(), getE2().db() )
            ; }
    }
}
```

Figure 3.17: Implementing the double feature using object algebras.

In summary, implementing the more challenging `double` feature is possible. However, we must use a number of techniques and type-parametrization. So this solution is no longer trivial.

3.8 Discussion and Limitations

It may seem surprising that this simple solution to the EP in mainstream languages (Sections 3.2 and 3.4) has not been proposed before in the literature. One possible explanation is that although many languages support covariant type refinement in some form, only Scala allows a straightforward solution using type refinement of immutable fields. In Java, some more ingenuity (and code) is required to make use of covariant return types. Although the Java solution has some boilerplate (because `Final` classes are needed), that code is mechanical and could be automatically generated. The Java solution can also support independent extensibility by changing classes into interfaces with default methods (supported in Java 8).

Although the idea of covariant refinement has not been applied before to solutions of the EP in mainstream OO languages, it has been a fundamental part of various new language designs aimed at solving extensibility problems. For example, languages that support family polymorphism rely on the fact that family extensions allow covariant type refinements. In languages supporting family polymorphism, it is also possible to have a simple solution to

```
abstract class NewFeature<E extends NewFeature.Exp<E>> extends
    DbFeature<E> {
  interface Exp<E> extends DbFeature.Exp<E> { abstract void foo(); }
  abstract class Lit extends DbFeature<E>.Lit implements Exp<E> {
      public E db() { return super.db(); }
  }
  abstract class Add extends DbFeature<E>.Add implements Exp<E> {
      public E db() { return super.db(); }
  }
}
```

Figure 3.18: Extending DbFeature with NewFeature using object algebras.

Wadler's EP [Ernst, 2004]. One important difference to more conventional type systems like Java is that in family polymorphism covariant type-refinement is also possible for arguments of methods. In contrast, Java (or Scala) only allows type-refinements for types used in positive positions (that is, return or field types). There is a good reason for such restriction: it is well-known that naively allowing covariant type-refinement everywhere would lead to type unsoundness. Type systems for family polymorphism need to take special care to ensure that covariant type-refinement can happen everywhere.

Binary and Producer Methods The restriction of type-refinement to types in positive positions implies that binary methods pose extra challenges for extensibility. For example, if expressions were to support a (binary) equality method, then we would want to refine the argument type of the equality method in the extension. However, this is not possible in Scala or Java. Producer methods that transform one expression and produce another are possible using the techniques presented here. However, they introduce some code duplication (as discussed in Section 3.7.1) because the original code of the method cannot be reused in extensions. In the original Wadler's EP, the two operations (printing and evaluation) are consumer methods where the recursive type does not occur anywhere in the signature of the method. It is for this special class of methods that our techniques shine and lead to particularly simple solutions.

Mutability Another limitation of the approach presented here is the lack of mutability of the sub-expressions: the Scala solution relies on immutable fields; and the Java solution relies on getters. We do not know how to support mutability using only subtyping. However, if we also allow the use of generics, then we can obtain a variant of the solution presented here that supports mutability and even removes the need for final classes in Java. The idea is to abstract over the type of expressions in classes with sub-expressions. For example, instead of the classes Add (in Figure 3.5) and AddP (in Figure 3.6), the following classes would be used:

```
class Add<E extends Exp> implements Exp {
   E e1, e2;
   public int eval() {return e1.eval() + e2.eval();}
}
class AddP<E extends ExpP> extends Add<E> implements ExpP {
   public String print() {
      return e1.print() + " + " + e2.print();
   }
}
```

Now, the fields e1 and e2 are mutable, and the types of the fields are refined via the bounds of E. This solution can be viewed as a simplification of Torgersen's solution to the EP [Torgersen, 2004], that avoids uses of F-bounds [Canning et al., 1989] and excessive type-parametrization. The full source code for this variant, including instantiation code, can be found online. If we go all the way to Torgersen's solution it is even possible to deal with binary and producer methods. However this comes at the cost of simplicity, as now the code gets filled with numerous type annotations and bounds.

Family Polymorphism As we have discussed in Section 3.5.3, every family like BaseFeature requires a companion implementation class like BaseFinal. This makes the approach cumbersome to use. However, technically, this boilerplate code can be automatically generated with the help of techniques such as annotation processing. However, in addition to the mechanical code, another limitation is the efficiency problem. As we can see in the client code calling i.Lit(3), each time a new object is created, at the same time, a new anonymous inner class that implements the interface Lit will be created. As the objects in the system accumulate, the memory that the system occupies can go beyond control.

4 Classless Java

This chapter presents an OO style without classes, which we call interface-based object-oriented programming (IB). IB is a natural extension of closely related ideas such as traits [Schärli et al., 2003, Ducasse et al., 2006]. *Abstract state operations* provide a new way to deal with state, which allows for flexibility not available in class-based languages. In IB, state can be type-refined in subtypes. The combination of a purely IB style and type-refinement enables powerful idioms using multiple inheritance and state. To introduce IB to programmers, we created **Classless Java**: an embedding of IB directly into Java. Classless Java uses annotation processing for code generation and relies on new features of Java 8 for interfaces. The code generation techniques used in Classless Java have interesting properties, including guarantees that the generated code is type-safe and has good integration with IDEs. Usefulness of IB and Classless Java is shown with examples and case studies.

4.1 Introduction

As mentioned in Chapter 1, for better code reuse in Object-oriented languages, two main OO models CB (class-based languages) and PB (prototype-based languages) are proposed. However, there are various limitations related to conflicts or fields such as the diamond problem, field initialization problem, mutable field problem, etc. To address those limitations, this chapter presents a third alternative OO model called *interface-based* object-oriented programming languages (IB), where objects implement interfaces directly and fields are not directly supported. In IB interfaces own the implementation for the behavior, which is structurally defined in their interface. Programmers do not define objects directly but delegate the task to *object interfaces*, whose role is similar to non-abstract classes in CB. Objects are instantiated by static factory methods in object interfaces.

Due to the absence of fields, a key challenge in IB lies in how to model state, which is fundamental to having stateful objects. All abstract operations in an object interface are interpreted as *abstract state operations*. The abstract state operations include various common utility methods (such as getters and setters, or clone-like methods). Objects are only responsible for defining the ultimate behavior of a method. Anything related to state is completely contained in the instances and does not leak into the inheritance logic. In CB, the structure of the state is fixed and can only grow by inheritance. In contrast, in IB the state is never fixed, and methods such as abstract setters and getters can always receive an explicit implementation down in the inheritance chain, improving **modularity and flexibility**. That is, the concept of abstract state is more fluid.

Object interfaces provide support for automatic type-refinement. In contrast, in CB special care and verbose explicit type-refinement are required to produce code that deals with subtyping adequately. We believe that such verbosity hinders and slows down the discovery of useful programming patterns involving type-refinement. Our previous work in Chapter 3 on the Expression Problem [Wadler, 1998] in Java-like languages shows how easy it is to solve the problem using only type-refinement. However, it took nearly 20 years since the formulation of the problem for that solution to be presented in the literature. In IB, due to its emphasis on type-refinement, that solution should have been more obvious.

One advantage of abstract state operations and type-refinement is that it allows a new approach to the *type-safe covariant mutable state*. That is, in IB, it is possible to type-refine *mutable* state in subtypes. This is typically forbidden in CB: it is widely known that *naive* type- refinement of mutable fields is not type-safe. Although covariant refinement of mutable fields is supported by some type systems [Bruce et al., 1998, Bruce, 1994, Ernst et al., 2006, Saito and Igarashi, 2013], this requires significant complexity and restrictions to ensure that all uses of the covariant state are indeed type-safe.

IB could be explained by defining a novel language (e.g., **FHJ+** in Chapter 6), with new syntax and semantics. However, this would have a steep learning curve. Therefore, we take a different approach instead. For the sake of providing a more accessible explanation, we will embed our ideas directly into Java. Our IB embedding relies on the new features of Java 8: interface *static methods* and *default methods*, which allow interfaces to have method implementations. In the context of Java, what we propose is a programming style, where we never use classes (more precisely, we never use the `class` keyword). We call this restricted version of Java *Classless Java*.

Using Java annotation processors, we produce an implementation of Classless Java, which allows us to stick to pure Java 8. The implementation works by performing AST rewriting, allowing most existing Java tools (such as IDEs) to work out-of-the-box with our implementation. Moreover, the implementation blends Java's conventional CB style and IB smoothly. We apply object interfaces to several interesting Java programs and conduct various case studies. Finally, we also discuss the behavior of Classless Java and its properties.

In summary, the contributions of this chapter are:

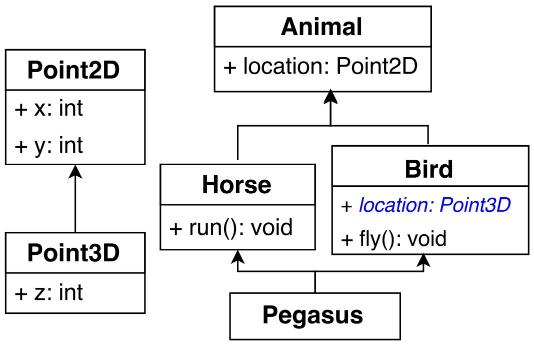


Figure 4.1: The complete structure of the animal system

- **IB and Object Interfaces:** we activate powerful programming idioms using multiple-inheritance, type-refinement and abstract state operations.
- **Classless Java:** we provide a practical realization of IB in Java. Classless Java is implemented using annotation processing, allowing most tools to work transparently with our approach. Existing Java projects can use our approach and still be backward compatible with their clients, in a way that is specified by our safety properties.
- **Type-safe covariant mutable state:** we show how the combination of abstract state operations and type-refinement enables a form of mutable state that can be covariantly refined in a type-safe way.
- **Applications and case studies:** we illustrate the usefulness of IB through various examples and case studies[1]. An extended version with a formal translation to Java can be found in Appendix A.1.

4.2 A Running Example: Animals

This section illustrates how our programming style, supported by `@Obj`, enables powerful programming idioms based on multiple inheritance and type refinements. We propose a standard example: `Animals` with a 2-dimensional `Point2D` representing their `location`, subtypes `Horses`, `Birds`, and `Pegasus`. Birds can `fly`, thus their locations need to be 3-dimensional `Point3Ds` (field type refinement). We model `Pegasus` (a well-known creature in Greek mythology) as a kind of `Animal` with the skills of both `Horses` and `Birds` (multiple inheritance). A simple class diagram illustrating the basic system is given in Fig. 4.1.

[1] https://github.com/YanlinWang/classless-java

```
interface Animal {} // no points yet!
interface Horse extends Animal {
   default void run(){out.println("run!");}
}
interface Bird extends Animal {
   default void fly(){out.println("fly!");}
}
interface Pegasus extends Horse, Bird {}
```

Figure 4.2: Code for the simplified animal system

4.2.1 Simple Multiple Inheritance with Default Methods

Before modelling the complete animal system, we start with a simple version without locations. This version serves the purpose of illustrating how Java 8 default methods can already model simple forms of multiple inheritance. `Horse` and `Bird` are subtypes of `Animal`, with methods `run()` and `fly()`, respectively. Pegasus can not only *run* but also *fly*! This is the place where *"multiple inheritance"* is necessary because `Pegasus` needs to inherit the `fly` and `run` functionalities from both `Horse` and `Bird`. The first attempt to model the animal system is depicted in Fig. 4.2. Note that the implementations of the methods `run` and `fly` are defined inside interfaces, using default methods. Moreover, because interfaces support multiple interface inheritance, the interface for `Pegasus` can inherit behavior from both `Horse` and `Bird`. Although Java interfaces do not allow instance fields, no form of state is needed so far to model the animal system.

Instantiation To use `Horse`, `Bird`, and `Pegasus`, some objects must be created first. A first problem with using interfaces to model the animal system is that interfaces cannot be directly instantiated. Classes, such as:

```
class HorseImpl implements Horse {}
class BirdImpl implements Bird {}
class PegasusImpl implements Pegasus {}
```

are needed for instantiation. Now a `Pegasus` animal can be created using the class constructor:

```
Pegasus p = new PegasusImpl();
```

There are some annoyances here. Firstly, the sole purpose of the classes is to provide a way to instantiate objects. Although (in this case) it takes only one line of code to provide each of those classes, this code is essentially boilerplate code, which does not add behavior to the system. Secondly, the namespace gets filled with three additional types. For example, both

`Horse` and `HorseImpl` are needed: `Horse` is needed as an interface so that `Pegasus` can use multiple inheritance; and `HorseImpl` is required to provide object instantiation. Note that, for this very simple animal system, plain Java 8 anonymous classes can be used to avoid these problems. We could have simply instantiated `Pegasus` using:

```
Pegasus p = new Pegasus() {}; // anonymous class
```

However, as we shall see, once the system gets a little more complicated, the code for instantiation quickly becomes more complex and verbose (even with anonymous classes).

4.2.2 Object Interfaces and Instantiation

To model the animal system with object interfaces all that a user needs to do is to add an `@Obj` annotation to the `Horse`, `Bird`, and `Pegasus` interfaces:

```
@Obj interface Horse extends Animal {
    default void run() {out.println("running!");}
}
@Obj interface Bird extends Animal {
    default void fly() {out.println("flying!");}
}
@Obj interface Pegasus extends Horse, Bird {}
```

The effect of the annotations is that a static *factory* method called `of` is automatically added to the interfaces. With the `of` method a `Pegasus` object is instantiated as follows:

```
Pegasus p = Pegasus.of();
```

The `of` method provides an alternative to a constructor, which is missing from interfaces. The followings show the code corresponding to the `Pegasus` interface after the `@Obj` annotation is processed:

```
interface Pegasus extends Horse, Bird {
  // generated code not visible to users
  static Pegasus of() { return new Pegasus() {}; }
}
```

Note that the generated code is transparent to users, who only see the original code with the `@Obj` annotation. Compared to the pure Java solution in Section 4.2.1, the solution using object interfaces has the advantage of providing a direct mechanism for object instantiation, which avoids adding boilerplate classes to the namespace.

4.2.3 Object Interfaces with State

The animal system modeled so far is a simplified version of the system presented in Fig. 4.1. The example is still not sufficient to appreciate the advantages of IB programming. Now we model the complete animal system where an `Animal` includes a `location` representing its position in space. We use 2D points to keep track of locations.

`Point2D`: simple immutable data with fields Points can be modeled with interfaces. In IB state is accessed and manipulated using abstract methods. The normal approach to model points in Java is to use a class with fields for the coordinates. In Classless Java, interfaces are used instead:

```
interface Point2D { int x(); int y(); }
```

The encoding over Java is now inconvenient: creating a new point object is cumbersome, even with anonymous classes:

```
Point2D p = new Point2D() {
  public int x() {return 4;}
  public int y() {return 2;}
}
```

However, this cumbersome syntax is not required for every object allocation. As programmers do, for ease or reuse, the boring and repetitive code can be encapsulated in a method. A generalization of the `of` static factory method is appropriate:

```
interface Point2D { int x(); int y();
  static Point2D of(int x, int y) {
    return new Point2D() {
      public int x(){return x;}
      public int y(){return y;}
    };
  }
}
```

`Point2D` with object interfaces This obvious "constructor" code is generated by the `@Obj` annotation. By annotating the interface `Point2D`, a variation of the shown static method `of` will be generated, mimicking the functionality of a simple-minded constructor. `@Obj` first looks at the abstract methods and detects what the fields are, then generates an `of` method with one parameter for each of them. We can just write:

```
@Obj interface Point2D { int x(); int y(); }
```

A field or factory parameter is generated for every abstract method that takes no parameters. An example of using `Point2D`, where we "clone" an existing point but use `42` as the x-coordinate, is:

```
Point2D p = Point2D.of(42,myPoint.y());
```

`with-` methods in object interfaces The pattern of creating a new object by reusing most information from an old object is very common when programming with immutable data-structures. As such, it is supported by `@Obj` as `with-` methods:

```
@Obj interface Point2D {
    int x();  int y(); // getters
    // with- methods
    Point2D withX(int val);
    Point2D withY(int val);
}
```

Using `with-` methods, the point `p` can also be created by:

```
Point2D p = myPoint.withX(42);
```

If there is a large number of fields, `with-` methods will save programmers from writing large amounts of tedious code that simply copies field values. Moreover, if the programmer wants a different implementation, he may provide an alternative implementation using `default` methods. For example:

```
@Obj interface Point2D {
    int x(); int y();
    default Point2D withX(int val){ /*myCode*/ }
    Point2D withY(int val);
}
```

is expanded into

```
interface Point2D {
    int x(); int y();
    default Point2D withX(int val) { /*myCode*/ }
    Point2D withY(int val);
    static Point2D of(int _x, int _y) {
      return new Point2D() {
        int x=_x;
        int y=_y;
        public int x() { return x; }
        public int y() { return y; }
        public Point2D withY(int val) {
          return of(x(),val);
        }
      };
    }
}
```

Only code for methods needing implementation is generated. Thus, programmers can easily customize the behavior for their special needs. Also, since `@Obj` interfaces offer the `of` factory method, only interfaces where all the abstract methods can be synthesized can be object interfaces. A non-`@Obj` interface is like an abstract class in Java.

`Animal` and `Horse`: simple mutable data with fields 2D points are mathematical entities, thus we choose an immutable data structure to model them. Animals are real-world entities, and when an animal moves, it is the *same* animal with a different location. We model this with mutable state:

```
interface Animal {
   Point2D location();
   void location(Point2D val);
}
```

Here we declare an abstract getter and a setter for the mutable "field" `location`. Without the `@Obj` annotation, there is no convenient way to instantiate `Animal`. For `Horse`, the `@Obj` annotation is used, and an implementation of `run()` is defined using a `default` method. The implementation of `run()` further illustrates the convenience of `with-` methods:

```
@Obj interface Horse extends Animal {
   default void run() {
      location(location().withX(location().x()+20));
   }
}
```

Creating and using `Horse` is quite simple:

```
Point2D p = Point2D.of(0, 0);
Horse horse = Horse.of(p);
horse.location(p.withX(42));
```

Note how the `of`, `withX` and `location` methods (generated automatically) give a basic interface for dealing with animals.

In summary, state (mutable or not) in object interfaces relies on a notion of abstract state, and state is not directly available to programmers. Instead, programmers use methods, called *abstract state operations*, to interact with state.

4.2.4 Object Interfaces and Subtyping

`Birds` are `Animals`, but while `Animals` only need 2D locations, `Birds` need 3D locations. Therefore when the `Bird` interface extends the `Animal` interface, the notion of points needs to be *refined*. Such kind of refinement is challenging in typical class-based approaches. Fortunately, with object interfaces, we are able to provide a simple but effective solution.

Unsatisfactory class-based solutions to field type refinement In Java, if we want to define an animal class with a field, we have a set of unsatisfactory options to choose from:

- Define a `Point3D` field in `Animal`: this is bad since all animals would require more than needed. Also, it requires adapting the old code to accommodate for new evolutions.
- Define a `Point2D` field in `Animal` and define an extra **`int`** `z` field in `Bird`. This solution is very ad-hoc, requiring to basically duplicate the difference between `Point2D` and `Point3D` inside `Bird`. The most dramatic criticism is that it would not scale to a scenario when `Bird` and `Point3D` are from different programmers.
- Redefine getters and setters in `Bird`, always put `Point3D` objects in the field and cast the value out of the `Point2D` field to `Point3D` when implementing the overridden getter. This solution scales to the multiple programmers approach, but requires ugly casts and can be implemented in a wrong way leading to bugs.

We may be tempted to assume that a language extension is needed. Instead, the *restriction* of (object) interfaces to have no fields enlightens us that another approach is possible; often in programming languages "freedom is slavery."

Field type refinement with object interfaces Object interfaces address the challenge of type-refinement as follows:

- by *covariant method overriding*, the return type of `location()` is refined to `Point3D`;
- by *overloading*, a new setter for location is defined with a more precise type;
- a **`default`** setter implementation with the old signature is provided by the programmer.

Thus the code for the `Bird` interface is:

```
@Obj interface Bird extends Animal {
    Point3D location();
    void location(Point3D val);
    default void location(Point2D val) {
        location(location().with(val));
    }
    default void fly() {
        location(location().withX(location().x() + 40));
    }
}
```

From the type perspective, the key is the covariant method overriding of `location()`. However, from the semantics perspective the crux is the implementation for the setter with the old signature (`location(Point2D)`). The core part of the setter implementation is a new type of **`with`** method, called a (functional) property updater.

`Point3D` and property updaters The `Point3D` interface is defined as follows:

```
@Obj interface Point3D extends Point2D {
    int z();
    Point3D withZ(int z);
    Point3D with(Point2D val);
}
```

`Point3D` includes a **`with`** method, taking a `Point2D` as an argument. Other wither methods (such as `withX`) functionally update a field one at a time. This can be inefficient, and sometimes hard to maintain. Often we want to update multiple fields simultaneously, for example using another object as source. Following this idea, the method **`with`**`(Point2D)` is an example of a (functional) property updater: it takes an object and returns a copy of the current object where all the fields that match fields in the parameter object are updated to the corresponding value in the parameter. The idea is that the result should be like **`this`**, but modified to be as similar as possible to the parameter.

With the new **`with`** method, we may use the information for `z` already stored in the object to forge an appropriate `Point3D` to store. Note how all the information about what fields sit in `Point3D` and `Point2D` is properly encapsulated in the **`with`** method and is transparent to the implementer of `Bird`.

Property updaters never break class invariants, since they internally call operations that were already deemed safe by the programmer. For example, a list object would not offer a setter for its `size` field (which should be kept hidden). Thus a property updater would not attempt to set it.

Generated boilerplate To give an idea of how much code `@Obj` is generating, we show the generated code for `Point3D` in Figure 4.3. Writing such code by hand is error-prone. For example a distracted programmer may swap the arguments of calls to `Point3D.of`. Note how **`with`**- methods are automatically refined in their return types, so that code like:

```
Point3D p = Point3D.of(1,2,3);
p = p.withX(42);
```

will be accepted. If the programmer wishes to suppress this behavior and keep the signature as it was, it is sufficient to redefine the **`with`**- methods in the new interface repeating the old

```
interface Point3D extends Point2D {
    int z();
    Point3D withZ(int val);
    Point3D with(Point2D val);
    // generated code
    Point3D withX(int val);
    Point3D withY(int val);
    public static Point3D of(int _x, int _y, int _z) {
        int x=_x; int y=_y; int z=_z;
        return new Point3D() {
            public int x(){return x;}
            public int y(){return y;}
            public int z(){return z;}
            public Point3D withX(int val){
                return Point3D.of(val, this.y(), this.z());
            }
            public Point3D withY(int val){
                return Point3D.of(this.x(), val, this.z());
            }
            public Point3D withZ(int val){
                return Point3D.of(this.x(), this.y(), val);
            }
            public Point3D with(Point2D val){
                if(val instanceof Point3D)
                    return (Point3D)val;
                return Point3D.of(val.x(), val.y(), this.z());
            }
        };
    }
}
```

Figure 4.3: Generated boilerplate code.

	Operation	**Example**	**Description**
State operations (for a field `x`)	**"fields"/getters**	`int x()`	Retrieves value from field `x`.
	withers	`Point2D withX(int val)`	Clones object; updates field `x` to `val`.
	setters	`void x(int val)`	Sets the field `x` to a new value `val`.
	fluent setters	`Point2D x(int val)`	Sets the field `x` to `val` and returns `this`.
Other operations	**factory methods**	`static Point2D of(int _x,int _y)`	Factory method (generated).
	functional updaters	`Point3D with(Point2D val)`	Updates all matching fields in `val`.

Figure 4.4: Abstract state operations for a field `x`, together with other operations, supported by the `@Obj` annotation.

signature. Again, the philosophy is that if the programmer provides something directly, `@Obj` does not touch it. The cast in `with(Point2D)` is trivially safe because of the `instanceof` test. The idea is that if the parameter is a subtype of the current exact type, then we can just return the parameter, as something that is just "more" than `this`.

Summary of operations in Classless Java In summary, object interfaces provide support for different types of abstract state operations: four field-based state operations; and functional updaters. Object instantiation is directly supported by `of` factory methods. Figure 4.4 summarizes the six operations supported by `@Obj`. The field-based abstract state operations are determined by naming conventions and the types of the methods. Fluent setters are a variant of conventional setters and are discussed in more detail in Section 4.4.2.

4.2.5 Advanced Multiple Inheritance

Finally, defining `Pegasus` is as simple as we did in the simplified (and stateless) version in Fig. 4.2. Note how even the non-trivial pattern for field type refinement is transparently composed, and `Pegasus` has a `Point3D location`:

```
@Obj interface Pegasus extends Horse, Bird {}
```

4.3 Bridging between IB and CB in Java

Creating a new language/extension would be an elegant way to illustrate the idea of IB. However, a significant amount of engineering effort would be needed to build a practical language and achieve a similar level of integration and tool support as Java. To be practical, we have instead implemented `@Obj` as an annotation in Java 8, and a *compilation agent*. That is, the Classless Java style of programming is supported by the library.

Disciplined use of Classless Java (avoiding class declarations as done in Section 4.2) illustrates what *pure* IB is like. However, using `@Obj`, CB and IB programming can be mixed together, harvesting the practical convenience of using existing Java libraries, the full Java

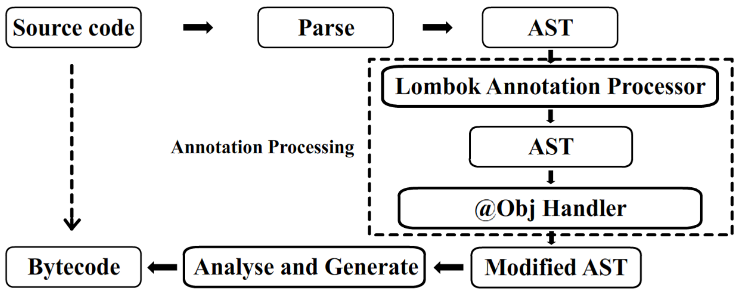


Figure 4.5: The flow chart of `@Obj` annotation processing.

language and IDE support. The key to our implementation is a compilation agent, which allow us to rewrite the Java abstract syntax tree (AST) just before compilation. We discuss the advantages and limitations of our approach in the following.

4.3.1 Compilation Agents

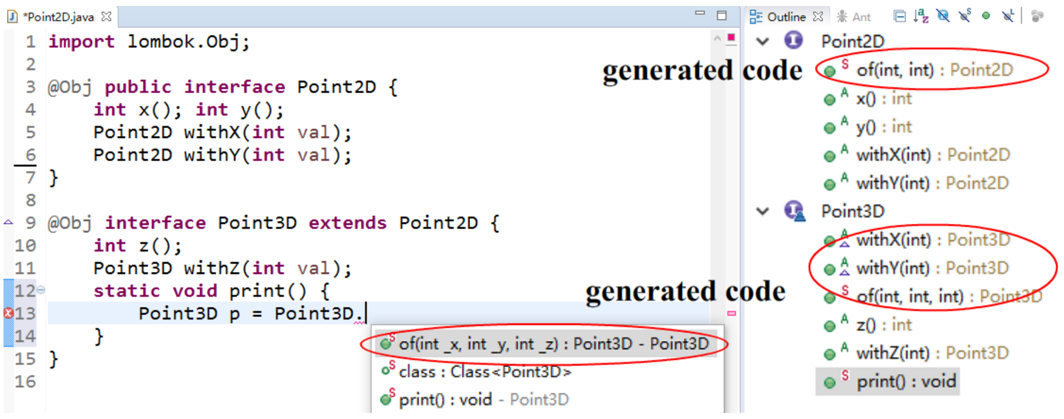
As we have reviewed in Section 2.4, Java supports compilation agents, where Java libraries can interact with the Java compilation process, acting as a man in the middle between the generation of AST and bytecode.

Lombok is one such compilation agent which `@Obj` was created based on. Figure 4.5 [Neildo, 2011] illustrates the flow of the `@Obj` annotation. First Java source code is parsed into an AST. The AST is then captured by Lombok: each annotated node is passed to the corresponding (Eclipse or Javac) handler. The handler is free to modify the information of the annotated node, or even inject new nodes (like methods, inner classes, etc). Finally, the Java compiler works on the modified AST to generate bytecode.

Features written in Lombok are supported directly in the language and are also supported by most tools. In Figure 4.6, `@Obj` generates an `of` method in `Point2D`, and `of`, `withX`, `withY` methods in `Point3D`. In Eclipse, the processing is performed transparently and the information of the interface from the compilation is captured in the “Outline” window.

4.3.2 `@Obj` AST Reinterpretation

Of course, a careless reinterpretation of the AST could still be surprising for poorly designed rewritings. `@Obj` reinterprets the syntax with the sole goal of *enhancing and completing code*: we satisfy the behavior of abstract methods; add method implementations; and refine return types. We consider this to be quite easy to follow and reason about since it is similar to what happens in normal inheritance. Refactoring operations like renaming and moving should



```java
import lombok.Obj;

@Obj public interface Point2D {
    int x(); int y();
    Point2D withX(int val);
    Point2D withY(int val);
}

@Obj interface Point3D extends Point2D {
    int z();
    Point3D withZ(int val);
    static void print() {
        Point3D p = Point3D.
    }
}
```

Figure 4.6: Generated methods shown in the Outline window of Eclipse and auto-completion.

work transparently in conjunction with our annotation, since they rely on the overall type structure of the class, which we do not arbitrarily modify but just complete.

Thus, in addition to the advantages of Lombok, Classless Java offers more advantages with respect to arbitrary (compilation agent driven) AST rewriting.

Syntax and type errors Some preprocessors (like the C one) can produce syntactically invalid code. Lombok ensures only syntactically valid code is produced. Classless Java additionally guarantees that no type errors are introduced in generated code and client code. We discuss these two guarantees in more detail as follows:

- **Self coherence**: the generated code itself is well-typed. In our case, it means that either `@Obj` produces (in a controlled way) an understandable error or the interface can be successfully annotated and the generated code (e.g., the `of` methods in Figure 4.6) is well-typed.

- **Client coherence**: all the client code (for example method calls) that is well-typed before code generation is also well-typed after the generation. The annotation just adds more behavior without removing any functionality.

Heir coherence Another form of guarantee that could be useful in AST rewriting is heir coherence. That is, interfaces (and in general classes) inheriting the instrumented code are well-typed if they were well-typed without the instrumentation. In a strict sense, our rewriting *does not* guarantee heir coherence. The reason is that this would forbid adding any (default or abstract) method to the annotated interfaces or even doing type refinement. Indeed consider the following:

```
interface A { int x(); A withX(int x); }
@Obj interface B extends A {}
interface C extends B { A withX(int x); }
```

This code is correct before the translation, but `@Obj` would generate in `B` a method "`B withX (int x);`". This would break `C`. Similarly, an expression of the form "`new B(){.. A withX(int x){..}}`" would be correct before translation, but ill-typed after the translation.

Our automatic type refinement is a useful and convenient feature, but not transparent to the heirs of the annotated interface. They need to be aware of the annotation semantics and provide the right type while refining methods. To support heir coherence, we need to give up automatic type refinement, which is an essential part of IB programming. However, Java libraries almost always break heir coherence during evolution and still claim backward compatibility. In practice, adding any method to any non-final class of a Java library is sufficient to break heir coherence. We think return type refinement breaks heir coherence "less" than normal library evolution, and if no automatic type- refinements are needed, then `@Obj` can claim a form of heir coherence. Formal definition/proofs for our safety claims can be found in Appendix A.1.

4.3.3 Limitations

Our prototype implementation has certain limitations:

- Lombok allows writing handlers for either javac or ejc (Eclipse's own compiler). Our current implementation only realizes ejc version. The implementation for the `javac` version is still missing.
- Simple generics is supported: type parameters can be used, but generic method typing is delegated to the Java compiler instead of being explicitly checked by `@Obj`.
- Due to limited support in Lombok for separate compilation, i.e., accessing information of code defined in different files, `@Obj` requires that all related interfaces have to appear in a single Java file. Reusing the logic inside the experimental Lombok annotation `@ Delegate`, we also offer a less polished annotation supporting separate compilation.

4.4 Applications and Case Studies

This section illustrates applications and larger case studies for Classless Java. The first application shows how a useful pattern, using multiple inheritance and type-refinement, can be conveniently encoded in Classless Java. The second application shows how to model embedded DSLs (domain specific languages) based on fluent APIs. Then two larger case studies refactor existing projects into Classless Java. The first one shows a significant reduction in code size, while the second one maintains the same amount of code, but improves modularity.

4.4.1 The Expression Problem with Object Interfaces

As the first application for Classless Java, we illustrate a useful programming pattern that improves the modularity and extensibility of programs. This useful pattern is based on an existing solution to the *Expression Problem* (EP) [Wadler, 1998], which is a well-known problem about modular extensibility issues in software evolution. Actually, our solution in Chapter 3 using only covariant type refinement was proposed. When this solution is modeled with interfaces and default methods, it can even provide independent extensibility [Zenger and Odersky, 2005]: the ability to assemble a system from multiple, independently developed extensions. Unfortunately, the required instantiation code makes a plain Java solution verbose and cumbersome to use. The `@Obj` annotation is sufficient to remove the boilerplate code, making the presented approach very appealing. Our last case study, presented in Section 4.4.4, is essentially a (much larger) application of this pattern to an existing program. Here we illustrate the pattern in the much smaller Expression Problem.

Initial system In the formulation of the EP, there is an initial system that models arithmetic expressions with only literals, addition, and an initial operation `eval` for expression evaluation. As shown in Figure 4.7, `Exp` is the common super-interface with operation `eval()` inside. Sub-interfaces `Lit` and `Add` extend interface `Exp` with default implementations for the `eval` operation. The number field `x` of a literal is represented as a getter method `x()` and expression fields (`e1` and `e2`) of an addition as getter methods `e1()` and `e2()`.

Adding a new type of expressions In the OO paradigm, it is easy to add new types of expressions. For example, the following code shows how to add subtraction.

```
@Obj interface Sub extends Exp {
    Exp e1(); Exp e2();
    default int eval() {
        return e1().eval() - e2().eval();
    }
}
```

```
interface Exp { int eval(); }
@Obj interface Lit extends Exp {
   int x();
   default int eval() {return x();}
}
@Obj interface Add extends Exp {
   Exp e1(); Exp e2();
   default int eval() {
      return e1().eval() + e2().eval();
   }
}
```

Figure 4.7: The Expression Problem: initial system.

```
interface ExpP extends Exp {String print();}
@Obj interface LitP extends Lit, ExpP {
   default String print() {return "" + x();}
}
@Obj interface AddP extends Add, ExpP {
   ExpP e1(); //return type refined!
   ExpP e2(); //return type refined!
   default String print() {
      return "(" + e1().print() + " + "
             + e2().print() + ")";}
}
```

Figure 4.8: The Expression Problem: code for adding print operation.

Adding a new operation The difficulty of the EP in OO languages arises from adding new operations. For example, adding a pretty printing operation would typically change all existing code. However, a solution should add operations in a type-safe and modular way. This turns out to be easily achieved with the assistance of `@Obj`. The code in Figure 4.8 shows how to add the new operation `print`. Interface `ExpP` extends `Exp` with the extra method `print()`. Interfaces `LitP` and `AddP` are defined with default implementations of `print()`, extending base interfaces `Lit` and `Add`, respectively. Importantly, note that in `AddP`, the types of "*fields*" (i.e., the getter methods) `e1` and `e2` are refined. If the types were not refined then the `print()` method in `AddP` would fail to type-check.

Independent extensibility To show that our approach supports independent extensibility, a new operation `collectLit` which collects all literal components in an expression is defined. For space reasons, we omit some code:

```
interface ExpC extends Exp {
   List<Integer> collectLit();
}
@Obj interface LitC extends Lit, ExpC {...}
@Obj interface AddC extends Add, ExpC {
   ExpC e1();
   ExpC e2(); ...
}
```

Now we combine the two extensions together:

```
interface ExpPC extends ExpP, ExpC {}
@Obj interface LitPC extends ExpPC, LitP, LitC {}
@Obj interface AddPC extends ExpPC, AddP, AddC {
   ExpPC e1();
   ExpPC e2();
}
```

`ExpPC` is the new expression interface supporting `print` and `collectLit` operations; `LitPC` and `AddPC` are the extended variants. Notice that except for the routine of **extends** clauses, no glue code is required. Return types of `e1,e2` must be refined to `ExpPC`. Creating a simple expression of type `ExpPC` is as simple as:

```
ExpPC e8 = AddPC.of(LitPC.of(3), LitPC.of(4));
```

Without Classless Java, tedious instantiation code would need to be defined manually.

4.4.2 Embedded DSLs with Fluent Interfaces

Since the style of fluent interfaces was invented in Smalltalk as method cascading, more and more languages (Java, C++, Scala, etc.) came to support fluent interfaces. In most languages, to create fluent interfaces, programmers have to either hand-write everything or create a wrapper around the original non-fluent interfaces using **this**. In Java, there are several libraries (including jOOQ, op4j, fluflu, JaQue, etc) providing useful fluent APIs. However, most of them only provide a fixed set of predefined fluent interfaces.

The `@Obj` annotation can also be used to create fluent interfaces. When creating fluent interfaces with `@Obj`, there are two main advantages:

1. Instead of forcing programmers to hand-write code using **return this**, our approach with `@Obj` annotation removes this verbosity and automatically generates fluent setters.

2. The approach supports extensibility: the return types of fluent setters are automatically refined.

We use embedded DSLs of two simple SQL query languages to illustrate. The first query language `Database` models `select`, `from` and `where` clauses:

```
@Obj interface Database {
   String select(); Database select(String select);
   String from(); Database from(String from);
   String where(); Database where(String where);
   static Database of() {
      return of("", "", "");
   }
}
```

The main benefit that fluent methods give us is the convenience of method chaining:

```
Database query1 = Database.of().select("a, b").from("Table").where("c
     > 10");
```

Note how all the logic for the fluent setters is automatically provided by the `@Obj` annotation.

Extending the query language The previous query language can be extended with a new feature `orderBy` which orders the result records by a field that users specify. With `@Obj` programmers just need to extend the interface `Database` with new features, and the return type of fluent setters in `Database` is automatically refined to `ExtendedDatabase`:

```
@Obj interface ExtendedDatabase extends Database {
   String orderBy();
   ExtendedDatabase orderBy(String orderBy);
   static ExtendedDatabase of() {
      return of("", "", "","");
   }
}
```

In this way, when a query is created using `ExtendedDatabase`, all the fluent setters return the correct type instead of the old `Database` type, which would prevent calling `orderBy`.

```
ExtendedDatabase query2 = ExtendedDatabase.of().select("a, b").from("
   Table").where("c > 10").orderBy("b");
```

Languages like Smalltalk and Dart offer method cascading and avoid the need for fluent setters. This is achieved at the price of introducing additional syntax and intrinsically relies on an imperative setting. Our approach supports both fluent setters and (functional) fluent withers.

4.4.3 A Maze Game

This case study is a simplified variant of a Maze game, which is often used [Gamma et al., 1995, Bono et al., 2014] to evaluate code reuse ability related to

	SLOC	# of classes/interfaces
Bono et al.	335	14
Ours	199	11
Reduced by	40.6%	21.4%

Table 4.1: Maze game code size comparison

Code	SLOC	Code	SLOC
original (`eval`)	626	original (`eval+print`)	661
refactored (`eval`)	560	refactored (`eval+print`)	677

Table 4.2: Interpreter code size comparison

inheritance and design patterns. In the game, there is a player with the goal of collecting as many coins as possible. She may enter a room with several doors to be chosen among. This is a good example because it involves code reuse (different kinds of doors inherit a common type, with different features and behavior), multiple inheritance (a special kind of door may require features from two other door types) and it also shows how to model operations `symmetric sum`, `override` and `alias` from trait-oriented programming. The game has been implemented using plain Java 8 and default methods by Bono et al. [Bono et al., 2014], and the code for that implementation is available online. We refactored the game using `@Obj`[2].

We summarize the number of lines of code and classes/interfaces in each implementation in Table 4.1. The `@Obj` annotation reduced the interfaces/classes used in Bono et al.'s implementation by 21.4% (from 14 to 11) and the number of source lines of code (SLOC) is reduced by 40%. We show some excerpts of Bono's implementation in Appendix A.5.1 and **Classless Java** implementation in Appendix A.5.2. To implement `KnockDoor`, Bono's implmentation needs four components: `TDoor`, `TCounter`, `TKnockDoor` and `KnockDoor` with 124 lines of code. **Classless Java**'s implementation only needs three components: `TDoor`, `TCounter` and `TKnockDoor`. As we can see, the number of classes is reduced due to the replacement of instantiation class `KnockDoor` in Appendix A.5.1 with the generated of method in `TKnockDoor` in Appendix A.5.2. The SLOC is reduced due to both the removal of instantiation overhead and generation of getters/setters.

[2] `https://github.com/YanlinWang/classless-java/tree/master/bundle/UseMixinLombok/src/casestudy/mazegame`

4.4.4 Refactoring an Interpreter

The last case study refactors the code from an interpreter for a Lisp-like language `Mumbler`[3], which is created as a tutorial for the Truffle Framework [Würthinger et al., 2013]. Keeping a balance between simplicity and useful features, `Mumbler` contains numbers, booleans, lists (encoding function calls and special forms such as if-expression, lambdas, etc.). In the original code base, which consists of 626 SLOC of Java, only one operation `eval` is supported. Extending Mumbler to support one more operation, such as a pretty printer `print`, would normally require changing the existing code base directly.

Our refactoring applies the pattern presented in Section 4.4.1 to the existing Mumbler code base to improve its modularity and extensibility. Using the refactored code base, it becomes possible to add new operations modularly and to support independent extensibility. We add one more operation `print` to both the original and the refactored code base. In the original code base, the pretty printer is added non-modularly by modifying the existing code. As shown in table 4.2 the pretty printer in the refactored code is added modularly with more interfaces. Thus the code is slightly increased by 2.4% SLOC. However, the modularity is greatly increased, allowing for improved reusability and maintainability.

[3] `https://github.com/cesquivias/mumbler/tree/master/simple`

Part II

Revisiting Models of Multiple Inheritance

5 FHJ

In terms of method conflicts in multiple inheritance, numerous existing work provides solutions for conflicts which arise from *diamond inheritance*: i.e., conflicts that arise from implementations sharing a common ancestor. However, most mechanisms are inadequate to deal with *unintentional method conflicts*: conflicts which arise from two unrelated methods that happen to share the same name and signature.

This chapter presents a new model called *Featherweight Hierarchical Java* (**FHJ**) that deals with unintentional method conflicts. In our new model, which is partly inspired by C++, conflicting methods arising from unrelated methods can coexist in the same class, and *hierarchical dispatching* supports unambiguous lookups in the presence of such conflicting methods. To avoid ambiguity, hierarchical information is employed in method dispatching, which uses a combination of static and dynamic type information to choose the implementation of a method at run-time. Furthermore, unlike all existing inheritance models, our model supports *hierarchical method overriding*: that is, methods can be *independently overridden* along the multiple inheritance hierarchy. We give illustrative examples of our language and features and formalize **FHJ** as a minimal Featherweight-Java style calculus.

5.1 Introduction

Inheritance in OOP offers a mechanism for code reuse. However many OOP languages are restricted to single inheritance, which is less expressive and flexible than multiple inheritance. Nevertheless, different flavours of multiple inheritance have been adopted in some popular OOP languages. C++ has had multiple inheritance from the start. Scala adapts the ideas from traits [Schärli et al., 2003, Ducasse et al., 2006, Liquori and Spiwack, 2008] and mixins [Bracha and Cook, 1990, Flatt et al., 1998, Limberghen and Mens, 1996, Ancona et al., 2003, Hendler, 1986] to offer a disciplined form of multiple inheritance. Java 8 provides a simple variant of traits, disguised as interfaces with default methods [Goetz and Field, 2012].

A reason why programming languages have resisted to multiple inheritance in the past is due to the difficulty of implementing multiple inheritance. One of the most sensitive and critical issues is perhaps the ambiguity introduced by multiple inheritance. One case is the famous *diamond problem* [Sakkinen, 1989, Singh, 1995] (also known as the *fork-join inheritance* [Sakkinen, 1989]). In the diamond problem, inheritance allows one feature to be inherited from multiple parent classes that share a common ancestor. Hence conflicts arise. The variety of strategies for resolving such conflicts urges the occurrence of different multiple inheritance models, including traits, mixins, CZ [Malayeri and Aldrich, 2009], and many others. Existing languages and research have addressed the issue of diamond inheritance extensively. Other issues including how multiple inheritance deals with state, have also been discussed quite extensively [Wang et al., 2016, Malayeri and Aldrich, 2009, Stroustrup, 1995].

In contrast to diamond inheritance, the second case of ambiguity is *unintentional method conflicts* [Schärli et al., 2003]. In this case, conflicting methods do not actually refer to the same feature, meaning that methods can be designed for different functionality but happen to have the same names (and signatures). A simple example of this situation is two `draw` methods that are inherited from a deck of cards and a drawable widget, respectively. In such a context, the two `draw` methods have very different meanings, but they happen to share the same name. When inheritance is used to compose these classes, a compilation error happens due to conflicts. However, unlike the diamond problem, the conflicting methods have very different meanings and do not share a common parent. We call such a case *fork inheritance*, in analogy to diamond inheritance.

When unintentional method conflicts happen, they can have severe effects in practice if no appropriate mechanisms to deal with them are available. In practice, existing languages only provide limited support for the issue. In most languages, the mechanisms available to deal with this problem are the same as the diamond inheritance. However, this is often inadequate and can lead to tricky issues in practice. This is especially the case when it is necessary to combine two large modules and their features, but the inheritance is simply prohibited by a small conflict. As a workaround from the diamond inheritance side, it is possible to define a new method in the child class to override those conflicting methods. However, using one method to fuse two unrelated features is clearly unsatisfactory. Therefore we need a better solution to keep both features separately during inheritance, so as not to break *independent extensibility* [Zenger and Odersky, 2005].

C++ and C# do allow for two unintentionally conflicting methods to coexist in a class. C# allows this by interface multiple inheritance and explicit method implementations. But since C# is a single inheritance language, it is only possible to *implement* multiple interfaces (but not multiple classes). C++ accepts fork inheritance and resolves the ambiguity by specifying the expected path by *upcasts*. However, neither the C# nor C++ approaches allow such conflicting methods to be further overridden. Some other workarounds or approaches in-

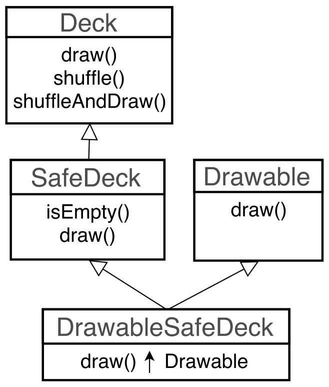


Figure 5.1: `DrawableSafeDeck`: an illustration of hierarchical overriding.

clude delegation and renaming/exclusion in the trait model. However, renaming/exclusion can break the subtyping relation between a subclass and its parent. This is not adequate for the class model commonly used in mainstream OOP languages, where the subclass is always expected to be a subtype of the parent class.

In this chapter, we propose two mechanisms to deal with unintentional method conflicts: *hierarchical dispatching* and *hierarchical overriding*. Hierarchical dispatching is inspired by the mechanisms in C++ and provides an approach to method dispatching, which combines static and dynamic information. Using hierarchical dispatching, the method binder will look at both the *static type* and the *dynamic type* of the receiver during runtime. When there are multiple branches that cause unintentional conflicts, the static type can specify one branch among them for unambiguity, and the dynamic type helps to find the most specific implementation. In that case, both unambiguity and extensibility are preserved. The main novelty over existing work is the formalization of the essence of a hierarchical dispatching algorithm, which (as far as we know) has not been formalized before.

Hierarchical overriding is a novel language mechanism that allows method overriding to be applied only to one branch of the class hierarchy. Hierarchical overriding adds expressive power that is not available in languages such as C++ or C#. In particular, it allows overriding to work for classes with multiple (conflicting) methods sharing the same names and signatures. An example is presented in Figure 5.1. In this example, there are four classes/interfaces. Two classes `Deck` and `Drawable` model a deck of cards and a drawable widget, respectively. The class `SafeDeck` adds functionality to check whether the deck is empty so as to prevent drawing a card from an empty deck. The interesting class is `DrawableSafeDeck`, which inherits from both `SafeDeck` and `Drawable`. Hierarchical overriding is used in `DrawableSafeDeck` to keep two separate `draw` methods for each parent, but override *only* the `draw` method coming from `Drawable`, in order to draw a widget with a deck of cards.

Note that hierarchical overriding is denoted in the UML diagram with the notation `draw()` ↑`Drawable`, expressing that the `draw` method from `Drawable` is overridden. Although in this example only one of the `draw` methods is overridden (and the other is simply inherited), hierarchical overriding supports multiple conflicting methods to be independently overridden as well.

To present hierarchical overriding and dispatching, we introduce a formalized model **FHJ** in Section 5.3 based on Featherweight Java [Igarashi et al., 2001], together with theorems and proofs for type soundness. We also have a prototype implementation of an **FHJ** interpreter written in Scala. The implementation validates all the examples presented in this chapter. One nice feature of the implementation is that it can show the detailed step-by-step evaluation of the program, which is convenient for understanding and debugging programs & semantics.

In summary, our contributions are:

- **A formalization of the hierarchical dispatching algorithm** that integrates both the static type and dynamic type for method dispatch, and ensures unambiguity as well as extensibility in the presence of unintentional method conflicts.

- **Hierarchical overriding:** a novel notion that allows methods to override individual branches of the class hierarchy.

- **FHJ:** a formalized model based on Featherweight Java, supporting the above features. We provide the static and dynamic semantics and prove the type soundness of the model.

- **Prototype implementation**[1]**:** a simple implementation of **FHJ** interpreter in Scala. The implementation can type-check and run variants of all the examples shown in this chapter.

5.2 A Running Example: Drawable Deck

This section illustrates the problem of unintentional method conflicts, together with the features of our model for addressing this issue, by a simple running example. In the following text, we will introduce three problems one by one and have a discussion on possible workarounds and our solutions. Problems 1 and 2 are related to hierarchical dispatching, and in C++ it is possible to have similar solutions to both problems. Hence it is important

[1]The implementation is available at `https://github.com/YanlinWang/MIM/tree/master/Calculus`

to emphasize that, with respect to hierarchical dispatching, our model is not a novel mechanism. Instead, inspired by the C++ solutions, our contribution is formalizing a minimal calculus of this feature together with a proof of type soundness. However, for the final problem, there is no satisfactory approach in existing languages, thus what we propose is a novel feature (hierarchical overriding) with the corresponding formalization of that feature.

In the rest of this chapter, we use a Java-like syntax for programs. All types are defined with the keyword **`interface`**; the concept is closely related to Java 8 interfaces with default methods [Bono et al., 2014] and traits. In short, an interface in our model has the following characteristics:

- It allows multiple inheritance.
- Every method is either abstract or implemented with a body (like Java 8 default methods).
- The **`new`** keyword is used to instantiate an interface.
- It cannot have state.

5.2.1 Problem 1: Basic Unintentional Method Conflicts

Suppose that two components `Deck` and `Drawable` have been developed in a system. `Deck` represents a deck of cards and defines a method `draw` for drawing a card from the deck. `Drawable` is an interface for graphics that can be drawn and also includes a method called `draw` for visual display. For simple illustration, the default implementation of the `draw` in `Drawable` only creates a blank canvas on the screen, while the `draw` method in `Deck` simply prints out a message `"Draw a card."`.

```
interface Deck {
  void draw() { // draws a card from the Deck
    println("Draw a card.");
  }
}
interface Drawable {
  void draw() { // create a blank canvas
    JFrame frame = new JFrame();
    frame.setVisible(true);
  }
}
```

In `Deck`, `draw` uses `println`, which is a library function. The two `draw` methods can have different return types, but for simplicity, the return types are both `void` here. Note that, similarly to Featherweight Java [Igarashi et al., 2001], `void` is unsupported in our formalization.

We could have also defined an interface called `Void` and return an object of that type instead. To be concise, however, we use `void` in our examples. In interface `Drawable`, the `draw` method creates a blank canvas.

Now, suppose that a programmer is designing a card game with a GUI. He may want to draw a deck on the screen, so he first defines a drawable deck using multiple inheritance:

```
interface DrawableDeck extends Drawable, Deck {}
```

The point of using multiple inheritance is to compose the features from various components and to achieve code reuse, as supported by many mainstream OO languages. Nevertheless, at this point, languages like Java simply treat the two `draw` methods as the same, hence the compiler fails to compile the program and reports an error.

This case is an example of a so-called *unintentional method conflict*. It arises when two inherited methods happen to have the same name and parameter types, but they are designed for different functionalities with different semantics. Now one may quickly come up with a workaround, which is to manually merge the two methods by creating a new `draw` method in `DrawableDeck` to override the old ones. However, merging two methods with totally different functionalities does not make any sense. This non-solution would hide the old methods and break independent extensibility.

Problem and Possible Workarounds The essential problem is how to resolve unintentional method conflicts and invoke the conflicting methods separately without ambiguity. To tackle this problem, there are several other workarounds that come to our mind. We briefly discuss those potential fixes and workarounds next:

- *I. Delegation.* As an alternative to multiple inheritance, delegation can be used by introducing two fields (or field methods) with the `Drawable` type and `Deck` type, respectively. Although it avoids method conflicts, it is known that using delegation makes it hard to correctly maintain self-references in an extensible system and also introduces a lot of boilerplate code.

- *II. Refactor `Drawable` and/or `Deck` to rename the methods.* If the source code for `Drawable` or `Deck` is available, then it may be possible to rename one of the `draw` methods. However, this approach is non-modular, as it requires modifying existing code and becomes impossible if the code is unavailable.

- *III. Method exclusion/renaming.* Eiffel [Meyer, 1987] and some trait models support method exclusion/renaming. Those features can eliminate conflicts, although most programming languages do not support them. In a traditional OO system, they can break the subtyping relationship. Moreover, in contrast with exclusion, renaming can indeed preserve both conflicting behaviours. However, it is cumbersome in practice, as introducing new names can affect other code blocks.

FHJ's solution To solve this problem, it is important to preserve both conflicting methods during inheritance instead of merging them into a single method. Therefore **FHJ** accepts the definition of `DrawableDeck`. To disambiguate method calls, we can use *upcasts* in **FHJ** to specify the "branch" in the inheritance hierarchy that should be called. The following code illustrates the use of upcasts for disambiguation:

```
interface Deck { void draw() {...} }
interface Drawable { void draw() {...} }
interface DrawableDeck extends Drawable, Deck {}
// main program
((Deck) new DrawableDeck()).draw() // calls Deck.draw
// new DrawableDeck().draw()   // this call is ambiguous and rejected
```

In our language, a program consists of interfaces declarations and a main expression which produces the final result. In the above main expression `((Deck)new DrawableDeck()).draw()`, the cast indicates that we expect to invoke the `draw` method from the branch `Deck`. Similarly, we could have used an upcast to `Drawable` to call the `draw` method from `Drawable`. Without the cast, the call would be ambiguous and **FHJ**'s type system would reject it.

This example illustrates the basic form of fork inheritance, where two unintentionally conflicting methods are accepted by multiple inheritance. Note that C++ supports this feature and also addresses the ambiguity by upcasts. The code for the above example in C++ is similar.

5.2.2 Problem 2: Dynamic Dispatching

Using explicit upcasts for disambiguation helps when making calls to classes with conflicting methods, but things become more complicated with dynamic dispatching. Dynamic dispatching is very common in OO programming for code reuse. Let us expand the previous example a bit, by redefining those interfaces with more features:

```
interface Deck {
  void draw() {...}
  void shuffle() {...}
  void shuffleAndDraw() { this.shuffle(); this.draw(); }
}
```

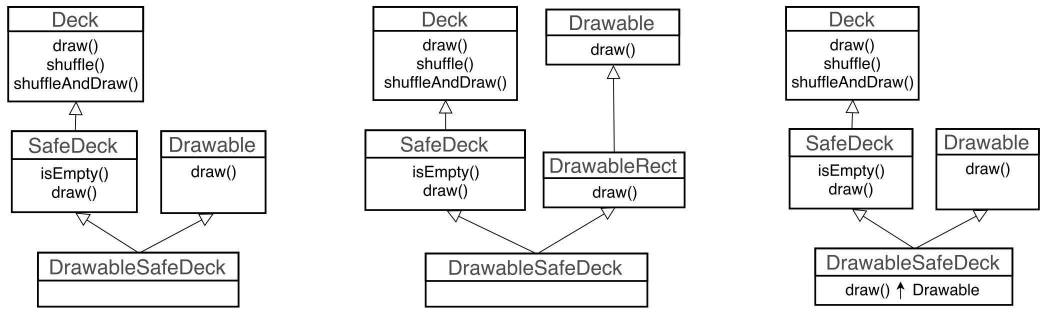


Figure 5.2: UML diagrams for 3 variants of `DrawableSafeDeck`.

Here `shuffleAndDraw` invokes `draw` from its own enclosing type. In **FHJ**, this invocation is dynamically dispatched. This is important, because a programmer may define a subtype of `Deck` and override the method `draw`:

```
interface SafeDeck extends Deck {
  boolean isEmpty() {...}
  void draw() { // overriding
    if (isEmpty()) println("The deck is empty.");
    else println("Draw a card");
  }
}
```

Without dynamic dispatching, we may have to copy the `shuffleAndDraw` code into `SafeDeck`, so that `shuffleAndDraw` calls the new `draw` defined in `SafeDeck`. Dynamic dispatching immediately saves us from the duplication work, since the method becomes automatically dispatched to the most specific one. Nevertheless, as seen before, dynamic dispatch would potentially introduce ambiguity. For instance, when we have the class hierarchy structure shown in Figure 5.2(left) with the following code:

```
interface DrawableSafeDeck extends Drawable, SafeDeck {}
new DrawableSafeDeck().shuffleAndDraw()
```

Indeed, using reduction steps following the reduction rules in FeatherweightJava-like languages, where no static types are tracked, the reduction steps would roughly be:

```
   new DrawableSafeDeck().shuffleAndDraw()
-> new DrawableSafeDeck().shuffle(); new DrawableSafeDeck().draw()
-> ...
-> new DrawableSafeDeck().draw()
-> <<error: ambiguous call!!!>>
```

When the `DrawableSafeDeck` object calls `shuffleAndDraw`, the implementation in `Deck` is dispatched. But then `shuffleAndDraw` invokes "**`this`**`.draw()`", and at this point, the receiver is replaced by the object **`new`** `DrawableSafeDeck()`. From the perspective of `DrawableSafeDeck`, the `draw` method seems to be ambiguous since `DrawableSafeDeck` inherits two `draw` methods from both `SafeDeck` and `Drawable`. But ideally, we would like `shuffleAndDraw` to invoke `SafeDeck.draw` because they belong to the same class hierarchy branch.

FHJ's solution The essential problem is how to ensure that the correct method is invoked. To solve this problem, **FHJ** uses a variant of method dispatching that we call *hierarchical dispatching*. In hierarchical dispatching, both the static and dynamic type information is used to select the right method implementation. During runtime, a method call makes use of both the static type and the dynamic type of the receiver, so it is a combination of static and dynamic dispatching. Intuitively, the static type specifies one branch to avoid ambiguity, and the dynamic type finds the most specific implementation on that branch. To be specific, the following code is accepted by **FHJ**:

```
interface Deck {
  void draw() {...}
  void shuffle() {...}
  void shuffleAndDraw() { this.shuffle(); this.draw(); }
}
interface Drawable {...}
interface SafeDeck extends Deck {...}
interface DrawableSafeDeck extends Drawable, SafeDeck {}
new DrawableSafeDeck().shuffleAndDraw() // SafeDeck.draw is called
```

The computation performed in **FHJ** is as follows:

```
   new DrawableSafeDeck().shuffleAndDraw()
-> ((DrawableSafeDeck) new DrawableSafeDeck()).shuffleAndDraw()
-> ((Deck) new
    DrawableSafeDeck()).shuffle(); ((Deck) new DrawableSafeDeck()).draw()
-> ...
-> ((Deck) new DrawableSafeDeck()).draw()
-> ... // SafeDeck.draw
```

Notably, we track the static types by adding upcasts during reduction. In contrast to FJ, where `new C()` is a value, in **FHJ** such an expression is not a value. Instead, an expression of the form `new C()` is reduced to `(C) new C()`, which is a value in **FHJ** and the cast denotes the static type of the expression. This rule is applied in the first reduction step. In the second reduction step, when `shuffleAndDraw` is dispatched, the receiver `(DrawableSafeDeck)new DrawableSafeDeck()` replaces the special variable `this` by `(Deck)new DrawableSafeDeck()`. Here, the static type used in the cast (`Deck`) denotes the origin of the `shuffleAndDraw` method, which is discovered during method lookup. Later, in the fourth step, `((Deck)new DrawableSafeDeck()).draw()` is an instance of *hierarchical invocation*, which can be read as "finding the most specific `draw` above `DrawableSafeDeck` and along path `Deck`". The meaning of "above `DrawableSafeDeck`" implies its supertypes, and "along path Deck" specifies the branch. Finally, in the last reduction step, we find the most specific version of draw in SafeDeck. In this sequence of reduction steps, the cast that tracks the origin of shuffleAndDraw is crucial to unambiguously find the correct implementation of draw. The formal procedure will be introduced in Section 5.3 and Section 5.4.

5.2.3 Problem 3: Overriding on Individual Branches

Method overriding is common in Object-Oriented Programming. With diamond inheritance, where conflicting methods are intended to have the same semantics, method overriding is not a problem. If conflicting methods arise from multiple parents, we can override all those methods in a single unified (or merged) method in the subclass. Therefore further overriding is simple because there is only one method that can be overridden.

With unintentional method conflicts, however, the situation is more complicated because different, separate, conflicting methods can coexist in one class. Ideally, we would like to support overriding for those methods too, in exactly the same way that overriding is available for other (non-conflicting) methods. However, we need to be able to override the individual conflicting methods, rather than overriding all conflicting methods into a single merged one.

We illustrate the problem and the need for a more refined overriding mechanism with an example. Suppose that the programmer defines a new interface DrawableSafeDeck (based on the code in Section 5.2.2 without the old DrawableSafeDeck), but he needs to override Drawable.draw and give a new implementation of drawing so that the deck can indeed be visualized on the canvas.

Potential solutions/workarounds in existing languages Unfortunately, in all languages we know of (including C++), the existing approaches are unsatisfactory. One direction is to simply avoid this issue, by putting overriding before inheritance. For example, as shown in Figure 5.2(middle), we define a new component DrawableRect that extends Drawable, which simply draws the deck as a rectangle, and modifies the hierarchy:

```
interface DrawableRect extends Drawable {
  void draw() {
    JFrame frame = new JFrame("Canvas");
    frame.setSize(600, 600);
    frame.getContentPane().setBackground(Color.red);
    frame.getContentPane().add(new Square(10,10,100,100)); ...
  }
}
interface DrawableSafeDeck extends DrawableRect, SafeDeck {}
```

This workaround seems to work, but there are severe issues:

- It changes the hierarchy and existing code, hence breaks the modularity.
- Separate overriding is required to come after the fork inheritance, especially when the implementation needs functionality from both parents. In the above code, we have assumed that the overriding is unrelated to `Deck`. But when the drawing relies on some information of the `Deck` object, we have to either introduce field methods for delegation or change the signature of `draw` to take a parameter. Either way introduces unnecessary complexity and affects extensibility.

There are more involved workarounds in C++ using templates and complex patterns, but such patterns are complex to use and there are still issues. A more detailed discussion of such an approach is presented in Section 7.4.

FHJ's solution An additional feature of our model is *hierarchical overriding*. It allows conflicting methods to be overridden on individual branches, hence offers independent extensibility. The above example can be easily realized by:

```
interface DrawableSafeDeck extends Drawable, SafeDeck {
  void draw() override Drawable {
    JFrame frame = new JFrame("Canvas");
    frame.setSize(600, 600);
    frame.getContentPane().setBackground(Color.red);
    frame.getContentPane().add(new Square(10,10,100,100)); ...
  }
}
((Drawable)new DrawableSafeDeck()).draw(); //calls the draw in
    DrawableSafeDeck
```

The UML graph is shown in Figure 5.2(right), where the up-arrow $\uparrow$ is short for `override`. Here the idea is that *only* `Drawable.draw` is overridden. This is accomplished by specifying, in the method definition, that the method only overrides the `draw` from `Drawable`. The individual overriding allows us to make use of the methods from `SafeDeck` as well. In the formalization, the hierarchical overriding feature is an important feature, involved in the algorithm of hierarchical dispatch.

Note that, although the example here only shows one conflicting method being overridden, hierarchical overriding allows (as expected) multiple conflicting methods to be overridden in the same class.

Terminology In `Drawable`, `Deck`, and `SafeDeck`, the `draw` methods are called *original methods* here, because they are originally defined by the interfaces. In contrast, `DrawableSafeDeck` defines a *hierarchical overriding method*. The difference is that traditional method overriding overrides all branches by defining another original method, whereas hierarchical overriding only refines one branch.

A special rule for hierarchical overriding is: it can only refine *original* methods, and cannot jump over original methods with the same signature. For instance, writing "`void draw() override Deck {...}`" is disallowed in `DrawableSafeDeck`, because the existing two branches are `Drawable.draw` and `SafeDeck.draw`, while `Deck.draw` is already covered. It does not really make sense to refine the old branch `Deck`.

A peek at the hierarchical dispatching algorithm In **FHJ**, fork inheritance allows several original methods (branches) to coexist, and hierarchical dispatch first finds the most specific original method (branch), then it finds the most specific hierarchical overriding on that branch.

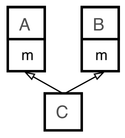
Before the formalized algorithm, Figure 5.3 gives a peek at the behavior using a few examples. The UML diagrams present the hierarchy. In (d) and (e), a cross mark indicates that the interface fails to type-check. Generally, **FHJ** rejects the definition of an interface during compilation if it reaches a diamond with ambiguity. `mbody` is the method lookup function for hierarchical dispatch, formally defined in Section 5.4.1. In general, $\texttt{mbody}(m, X, Y) = (Z, ...)$ reflects that the source code `((Y) new X()).m()` calls `Z.m` at runtime. It is undefined when method dispatch is ambiguous.

In Figure 5.3, (a) is the base case for unintentional conflicts, namely the fork inheritance. (b) uses overriding to merge the conflicting methods explicitly. (c) represents hierarchical overriding.

Furthermore, our model supports diamond inheritance and can deal with diamond problems. For example, (d) and (e) are two base cases of diamond inheritance in **FHJ** and the definition of each C is rejected because T is an ambiguous parent to C. One solution for diamond inheritance is to merge methods coming from different parents. (f) gives a common solution to the diamond as in Java or traits, which is to explicitly override $A.m$ and $B.m$ in C. Our calculus supports this kind of merging methods. In the last three examples, conflicting methods $A.m$ and $B.m$ should be viewed as intentional conflicts, as they come from the same source T.

5.3 Formalization

In this section, we present a formal model called **FHJ** (*Featherweight Hierarchical Java*), following a similar style to Featherweight Java [Igarashi et al., 2001]. **FHJ** is a minimal core

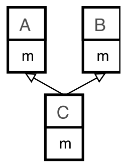


(a)

$\mathtt{mbody}(m, C, A) = (A, \ldots)$

$\mathtt{mbody}(m, C, B) = (B, \ldots)$

$\mathtt{mbody}(m, C, C) = \mathtt{undefined}$

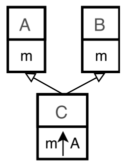


(b)

$\mathtt{mbody}(m, C, A) = (C, \ldots)$

$\mathtt{mbody}(m, C, B) = (C, \ldots)$

$\mathtt{mbody}(m, C, C) = (C, \ldots)$

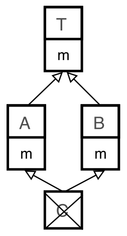


(c)

$\mathtt{mbody}(m, C, A) = (C, \ldots)$

$\mathtt{mbody}(m, C, B) = (B, \ldots)$

$\mathtt{mbody}(m, C, C) = \mathtt{undefined}$

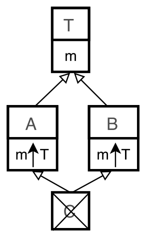


(d)

$\mathtt{mbody}(m, T, T) = (T, \ldots)$

$\mathtt{mbody}(m, C, T) = \mathtt{undefined}$

C rejected by type-check

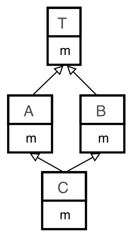


(e)

$\mathtt{mbody}(m, T, T) = (T, \ldots)$

$\mathtt{mbody}(m, C, T) = \mathtt{undefined}$

C rejected by type-check



(f)

$\mathtt{mbody}(m, C, T) = (C, \ldots)$

$\mathtt{mbody}(m, C, A) = (C, \ldots)$

$\mathtt{mbody}(m, C, B) = (C, \ldots)$

$\mathtt{mbody}(m, C, C) = (C, \ldots)$

Figure 5.3: Examples in **FHJ**. "$m \uparrow A$" stands for hierarchical overriding "m `override` A".

calculus that formalizes the core concept of hierarchical dispatching and overriding. The syntax, typing rules and small-step semantics are presented.

5.3.1 Syntax

The abstract syntax of **FHJ** interface declarations, method declarations, and expressions is given in Figure 5.4. The multiple inheritance feature of **FHJ** is inspired by Java 8 interfaces, which supports method implementations via default methods. This feature is closely related to *traits*. To demonstrate how unintentional method conflicts are untangled in **FHJ**, we only focus on a small subset of the interface model. For example, all methods declared in an interface are either default methods or abstract methods. Default methods provide default implementations for methods. Abstract methods do not have a method body. Abstract methods can be overridden with future implementations.

Notations The metavariables I, J, K range over interface names; x ranges over variables; m ranges over method names; e ranges over expressions; and M ranges over method declarations. Following Featherweight Java, we assume that the set of variables includes the special variable `this`, which cannot be used as the name of an argument to a method. We use the same conventions as FJ; we write $\overline{I}$ as shorthand for a possibly empty sequence $I_1, ..., I_n$, which may be indexed by I_i; and write $\overline{M}$ as shorthand for $M_1...M_n$ (with no commas). We also abbreviate operations on pairs of sequences in an obvious way, writing $\overline{I}\ \ \overline{x}$ for $I_1\ x_1, ..., I_n\ x_n$, where n is the length of $\overline{I}$ and $\overline{x}$.

Interfaces In order to achieve multiple inheritance, an interface can have a set of parent interfaces, where such a set can be empty. Moreover, as usual in class-based languages, the extension relation over interfaces is acyclic. The interface declaration $\texttt{interface I extends}\ \overline{I}\ \{\overline{M}\}$ introduces an interface named I with parent interfaces $\overline{I}$ and a suite of methods $\overline{M}$. The methods of I may either override methods that are already defined in $\overline{I}$ or add new functionality special to I, we will illustrate this in more detail later.

Methods Original methods and hierarchically overriding methods share the same syntax in our model for simplicity. The concrete method declaration $I\ m(\overline{I_x}\ \ \overline{x})\ \texttt{override}\ J\ \{\texttt{return}\ e;\}$ introduces a method named m with result type I, parameters $\overline{x}$ of type $\overline{I_x}$ and the overriding target J. The body of the method simply includes the returned expression e. Notably, we have introduced the `override` keyword for two cases. Firstly, if the overridden interface is exactly the enclosing interface itself, then such a method is seen as *originally defined*. Note that the case of merging methods from different branches is also regarded as originally defined. Secondly, for all other cases, the method is considered a *hierarchical overriding method*. Note that in an interface J, $I\ m(\overline{I_x}\ \overline{x})\ \{\texttt{return}\ e;\}$ is syntactic sugar for $I\ m(\overline{I_x}\ \overline{x})\ \texttt{override}\ J\ \{\texttt{return}\ e;\}$, which is the standard way to define methods in Java-like languages. The definition of abstract methods is written as $I\ m(\overline{I_x}\ \overline{x})\ \texttt{override}\ J\ ;$, which is similar to a concrete method but

Interfaces	IL	::=	$\texttt{interface}\ I\ \texttt{extends}\ \overline{I}\ \{\overline{M}\}$
Methods	M	::=	$I\ m(\overline{I_x\ x})\ \texttt{override}\ J\ \{\texttt{return}\ e;\} \mid I\ m(\overline{I_x\ x})\ \texttt{override}\ J\ ;$
Expressions	e	::=	$x \mid e.m(\overline{e}) \mid \texttt{new}\ I() \mid\ (I)e$
Context	Γ	::=	$\overline{x} : \overline{I}$
Values	v	::=	$(I)\texttt{new}\ J()$

Figure 5.4: Syntax of **FHJ**.

without the method body. For simplicity, overloading is not modelled for methods, which implies that we can uniquely identify a method by its name.

Expressions & Values Expressions can be standard constructs such as variables, method invocation, object creation, together with cast expressions. Object creation is represented by `new I()`[2]. Fields and primitive types are not modelled in **FHJ**. The casts are merely safe upcasts, and in fact, they can be viewed as annotated expressions, where the annotation indicates its static type. The coexistence of static and dynamic types is the key to hierarchical dispatch. A value "(I)new J()" is the final result of multiple reduction steps for evaluating an expression.

For simplicity, **FHJ** does not formalize statements like assignments and so on because they are orthogonal features to the hierarchical dispatching and overriding feature. A program in **FHJ** consists of a list of interface declarations, plus a single expression.

5.3.2 Subtyping and Typing Rules

Subtyping The subtyping of **FHJ** consists of only a few rules shown at the top of Figure 5.5. In short, subtyping relations are built from the inheritance in interface declarations. Subtyping is both reflexive and transitive.

Type-checking Details of type-checking rules are displayed at the bottom of Figure 5.5, including expression typing, well-formedness of methods and interfaces. As a convention, an environment Γ is maintained to store the types of variables, together with the self-reference `this`.

(T-Invk) is the typing rule for method invocation. Naturally, the receiver and the arguments are required to be well-typed. `mbody` is our key function for method lookup that implements the hierarchical dispatching algorithm. The formal definition will be introduced in

[2]In Java the corresponding syntax is `new I(){}`.

$$\boxed{I <: J} \qquad I <: I$$

$$\frac{I <: J \qquad J <: K}{I <: K} \qquad \frac{\texttt{class}\ I\ \texttt{extends}\ I_1, I_2, \ldots, I_n\ \{\ldots\}}{I <: I_1, I <: I_2, \ldots, I <: I_n}$$

$$\boxed{\Gamma \vdash e : I} \qquad (\textsc{T-Var})\ \Gamma \vdash x : \Gamma(x)$$

$$(\textsc{T-Invk})\ \frac{\Gamma \vdash e_0 : I_0 \qquad \texttt{mbody}(m, I_0, I_0) = (K, \overline{J}\ \overline{x}, I\ \_) \qquad \Gamma \vdash \overline{e} : \overline{I} \qquad \overline{I} <: \overline{J}}{\Gamma \vdash e_0.m(\overline{e}) : I}$$

$$(\textsc{T-New})\ \frac{\texttt{interface}\ I\ \texttt{extends}\ \overline{I}\ \{\overline{M}\} \qquad \texttt{canInstantiate}(I)}{\Gamma \vdash \texttt{new}\ I() : I}$$

$$(\textsc{T-Anno})\ \frac{\Gamma \vdash e : I \qquad I <: J}{\Gamma \vdash (J)e : J}$$

$$(\textsc{T-Method})\ \frac{\begin{array}{c} I <: J \qquad \texttt{findOrigin}(m, I, J) = \{J\} \\ \texttt{mbody}(m, J, J) = (K, \overline{I_x}\ \overline{x}, I_e\ \_) \qquad \overline{x} : \overline{I_x}, \texttt{this} : I \vdash e_0 : I_0 \qquad I_0 <: I_e \end{array}}{I_e\ m(\overline{I_x}\ \overline{x})\ \texttt{override}\ J\ \{\texttt{return}\ e_0;\}\ \text{OK IN}\ I}$$

$$(\textsc{T-AbsMethod})\ \frac{\begin{array}{c} I <: J \quad \texttt{findOrigin}(m, I, J) = \{J\} \\ \texttt{mbody}(m, J, J) = (K, \overline{I_x}\ \overline{x}, I_e\ \_) \end{array}}{I_e\ m(\overline{I_x}\ \overline{x})\ \texttt{override}\ J\ ;\ \text{OK IN}\ I}$$

$$(\textsc{T-Intf})\ \frac{\begin{array}{c} \overline{M}\ \text{OK IN}\ I \\ \forall J >: I \text{ and } m, \texttt{mbody}(m, J, J) \text{ is defined} \Rightarrow \texttt{mbody}(m, I, J) \text{ is defined} \\ \forall J >: I \text{ and } m, I[m\ \texttt{override}\ I] \text{ and } J[m\ \texttt{override}\ J] \text{ defined} \Rightarrow \texttt{canOverride}(m, I, J) \end{array}}{\texttt{interface}\ I\ \texttt{extends}\ \overline{I}\ \{\overline{M}\}\ \text{OK}}$$

Figure 5.5: Subtyping and typing rules of **FHJ**

Section 5.4. Here $\texttt{mbody}(m, I_0, I_0)$ finds the most specific m above I_0. “Above I_0” specifies the search space, namely the supertypes of I_0 including itself. For the general case, however, the hierarchical invocation $\texttt{mbody}(m, I, J)$ finds “the most specific m above I and along the path/branch J”. “Along path J” additionally requires the result to relate to J, that is to say, the most specific interface that has a subtyping relationship with J.

In (T-Invk), as the compilation should not be aware of the dynamic type, it only requires that invoking m is valid for the static type of the receiver. The result of mbody contains the interface that provides the most specific implementation, the parameters and the return type. We use underscore for the return expression, matching both implemented and abstract methods.

(T-New) is the typing rule for object creation `new I()`. The auxiliary function `canInstantiate`(I) (see definition in Section 5.4.4) checks whether an interface I can be instantiated or not. Since fork inheritance accepts conflicting branches to coexist, the check requires that the most specific method is concrete for each method on each branch.

(T-Method) is more interesting since a method can either be an original method or a hierarchical overriding, though they share the same syntax and method typing rule. `findOrigin`(m, I, J) is a fundamental function, used to find "the most specific interfaces that are above I and along the path J, and originally defines m" (see Section 5.4 for full definition). By "most specific interfaces", it implies that the inherited supertypes are excluded. Thus the condition `findOrigin`(m, I, J) $= \{J\}$ indicates a characteristic of a hierarchical overriding: it must override an original method; the overriding is direct and there does not exist any other original method m in between. Then `mbody`(m, J, J) provides the type of the original method, so hierarchical overriding has to preserve the type. Finally, the return expression is type-checked to be a subtype of the declared return type. For the definition of an original method, I equals J and the rule is straightforward. (T-AbsMethod) is a similar rule but works on abstract method declarations.

(T-Intf) defines the typing rule on interfaces. The first condition is obvious, namely, its methods need to be well checked. The third condition checks whether the overriding between original methods preserves typing. In this condition, we again use some helper functions defined in Section 5.4. I[m `override` I] is defined if I originally defines m, and `canOverride`(m, I, J) checks whether I.m has the same type as J.m. Generally the preservation of method type is required for any supertype J and any method m.

The second condition of (T-Intf) is more complex and is the key to type soundness. Unlike C++ which rejects on ambiguous calls, **FHJ** rejects on the definition of interfaces when they form a diamond. Consider the case when the second condition is broken: `mbody`(m, J, J) is defined but `mbody`(m, I, J) is undefined for some J and m. This indicates that m is available and unambiguous from the perspective of J, but is ambiguous to I on branch J. It means that there are multiple overriding paths of m from J to I, which form a diamond. Hence rejecting that case meets our expectation. Below is an example (Figure 5.3 (e)) that illustrates the reason why this condition is needed:

```
interface T             { T m() override T { return new T(); } }
interface A extends T   { T m() override T { return new A(); } }
interface B extends T   { T m() override T { return new B(); } }
interface C extends A, B {}
((T) new C()).m()
```

This program does not compile on interface C, because of the second condition in (T-Intf), where I equals C and J equals T. By the algorithm, `mbody`(m, T, T) will refer to T.m, but `mbody`$(\mathrm{m}, \mathrm{C}, \mathrm{T})$ is undefined since both $\mathrm{A.m}$ and $\mathrm{B.m}$ are most specific to C along the path T, which forms a diamond. The expression `((T) new C()).m()` is one example of triggering ambiguity, but **FHJ** simply rejects the definition of C. To resolve the issue, the programmer needs to have an overriding method in C, to explicitly merge the conflicting ones.

Finally, rule (T-Anno) is the typing rule for a cast expression. By the rule, only upcasts are valid.

5.3.3 Small-step Semantics and Propagation

Figure 5.6 defines the small-step semantics and propagation rules of **FHJ**. When evaluating an expression, they are invoked and produce a single value in the end.

Semantic Rules (S-Invk) is the only computation rule we need for method invocation. As a small-step rule and by congruence, it assumes that the receiver and the arguments are already values. Specifically, the receiver (J)`new` $\mathrm{I}()$ indicates the dynamic type I together with the static type J. Therefore `mbody`$(\mathrm{m}, \mathrm{I}, \mathrm{J})$ carries out hierarchical dispatching, acquires the types, the return expression e_0 and the interface I_0 which provides the most specific method. Here we use e_0 to imply that the return expression is forced to be non-empty because it requires a concrete implementation. Now the rule reduces method invocation to e_0 with substitution. Parameters are substituted with arguments, and the `this` reference is substituted with the receiver, and in the meanwhile, the static types are recorded via annotations. Finally, the return type I_e is put in the front as an annotation.

Propagation Rules (C-Receiver), (C-Args) and (C-FReduce) are natural propagation rules on receivers, arguments, and cast-expressions, respectively. (C-StaticType) automatically adds an annotation I to the new object `new I()`. (C-AnnoReduce) merges nested upcasts into a single upcast with the outermost type.

5.4 Key Algorithms and Type-Soundness

In this section, we present the fundamental algorithms and auxiliary definitions used in our formalization and show that the resulting calculus is type sound. The functions presented in this section are the key components that implement our algorithm for method lookup.

5.4.1 The Method Lookup Algorithm in `mbody`

`mbody`$(\mathrm{m}, \mathrm{I}_d, \mathrm{I}_s)$ denotes the method body lookup function. We use $\mathrm{I}_d, \mathrm{I}_s$, since `mbody` is usually invoked by a receiver of a method m, with its dynamic type I_d and static type I_s.

Such a function returns the most specific method implementation. More accurately, `mbody` returns $(J, \overline{I_x\ x}, I_e\ e_0)$ where J is the found interface that contains the desired method; $\overline{I_x\ x}$ are the parameters and its types, e_0 is the returned expression (empty for abstract methods). It considers both originally-defined methods and hierarchical overriding methods, so `findOrigin` and `findOverride` (see the definition in Section 5.4.2 and Section 5.4.3) are both invoked. The formal definition gives the expected results for the earlier examples in Figure 5.3.

▷ *Definition of* $\texttt{mbody}(m, I_d, I_s)$:

- $\texttt{mbody}(m, I_d, I_s) = (J, \overline{I_x\ x}, I_e\ e_0)$

  with: $\texttt{findOrigin}(m, I_d, I_s) = \{I\}$

  $\texttt{findOverride}(m, I_d, I) = \{J\}$

  $J[m\ \texttt{override}\ I] = I_e\ m(\overline{I_x\ x})\ \texttt{override}\ I\ \{\texttt{return}\ e_0;\}$

- $\texttt{mbody}(m, I_d, I_s) = (J, \overline{I_x\ x}, I_e\ )$

  with: $\texttt{findOrigin}(m, I_d, I_s) = \{I\}$

  $\texttt{findOverride}(m, I_d, I) = \{J\}$

  $J[m\ \texttt{override}\ I] = I_e\ m(\overline{I_x\ x})\ \texttt{override}\ I\ ;$

To calculate $\texttt{mbody}(m, I_d, I_s)$, the invocation of `findOrigin` looks for the most specific original methods and their interfaces, and expects a singleton set, so as to achieve unambiguity. Furthermore, the invocation of `findOverride` also expects a unique and most specific hierarchical override. And finally, the target method is returned.

5.4.2 Finding the Most Specific Origin: `findOrigin`

We proceed to give the definitions of two core functions that support method lookup, namely, `findOrigin` and `findOverride`. Generally, $\texttt{findOrigin}(m, I, J)$ finds the set of most specific interfaces where m is originally defined. Interfaces in this set should be above interface I and along path J. Finally with `prune` (defined in Section 5.4.4) the overridden interfaces will be filtered out.

▷ *Definition of* $\texttt{findOrigin}(m, I, J)$: (5.1)

- $\texttt{findOrigin}(m, I, J) = \texttt{prune}(\texttt{origins})$ (5.2)

  with: $\texttt{origins} = \{K \mid I <: K,\ \text{and}\ K <: J\ \vee\ J <: K,$ (5.3)

  $\text{and}\ K[m\ \texttt{override}\ K]\ \text{is defined}\}$ (5.4)

$$(\text{S-Invk})\ \frac{\texttt{mbody}(m, I, J) = (I_0, \overline{I_x}\ \overline{x}, I_e\ e_0)}{((J)\texttt{new I()}).m(\overline{v}) \rightarrow (I_e)[\overline{(I_x)v}/\overline{x}, (I_0)\texttt{new I()}/\texttt{this}]e_0}$$

$$(\text{C-Receiver})\ \frac{e_0 \rightarrow e_0'}{e_0.m(\overline{e}) \rightarrow e_0'.m(\overline{e})} \qquad (\text{C-Args})\ \frac{e \rightarrow e'}{e_0.m(\ldots, e, \ldots) \rightarrow e_0.m(\ldots, e', \ldots)}$$

$$(\text{C-StaticType})\ \texttt{new I()} \rightarrow\ (I)\texttt{new I()}$$

$$(\text{C-FReduce})\ \frac{e \rightarrow e' \qquad e \neq \texttt{new J()}}{(I)e \rightarrow\ (I)e'}$$

$$(\text{C-AnnoReduce})\ (I)((J)\texttt{new K()}) \rightarrow (I)\texttt{new K()}$$

Figure 5.6: Small-step semantics.

By definition, an interface belongs to `findOrigin`(m, I, J) if and only if:

- It originally defines m;
- It is a supertype of I (including I);
- It is either a supertype or a subtype of J (including J);
- No subtype of it belongs to the same result set because of `prune`.

5.4.3 Finding the Most Specific Overriding: `findOverride`

The `findOrigin` function only focuses on original method implementations, where all the hierarchical overriding methods are omitted during that step. On the other hand, `findOverride`(m, I, J) has the assumption that J defines an original m, and this function tries to find the interfaces with the most specific implementations that hierarchically overrides such an m. Formally,

▷ *Definition of* `findOverride`(m, I, J) :

• `findOverride`$(m, I, J) = \texttt{prune}(overrides)$

with: $overrides = \{K \mid I <: K,\ K <: J$ and $K[m\ \texttt{override}\ J]$ is defined

By definition, an interface belongs to `findOverride`(m, I, J) if and only if:

- it is between I and J (including I, J);

- it hierarchically overrides $\mathtt{J.m}$;

- any subtype of it does not belong to the same set.

5.4.4 Other Auxiliaries

Below we give other minor definitions of the auxiliary functions that are used in previous sections.

▷ *Definition of* $I[m\ \texttt{override}\ J]$:

- $I[m\ \texttt{override}\ J] = I_e\ m(\overline{I_x\ x})\ \texttt{override}\ J\ \{\texttt{return}\ e_0;\}$

  with: $\texttt{interface}\ I\ \texttt{extends}\ \overline{I}\ \{I_e\ m(\overline{I_x\ x})\ \texttt{override}\ J\ \{\texttt{return}\ e_0;\}\ldots\}$
- $I[m\ \texttt{override}\ J] = I_e\ m(\overline{I_x\ x})\ \texttt{override}\ J\ ;$

  with: $\texttt{interface}\ I\ \texttt{extends}\ \overline{I}\ \{I_e\ m(\overline{I_x\ x})\ \texttt{override}\ J\ ;\ldots\}$

Here $I[m\ \texttt{override}\ J]$ is basically a direct lookup for method m in the body of I, where such a method overrides J (like static dispatch). The method can be either concrete or abstract, and the body of definition is returned. Notice that by our syntax, $I[m\ \texttt{override}\ I]$ is looking for the originally-defined method m in I.

▷ *Definition of* $\texttt{prune}(set)$:

- $\texttt{prune}(set) = \{I \in set \mid \nexists J \in set \setminus I, J <: I\}$

The prune function takes a set of types, and filters out those that have subtypes in the same set. In the returned set, none of them has subtyping relation to one another, since all supertypes have been removed.

▷ *Definition of* $\texttt{canOverride}(m, I, J)$:

- $\texttt{canOverride}(m, I, J)$ holds

  iff: $I[m\ \texttt{override}\ I] = I_e\ m(\overline{I_x\ x})\ \texttt{override}\ I\ldots$

  $J[m\ \texttt{override}\ J] = I_e\ m(\overline{I_x\ y})\ \texttt{override}\ J\ldots$

$\texttt{canOverride}$ just checks that two original m in I and J have the same type.

▷ *Definition of* $\texttt{canInstantiate}(\mathrm{I})$:

- $\texttt{canInstantiate}(\mathrm{I})$ holds

  iff: $\forall m, \forall J \in \texttt{findOrigin}(m, I, I), \texttt{findOverride}(m, I, J) = \{K\}$,

  and $K[m\ \texttt{override}\ J] = I_e\ m(\overline{I_x\ x})\ \texttt{override}\ J\ \{\texttt{return}\ e_0;\}$

$\texttt{canInstantiate}(\mathrm{I})$ checks whether interface I can be instantiated by the keyword `new`. $\texttt{findOrigin}(m, I, I)$ represents the set of branches that I inherits on method m. I can be instantiated if and only if, for every branch, the most specific implementation is non-abstract.

5.4.5 Properties

We present the type soundness of the model by a few theorems below, following the standard technique of subject reduction and progress proposed by Wright and Felleisen [Wright and Felleisen, 1994]. The proof, together with some lemmas, is presented in the Appendix. Type soundness states that if an expression is well-typed, then after many reduction steps it must reduce to a value, and its annotation is the same as the static type of the original expression.

Theorem 1 (Subject Reduction). *If* $\Gamma \vdash e : I$ *and* $e \to e'$*, then* $\Gamma \vdash e' : I$.

Proof. See Appendix A.6. □

Theorem 2 (Progress). *Suppose* e *is a well-typed expression, if* e *includes* $((J)\texttt{new}\ I()).m(\overline{v})$ *as a sub-expression, then* $\textit{mbody}(m, I, J) = (I_0, \overline{I_x\ x}, I_e\ e_0)$ *and* $\#(\overline{x}) = \#(\overline{v})$ *for some* I_0, $\overline{I_x}$, $\overline{x}$, I_e *and* e_0.

Proof. See Appendix A.6. □

Theorem 3 (Type Soundness). *If* $\vdash e : I$ *and* $e \to^* e'$ *with* e' *a normal form, then* e' *is a value* v *with* $\vdash v : I$.

Proof. Immediate from Theorem 1 and Theorem 2. □

Note that in Theorem 2, "$\#(\overline{x})$" denotes the length of $\overline{x}$.

Our theorems are stricter than those of Featherweight Java [Igarashi et al., 2001]. In FJ, the subject reduction theorem states that after a step of reduction, the type of an expression may change to a subtype due to subtyping. However, in **FHJ**, the type remains unchanged because we keep track of the static types and use them for casting during reduction.

Finally we show that one-step evaluation is deterministic. This theorem is helpful to show that our model of multiple inheritance is not ambiguous (or non-deterministic).

Theorem 4 (Determinacy of One-Step Evaluation). *If* $\mathtt{t} \to \mathtt{t}'$ *and* $\mathtt{t} \to \mathtt{t}''$, *then* $\mathtt{t}' = \mathtt{t}''$.

Proof. See Appendix A.6. □

5.5 Discussion

In this section, we will discuss the design space and reflect on some of the design decisions of our work. We relate our language to traits, Java interfaces as well as other languages. Furthermore, we discuss ways to improve our work.

5.5.1 Abstract Methods

Abstract methods are one of the key features in most general OO languages. For example, Java interfaces (prior to Java 8) were designed to include only method declarations, and those abstract methods can be implemented in a class body. The formal Featherweight Java model [Igarashi et al., 2001] does not include abstract methods because of the orthogonality to the core calculus. In traits, the similar idea is to use keywords like "**require**" for abstract method declarations [Schärli et al., 2003]. Abstract methods provide a way to delay the implementations to future subtypes. Using overriding, they also help to "exclude" existing implementations.

In our formalized calculus, however, abstract methods are not a completely orthogonal feature. The `canInstantiate` function has to check whether an interface can be instantiated by looking at all the inherited branches and checking if each most specific method is concrete or not.

Our formalization has a simple form of abstract methods, which behave similarly to conventional methods with respect to conflicts. Other languages may behave differently. For instance, in Java 8 when putting two identical abstract methods together by multiple inheritance, there is no conflict error. In Figure 5.7, we use italic *m* to denote abstract methods. In both cases, the Java compiler accepts the definition of C and automatically merges the two inherited methods $\mathtt{m}$ into a single one. **FHJ** behaves differently from Java in both cases. In the fork inheritance case (left), C will have two distinct abstract methods corresponding to $\mathtt{A.m}$ and $\mathtt{B.m}$. In the diamond inheritance case, the definition of C is rejected. There are two reasons for this difference in behaviour. Firstly, our formalization just treats abstract methods as concrete methods with an empty body, and that simplifies the rules and proofs

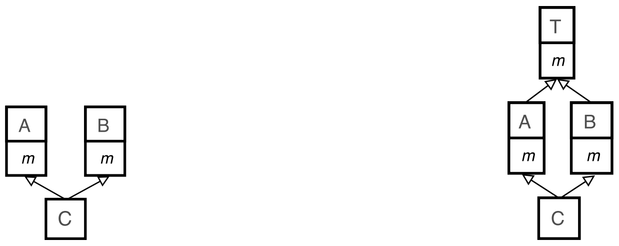


Figure 5.7: Fork inheritance (left) and diamond inheritance (right) on abstract methods.

a lot. Secondly, and more importantly, we distinguish and treat differently conflicting methods, since they may represent different operations, even if they are abstract. Thus our model adopts a very conservative behavior rather than automatically merging methods by default (as done in many languages). Arguably, the diamond case it is actually an intentional conflict due to the same source T. Therefore our model conservatively rejects this case. It is possible to change our model to account for other behaviors for abstract methods, but we view this as a mostly orthogonal change to our work, and should not affect the essence of the model presented here.

5.5.2 Orthogonal & Non-Orthogonal Extensions

Our model is designed as a minimal calculus that focuses on resolving unintentional conflicts. Therefore, we have omitted a number of common orthogonal features including primitive types, assignments, method overloading, covariant method return types, static dispatch, and so on. Those features can, in principle, be modularly added to the model without breaking type soundness. For example, we present the additional syntax, typing and semantic rules of static invocation below as an extension:

$$\text{Expressions} \quad e \quad ::= \quad \ldots \mid e.J_0@J_1 :: m(\overline{e})$$

$$\text{(T-StaticInvk)} \; \frac{\begin{array}{c} J_0[m\ \mathtt{override}\ J_1] = I\ m(\overline{J}\ \overline{x})\ \mathtt{override}\ J_1\ \{\mathtt{return}\ e;\} \\ \Gamma \vdash e_0 : I_0 \qquad I_0 <: J_0 \qquad \Gamma \vdash \overline{e} : \overline{I} \qquad \overline{I} <: \overline{J} \end{array}}{\Gamma \vdash e_0.J_0@J_1 :: m(\overline{e}) : I}$$

$$\text{(S-StaticInvk)} \; \frac{J_0[m\ \mathtt{override}\ J_1] = I_e\ m(\overline{I_x}\ \overline{x})\ \mathtt{override}\ J_1\ \{\mathtt{return}\ e_0;\}}{((J)\mathtt{new}\ I()).J_0@J_1 :: m(\overline{v}) \rightarrow (I_e)[\overline{(I_x)v}/\overline{x}, (J_0)\mathtt{new}\ I()/\mathtt{this}]e_0}$$

A static invocation $\mathtt{e.J_0@J_1 :: m(\overline{e})}$ aims at finding the method $\mathtt{m}$ in $\mathtt{J_0}$ that hierarchically overrides $\mathtt{J_1}$. Thus $\mathtt{J_0[m\ override\ J_1]}$ is invoked. As shown in (S-StaticInvk), static dispatch needs a receiver for the substitution of the "`this`" reference, so as to provide the latest implementations. In fact, static dispatch is common in OO programming, as it provides a shortcut to the reuse of old implementation easily, and super calls can also rely on this feature. For convenience, we just make it simple above, whereas in languages like C++ or Java, the static or super invocations are more flexible, as they can climb the class hierarchy.

One non-orthogonal extension to **FHJ** could be to generalize the model to allow multiple hierarchical method overriding, meaning that, we allow overriding methods to update multiple branches instead of only one branch. This feature offers a more fine-grained mechanism for merging and can be helpful to easily understand the structure of the hierarchy. Multiple overriding would be useful in the following situation, for example:

```
interface A { void m() {...} }
interface B { void m() {...} }
interface C { void m() {...} }
interface D extends A, B, C {
   void m() override A,B {...} // overrides branches A and B only
   void m() override C {...}  // overrides branch C
}
```

Here `D` inherits from three interfaces `A, B, C` with conflicting methods `m`, but only merges two of those methods. While we can simulate D without multiple overriding in our calculus (by introducing an intermediate class), a better approach would be to support multiple overriding natively.

We present the modification of syntax, typing and semantic rules below (abstract methods omitted):

$$\text{Methods}\quad M \quad ::= \quad \dots \mid I\ \mathtt{m}(\overline{I_x}\ \overline{x})\ \mathtt{override}\ \overline{J}\ \{\mathtt{return}\ e;\}$$

$$\text{(T-MoMethod)}\ \frac{\begin{array}{c}\forall J_i \in \overline{J}, I <: J_i \\ \mathtt{findOrigin}(\mathtt{m}, I, J_i) = \{J_i\} \qquad \mathtt{mbody}(\mathtt{m}, J_i, J_i) = (K, \overline{I_x}\ \overline{x}, I_e\ \_) \\ \overline{x} : \overline{I_x}, \mathtt{this} : I \vdash e_0 : I_0 \qquad I_0 <: I_e\end{array}}{I_e\ \mathtt{m}(\overline{I_x}\ \overline{x})\ \mathtt{override}\ \overline{J}\ \{\mathtt{return}\ e_0;\}\ \text{OK IN}\ I}$$

Semantic rules themselves remain unchanged, however, we need to change slightly the definition of `findOverride` in `mbody`:

▷ *Definition of* $\texttt{findOverride}(m, I, J)$:

- $\texttt{findOverride}(m, I, J) = \texttt{prune}(\texttt{overrides})$

  with: $\texttt{overrides} = \{K \mid I <: K,\ K <: J \text{ and } K[m\ \texttt{override}\ \bar{J}] \text{ where } J \in \bar{J}$

With this approach, branches A and B are merged in the sense that they share the same code, which can be separately updated in future interfaces. Another approach would be to deeply merge the branches, with similar effect as introducing an intermediate interface AB to explicitly merge the two branches. However, this approach is problematic because there is no clear mechanism for identifying and further updating the merged branches. This could be an interesting future work to explore.

Other typical non-orthogonal extensions to **FHJ** could be to have fields. The design of **FHJ** can be viewed as a variant of Java 8 with default methods which allows for unintentional method conflicts. Like Java interfaces and traits, state is forbidden in **FHJ**. There are some inheritance models that also account for fields, such as C++ that uses virtual inheritance [Ellis and Stroustrup, 1990]. In our model, however, we can perhaps borrow the idea of *interface-based programming* [Wang et al., 2016], which models state with abstract state operations. This can be realized by extending our current model with static methods and anonymous classes from Java. We will discuss the extension with fields in the next chapter.

5.5.3 Loosening the Model: Reject Early or Reject Later?

FHJ rejects the following case of diamond inheritance:

```
interface A { void m() {...} }
interface B extends A { void m() {...} }
interface C extends A { void m() {...} }
interface D extends B, C {}
```

Here both B.m and C.m override A.m, and D inherits both conflicting methods without an explicit override. In this case, automatically merging the two methods (to achieve diamond inheritance) is not possible, which is why many models (like traits and Java 8) reject such programs. Moreover, keeping the two method implementations in D is problematic. In essence, hierarchical information is not helpful to disambiguate later method calls, since the two methods share the same origin (A.m). Our calculus rejects such conflicts by the (T-Intf) rule, where D is considered to be ill-formed. We believe that rejecting D follows the principle of models like traits and Java 8 interfaces, where the language/type-system is meant to alert the programmer for a possible conflict early.

Nonetheless, C++ accepts the definition of `D` but forbids later upcasts from `D` to `A` because of ambiguity. Our language is more conservative on definitions of interfaces compared to C++, but on the upside, upcasts are not rejected. We could also loosen the model to accept definitions such as `D`, and perform ambiguity check on upcasts and other expressions. Then, we would need to handle more cases than C++ because of the complication caused by the hierarchical overriding feature.

6 FHJ+

This chapter continues the topic of unintentional conflicts from Chapter 5. Besides unintentional method conflicts, we will extend **FHJ** to **FHJ+**, which supports state and state conflict resolution. Moreover, **FHJ+** is also a general solution to the diamond problem. We provide a formal calculus with concrete solutions to the corresponding issues.

6.1 Motivation

As stated by Taivalsaari [Taivalsaari, 1996], object-oriented systems are usually built around classes, where a class represents a generic concept, and an instance represents an individual. Classes are the templates for creating a set of similar objects while an instance holds the local data representing the state of the object.

As we discussed in the last chapter, programmers have been struggling to solve various issues caused by multiple inheritance. In the last chapter, **FHJ** tries to solve the unintentional conflicts problem in multiple inheritance. It provides the algorithm for hierarchical dispatching and a novel mechanism for disambiguation, even in the presence of unintentional conflicts.

In this chapter, based on **FHJ**, we will deal with a related problem: multiple inheritance with state. When it comes to state, new issues occur and the current language solutions/-models are not satisfying. Developing full and faithful type systems for object-oriented languages is a well-known and challenging research problem. Moreover, in many cases, the existence of state is the reason that causes this difficulty. As discussed in the original trait paper [Ducasse et al., 2006], in single inheritance, inheriting state does not cause complications and a simple mechanism such as using the keyword `super` is enough to achieve the goal. However, when it comes to multiple inheritance, state entangles with multiple inheritance issues, which makes the situation more complicated. Existing languages/models either ignore the problem or only provide limited support.

The trait model [Schärli et al., 2003] is a novel multiple inheritance model. However, it does not introduce state. As reviewed in Section 2.2.1, a trait is a collection of methods

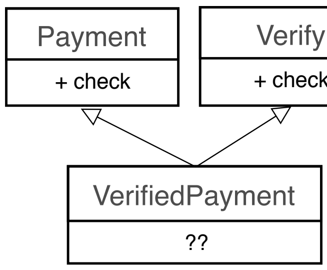


Figure 6.1: The UML diagram of the transaction example.

without state; it can be viewed as an incomplete stateless class, to avoid the complexity caused by state. As Schärli et al. [Schärli et al., 2003] summarized the trait model with the following equation:

$$\mathsf{Class} = \mathsf{Superclass} + \mathsf{State} + \mathsf{Traits} + \mathsf{Glue}$$

From this equation, it is obvious that classes in their system can have state. However, traits themselves (which provide the multiple inheritance functionality) do not support state. In other words, it is hard to embed state into multiple inheritance models with traits directly.

The mixin model [Bracha and Cook, 1990] supports state. The main difference between mixins and traits is that in the mixin model, when composing multiple mixins, a linear order requirement is enforced. This linear order restriction causes significant fragility problems and may make code maintainability difficult [Schärli et al., 2003]. Moreover, because of the state, mixin models must also deal with constructor problems, which can be another source of fragility since it is hard to predict what the interface of the super-class constructor will be [Reppy and Turon, 2006].

C++ is problematic with state, especially in the diamond problem. For example, when a class `C` inherits an ancestor `A` through more than one path where `A` has fields, should `C` inherit multiple copies of the fields or just one? With virtual inheritance, `C` expects to inherit only one copy of the fields from `A`. Then the object initialization would be problematic because we cannot ensure that the object initializer of `A` is called only once [Malayeri and Aldrich, 2009]. The object initialization problem occurs in this semantics and it depends on how and when the superclass constructor is called [Singh, 1995, Snyder, 1986].

The existence of state poses additional difficulty in the scenario of unintentional conflicts. We will use the following example for illustration. Figure 6.1 is the UML diagram for a bank transaction system. Class `Payment` represents a payment process made by users, where it is paid by check, and the field `check` represents the check ID. Class `Verify` is a class representing the verification process with a flag field `check` to indicate whether the verification is checked or not. Class VerifiedPayment inherits both Payment and Verify, describing the information of verified payments. Inside class VerifiedPayment, the two fields check from Payment and Verify are in conflict. This kind of conflict is an unintentional conflict. The two fields have completely different meanings/domains which happen to have the same name. Let us consider how to express this case in different languages and how they are treated.

In Java-like languages, this state conflict case cannot even be directly expressed! The reason is that multiple class inheritance is not allowed in Java. However, interfaces can be used to mimic classes, and we can use state operations to mimic state. For example, the code in Java with the **Classless Java**-style is shown below:

```
interface Payment {
   String check(); // represents an entity 'check'
}
interface Verify {
   // represents whether the verification is checked or not.
   boolean check();
}
interface VerifiedPayment extends Payment, Verify {}
```

In the code above, the two methods check() are used to represent the two fields of Payment and Verify, respectively. We want to keep both fields (separately) in VerifiedPayment, but the program is rejected by Java because 'the return types are incompatible for the inherited methods Payment.check(), Verify.check()'. This method conflict is the limitation of abstract state operations in **Classless Java**. Even if in other scenarios where the return types are the same, we still expect the two fields are inherited separately instead of being treated as one since they are unintentionally conflicted.

C++ partly supports this, just like how C++ supports unintentional method conflicts. However, besides the drawbacks of the initialization problem we mentioned before, C++ cannot handle the case where fields need to be refined or two fields with conflicts need to be merged. The following code illustrates the representation of the same example. Class VerifiedPayment inherits both Payment and Verify, and class Payment has a field check with the type of A*; class Verify has a same-named field check but with a different type bool. It compiles and runs correctly, showing that C++ does support unintentional state conflicts to a certain extent.

```
class A {};
class Payment {
   public:
   A* check;
};
class Verify {
   public:
   bool check;
};
class VerifiedPayment : public Payment, public Verify {};

int main()
{
   VerifiedPayment* vp = new VerifiedPayment();
   vp->Verify::check = true;
}
```

Next, we will use a modified version of the transaction system to illustrate the issues:

```
class A {};
class B:A {};
class Payment {
   public:
   A* check;
};
class Verify {
   public:
   bool check;
};
class VerifiedPayment : public Payment, public Verify {
   public:
   B* check;
};

int main()
{
   VerifiedPayment* vp = new VerifiedPayment();
   vp->Verify::check = true;
   vp->check = new B();
   cout << vp->Verify::check << ", ";
   cout << vp->Payment::check << endl;
}
```

Here, class `VerifiedPayment` still inherits `Payment` and `Verify` and would like to refine the type of field `check` from a pointer to `A` to `B`. The desired semantics is that class `VerifiedPayment` only contains two fields and running result should be `1, #a_pointer_to_check`. However, our experimental result is `1, 0x0`, meaning that the pointer to `Payment.check` is **`null`**. The experimental result shows that in class `VerifiedPayment`, `Payment.check`, `Verify.check` and `VerifiedPayment.check` are three unrelated fields. Thus the solution in C++ is not satisfying. Note that in our current formalization, field refinement via covariant return types is not supported yet.

6.2 Overview

Knowing that multiple inheritance gets complicated when dealing with state, in this section, we will give an overview of **FHJ+** and illustrate the issues in detail. Especially we summarize the issues into several problems. We will introduce them one by one and show how **FHJ+** solves them.

6.2.1 Abstract State Operations

We first quickly go through how to represent the fields. As discussed in Section 6.1, we may use abstract state operations as in **Classless Java** to mimic state. An example program in **FHJ+** to represent fields would look like this:

```
class Payment {
   int check();
   //void check(int x);
}
class Verify {
   bool check();
}
```

Here, component `Payment` has a method (getter) `check()` to represent the field `check`. The commented out method `void check(int x)` can be regarded as the setter of `check`. Similarly, component `Verify` has a method (getter) `check()` to represent the field `check`. These getters and setters are *abstract state operations* to represent state in Java.

6.2.2 Constructors in FHJ+

In terms of constructors in **FHJ+**, they are basically generated with some specification from programmers. For example, the following is the code for `VerifiedPayment` with the constructor and code for initialization. The code `new (int Payment.check, bool Verify .check);` specifies a constructor with two arguments (also two fields) `Payment.check` and `Verify.check`.

```
// VerifiedPayment with constructor
class VerifiedPayment extends Payment, Verify {
   new (int Payment.check, bool Verify.check);
}
```

```
// instantiation
VerifiedPayment vp = new VerifiedPayment(1, false);
(Payment)vp.check()
```

At the call site, for an object of class `VerifiedPayment`, the disambiguation of its two fields from parent/branch `Payment` can be achieved through the type annotation `Payment`.

In C++ (and other OO languages), when a member of a class is declared, it is either declared to be a field or a method. This member can have one and only one role. However, there might be cases where people want to bind the role to a member later, to determine the role in the future. In this case, **FHJ+** can also help to achieve this goal with constructors.

```
class Payment {
    int check();
    int date();
    new (int Payment.check);
}
class Verify {
    int check();
    new (int Verify.check);
}
class VerifiedPayment extends Payment, Verify {
    new (int Payment.check, int Payment.date, int Verify.check);
}
```

In class `Payment` and `Verify`, both `check` are fields because they are both declared in the constructors of `Payment` and `Verify`. Notice that in class `Payment`, although the method `date()` takes the same form as method `check()`, since `date` does not exist in the constructor of `Payment`, it is just an abstract method. However, in `VerifiedPayment`, we choose to make `date` a field, so we can just put it into the constructor of `VerifiedPayment`.

In the rest of the thesis, we also simply use *constructors* to stand for *constructor specification*. Constructors also play the role of specifying the order of fields, this is important in the case of fields with the same type or conflicting fields. For example, in class `VerifiedPayment`, the constructor `new (int Payment.check, int Payment.date, int Verify.check)` specifies that all the three members will be treated as fields and the order is also specified so that at the call site, with the constructor signature, the order would not confuse the programmers.

6.2.3 Problem 1: Unintentional State Conflicts

Section 6.1 illustrates what is the unintentional state conflicts problem, now how to solve it? Assuming that we use abstract state operations to represent state, an example program in the desired language **FHJ+** should look like this:

```
class Payment {
    int check();
    //void check(int x);
}
class Verify {
    bool check();
}
class VerifiedPayment extends Payment, Verify {
    ...
}
```

When class `VerifiedPayment` inherits/extends classes `Payment` and `Verify`, both the two fields `Payment.check`, and `Verify.check` will be inherited by class `VerifiedPayment` without any conflicts. This solution in **FHJ+** is simple and straightforward to use.

6.2.4 Problem 2: The Diamond Problem

The diamond problem is a classic problem when talking about multiple inheritance. Here, it is the same when we take state into consideration: two classes `B` and `C` inherit from `A` (with state), and class `D` inherits from both `B` and `C`. Current languages have not treated this issue seriously, and some languages cannot manage the issue in a way we want it to be. In C++, both ordinary inheritance and virtual inheritance are supported. However, neither approach is enough to solve the problem.

How to solve the diamond problem with state? Let us look at the example in Fig. 6.2. If the diamond case is represented with ordinary inheritance in C++, the code does not even compile, with the error message "non-static member 'check' found in multiple base-class subobjects of type 'Verify'". Note that if we only declare classes `Verify`, `Verify1`, `Verify2` and `Verify3` but not use them (call site), then C++ would not report any errors.

Next, Fig. 6.3 shows the usefulness of C++ virtual inheritance, which supports the representation of diamond problem with fields without any error. **FHJ+** also support this functionality. As shown in Fig. 6.4, this solution is similar to C++.

However, there are further issues in C++: the object initialization problem. The semantics is non-deterministic, it depends on how and when the superclass constructor or initializer is called [Singh, 1995, Snyder, 1986]. For example, in Fig. 6.5, the code is rejected with the error message "implicit default constructor for 'Verify3' must explicitly initialize the base class 'Verify' which does not have a default constructor ". In our **FHJ+** model, the problem does not exist, because we have a restrictive form of fields with state operations and constructors, as shown in Section 6.2.2.

```
class Verify {
    bool check;
};
class Verify1 : Verify {};
class Verify2 : Verify {};
class Verify3 : Verify1, Verify2 {};

int main(int argc, char const *argv[])
{
    Verify3 *v = new Verify3();
    cout << v -> check << endl; //member found by ambiguous name
        lookup
    return 0;
}
```

Figure 6.2: Diamond problem with C++ ordinary inheritance.

In the following section, more details will be given on the calculus, including the syntax, typing rules, semantics, and proofs.

6.3 Formalization

In this section, we present a formal model called **FHJ+** (Featherweight Hierarchical Java Plus), also following the Featherweight Java [Igarashi et al., 2001] model. Based on **FHJ**, **FHJ+** makes use of the hierarchical dispatching algorithm and adds the favor of supporting unintentional state conflicts, which integrates the ideas from **Classless Java** to support state with abstract state operations. The syntax, typing rules, and small-step semantics are presented. Note that in our current formalization, covariant field type refinement is not supported yet for two reasons: time limitation and the complexity caused by adding covariant field type refinement.

6.3.1 Syntax

The abstract syntax of our model **FHJ+** is presented in Fig. 6.6. **FHJ+** supports features including multiple inheritance, default and abstract methods, cast expressions, let expressions, etc. Fig. 6.6 also presents the syntax for class declarations, method declarations, various kinds of expressions and values.

Notations The meta-variables I, J, K range over class names; x ranges over variables; m ranges over method names; e ranges over expressions; M ranges over method declarations;

```
class Verify {
public:
   int num;
};
class Verify1 : public virtual Verify {};
class Verify2 : public virtual Verify {};
class Verify3 : public Verify1, public Verify2 {
};
int main(int argc, char const *argv[])
{
   Verify3 *v = new Verify3();
   cout << v -> num << endl; //0, 0
   return 0;
}
```

Figure 6.3: Diamond problem with C++ virtual inheritance.

```
class Verify {
   Int num() override Verify;
}
class Verify1 extends Verify {}
class Verify2 extends Verify {}
class Verify3 extends Verify1, Verify2 {
   new (Int Verify.num);
}

new Verify3(3)
```

Figure 6.4: Diamond problem in **FHJ+**

and o ranges over object identifiers (see details in Section 6.3.3). Following Featherweight Java, we assume that the set of variables includes the special variable `this`, which cannot be used as the name of an argument to a method. Following **FHJ**, we write $\overline{I}$ as a shorthand for a possibly empty sequence $I_1, ..., I_n$, which may be indexed by I_i; and write $\overline{M}$ as shorthand for $M_1 ... M_n$ (with no commas). We also abbreviate operations on pairs of sequences in an obvious way, writing $\overline{I}\ \overline{x}$ for $I_1\ x_1, ..., I_n\ x_n$, where n is the length of $\overline{I}$ and $\overline{x}$.

Classes Just like **FHJ**, in order to achieve multiple inheritance, a class can have a set of parent classes, where such a set can be empty. Moreover, as usual in class-based languages, the extension relation over classes is acyclic. The class declaration `class I extends` $\overline{I}$ $\{MC?\ \overline{M}\}$ introduces a class named I with parent classes $\overline{I}$, a constructor MC and a suite of methods $\overline{M}$. The methods of I may either override methods that

```
class Verify {
public:
    Verify(int check_) {check = check_;};
public:
    int check;

};

class Verify1 : public virtual Verify {
public:
    Verify1(): Verify(1) {};
};

class Verify2 : public virtual Verify {
public:
    Verify2(): Verify(2) {};
};

class Verify3 : public Verify1, public Verify2 {
public:
    bool flag;
};

int main(int argc, char const *argv[])
{
    Verify3 *v = new Verify3();
    cout << v ->flag << ", " << v -> check << endl; //0, 0
    return 0;
}
```

Figure 6.5: C++ constructor initialization problem

are already defined in $\overline{I}$ or add new functionality special to I, and they could be ordinary methods or getter/setters. We will illustrate this in more detail later.

Methods The syntax of methods (both default and abstract) follows **FHJ**. However, the semantics is different in terms of state operations. Original methods and hierarchically overriding methods share the same syntax in our model for simplicity. The concrete method declaration $I\ m(\overline{I_x\ x})$ `override` $J\ \{$`return` $e;\}$ introduces a method named m with result type I, parameters $\overline{x}$ of type $\overline{I_x}$ and the overriding target J. The body of the method simply includes the returned expression e. Notably, we have introduced the `override` keyword for two cases. Firstly, if the overridden class is exactly the enclosing class itself, then such a method is seen as *originally defined*. Note that the case of merging methods from differ-

Program	P	$::=$	$\overline{IL}\ e$
Classes	IL	$::=$	$\texttt{class}\ I\ \texttt{extends}\ \overline{I}\ \{MC?\ \overline{M}\}$
Constructors	MC	$::=$	$\texttt{new}(\overline{I_x}\ \overline{J.x})\ ;$
Methods	M	$::=$	$I\ m(\overline{I_x}\ \overline{x})\ \texttt{override}\ J\ \{\texttt{return}\ e;\} \mid I\ m(\overline{I_x}\ \overline{x})\ \texttt{override}\ J\ ;$
Expressions	e	$::=$	$x \mid e.m(\overline{e}) \mid (I)e \mid \texttt{new}\ I(\overline{e}) \mid I\ x = e_1; e_2$
Context	Γ	$::=$	$\overline{x} : \overline{I}$
Values	v	$::=$	$(I)o$

Figure 6.6: Syntax of **FHJ+**

ent branches also counts as originally defined. Secondly, for all other cases, the method is considered a *hierarchical overriding method*. Note that in a class J, $I\ m(\overline{I_x}\ \overline{x})\ \{\texttt{return}\ e;\}$ is syntactic sugar for $I\ m(\overline{I_x}\ \overline{x})\ \texttt{override}\ J\ \{\texttt{return}\ e;\}$, which is the standard way to define methods in Java-like languages. The definition of abstract methods is written as $I\ m(\overline{I_x}\ \overline{x})\ \texttt{override}\ J\ ;$, which is similar to a concrete method but without the method body. An abstract method without any parameters can represent a getter when the corresponding name appears in the object constructor. Similarly, an abstract method with one and only one parameter can represent a setter, when the corresponding getter is defined.

Constructors The syntax of object constructor declaration is $\texttt{new}(\overline{I_x}\ \overline{J.x})\ ;$. $\overline{x}$ are all the fields declared in the current class or inherited from the parents, with path annotation $\overline{J}$. Putting them together, we use $\overline{J.x}$ to denotes qualified fields. $\overline{I_x}$ are the types of fields $\overline{J.x}$. Corresponding to Figure 6.3.1, we will guarantee the coherence by checking all parameters $\overline{J.x}$ have the corresponding valid getters/setters.

Expressions Expressions are standard constructs such as variables, method invocation, object creation, cast expressions together with let expressions. Object creation is represented by $\texttt{new}\ I(\overline{e})$, which should conform to the constructor signature of the class. The casts are merely safe up-casts, just as in **FHJ**, they can be viewed as annotated expressions, where the annotation indicates its static type. In **FHJ+**, it can also be used for fields disambiguation when there are unintentional state conflicts. For the variable assignment, instead of using variable assignment directly, the approach we take is let expressions. The reason is that for modeling the single feature variable assignment, we will have to model statements and also variable declaration. In this case, let-expressions can achieve the same goal without expanding expressions to statements.

Values A value $(I)o$ is the result of multiple reduction steps for evaluating an expression. The notion o is the object reference created during the object creation process, pointing to a concrete object (address) in the heap, and I is the static type of the object. The static type I is a possible result of type annotation from programmers for disambiguation or generated during the reduction process.

A program in **FHJ+** consists of a list of class declarations, plus a single expression.

6.3.2 Subtyping and Typing Rules

The subtyping relation of **FHJ+** is shown at the top of Figure 6.7. It is traditional and built from the inheritance in class declarations. Subtyping is both reflexive and transitive.

Details of type-checking rules are shown in Figure 6.7, including expression typing, well-formedness of constructors, methods, and classes. As a convention, an environment Γ is maintained to store the types of variables, together with the self-reference `this`. The typing rules take the same approach as **FHJ**. The difference is just the value representation and let expressions. The simplicity of **FHJ+** makes it easy to learn and use.

(T-Invk) is the typing rule for method invocation. Naturally, the receiver and the arguments are required to be well-typed. `mbody` is our key function for method lookup that implements the hierarchical dispatching algorithm. The formal definition will be introduced in Section 6.4. Here $\texttt{mbody}(m, I_0, I_0)$ finds the most specific m above I_0. "Above I_0" specifies the search space, namely the supertypes of I_0 including itself. For the general case, however, the hierarchical invocation $\texttt{mbody}(m, I, J)$ finds "the most specific m above I and along the path/branch J". "Along the path J" additionally requires the result to relate to J, that is to say, the most specific class that has a subtyping relationship with J. The result of `mbody` contains the class that provides the most specific implementation, the parameters and the return type. We use the underscore character for the return expression, matching both implemented and abstract methods.

(T-New) is the typing rule for object creation $\texttt{new}\ I(\overline{e})$. The auxiliary function $\texttt{mconstr}(I_0) = \overline{I_x}\ \overline{J.x}$ (see definition in Section 6.4) retrieves the signature of the constructor and checks whether the arguments conform to the parameters specification. Since fork inheritance accepts conflicting branches to coexist, the check requires that the most specific method is concrete for each method on each branch.

(T-Cast) is the typing rule for the cast expressions. The rule checks the expression e to type I and checks that e is casting to J, which is a supertype of I.

(T-Let) is the typing rule for the let expressions. The rule checks the expression e_1 to type I_1, which is a subtype of I. And under the new context (with variable x), the expression e_2

$$\boxed{I <: J} \qquad I <: I$$

$$\frac{I <: J \qquad J <: K}{I <: K} \qquad \frac{\texttt{class}\ I\ \texttt{extends}\ I_1, I_2, ..., I_n\ \{...\}}{I <: I_1, I <: I_2, ..., I <: I_n}$$

$$\boxed{\Gamma \vdash e : I} \qquad (\text{T-Var})\ \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$(\text{T-Invk})\ \frac{\Gamma \vdash e_0 : I_0 \qquad \texttt{mbody}(m, I_0, I_0) = (K, \overline{I_x}\ \overline{x}, I\ \_) \qquad \Gamma \vdash \overline{e} : \overline{I} \qquad \overline{I} <: \overline{I_x}}{\Gamma \vdash e_0.m(\overline{e}) : I}$$

$$(\text{T-New})\ \frac{\texttt{mconstr}(I_0) = \overline{I_x}\ \overline{J.x} \qquad \Gamma \vdash \overline{e} : \overline{I} \qquad \overline{I} <: \overline{I_x}}{\Gamma \vdash \texttt{new}\ I_0(\overline{e}) : I_0}$$

$$(\text{T-Cast})\ \frac{\Gamma \vdash e : I \qquad I <: J}{\Gamma \vdash (J)e : J} \qquad (\text{T-Let})\ \frac{\Gamma \vdash e_1 : I_1 \qquad I_1 <: I \qquad \Gamma, x : I \vdash e_2 : I_2}{\Gamma \vdash I\ x = e_1; e_2 : I_2}$$

$$(\text{T-MC})\ \frac{\texttt{validMC}(I, \overline{I_x}, \overline{J.x})}{\texttt{new}(\overline{I_x}\ \overline{J.x})\ ;\ \text{OK IN}\ I} \qquad (\text{T-Prog})\ \frac{\forall I \in \overline{IL}, I\ \text{OK} \qquad \Gamma \vdash e : T}{\overline{IL}\ e\ \text{OK}}$$

$$(\text{T-Method})\ \frac{\begin{array}{c} I <: J \qquad \texttt{findOrigin}(m, I, J) = \{J\} \qquad \texttt{mbody}(m, J, J) = (K, \overline{I_x}\ \overline{x}, I_e\ \_) \\ \Gamma, \overline{x} : \overline{I_x}, \text{this} : I \vdash e_0 : I_0 \qquad I_0 <: I_e \end{array}}{I_e\ m(\overline{I_x}\ \overline{x})\ \texttt{override}\ J\ \{\texttt{return}\ e_0;\}\ \text{OK IN}\ I}$$

$$(\text{T-AbsMethod})\ \frac{\begin{array}{c} I <: J \quad \texttt{findOrigin}(m, I, J) = \{J\} \\ \texttt{mbody}(m, J, J) = (K, \overline{I_x}\ \overline{x}, I_e\ \_) \end{array}}{I_e\ m(\overline{I_x}\ \overline{x})\ \texttt{override}\ J\ ;\ \text{OK IN}\ I}$$

$$(\text{T-Class})\ \frac{\begin{array}{c} MC\ \text{OK IN}\ I \qquad \overline{M}\ \text{OK IN}\ I \\ \forall J >: I \text{ and } m, \texttt{mbody}(m, J, J) \text{ is defined} \Rightarrow \texttt{mbody}(m, I, J) \text{ is defined} \\ \forall J >: I \text{ and } m, I[m\ \texttt{override}\ I] \text{ and } J[m\ \texttt{override}\ J] \text{ defined} \Rightarrow \texttt{canOverride}(m, I, J) \end{array}}{\texttt{class}\ I\ \texttt{extends}\ \overline{I}\ \{MC?\ \overline{M}\}\ \text{OK}}$$

Figure 6.7: Typing rules of **FHJ+**

type checks to I_2. The type for the whole expression is determined by the type of e_2, which is I_2.

(T-Method) is more interesting since a method can either be an original method or a hierarchical overriding, though they share the same syntax and method typing rule. $\texttt{findOrigin}(m, I, J)$ is defined the same as that of **FHJ**. Then $\texttt{mbody}(m, J, J)$ provides the type of the original method, so hierarchical overriding has to preserve the type. Finally, the return expression is type-checked to be a subtype of the declared return type. The typing rule (T-AbsMethod) is a similar rule but works on abstract method declarations.

The rule (T-MC) defines the typing rule on constructors. The checking process is abstracted away with the auxiliary definition `validMC`, which is defined in Section 6.4.1. Given the current class, the fields and their types, `validMC` will check whether the constructor declaration is coherent and complete.

(T-CLASS) defines the typing rule on classes. The first two conditions are obvious, namely, its constructor declaration and methods need to be well checked. The fourth condition checks whether the overriding between original methods preserves typing. In this condition, we again use some helper functions defined in Section 6.4. $I[m\ \texttt{override}\ I]$ is defined if I originally defines m, and $\texttt{canOverride}(m, I, J)$ checks whether $I.m$ has the same type as $J.m$. Generally the preservation of method type is required for any supertype J and any method m. The third condition of (T-CLASS) is more complex and is the key to type soundness. Unlike C++, which rejects on ambiguous calls, **FHJ** rejects on the definition of classes when they form a diamond. Consider the case when the third condition is broken: $\texttt{mbody}(m, J, J)$ is defined but $\texttt{mbody}(m, I, J)$ is undefined for some J and m. This indicates that m is available and unambiguous from the perspective of J, but is ambiguous to I on branch J.

Finally, rule (T-PROG) is the typing rule for the whole program. It checks that all the declared classes and expression are well type checked.

6.3.3 Small-step Semantics

Figure 6.8 defines the reduction rules of **FHJ+**. When evaluating an expression, it is invoked with a heap environment and produces a single value with a final heap in the end. A heap environment μ maps object identifiers o to object states os:

$$\mu ::= \overline{o \mapsto os} \tag{6.1}$$

$$os ::= \texttt{new}\ I(\overline{o}) \tag{6.2}$$

(E-INVK) is an important computation rule we need for method invocation. As a small-step rule, it assumes that the receiver and the arguments are already values. $\mathsf{isField}(m, I)$ is an auxiliary function to check whether m is a field of class I (see the definition of isField in Section 6.4.1), where I is the type of o. Here the rule (E-INVK) handle the case where the method call is not a getter or setter. Specifically, the receiver $(J)\texttt{new}\ I()$ indicates the dynamic type I together with the static type J. Therefore $\texttt{mbody}(m, I, J)$ carries out hierarchical dispatching, acquires the types, the return expression e_0 and the class I_0 which provides the most specific method. Here we use e_0 to imply that the return expression is forced to be non-empty be-

$$\boxed{\mu \mid e \to \mu' \mid e'} \qquad \text{(E-INVK)}\ \frac{\begin{array}{c}\mu(o) = \texttt{new I}(...) \\ \neg\mathsf{isField}(m, I) \qquad \texttt{mbody}(m, I, J) = (I_0, \overline{I_x}\ \overline{x}, I_e\ e_0)\end{array}}{\mu \mid (J)o.\texttt{m}(\overline{v}) \to \mu \mid (I_e)e_0[(J)o/\texttt{this}, (\overline{I_x})\overline{v}/\overline{x}]}$$

$$\text{(E-NEW)}\ \frac{o \notin \mathrm{dom}(\mu) \quad \mu' = \mu, o \mapsto \texttt{new I}(\overline{v})}{\mu \mid \texttt{new I}(\overline{v}) \to \mu' \mid (I)o}$$

$$\text{(E-SETTER)}\ \frac{\begin{array}{c}\mu(o) = \texttt{new I}(o_1, ..., o_n) \qquad \texttt{mconstr}(I) = \texttt{new}(\overline{I_x}\ \overline{J.x})\ ; \\ J.f \textit{ is the i-th element of } \texttt{mconstr}(I) \qquad \texttt{isSetterDeclared}(J, f) \\ \mu' = \mu[o \mapsto \texttt{new I}(o_1, ..., o', ..., o_n)] \textit{ with } \mu(o) = \texttt{new I}(o_1, ..., o_i, ..., o_n)\end{array}}{\mu \mid (J)o.\mathsf{SET\_f}((I')o') \to \mu' \mid (I)o}$$

$$\text{(E-GETTER)}\ \frac{\begin{array}{c}\mu(o) = \texttt{new I}(o_1, ..., o_n) \\ \texttt{mconstr}(I) = \texttt{new}(\overline{I_x}\ \overline{J.x})\ ; \qquad I_F\ J.f \textit{ is the i-th element of } \texttt{new I}(...)\end{array}}{\mu \mid (J)o.f() \to \mu \mid (I_F)o_i}$$

$$\text{(E-LET)}\ \frac{}{\mu \mid I\ x = o; e \to \mu \mid e[(I)o/x]} \qquad \text{(E-CAST)}\ \frac{}{\mu \mid (J)((I)o) \to \mu \mid (J)o}$$

$$\text{(E-CTX)}\ \frac{\mu \mid e \to \mu' \mid e'}{\mu \mid \varepsilon\{e\} \to \mu' \mid \varepsilon\{e'\}}$$

$$\textit{where } \varepsilon ::= \emptyset \mid \varepsilon.m(\overline{e}) \mid (J)o.m(\overline{o}, \varepsilon, \overline{e}) \mid \texttt{new I}(\overline{o}, \varepsilon, \overline{e}) \mid I\ x = \varepsilon; e \mid (I)\varepsilon$$

Figure 6.8: Reduction rules of **FHJ+**

cause it requires a concrete implementation. Now the rule reduces the method invocation to e', which is a substitution of e_0. Parameters are substituted with arguments, and the **this** reference is substituted with the receiver, and in the meanwhile, the static types are recorded via annotations. Finally, the return type I_e is put in the front as an annotation.

The rule (E-NEW) guides the reduction process of object creation. Reducing an expression new $I(\overline{v})$ under the heap environment μ, will produce a fresh object o with static type annotation (I) on the heap, leading to the new heap environment μ'.

The rule (E-GETTER) retrieves the information of object o from the heap environment μ and checks that the getter method $f()$ corresponds to a field of I. An expression $(J)o.f()$ will be reduced to $(I_F)o_i$ where o_i is the value of the field and I_F the static type. The heap environment μ keeps unchanged.

The rule (E-SETTER) is similar to (E-GETTER) but more complex. It retrieves the information of the object o from the heap environment μ and checks that $J.f$ *does exit* in the constructor of class I. Then it updates the corresponding field of object o with the new value specified by the argument $(I')o'$. Note that the heap environment μ is updated to μ'.

Under the rule (E-Let), a let-expression $I\ x = o; e$ will be reduced to $e[(I)o/x]$ with the heap environment μ unchanged. The rule (E-Cast) is trivial, for an expression of the form $(J)((I)o)$, it simply drops the inner type annotation (I).

The last rule (E-CtX) is an abstraction over a set of rules, acting as the role of standard congruence. An expression $\varepsilon\{e\}$ (*outer expression*) containing a reducible *inner expression* e is also reducible. If e reduces to e', then $\varepsilon\{e\}$ reduces to $\varepsilon\{e'\}$. The heap environment changes accordingly. The outer expression ε can be: method invocation with reducible receiver $\varepsilon.m(\overline{e})$, method invocation with reducible arguments $(J)o.m(\overline{o}, \varepsilon, \overline{e})$, object creation with reducible arguments $\texttt{new}\ I(\overline{o}, \varepsilon, \overline{e})$, a let expression with reducible variable assignment $I\ x = \varepsilon; e$, and the cast expression $(I)\varepsilon$.

6.4 Auxiliary Definitions and Properties

In this section, we present the auxiliary definitions used in our formalization and show the important properties of the calculus. Note that we will omit the definition of `mbody` since it reuses the definition from **FHJ** (see Section 5.4).

6.4.1 Auxiliary Definitions

▷ **Constructor Retrieve: `mconstr`**

Definition of $\texttt{mconstr}(I) = \overline{I_x}\ \overline{J.x}$:

- If $\texttt{new}(\overline{I_x}\ \overline{J.x})$ defined in class I, $\texttt{mconstr}(I) = \overline{I_x}\ \overline{J.x}$.
- Otherwise, $\texttt{mconstr}(I) = \mathtt{Undefined}$:

Given a class name I, `mconstr` retrieves the object constructor signature from the class declaration. $\overline{J.x}$ and $\overline{I_x}$ are the fields and their types, respectively.

▷ **Constructor Checker: `validMC`**

Definition of $\texttt{validMC}(I, \overline{I_x}, \overline{J.x})$:
$\texttt{validMC}(I, \overline{I_x}, \overline{J.x}) = \mathtt{true}$, if and only if
$\forall I_x J.x \in \overline{I_x J.x}$, $J.x$ is a valid field of I, i.e., method $I_x\ x()\ \mathtt{override}\ J;$ is defined in J.

▷ **Field Checker: isField**

Definition of $\mathsf{isField}(m, I)$:
$\mathsf{isField}(m, I) = \mathit{true}$ if and only if:

- $\mathtt{mconstr}(I) = \overline{I_x}\ \overline{J.x}$ and $m \in \overline{x}$.

▷ **Is Field Setter Declared: `isSetterDeclared`**

Definition of $\mathtt{isSetterDeclared}(I, m)$:
$\mathtt{isSetterDeclared}(I, m) = \mathit{true}$ if and only if:

- The setter declaration $\mathit{VoidSET}_m(...)\mathit{override}I;$ is declared in I.

▷ **Method Override:** $I[m\ \mathtt{override}\ J]$

Definition of $I[m\ \mathtt{override}\ J]$:

- $I[m\ \mathtt{override}\ J] = I_e\ m(\overline{I_x}\ \overline{x})\ \mathtt{override}\ J\ \{\mathtt{return}\ e_0;\}$
  with: $\mathtt{class}\ I\ \mathtt{extends}\ \overline{I}\ \{I_e\ m(\overline{I_x}\ \overline{x})\ \mathtt{override}\ J\ \{\mathtt{return}\ e_0;\}\ldots\}$
- $I[m\ \mathtt{override}\ J] = I_e\ m(\overline{I_x}\ \overline{x})\ \mathtt{override}\ J\ ;$
  with: $\mathtt{class}\ I\ \mathtt{extends}\ \overline{I}\ \{I_e\ m(\overline{I_x}\ \overline{x})\ \mathtt{override}\ J\ ;\ldots\}$

Here $I[m\ \mathtt{override}\ J]$ is basically a direct lookup for method m in the body of I, where such a method overrides J (like static dispatch). The method can be either concrete or abstract, and the body of definition is returned. Notice that by our syntax, $I[m\ \mathtt{override}\ I]$ is looking for the originally-defined method m in I.

▷ **Prune Set: `prune`**

Definition of $\mathtt{prune}(\mathit{set})$:

- $\mathtt{prune}(\mathit{set}) = \{I \in \mathit{set} \mid \nexists J \in \mathit{set} \setminus I, J <: I\}$

The prune function takes a set of types, and filters out those that have subtypes in the same set. In the returned set, none of them has subtyping relation to one another, since all supertypes have been removed.

6.4.2 Properties

In this section, we present the type safety theorem of **FHJ+** by a few theorems below, following the standard techniques of subject reduction and progress proposed by Wright and Felleisen [Wright and Felleisen, 1994]. The detailed proof and related lemmas are presented in Appendix. Theorem 5 is the subject reduction theorem, and it states that under one-step reduction following the reduction rules, when a term is reduced to another term, the type remains the same. Theorem 6 is the progress theorem, it states that if an expression is well-typed and includes a sub-expression with the method invocation form, then it is guaranteed that this expression can be further reduced, and it can take the form of either getter, setter or normal method invocation. Theorem 7 is the type soundness theorem, it states that if an expression is well-typed, then after many reduction steps it must reduce to a value, and the type of the value is the same as the original expression. Theorem 8 is the theorem of the determinacy of one-step evaluation. It states that given a heap environment and a term, there is one and only one possible rule and result for the one-step evaluation.

Theorem 5 (Subject Reduction). *If* $\Gamma \vdash e : I$ *and* $\mu|e \rightarrow \mu'|e'$, *then* $\Gamma \vdash e' : I$.

Proof. See Appendix A.7. $\square$

Theorem 6 (Progress). *Suppose* e *is a well-typed expression, if* e *includes* $(J)o.m(\overline{v})$ *as a sub-expression, where* $\mu(o) = \mathit{new}\ I(\dots)$, *then one of the following conditions holds:*

1. $\#\overline{v} = 0$, *validGetter(m, I, J)*

2. $\#\overline{v} = 1$, *validSetter(m, I, J)*

3. $\mathit{mbody}(m, I, J) = (I_0, \overline{I_x}\ \overline{x}, I_e\ e_0)$ *and* $\#(\overline{x}) = \#(\overline{v})$ *for some* I_0, $\overline{I_x}$, $\overline{x}$, I_e *and* e_0.

Proof. See Appendix A.7. $\square$

Theorem 7 (Type Soundness). *If* $\vdash e : I$ *and* $|e \rightarrow^* \mu|e'$ *with* e' *a normal form, then* e' *is a value* v *with* $\vdash v : I$.

Proof. Immediate from Theorem 5 and Theorem 6. $\square$

Theorem 8 (Determinacy of One-Step Evaluation). *If* $\mu \mid t \rightarrow \mu' \mid t'$ *and* $\mu \mid t \rightarrow \mu'' \mid t''$, *then* $t' = t''$ *and* $\mu' = \mu''$.

Proof. See Appendix A.7. $\square$

```
class A {
    new(Int A.x);
    Int x() override A;
    Void SET_x(Int x) override A;
}

A m = new A(3);
m.SET_x(2);
m.x()
```

Figure 6.9: **FHJ+** program example.

6.5 Implementation

We have a prototype implementation [1] of an **FHJ+** interpreter written in Scala. The implementation validates all the examples presented in this chapter. The prototype is based on the implementation of **FHJ**, so it can also show the detailed step-by-step evaluation of the program, which is convenient for understanding and debugging programs & semantics. For example, Fig. 6.9 is a simple program example. Fig. 6.10 is the output by our prototype, which contains rich information: the AST, return type of the expression, and the step-by-step evaluation with heap change.

6.6 Discussions & Limitations

The current model of **FHJ+** is not the only possible model to implement our requirement for handling unintentional state conflicts. In this section, we will briefly discuss other possibilities and discuss the thoughts we have about the design space in the process of working on this project and explain why the reasons for taking the current approach.

6.6.1 Middleweight Java

Middleweight Java (MJ) [Bierman et al., 2003], as a contender for a minimal imperative core calculus for Java, is proposed by Bierman et al. in 2003. MJ is an extension of FJ that is big enough to include the essential imperative features of Java. In addition to FJ, they model language features such as object identity, field assignment, constructor methods and block structure. Since MJ is intended to be a starting point for the study of Java models, in the

[1]The implementation is available at `https://github.com/YanlinWang/MIM/tree/master/StateMIM-small`

```
Program(List(
    TypeDef(Void,List(),List(),Some(Constructor(List()))),
    TypeDef(Int,List(),List(),None),
    TypeDef(String,List(),List(),None),
    TypeDef(A,List(),
            List(MethDef(Int,x,List(),A,None),
                 MethDef(Void,SET_x,List(Parameter(Int,x)),A,None)),
            Some(Constructor(List(Field(Int,A,x)))))),
    LetExpr(A,m,InvkStatic(A,List(Num(3))),
    LetExpr(Void,TEMP,InvkSetter(Var(m),x,Num(2)),
    Invk(Var(m),x,List()))))

Type check: ==> Int
==> Config(H(Map(),1),LetExpr(A,m,InvkStatic(A,List(Num(3))),
    LetExpr(Void,TEMP,InvkSetter(Var(m),x,Num(2)),Invk(Var(m),x,List()))))
==> Config(H(Map(1 -> A),2),LetExpr(A,m,Object(A,1),
    LetExpr(Void,TEMP,InvkSetter(Var(m),x,Num(2)),Invk(Var(m),x,List()))))
==> Config(H(Map(1 -> A),2),
    LetExpr(Void,TEMP,InvkSetter(Object(A,1),x,Num(2)),Invk(Object(A,1),x,List())))
==> Config(H(Map(1 -> A),2),LetExpr(Void,TEMP,
    InvkStatic(Void,List()),Invk(Object(A,1),x,List())))
==> Config(H(Map(1 -> A, 2 -> Void),3),
    LetExpr(Void,TEMP,Object(Void,2),Invk(Object(A,1),x,List())))
==> Config(H(Map(1 -> A, 2 -> Void),3),Invk(Object(A,1),x,List()))
==> Config(H(Map(1 -> A, 2 -> Void),3),Num(2))
(type = Int,res = 2)
```

Figure 6.10: **FHJ+** step-by-step evaluation.

beginning stage of modeling **FHJ+**, we also did the formalization based on MJ. The details of that draft formalization can be found in Appendix A.9. In that version, **FHJ+** supports state, void setters, and statements. We make use of the heaps and stacks of MJ to model our state-related features. A program consists of a list of interface declarations and a list of statements. A statement can be an expression, variable declaration or variable assignment. We use the same operational semantics for **FHJ+** as in MJ, where operational semantics of MJ is defined as transitions between *configurations*. A configuration is defined as a four-tuple, containing the following information:

1. Heap: A finite partial function that maps oids to heap objects. This is the key to store state.
2. Variable Stack: It maps variable names to oids.
3. Term: The main term to be evaluated.
4. Frame stack: This is essentially the program context in which the term is currently being evaluated.

That version of formalization works, however, the model is significantly more complex. Later on, a more straightforward and concise model came to our attention. The model is proposed by Lagorio and Servetto [Lagorio and Servetto, 2011]. Instead of transitions between four-tuples, the new model only uses heaps, therefore simplifies the model a lot. Now the transition $\mu \mid e \rightarrow \mu' \mid e'$ means "the reduction of expression e, in a heap μ, produces an expression e' and a (possibly) updated heap μ'". Actually, the simple model is enough for achieving our goals, thus, we switch the base model to the later.

6.6.2 Statements

In one version of our model, besides modeling fields, we also model statements. The reason is that normally in object-oriented programming, programmers are used to manipulate state with state declaration and assignment, which are statements in common programming languages. We modeled that in the beginning, however, after we apply the simpler model as we described in Section 6.6.1, we removed variable declaration and assignment feature. Instead, we now support let expressions, partly to replace statements. The reason is that with let expressions we can mimic some of the functions of statements, however, we cannot express assignment, while the model is still simple enough. Therefore, it's a trade-off between expressiveness and simplicity. Since the simple model does not affect the core of our calculus, finally we choose to use the simpler model.

6.6.3 Constructors

The constructor design in object-oriented languages is tricky. In Java-like languages, there is a default constructor provided which programmers can use for object creation. Also, programmers can define additional constructors for field initialization and other stuff. In **Classless Java**, our system provides default static methods called *of* to replace the common constructor methods. Of course, programmers have the freedom to define their customized constructors. **FHJ** does not touch the difficult state-related issues. Therefore, there is also no need to care about constructors. For **FHJ+**, the scenario is harder. The reason is that: firstly, we need constructors to cater fields; secondly, these fields might be conflicting, a new representation is needed to take care of conflicts.

There are two possible approaches in our minds, the first is default constructors and the second is our current approach. If we take the first approach, then we need to provide a default constructor, which initializes all the fields. And the fields are automatically detected as valid fields once it satisfies our rules for judging fields (e.g., an abstract method without arguments). However, this is problematic because sometimes people may define an abstract method first, and later decide whether he/she wants it to represent a field or method.

Taking all of these thoughts into consideration, the constructors in our current formalization take the form of $\texttt{new}(\overline{\texttt{I}_\texttt{x}}\ \overline{\texttt{J.x}})$;. The detailed meaning is explained in Section 6.3. With this representation, we can achieve both goals:

- Customized constructors.
- Allows unintentionally conflicted fields.

However, compared to constructors for classes in other mature language models (e.g., C++), our constructors are still limited. There is a trade-off between flexibility (in terms of initialization and constraints) and a nice and clean model of multiple inheritance. In terms of constraints in the constructor design, for example, a class called `Employee` contains a field called `age`, one might want to check whether `age` is a valid value by checking whether it is between 0 and 100. In Java, one can check this condition directly and throw an exception if it does not hold. However, for simplicity, both **FHJ+** model and the prototype implementation do not support such flexibility. One possible way to overcome this limitation is to support special annotations such as `@Range(low = 0, high = 100)Int age();` that specify the condition which will be integrated in the generated code.

6.6.4 State Operations and Static methods

As we discussed before, the ideas of **FHJ+** partially come from the state operations of **Classless Java**. However, currently in **FHJ+**, only two state operations are supported, which are setters and getters. What about others?

In **Classless Java**, withers take the form of `Point2D withX(int val)`, with the functionality of cloning an object and updating the field `x` with the value `val`. **FHJ+** could potentially incorporate this operation without much effort since the wither operation is orthogonal with the hierarchical dispatching feature. In the future, it could be an extension to **FHJ+**.

Withers update one field value at a time, if programmers wish to update multiple fields simultaneously, we may also incorporate the functional updater state operation in **Classless Java** into **FHJ+**. In **Classless Java**, property updaters take the form of `with(Point2D)`. It takes a certain type of object and returns a new object, which is a copy of the current object, however, with all matching fields (with the argument object) updated with the new values. It can also be easily added to **FHJ+**.

Remember in **Classless Java**, static methods are supported, so that customized constructors are possible. In **Classless Java**, there is a default static `of` method as the object constructor for the object interface. Meanwhile, programmers may define their own constructors with static methods. In this way, specifying field constraints and field initialization would be possible. In principle, this can also be supported in **FHJ+**. However, in the current version of **FHJ+**, this is not available. Only an abstract method for specifying which ones are fields are allowed to represent object constructors.

As discussed in Section 4.2.4, covariant field type refinement is supported in **Classless Java**. For getters, it is supported in by method covariant return types. For setters, it is supported by the ordinary method overriding and overloading. Incorporate this feature into **FHJ+** will make the fields refinable. However, tangled with the unintentional field conflicts scenario, we would need to modify this approach a little bit and to add hierarchical overriding in the signature of the new getters and setters.

6.7 A Larger Example: Office Clerk

In this section, to show that **FHJ+** is scalable to real applications, we will present a slightly larger example: office clerk. The code written in **FHJ+** is presented in Appendix A.8.1. The UML diagram (for the primary classes) is shown in Fig. 6.11.

In this example, classes `Manager`, `Director`, and `Officer` represent for different types of titles (class `Title`) in a company. Class `Point` contains the two coordinates `x` and `y`, modeled as two state operations here inside class `Point`.

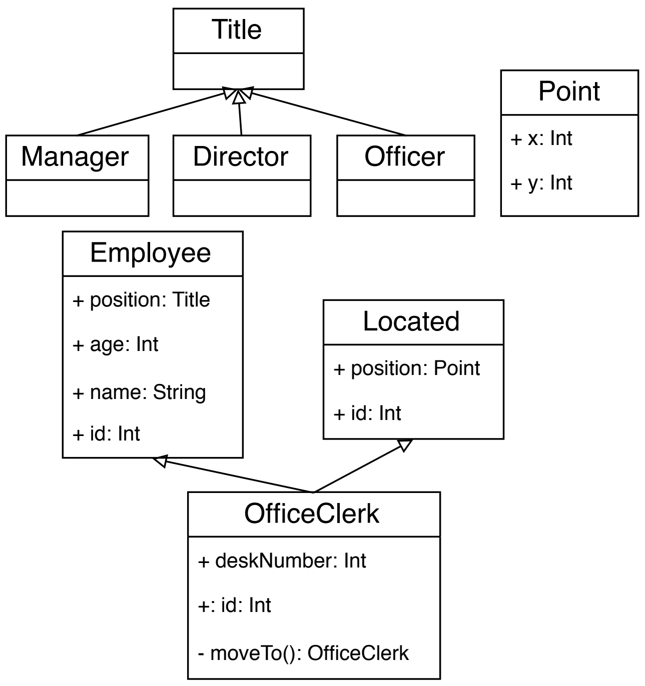


Figure 6.11: UML diagram for OfficeClerk

Class `Employee` contains four features (fields) of an employee, all represented by abstract state operations:

- `age` of type `Int`: the age of an employee.
- `name` of type `String`: the name of an employee.
- `position` of type `Title`: the position of an employee in the company, could be a `Manager`, `Director`, or `Officer`.
- `id` of type `Int`, the identity number of an employee.

Next is the core component `OfficeClerk`, which inherits from both `Located` and `Employee`. Note that there are two kinds of conflicts here (of fields `position` and `id`):

1. The two `position` fields inherited from `Located` and `Employee` coexist in the class `OfficeClerk`. This is what we called the *unintentional* state conflict, which we would expect to keep both of the fields in `OfficeClerk` without any conflicts.
2. Classes `Located` and `Employee` both contain the field `id` and in class `OfficeClerk`, we would like to merge the two fields into one. To achieve this, a simple line of method overridden `Int id()`**`override`** `OfficeClerk;` would satisfy the requirement.

	FHJ+	C++
LOC	45	57
Unintentional field conflicts	Yes	Yes
Field merge	Yes	No
Field refine	No (Not supported in the current version, but refer to \cj implementation, it is possible to incorporate this feature in the future)	No

Table 6.1: OfficeClerk comparison between **FHJ+** and C++

Moreover, the function `moveTo` shows how to do office move of a clerk from one place to another while keeping his/her title. Note that when calling the setter and getter of the field `position`, programmers can disambiguate the `position` from the `Located` branch from the `position` from the `Employee` by the prefix annotation `Located`.

Since programmers are more familiar with assignment statements, it seems like nothing can be done without assignments because when people write programs, it is very natural to define variables and assign new values to them. Therefore, for convenience, in the prototype implementation, we support the assignment statements as syntactic sugar to let expressions.

6.7.1 C++ Comparison

We also provide a C++ implementation of the `OfficeClerk` example as comparison (see code in Appendix A.8.2) and provide a simple comparison table (see Table 6.1).

1. In terms of lines of code, **FHJ+** reduces the SLOC by 21% (from 57 to 45, excluding blank lines), which is mainly due to the code required for constructors. For example, the constructor for class `Point` in the **FHJ+** implementation only requires one line of code (since it is actually a constructor specification):

   ```
   new(Int Point.x, Int Point.y);
   ```

   In contrast, the constructor for class `Point` in the C++ implementation requires 4 lines of code:

   ```
   Point(Int* _x, Int* _y) {
      x = _x;
      y = _y;
   }
   ```

2. Another difference is that **FHJ+** can do field merge, while in C++, there is no field merge. For example, to merge the fields `id` from classes `Located` and `Employee`, in **FHJ+**:

```
class Located {
    ...
    Int id() override Located;
}

class Employee {
    ...
    Int id() override Employee;
}

class OfficeClerk extends Located, Employee {
    ...
    Int id() override OfficeClerk;
    ...
}
```

In C++, if we do not declare a new field `id` in class `OfficeClerk` then there would be two separate `id` fields in `OfficeClerk`. If we do declare a new field `id` as we have presented in Appendix A.8.2, then there would be three separate `id` fields in `OfficeClerk`:

```
class Located {
    ...
    int id;
};

class Employee {
    ...
    int id;
};

class OfficeClerk : public Located, public Employee {
    ...
    int id;
    ...
};
```

This meaning that, one way or another, the fields would never be merged in C++.

7 Related Work

There are lots of related work to this thesis, some of them have been discussed in Chapter 2. In this chapter, we will discuss additional related work starting with previous solution to the EP. Then we will describe mainstream popular multiple inheritance models and then some specific models which are related to solving unintentional method conflicts. At last, we will talk about other topics including formalization based on Featherweight Java, code generation techniques, ThisType and MyType.

7.1 Previous Solutions to the Expression Problem

The Expression Problem in Mainstream Languages The solutions to the EP presented in Chapter 3 only use subtyping. This is in sharp contrast to existing solutions in various widely-used languages. Essentially all the solutions that we know of rely on various techniques and a combination of two mechanisms: subtyping and some form of type-parametrization.

Wadler proposed a solution using generics in Generic Java [Bracha et al., 1998] to solve the EP. However he later found a subtle typing problem. Torgersen's [Torgersen, 2004] presents four solutions that use a combination of Java generics and subtyping. His solutions use advanced features of generics, such as F-bounds [Canning et al., 1989] or wildcards. The solutions in Sections 3.4 and 3.2 follow the same structure as Torgersen's first solution. The difference is that covariant return types are used, instead of F-bounded type parameters, to type occurrences of recursive types. Similarly to our Java code, where we need some `Final` classes, Torgersen's solution requires a set of classes that close the fixpoint of the F-bounds. Object algebras [Oliveira and Cook, 2012] are an alternative approach to solving the EP in Java-like languages. While object algebras do not require advanced features of generics, such as F-bounds or wildcards, they still require a simple form of generics. Moreover the style in which code is written with object algebras can be considerably different from conventional OO style. Nevertheless, the combination of the techniques presented in our paper with object algebras is useful to solve problems that go beyond Wadler's EP.

There have been a few solutions to the EP in Scala. Zenger and Odersky [Zenger and Odersky, 2005] proposed a solution to the Expression Problem in

Scala using virtual types [Bruce et al., 1998]. The approach they present, commonly known in the Scala community as the "Cake pattern", is a way to use various Scala features to emulate some of the benefits of *virtual classes* [Ernst et al., 2006]. Their approach is closely related to our solution in Section 3.7. Where we use a type parameter with F-Bounded quantification, they use virtual type members. Instead of object algebras, they use factory methods directly in the family traits. In some sense it could be said that Section 3.7 approach is a poor man's version of the "Cake pattern", relying only in the features available in Java. Oliveira [Oliveira, 2009] presented two variations of the Visitor pattern, which allowed for extensibility. However this work used advanced features of generics including type constructor polymorphism, variance annotations, self-type annotations and mixins.

There are also solutions to the EP in languages like Haskell. A popular Haskell solution to the Expression Problem is based on folds (and F-algebras) [Duponcheel, 1995, Swierstra, 2008]. This solution requires some advanced features of Haskell and it does not translate well to object-oriented programming because most OO languages do not have native support for sums-of-products, which are needed in that solution. Another solution to the Expression Problem in Haskell is to encode fold-algebras with type-classes [Oliveira et al., 2006]. This solution is closely related to the work on object algebras.

Language-based solutions to the Expression Problem Various approaches, based on new programming languages or programming language features, can be used to solve the EP. Examples of these include: *multi-methods* [Chambers and Leavens, 1995]; *open classes* [Clifton et al., 2000]; *virtual classes* [Madsen and Moller-Pedersen, 1989, Ernst et al., 2006, Nystrom et al., 2006]; *virtual types* [Bruce et al., 1998]; *units* [McDirmid et al., 2001]; *polymorphic variants* [Garrigue, 1998]; and others [Zenger and Odersky, 2001, Löh and Hinze, 2006, Wehr and Thiemann, 2011]. Because these approaches create new language constructs targeted at solving extensibility problems, solutions to the EP can be expressed quite naturally. However we believe that the Scala solution in Section 3.2 is comparable in simplicity to language-based approaches. Language-based approaches still have an advantage at dealing with some of the harder challenges in extensibility. For example, with virtual classes, expressing the advanced forms of family polymorphism is still quite natural, whereas with our design-pattern based approach gets significantly more complex.

7.2 Lightweight Encodings of Family Polymorphism

There has been some work on encoding family polymorphism [Ernst, 2001] in existing languages, or lightweight encodings of family polymorphism. We already mentioned the "Cake Pattern" in Scala [Zenger and Odersky, 2005], which can emulate many of the features of family polymorphism and virtual classes. Various lightweight encodings of family polymorphism [Kamina and Tamai, 2007, Saito et al., 2008, Saito and Igarashi, 2008, Kamina and Tamai, 2008] and related ideas have been proposed as extensions of Featherweight Java (FJ) [Igarashi et al., 2001]. Notable among these proposals is the work by Saito and Igarashi 's [Saito and Igarashi, 2008] on the essence of family polymorphism. They proposed a lightweight version of family polymorphism, which relies on a simple extension to Featherweight Generic Java (FGJ) [Igarashi et al., 2001]: *self-type variables*. Self type variables allow giving the self references `this` a more precise type. This is necessary to allow type-checking certain definitions, which take the self-reference as an argument. Saito and Igarashi argue that this is the essential difference between their lightweight polymorphism approach and encodings of family polymorphism using F-bounded polymorphism only. Although self-type variables are not expressible directly in FGJ or Java, Torgersen has shown a simple way to work around the issue [Torgersen, 2004]. We believe Torgersen's techniques could be combined with our approaches to allow type-checking definitions that require such uses of self-references. However this would add additional complexity to the encoding. Saito and Igarashi's approach does not support virtual constructors: instantiation of classes with self type variables is disallowed. In contrast, the approach in Section 3.7 uses object algebras to emulate virtual constructors.

7.3 Language-related Techniques and Issues in Multiple Inheritance

Multiple inheritance is a useful feature in object-oriented programming. It is very expressive but difficult to model and implement. It can cause various problems (e.g., the famous diamond problem [Bracha and Cook, 1990, Sakkinen, 1989, Singh, 1995], conflicting methods, etc.) in reasoning about programs. To allow for expressive power and simplicity, many models have been proposed, including C++ virtual inheritance, traits, mixins [Bracha and Cook, 1990], traits [Schärli et al., 2003], and hybrid models such as CZ [Malayeri and Aldrich, 2009]. They provide novel programming architecture models in the OO paradigm. In terms of restrictions set on these models, C++ virtual inheritance aims at a relatively general model; the mixin model adds some restrictions; and the trait model is the most restricted one (excluding state, instantiation, etc.). In this section, we will discuss these models in detail and relate to the present work.

7.3.1 C++ Model and the Middleman Approach

C++ has a general solution to multiple inheritance by virtual inheritance, dealing with the diamond problem by keeping only one copy of the base class [Ellis and Stroustrup, 1990]. However, it suffers from the object initialization problem [Malayeri and Aldrich, 2009]. It bypasses constructor calls to virtual superclasses, which can cause serious semantic errors. In our approach in Chapter 4, the `@Obj` annotation has full control over object initialization, and the mechanism is transparent to users. If users are not satisfied with the default `of` method, customized factory methods can be provided.

C++ allows the existence of unintentional conflicts and users may specify a hierarchical path via casts for disambiguation, as discussed in Section 5.2. With virtual methods, dynamic dispatch is used and the method lookup algorithm will find the most specific method definition. A contribution of our work in Chapter 5 is to provide a minimal formal model of hierarchical dispatching, whereas C++ can be viewed as a real-world implementation. There are several formalizations [Wasserrab et al., 2006, Ramananandro, 2012, Ramalingam and Srinivasan, 1997] in the literature modeling various C++ features. However, as far as we know, there is no formal model that captures this aspect of the C++ method dispatching model. Apart from this, as discussed in Section 5.5.3, **FHJ** conservatively rejects some interface/class definitions that C++ accepts, and upcasts are never rejected since the ambiguity is prevented beforehand.

Although C++ supports hierarchical dispatching, it does not support hierarchical overriding. However, there are some possible workarounds that can mimic hierarchical overriding, including the *MiddleMan* approach [1], the *interface classes* pattern as described in Section 25.6 of [Stroustrup, 1995], the *LotterySimulation* discussion in [Stroustrup, 1994]. Since these workarounds share the same spirit, we will discuss in detail the *MiddleMan* approach, with the code shown in Figure 7.1. In this example, classes `A` and `B` are two classes that both define a method with the same name `m` unintentionally.

Class `MiddleMan`, as its name suggests, acts as a middleman between its class `C` and its parents `A, B`. `MiddleMan` defines a virtual method `m` that overrides a parent method `m` and delegates the implementation to another method `m_impl` that takes `this` as a parameter. C++ supports method overloading so that multiple `m_impl` methods with different parameter types can coexist. When defining class `C`, we specify the parents to be `MiddleMan<A>`,

[1] https://stackoverflow.com/questions/44632250/can-i-do-mimic-things-likes-this-partial-override-in-c

```
class A { public: virtual void m() {cout << "MA" << endl;}};
class B { public: virtual void m() {cout << "MB" << endl;}};
template<class C>
class MiddleMan : public C {
    void m() override final { m_impl(this); }
  protected:
    virtual void m_impl(MiddleMan*) { return this->C::m(); }
};
class C : public MiddleMan<A>, public MiddleMan<B> {
private:
    void m_impl (MiddleMan<A>*) override {cout << "MA2" << endl;}
    void m_impl (MiddleMan<B>*) override {cout << "MB2" << endl;}
};
int main()
{
    C* c = new C();
    ((A*)c)->m();      //print "MA2"
    return 0;
}
```

Figure 7.1: The *MiddleMan* approach.

`MiddleMan<B>` instead of `A, B`. In this way, programmers may define new versions of `A.m` and `B.m` in class `C` by providing the corresponding `m_impl` methods. Then in the client code, the method call `((A*)c)->m()` will print out the string `"MA2"`, as expected. Although this workaround can help us defining partial method overrides to a certain extent, the drawbacks are obvious. Firstly, the approach is complex and requires the programmer to fully understand this approach. Moreover, the lack of direct syntax support makes MiddleMan code cumbersome to write. Finally, the approach is ad-hoc, meaning that the class `MiddleMan` shown in Figure 7.1 is not general enough to be used in other cases: more middlemen are needed if partial method overrides happen in other classes; and it is even worse when return types differ.

7.3.2 Traits

Traits and Java's default methods Simplifying the mixins approach, traits [Schärli et al., 2003] draw a strong line between units of reuse and object factories. Traits, as units of reusable code, contain only methods as reusable functionality, ignoring state and state initialization. Classes, as object factories, require functionality from (multiple) traits. Java 8 interfaces are closely related to traits: concrete method implementations are allowed (via the **default** keyword) inside interfaces. The introduction of default methods opens the gate for various flavors of multiple inheritance in Java. Traits offer an algebra of composition operations like sum, alias, and exclusion, providing explicit conflict resolution. Former work [Bono et al., 2014] provides details on mimicking the trait algebra through Java 8 interfaces.

There are also proposals for extending Java with traits. For example, FeatherTrait Java (FTJ) [Liquori and Spiwack, 2008] extends FJ [Igarashi et al., 2001] with statically-typed traits, adding trait-based inheritance in Java. Except for few, mostly syntactic details, their work can be emulated with Java 8 interfaces. There are also extensions to the original trait model, with operations (e.g., renaming [Reppy and Turon, 2006], which breaks structural subtyping) that default methods and interfaces cannot model.

Traits vs. object interfaces. We consider object interfaces an alternative to traits or mixins. In trait model two concepts (traits and classes) coexist and cooperate. Some authors [Bettini et al., 2013] see this as good language design fostering good software development by helping programmers think about the program structures. However, others see the need of both concepts and the absence of state as the drawbacks of this model [Malayeri and Aldrich, 2009]. Object interfaces are units of reuse, and meanwhile provide factory methods for instantiation and support state. Our approach promotes the use of interfaces in order to exploit the modular composition offered by interfaces. Since Java was designed for classes, a direct classless programming style is verbose and unnatural. However, annotation-driven code generation is enough to overcome this difficulty and the resulting programming style encourages modularity, composability and reusability. In that sense, we promote object interfaces as being both units of reuse and object factories. Our practical experience shows that separating the two notions leads to lots of boilerplate code, and is quite limiting when multiple inheritance with state is required. Abstract state operations avoid the key difficulties associated with multiple inheritance and state, while still being quite expressive. Moreover, the ability to support constructors adds expressivity, which is not available in approaches such as Scala's traits/mixins.

7.3.3 Mixins and the Linearization Problem

In the previous chapter (see Section 2.2.3), we mentioned that mixin composition relies on linearization and researchers criticize this issue a lot. In this section, We will discuss what is the linearization and the issue it will cause.

What is Linearization? A mixin can be defined with no or multiple parent mixins, mixins choose an ordering of the composition, for both direct and inherited components. This ordering is important because different orderings will cause different behaviors. There are predefined rules to control the ordering of mixins. For example:

- A mixin itself precedes its parent mixins.
- The local ordering of a mixin is preserved.
- Duplicated mixins are removed from the ordering, keeping the left-most one.

Here is an example of hotpot, illustrating how the linearization is applied in real examples. A `hotpot` is made up of `meat` and `vegetable`, i.e., mixined from mixins `meat` and `vegetable`. Similarly for `meat`, `vegetable`, `pork` and `potato`, they are also composed from other mixins with a certain order.

```
(def hotpot () (meat vegetable))
(def meat () (pork))
(def vegetable () (potato))
(def pork () (food))
(def potato () (food))
(def food () ())
```

The result ordering of parent mixins for `hotpot` is: `(hotpot meat pork vegetable potato food)`. This ordering satisfies all the three rules mentioned above.

Linearization Issues In the above example, all the orderings are compatible. However, when programmers try to mix together mixins with incompatible orderings, conflicts happens. When no ordering of all the mixins can satisfy the constraints, the program is rejected and requires the programmers to fix. For example:

```
(def dinner () (pork hotpot))
```

There is no proper ordering of `dinner`. Because in the ordering constraint of `hotpot`, `hotpot` precedes `pork`, however in the ordering constraint of `dinner`, `pork` precedes `hotpot`.

Mixins are more restricted than the C++ approach. Mixins allow naming components that can be applied to various classes as reusable functionality units. However, the linearization (total ordering) of mixin inheritance cannot provide a satisfactory resolution in some cases and restricts the flexibility of mixin composition. To fight against this limitation, an algebra of mixin operators is introduced [Ancona and Zucca, 2002], but this raises the complexity, especially when constructors and fields are considered [Lagorio et al., 2009]. Scala traits [Odersky et al., 2004] are in fact more like linearized mixins. Scala avoids the object initialization problem by disallowing constructor parameters, causing no ambiguity in cases such as the diamond problem. However, this approach has limited expressiveness and suffers from all the problems of linearized mixin composition. Java interfaces and default methods do not use linearization: the semantics of Java **extends** clause in interfaces is unordered and symmetric.

7.3.4 The CZ Model

Malayeri and Aldrich proposed a model CZ [Malayeri and Aldrich, 2009] which aims to do multiple inheritance without the diamond problem. The design of CZ is based on the intuition that there are relationships between classes that are not captured by inheritance, and that if class hierarchies could express richer interconnections, inheritance diamonds need not exist. Suppose the concrete class `C` extends `A`, as noted by Schärli et al., it is beneficial to recognize that `C` serves two roles: (1) it is a generator of instances, and (2) it is a unit of reuse (through subclassing) [Schärli et al., 2003]. In the first role, inheritance is the implementation strategy and may not be omitted. In the second role, however, it is possible to transform the class hierarchy to one where an inheritance dependency between `C` and `A` is stated and where subclasses of `C` inherit from both `C` and `A`. Here, inheritance is divided into two concepts: inheritance dependency and implementation inheritance. The key distinguishing feature of CZ is this notion of inheritance dependency, because while multiple inheritance is permitted, inheritance diamonds are forbidden.

Using a combination of `requires` and `extends`, a program with diamond inheritance is transformed into one without diamonds. Moreover, fields and multiple inheritance can coexist. However untangling inheritance also untangles the class structure. Thus in CZ, one of the drawbacks is that not only the number of classes, but also the class hierarchy complexity increases. In contrast, IB does not complicate the hierarchical structure, and state also coexists with multiple inheritance.

7.4 Resolving Unintentional Method Conflicts

The above-mentioned models/languages support multiple inheritance, focusing on diamond inheritance. They handle method conflicts in the same way, by simply disallowing two methods with the same signature from two different units to coexist. In contrast, our work in Chapter 5 provides mechanisms that allow methods with the same signatures, but different parents to coexist in a class. Disambiguation is possible in many cases by using both static and dynamic type information during method dispatching. In the cases where real ambiguity exists, **FHJ**'s type system can reject interface definitions and/or method calls statically.

A few language implementations have realized the problem of unintentional conflicts and provided some support for it.

7.4.1 C# Explicit Method Implementations

Explicit method implementations is a special feature supported by C#. As described in C# documentation [Microsoft, 2003], a class that implements an interface can explicitly implement a member of that interface. When a member is explicitly implemented, it can only be accessed through an instance of the interface. Explicit interface implementations allow an interface to inherit multiple interfaces that share the same member names and give each interface member a separate implementation.

Explicit interface member implementations have two advantages. Firstly, they allow interface implementations to be excluded from the public interface of a class. This is particularly useful when a class implements an internal interface that is of no interest to a consumer of that class or struct. Secondly, they allow disambiguation of interface members with the same signature. However, there are two critical differences to **FHJ**: (1) default method implementations are not allowed in C# interfaces; (2) there is only one level of conflicting method implementations at the class that implements the multiple parent interfaces. Further overriding of those methods is not possible in subclasses.

7.4.2 Languages Using Hygienicity

In NextGen/MixGen [Allen et al., 2003], HygJava [Kusmierek and Bono, 2007] and Magda [Bono et al., 2012], *hygienicity* is proposed to deal with unintentional method conflicts. The idea is to give a method a unique identifier by prefixing the name with an unambiguous path. As shown in Figure 7.2, the prefix `HelloWorld` in the method call `(new HelloWorld []).HelloWorld.MainMatter()` is mandatory. So writing programs in these languages is tedious if not supported by a specialized IDE, that aids filling prefix/method information. The advantage of this approach, compared to ours, is that it does not require any additional notion for method dispatching. Indeed the compilation strategy is simple, just by generating conventional code (say in Java or C++) with method names attached with prefixes. Unfortunately, the disadvantage is that some expressive power is lost. In particular, *merging* methods arising from diamond inheritance is not possible because the methods have different prefixes. As shown in Figure 7.3, two methods `m` from different branches `A` and `B` cannot be overridden by the method `m` in `C` because they are regarded as unrelated methods, and `m` in `C` is just another new method that has nothing to do with `A.m` or `B.m`. The reason is that in these hygienic approaches, path names are

```
mixin HelloWorld of Object =
   new Object MainMatter()
   begin
      "Hello world".String.print();
   end;
end;
(new HelloWorld []).HelloWorld.MainMatter();
```

Figure 7.2: Full-qualified name of method calls in Magda.

```
mixin A of Object =
   new String m()
   begin
      return "A";
   end;
end;
mixin B of Object =
   new String m()
   begin
      return "B";
   end;
end;
mixin C of A, B =
   new String m()
   begin
      return "C";
   end;
end;
```

Figure 7.3: Code in Magda.

used to distinguish different methods. In contrast, our model can deal with unintentional conflicts, as well as merged methods because our semantics is not simply based on prefixing. Instead, our model keeps the names of methods unchanged, and our direct operational semantics takes static and dynamic type information into account at runtime when doing method dispatching. Finally, the multiple inheritance model in Magda is based on Mixins, whereas **FHJ** is based on traits. Thus, Magda inherits all limitations of Mixins (such as the linearization problem, etc.).

7.4.3 Hierarchical Dispatch in Self

As we have discussed before, although the mix of static and dynamic dispatch is particularly useful under certain circumstances, it has received little research attention. In the prototype-based language Self [Chambers et al., 1991], inheritance is a basic feature. Self does not include classes but instead allows individual objects to inherit from (or delegate to) other objects. Although it is different from class-based languages, the multiple inheritance model is somewhat similar. The Self language supports multiple (object) inheritance in a clever way. It not only develops the new inheritance relation with *prioritized parents* but also adopts *sender path tiebreaker rule* for method lookup. In Self "*if two slots with the same name are defined in equal-priority parents of the receiver, but only one of the parents is an ancestor or descendant of the object containing the method that is sending the message, then that parent's slot takes precedence over the over parent's slot.*" Similarly to our model, this sender path tiebreaker rule resolves ambiguities between unrelated slots. However, it is used in a prototype-based language setting and it does not support method hierarchical overriding as **FHJ** does.

7.5 Formalization Based on Featherweight Java

Featherweight Java (FJ) [Igarashi et al., 2001] is a minimal core calculus of the Java language, proposed by Igarashi et. al. There are many models built on Featherweight Java, including FeatherTrait [Liquori and Spiwack, 2008], Featherweight defenders [Goetz and Field, 2012], Jx [Nystrom et al., 2004], Featherweight Scala [Cremet et al., 2006], and so on. FJ provides the standard model of formalizing Java-like object-oriented languages and is easily extensible. In terms of formalization, the key novelty of our model is making use of various types (such as parameter types, method return types, etc.) to track the static types as well as the dynamic types during reduction. As far as we know, this technique has not appeared in the literature before. This notion is of vital importance in our hierarchical dispatch algorithm, and it allows for a more precise subject-reduction theorem as discussed in Section 5.3.

7.6 Code Generation Techniques

Automatic generation of getters and setters This is an old idea used in languages such as Self [Ungar and Smith, 1987b], Dart [Dar, 2016] and Newspeak [Bracha et al., 2008]. The programmers specify field signatures and (critically) the intention of storing such information, then the language generates getters and setters. Once state is abstracted away, it is well known that state access can be replaced with computation, but the type of the field stays the same. We do the opposite, the idea of the field is generated starting from signatures of getters and setters. In our approach in Chapter 4, the intention of storing the information is not expressed by the programmer and set in stone but can vary by inheritance. In this case, the underlying type of the field can be changed by our fluid state, and `with` methods provide the right injection from the old type to the new.

7.7 ThisType and MyType

Object interfaces support automatic type-refinement. Type refinement is part of a bigger topic in class-based languages: expressing and preserving type recursion and (nominal/structural) subtyping at the same time. One famous attempt in this direction is *MyType* [Bruce, 1994], representing the type of **this**, changing its meaning along with inheritance. However, when invoking a method with MyType in parameter positions, the exact type of the receiver must be known. This is a big limitation in class-based OO programming and is exasperated by the interface-based programming we propose: no type is ever going to be exact since classes are not explicitly used. A recent article [Saito and Igarashi, 2013] proposes two new features: exact statements and nonheritable methods. Both are related to our work: 1) any method generated inside the `of` method is indeed non-inheritable since there is no class name to extend from; 2) exact statements (a form of wild-card capture on the exact run-time type) could capture the "exact type" of an object even in a class-less environment.

Admittedly, MyType greatly enhances the expressivity and extensibility of object-oriented programming languages. Object interfaces use covariant return types to simulate some uses of MyType. However, this approach only works for refining return types, whereas MyType is more general, as it also works for parameter types. Our approach to covariantly refine state can recover some of the additional expressivity of MyType. As illustrated with our examples, object interfaces are still very useful in many practical applications, yet they do not require additional complexity from the type system.

8 Conclusions and Future Work

The previous chapters have discussed the main parts of the thesis. In this chapter, I will conclude the thesis, discuss potential limitations and discuss possible future work directions.

8.1 Conclusions

In this thesis we have explored the software modularity related issues in Object-Oriented Programming. We showed that multiple inheritance improves software modularity and also discussed the difficulty and problems in realizing multiple inheritance. Especially, we focused on the problems of extensibility, expressiveness, the diamond problem, and method/state conflicts. In this thesis, to solve these problems we have gradually introduced the following works:

EP Trivially After studying the canonical Expression Problem and discussing the importance of solutions to it, we proposed our simple solution using subtyping in both Scala and Java. The solution does not require any advanced language features except for covariant field type refinement (as in Scala) and covariant method return types (as in Java).

We believe that the results of our work provide two important insights. Firstly, while it has been widely believed that statically typed functional languages and OOP languages have equal difficulties in solving the Wadler's EP, our work shows that this is not true. Wadler's EP is in fact simpler to solve in OOP languages due to the native support for subtyping. Since traditional functional languages, such as Haskell or ML, have avoided native support for subtyping a similar solution does not directly apply. Secondly, our work shows that, as a benchmark for extensibility, Wadler's EP is perhaps "too easy". The bar can be set higher by requiring, not only consumer methods, but also binary and producer methods (such as a binary equality operation, or an operation that transforms expressions). We believe that our solution is valuable because of its simplicity and applicability in real-world applications.

Classless Java From Java 8, static and default methods are allowed in interfaces, which enable implementations inside interfaces. An important positive consequence that was probably overlooked is that the concept of class (in Java) is now (almost) redundant and unneeded. We propose a programming style, called Classless Java, where truly object-oriented programs and (reusable) libraries can be defined and used without ever defining a single class.

However, using this programming style directly in Java is very verbose. To avoid syntactic boilerplate caused by Java not being originally designed to work without classes, we introduce the `@Obj` annotation that provides default implementations for various methods (e.g., getters, setters, with-methods) and a mechanism to instantiate objects. We rely on annotation processing and the Lombok library, in this way `@Obj` is just a normal Java library. The `@Obj` annotation helps programmers to write less cumbersome code while coding in Classless Java. In summary, we believe that Classless Java with the `@Obj` annotation is lighter than full Java and it is convenient for programmers to adopt our proposed OO style without classes in their implementations.

FHJ & FHJ+ In these two chapters we have explored the issues related to conflicts in multiple inheritance. Previous approaches either do not support unintentional method conflicts, thus have to compromise between code reuse and type safety, or do not fully support overriding in the presence of unintentional conflicts. **FHJ** and **FHJ+** are proposed as two formalized multiple inheritance models to solve these issues.

To deal with unintentional method conflicts, we introduce two key mechanisms: hierarchical dispatching and hierarchical overriding. Hierarchical dispatching is inspired by the mechanisms in C++. We provide a minimal formal model of hierarchical dispatching in **FHJ**. Such an algorithm makes use of both dynamic type information and static information from either upcasts or parameters' information. It not only offers code reuse and dynamic dispatch but also ensures unambiguity using our hierarchical dispatching algorithm for method resolution. Additionally, we introduce *hierarchical overriding* to allow conflicting methods in different branches to be individually overridden. **FHJ** allows programmers to freely declare, inherit and update unintentional conflicted methods.

As an extension to **FHJ**, **FHJ+** aims at solving the problem of state conflicts. In addition, the way **FHJ+** treats the state conflicts can be seen as a general solution to the diamond problem. For both **FHJ** and **FHJ+**, we all provide the corresponding prototype implementation (in Scala), soundness properties and formal proofs.

FHJ and **FHJ+** are formalized following the style of Featherweight Java and proved to be sound. A prototype interpreter is implemented in Scala. We believe that the formalization of hierarchical dispatching features is general and can be safely embedded in other OO models, so as to have support for the fork inheritance.

8.2 Future Work

Although we have explored the software modularity related issues in Object- Oriented Programming and proposed several solutions to improve them in this thesis, there is still room for doing more tests, exploring new proposals, improving our solutions and prototypes, etc. We will briefly discuss them in the following and provide possible future directions.

8.2.1 Supporting More Scenarios in EP

Although our solution to the EP is a general solution in theory, we have only applied it to two languages: Java and Scala. In the future, we may apply the solution to more languages and scale the solution to larger programs/systems, which may in turn help us dive deeper into the EP, find the limitations, and give us more inspiration on how to improve the solution.

What is more, currently our solution to the EP supports consumer methods well. In terms of future work, one promising direction is to further support producer methods and binary methods. The ideal producer methods should be able to transform one expression to another and the binary methods should have the target (refinable) type in positive positions. To implement such extensions, one might need to create a new language model or modify the semantics in the existing languages, for example, using AST modification techniques in Java or Scala.

Last but not the least, another possible direction is to consider the extensibility in programming, as ASTs of our solution have to be immutable. The approach presented in this thesis lacks of mutability of the sub- expressions: the Scala solution relies on immutable fields; and the Java solution relies on getters. Although we present a variant which supports mutability by allowing the use of generics in Section 3.8, how to support mutability using only subtyping remains to be an interesting problem.

8.2.2 Possible Improvements on Classless Java

As discussed in Section 4.3.3, the current implementation for **Classless Java** has certain limitations. Some possible future work could be:

- Currently our implementation of the **Classless Java** annotation in Lombok only supports the ejc version. In the future, we can also implement the javac version so that users may choose freely.
- Support for separate compilation may be realized by reimplementing `@Obj` referring to `@Delegate` annotation (though this is an experimental feature), or find another annotation processing framework which supports separate compilation naturally.
- Better IDE support on the annotation, including code suggestions, error messages, etc. This would, of course, require much engineering effort.

Although the work of **Classless Java** is presented in Java, the underlying core idea is interface-based programming. Therefore, Java is not the only way to present the idea. One possible future work is to apply IB to more languages to impact users of other languages and also test the idea of IB. Another interesting avenue for future work would be to design a new language based on the idea of IB. With a proper language design, we would not need to restrict ourselves to the limitations of Java and its syntax.

8.2.3 Improving FHJ and FHJ+

Our current model of **FHJ** is the core calculus that contains only the most necessary features to support hierarchical dispatching and overriding. Similarly, the current model of **FHJ+** contains only the core features to support state-related features in addition to **FHJ**. However, to make them realistic languages to use, we should extend the language with more features and support more syntax sugars to make the language richer. As discussed in Section 5.5, there could be orthogonal and non-orthogonal extensions to be added. For example, we may add new features such as static invocation, assignments and static methods in the future.

Besides adding more features, as discussed in Section 3.8, there are lots of features that can be achieved with different designs. For example, in the future we may try to implement the constructors in different ways. We may support fields natively instead of using abstract state operations. Moreover, the future work relates to loosening the models without giving up its soundness, taking some inspirations from the C++ design.

Finally, programming is fun, and exploring the way to better programming is more fun. Actually, besides the future work we mentioned above, there is a large gap between the current programming practice and the ideal way in the future. In this thesis, the problems we have explored are only a small part of it. However, I hope the techniques proposed in this thesis can contribute a little to the programming community and inspire other researchers (including me) for further exploration.

A Appendix

A.1 Classless Java Translation: Formal Semantics

This section presents a formalization of Classless Java, which models the essence of Java interfaces with default methods. This formalization is used to define the semantics of object interfaces.

A.1.1 Syntax

Figure A.1 shows the syntax of Classless Java. The syntax formalizes a minimal subset of Java 8, focusing on interfaces, default methods and object creation literals. There is no syntax for classes. To help readability we use many metavariables to represent identifiers: $\mathtt{C}, \mathtt{x}, \mathtt{f}$ and $\mathtt{m}$; however they all map to a single set of identifiers as in Java. Expressions consist of conventional constructs such as variables (x), method calls ($e\texttt{.}m(\overline{e})$) and static method calls ($I\texttt{.}m(\overline{e})$). For simplicity the degenerate case of calling a static method over the **`this`** receiver is not considered. A more interesting type of expressions is super calls ($I\texttt{.}\mathtt{\textbf{super}}\texttt{.}m(\overline{e})$), whose semantics is to call the (non-static) method m over the **`this`** receiver, but statically dispatching to the version of the method as visible in the interface I. A simple form of field updates (x=e;e') is also modeled. In the syntax of field updates x is expected to be a field name. After updating the field x using the value of e, the expression e' is executed. To blend the statement based nature of Java and the expression based nature of our language, we consider a method body of the form **`return`** x=e;e' to represent x=e;**`return`** e' in Java. Finally, there is an object initialization expression from an interface I, where (for simplicity) all the fields are initialized with a variable present in scope. To be fully compatible with Java, the concrete syntax for an interface declaration with empty supertype list would also omit the **`extends`** keyword. Following standard practice, we consider a global Interface Table (IT) mapping from interface names I to interface declarations $\mathcal{I}$.

The environment Γ is a mapping from variables to types. As usual, we allow a functional notation for Γ to do variable lookup. Moreover, to help us define auxiliary functions, a functional notation is also allowed for a set of methods $\overline{meth}$, using the method name m as a key. That is, we define $\overline{meth}(m) = meth$ iff there is a unique $meth \in \overline{meth}$ whose name is m. For convenience, we define $\overline{meth}(m) =$ None otherwise; moreover $m \in \texttt{dom}(\overline{meth})$ iff $\overline{meth}(m) = meth$. For simplicity, we do not model overloading, thus for an interface to be well formed its methods must be uniquely identified by their names.

A.1.2 Typing

Typing statement $\Gamma \vdash e \in I$ reads "in the environment Γ, expression e has type I.". Before discussing the typing rules we discuss some of the used notation. As a shortcut, we write $\Gamma \vdash e \in I <: I'$ instead of $\Gamma \vdash e \in I$ and $I <: I'$.

We omit the definition of the usual traditional subtyping relation between interfaces, that is the transitive and reflexive closure of the declared **`extends`** relation. The auxiliary notation Γ^{mh} trivially extracts the environment from a method header, by collecting the all types and names of the method parameters. The notation m^{mh} and I^{mh} denotes respectively, extracting the method name and the return type from a method header. $\mathsf{mbody}(m, I)$, defined in Appendix A.1.3, returns the full method declaration as seen by I, that is the method m can be declared in I or inherited from another interface. $\mathsf{mtype}(m, I)$ and $\mathsf{mtypeS}(m, I)$ return the type signature from a method (using $\mathsf{mbody}(m, I)$ internally). $\mathsf{mtype}(m, I)$ is defined only for non static methods, while $\mathsf{mtypeS}(m, I)$ only for static ones. We use $\texttt{dom}(I)$ to denote the set of methods that are defined for type I, that is: $m \in \texttt{dom}(I)$ iff $\mathsf{mbody}(m, I) = meth$.

In Figure A.2 we show the typing rules. We discuss the most interesting rules, that is (T-OBJ) and (T-INTF). Rule (T-OBJ) is the most complex typing rule. Firstly, we need to ensure that all field initializations are type correct, by looking up the type of each variable assigned to a field in the typing environment and verifying that such type is a subtype of the field type. Finally, we check that all method bodies are well-typed. To do this the environment used to check the method body needs to be extended appropriately: we add all fields and their types; add **`this`** : I; and add the arguments (and types) of the respective method. Now we need to check if the object is a valid extension for that specific interface. This can be logically divided into two steps. First we check that all method headers are valid with respect to the corresponding method already present in I:

- $\mathsf{sigvalid}(mh_1 \dots mh_n, \mathrm{I}) =$
  $\forall i \in 1..n \;\; mh_i \, ; <: \mathsf{mbody}(m^{mh_i}, I)$

Here we require that for all newly declared methods, there is a method with the same name defined in the interface I, and that such method is a supertype of the newly introduced one. We define subtyping between methods in a general form that will also be useful later.

- $I \quad m(I_1 x_1 \ldots I_n x_n)$`;` $\quad <: \quad = \quad I <: I'$
  $I' m(I_1 x'_1 \ldots I_n x'_n)$`;`
- $meth <:$ **`default`** mh`{`**`return`** `_;}` $\quad = \quad meth <: mh;$
- **`default`** mh`{`**`return`** `_;}` $<: meth \quad = \quad mh; <: meth$

We allow return type specialization as introduced in Java 5. A method header with return type I is a subtype of another method header with return type I′ if all parameter types are the same, and I <: I′. A default method $meth_1$ is a subtype of another default method $meth_2$ iff mh^{meth_1} is a subtype of mh^{meth_2}. Secondly, we check that all abstract methods (which need to be explicitly overridden) in the interface have been implemented:

- $\mathsf{alldefined}(mh_1 \ldots mh_n, \mathrm{I}) \quad = \quad \forall m$ such that
  $\mathsf{mbody}(m, I) = mh; \exists i \in 1..n \; m^{mh_i} = m$

The rule (T-INTF) checks that an interface *I* is correctly typed. First we check that the body of all default and static methods are well-typed. Then we check that $\mathtt{dom}(I)$ is the same as $\mathtt{dom}(I_1) \cup \ldots \cup \mathtt{dom}(I_n) \cup \mathtt{dom}(\overline{meth})$. This is not a trivial check, since $\mathtt{dom}(I)$ is defined using mbody, which would be undefined in many cases: notably if a method $meth \in \overline{meth}$ is not compatible with some method in $\mathtt{dom}(I_1) \ldots \mathtt{dom}(I_n)$ or if there are methods in any $\mathtt{dom}(I_i)$ and $\mathtt{dom}(I_j)$ $(i, j \in 1..n)$ conflict.

A.1.3 Auxiliary Definitions

Defining mbody is not trivial, and requires quite a lot of attention to the specific model of Java interfaces, and to how it differs w.r.t. Java Class model. $\mathsf{mbody}(m, I)$ denotes the actual method *m* (body included) that interface *I* owns. The method can either be defined originally in *I* or in its supertypes, and then passed to *I* via inheritance.

- $\mathsf{mbody}(m, I_0) \quad = \quad \mathsf{override}(\overline{meth}(m), \mathsf{needed}(m, \bar{I}))$
  with $IT(I_0) = \; ann$ **`interface`** I_0 **`extends`** $I_1 \ldots I_n$
  `{` $\overline{meth}$ `}` and $I \in \bar{I}$ if $I_i <: I, i \in 1..n$

The definition of mbody reconstructs the full set of supertypes $\bar{I}$ and then delegates the work to two other auxiliary functions: $\mathsf{needed}(m, \bar{I})$ and $\mathsf{override}(meth, \overline{meth})$.

needed recovers from the interface table only the "needed" methods, that is, the non-static ones that are not reachable by another, less specific superinterface. Since the second parameter of needed is a set, we can choose an arbitrary element to be I_0. In the definition we denote by $\mathsf{originalMethod}(m, I) = meth$ the non-static method called *m* defined directly in *I*. Formally:

- $\mathsf{originalMethod}(m, I_0) \;=\; meth$
  with $IT(I_0) = ann$ **interface** I_0 **extends** $\bar{I}$`{` $\overline{meth}$ `}`,
  $meth \in \overline{meth}$ not static, $m = m^{meth}$
- $\mathsf{originalMethod}(m, I_0) \quad \in \;=$
  $\mathsf{needed}(m, I_0 \ldots I_n)$
  $\nexists i \in 1..n$ such that $\mathsf{originalMethod}(m, I_i)$ is defined
  and $I_i <: I_0$

override models how a method in an interface can override implementations in its super-interfaces, even in the case of conflicts. Note how the special value None is used, and how (the 5th case) overriding can solve a conflict.

- $\mathsf{override}(\mathsf{None}, \emptyset) \;=\; \mathsf{None}$
- $\mathsf{override}(meth, \emptyset) \;=\; meth$
- $\mathsf{override}(\mathsf{None}, meth) \;=\; meth$
- $\mathsf{override}(\mathsf{None}, \overline{mh;}) \;=\; \mathsf{mostSpecific}(\overline{mh;})$
- $\mathsf{override}(meth, \overline{meth}) \;=\; meth$
  with $\forall meth' \in \overline{meth} : meth <: meth'$

The definition mostSpecific returns the most specific method whose type is the subtype of all the others. Since method subtyping is a partial ordering, mostSpecific may not be defined, this in turn forces us to rely on the last clause of override; otherwise the whole mbody would not be defined for that specific m. Rule (T-INTF) relies on this behavior.

- $\mathsf{mostSpecific}(\overline{meth}) \;=\; meth$
  with $meth \in \overline{meth}$ and $\forall meth' \in \overline{meth} : meth <: meth'$

To illustrate the mechanism of mbody, we present an example. We compute mbody(`m`, `D`):

```
interface A { Object m(); }
interface B extends A { default Object m() {return this.m();} }
interface C extends A {}
interface D extends B, C { String m(); }
```

- First {`A`,`B`,`C`}, the full set of supertypes of `D` is obtained.
- Then we compute needed(`m`, {`A`,`B`,`C`}) = `default Object m(){...}`, that is `B.m`. That is, we do not consider either `C.m` (since `m` is not declared directly in `C`, hence originalMethod(`m`, `C`) is undefined) or `B.m` (that is a subtype of `A`, thus `B.m` hides `A.m`).
- The final step computes override(`D.m`, `B.m`) = `D.m`, by the last case of override we get that `D.m` hides `B.m` successfully (`String` is a subtype of `Object`). Finally we get mbody(`m`, `D`) = `D.m`.

e	$::=$	$x \mid e.m(\overline{e}) \mid I.m(\overline{e}) \mid I.\texttt{super}.m(\overline{e}) \mid x\texttt{=}e\texttt{;}e' \mid obj$	expressions
obj	$::=$	$\texttt{new}\ I()\{\ \overline{field}\ mh_1\{\ \texttt{return}\ e_1\texttt{;}\}\ \ldots\ mh_n\{\ \texttt{return}\ e_n\texttt{;}\}\}$	object creation
$field$	$::=$	$I\ f\texttt{=}x\texttt{;}$	field declaration
$\mathcal{I}$	$::=$	$ann\ \texttt{interface}\ I\ \texttt{extends}\ \overline{I}\{\ \overline{meth}\}$	interface declaration
$meth$	$::=$	$\texttt{static}\ mh\{\ \texttt{return}\ e\texttt{;}\} \mid \texttt{default}\ mh\{\ \texttt{return}\ e\texttt{;}\} \mid mh\texttt{;}$	method declaration
mh	$::=$	$I_0\ m\ (I_1\ x_1 \ldots I_n\ x_n)$	method header
ann	$::=$	$\texttt{@Obj} \mid \emptyset$	annotations
Γ	$::=$	$x_1{:}I_1 \ldots x_n{:}I_n$	environment

Figure A.1: Grammar of Classless Java

(T-INVK)
$$\frac{\Gamma \vdash e \in I_0 \qquad \forall i \in 1..n\ \ \Gamma \vdash e_i \in \_ <: I_i \qquad \mathsf{mtype}(m, I_0) = I_1 \ldots I_n \to I}{\Gamma \vdash e.m(e_1 \ldots e_n) \in I}$$

(T-STATICINVK)
$$\frac{\forall i \in 1..n\ \ \Gamma \vdash e_i \in \_ <: I_i \qquad \mathsf{mtypeS}(m, I_0) = I_1 \ldots I_n \to I}{\Gamma \vdash I_0.m(e_1 \ldots e_n) \in I}$$

(T-SUPERINVK)
$$\frac{\Gamma(\texttt{this}) <: I_0 \qquad \forall i \in 1..n\ \ \Gamma \vdash e_i \in \_ <: I_i \qquad \mathsf{mtype}(m, I_0) = I_1 \ldots I_n \to I}{\Gamma \vdash I_0.\texttt{super}.m(e_1 \ldots e_n) \in I}$$

(T-VAR)
$$\frac{\Gamma(x) = I}{\Gamma \vdash x \in I}$$

(T-OBJ)
$$\frac{\begin{array}{c}\forall i \in 1..k\ \ \Gamma(x_i) <: I_i \\ \forall i \in 1..n\ \ \Gamma, f_1{:}I_1, \ldots, f_k{:}I_k, \texttt{this}{:}I, \Gamma^{mh_i} \vdash e_i \in \_ <: I^{mh_i} \\ \mathsf{sigvalid}(mh_1 \ldots mh_n, \mathrm{I}) \qquad \mathsf{alldefined}(mh_1 \ldots mh_n, \mathrm{I})\end{array}}{\Gamma \vdash \texttt{new}\ I()\{\ I_1\ f_1\texttt{=}x_1\texttt{;} \ldots I_k\ f_k\texttt{=}x_k\texttt{;}\ mh_1\{\ \texttt{return}\ e_1\texttt{;}\} \ldots mh_n\{\ \texttt{return}\ e_n\texttt{;}\}\} \in I}$$

(T-UPDATE)
$$\frac{\Gamma \vdash e \in \_ <: \Gamma(x) \qquad \Gamma \vdash e' \in I}{\Gamma \vdash x\texttt{=}e\texttt{;}e' \in I}$$

(T-INTF)
$$\frac{\begin{array}{c}\mathsf{IT}(I) = ann\ \texttt{interface}\ I\ \texttt{extends}\ I_1 \ldots I_n\{\ \overline{meth}\} \\ \forall \texttt{default}\ mh\{\ \texttt{return}\ e\texttt{;}\} \in \overline{meth},\ \ \Gamma^{mh}, \texttt{this}{:}I \vdash e \in \_ <: I^{mh} \\ \forall \texttt{static}\ mh\{\ \texttt{return}\ e\texttt{;}\} \in \overline{meth},\ \ \Gamma^{mh} \vdash e \in \_ <: I^{mh} \\ \mathsf{dom}(I) = \mathsf{dom}(I_1) \cup \ldots \cup \mathsf{dom}(I_n) \cup \mathsf{dom}(\overline{meth})\end{array}}{I\ \mathrm{OK}}$$

Figure A.2: Typing rules of Classless Java

A.2 Classless Java: What `@Obj` Generates

This section gives an overview of what `@Obj` generates, and what formal properties are guaranteed in the translation.

We formalize syntax and typing for Classless Java in Appendix A.1, which models the essence of Java interfaces with default methods. Classless Java is just a proper subset of Java 8, so it is easy to understand the translation presented in this section without the syntax and typing rules of Classless Java. Since the formalized part of Classless Java does not consider casts or `instanceof`, the **`with`** method is not included in the formal translation. For the same reason `void` returning setters are not included, since they are just a minor variation over the more interesting fluent setters and they would require special handling just for the

- $[\![\texttt{@Obj}\ \textbf{interface}\ I_0\ \textbf{extends}\ \overline{I}\{\ \overline{meth}\}]\!] \;=\; [\![\texttt{@ObjOf}\ \textbf{interface}\ I_0\ \textbf{extends}\ \overline{I}\{\ \overline{meth}\ \overline{meth}'\}]\!]$
  with $\overline{meth}' = \mathsf{refine}(I_0, \overline{meth})$
- $[\![\texttt{@ObjOf}\ \textbf{interface}\ I_0\ \textbf{extends}\ \overline{I}\{\ \overline{meth}\}]\!] \;=\; \textbf{interface}\ I_0\ \textbf{extends}\ \overline{I}\{\ \overline{meth}\ \mathsf{ofMethod}(I_0)\}$
  with $\mathsf{valid}(I_0)$, $\texttt{of} \notin \texttt{dom}(I_0)$

Figure A.3: The translation functions of `@Obj` and `@ObjOf`.

- $I_0\ \textbf{with\#}m(I\ \texttt{\_val})\texttt{;} \in \mathsf{refine}(I_0, \overline{meth}) \qquad =$
  $\mathsf{isWith}(\mathsf{mbody}(\textbf{with\#}m, I_0), I_0),\ \textbf{with\#}m \notin \texttt{dom}(\overline{meth})$
- $I_0\ \texttt{\_}m(I\ \texttt{\_val})\texttt{;} \in \mathsf{refine}(I_0, \overline{meth}) \qquad =$
  $\mathsf{isSetter}(\mathsf{mbody}(\texttt{\_}m, I_0), I_0),\ \texttt{\_}m \notin \texttt{dom}(\overline{meth})$
- $\mathsf{valid}(I_0) \;=\; \forall m \in \texttt{dom}(I_0),\ \text{if}\ mh\texttt{;} = \mathsf{mbody}(m, I_0),$
  one of the following cases is satisfied:
  $\mathsf{isField}(meth)$, $\mathsf{isWith}(meth, I_0)$ or $\mathsf{isSetter}(meth, I_0)$
- $\mathsf{isField}(I\ m()\texttt{;}) \;=\; \text{not}\ \mathsf{special}(m)$
- $\mathsf{isWith}(I'\ \textbf{with\#}m(I\ x)\texttt{;}, I_0) =$
  $I_0 <: I', \mathsf{mbody}(m, I_0) = I\ m()\texttt{;}$ and not $\mathsf{special}(m)$
- $\mathsf{isSetter}(I'\ \texttt{\_}m(I\ x)\texttt{;}, I_0) \;=$
  $I_0 <: I', \mathsf{mbody}(m, I_0) = I\ m()\texttt{;}$ and not $\mathsf{special}(m)$

Figure A.4: The refine and valid functions and auxiliary functions

conventional `void` type. Since our properties are about preserving typing, we do not need to formalize Classless Java semantics to prove our statements.

A.2.1 Translation

For the purposes of the formalization, the translation is divided into two parts for more convenient discussion on formal properties later. To this aim we introduce the annotation `@ObjOf`. Its role is only in the translation process, hence is not part of the Classless Java language. `@ObjOf` generates the constructor method `of`, while `@Obj` automatically refines the return types and calls `@ObjOf`.

Figure A.3 presents the translation. In the first function, `@Obj` injects refined methods to interface I_0. The second function, `@ObjOf` invokes $\mathsf{ofMethod}(I_0)$ and generates the `of` method for I_0, if such a method does not exist in its domain, and all the abstract methods are valid for the annotation.

Figure A.4 presents more details on the auxiliary functions. The first two points of Figure A.4 define function refine. This function generates unimplemented **`with-`** and fluent setters in the interface, where the return types have been refined. To determine whether a method needs to be generated, we check if such **`with-`** or setter methods require an implementation in I_0, but are not declared directly in I_0. The third point gives the definition of

- ofMethod$(I_0) =$ `static` I_0 `of(`I_1 `_`m_1 `, ...` I_n `_`m_n`){`
  `return new` I_0`(){`
  $I_1\ m_1 =$ `_`m_1`; ...` $I_n\ m_n =$ `_`m_n`;`
  $I_1\ m_1$`(){`**`return`** m_1`;} ...` $I_n\ m_n$`(){`**`return`** m_n`;}`
  withMethod$(I_1, m_1, I_0, \bar{e}_1)\ldots$withMethod$(I_n, m_n, I_0, \bar{e}_n)$
  setterMethod$(I_1, m_1, I_0)\ldots$setterMethod(I_n, m_n, I_0)
  `};}`
  with $I_1\ m_1$`();, ...` $I_n\ m_n$`();` $=$ fields(I_0)
  and $\bar{e}_i = m_1$`, ... ,`m_{i-1}`,_val,`m_{i+1}`, ... ,`m_n
- *meth* $\in$ fields(I_0) $=$
  isField(*meth*) and *meth* $=$ mbody(m^{meth}, I_0)
- withMethod$(I, m, I_0, \bar{e})$ $=$
  I_0 **`with#`**m`(`I `_val){` **`return`** I_0`.of(`$\bar{e}$`);}`
  with mbody(**`with#`**m, I_0) having the form mh;
- withMethod$(I, m, I_0, \bar{e})$ $=$ $\emptyset$ otherwise
- setterMethod(I, m, I_0) $=$
  I_0 `_`m`(`I `_val){` m`= _val;`**`return this`**`;}`
  with mbody(`_`m, I_0) having the form mh;
- setterMethod(I, m, I_0) $=$ $\emptyset$ otherwise

Figure A.5: The generated `of` method and auxiliary functions.

valid: it is valid to annotate an interface if all abstract methods (that is, all those requiring an implementation) are valid. That is, we can categorize them in a pattern that we know how to implement (right column): it is either a field getter (first point), a with method (second point) or a setter (third point). Note that we write **`with#`**m to append m to **`with`**, following the camelCase rule. The first letter of m must be lower-case and is changed to upper-case upon appending. For example **`with#`**`foo=withFoo`. Special names special(m) are **`with`** and all identifiers of the form **`with#`**m.

Figure A.5 defines the ofMethod function, which generates the static method `of` as an object factory. It detects all the field methods of I_0 and use them to synthesize its arguments. The return statement instantiates an anonymous class which generates the needed getters, fluent setters and with-methods. The right column first point collects the getter methods, the second and third point generate implementations for `with-` methods if needed; similarly, the fourth and fifth point generate fluent setters if needed.

Some other features of `@Obj`, including non-fluent setters and the **`with`** method are not formalized here. Appendix A.3.2 gives a detailed but informal explanation of generation for those methods.

A.2.2 Results

Classless Java provides some guarantees regarding the generated code. Essentially, Classless Java ensures the *self coherence* and *client coherence* properties informally introduced in Section 4.3. Furthermore, we can show that *if there are no type-refinements*, then *heir coherence* also holds. The result about heir coherence is possible to prove because the translation is split into two parts. In essence heir coherence is a property of the translation of `@ObjOf`, but not of `@Obj`.

To formally characterize the behavior of our annotation and the two levels of guarantees that we offer, we provide some notations and two theorems:

- We denote with $I^{\mathcal{J}}$ and $\mathtt{m}^{meth}$ the name of an interface and of a method.
- An interface table IT is OK if under such interface table, all interfaces are OK, that is, well typed.
- Since interface tables are just represented as sequences of interfaces we write IT = $\mathcal{J}$ IT' to select a specific interface in a table.
- IT contains an heir of *I* if there is an interface that extends it, or a **`new`** that instantiates it.

Theorem 1 (@ObjOf)**.** *If a given interface table $\mathcal{J}$ IT is OK where $\mathcal{J}$ has `@ObjOf`, valid($I^{\mathcal{J}}$) and `of` $\notin$ dom($I^{\mathcal{J}}$), then the interface table $[\![\mathcal{J}]\!]$ IT is OK.*

Theorem 2 (@Obj)**.** *If a given interface table $\mathcal{J}$ IT is OK where $\mathcal{J}$ has `@Obj`, valid($I^{\mathcal{J}}$) and `of` $\notin$ dom($I^{\mathcal{J}}$), and there is no heir of $I^{\mathcal{J}}$, then the interface table $[\![\mathcal{J}]\!]$ IT is OK.*

Informally, the theorems mean that for a client program that type-checks before the translation is applied, if the annotated type has no subtypes and no objects of that type are created, then type safety of the generated code is guaranteed after the successful translation.

The second step of `@Obj`, namely what `@ObjOf` does in the formalization, is guaranteed to be type-safe for the three kinds of coherence by the `@ObjOf` theorem. The `@Obj` theorem is more interesting: since `@Obj` does not guarantee heir coherence, we explicitly exclude the presence of heirs. In this way the `@Obj` theorem guarantees only self and client coherence. The formal theorem proofs are available in Appendix A.4.

Type preservation Note that we preferred to introduce self, client and heir coherence instead of referring to conventional type preservation theorems. The reason is to better model how our approach behaves in a object-oriented software ecosystem with inheritance, where only some units may be translated/expanded. Note inheritance's crucial influence in heir coherence. Our formulation of client coherence allows us to discuss intermediate stages where only some code units are translated/expanded. Conventional type preservation refers only to completely translated program. Our coherence guarantees mean that developers and designers of Java libraries and frameworks can start using IB (and our `@Obj` annotation) in the evolution of their products and still retain backward compatibility with their clients.

A.3 Formal Definition of the Generated Methods by `@Obj`

This section presents a formal definition of the generated methods by `@Obj`.

A.3.1 Translation

The translation functions of `@Obj` and `@ObjOf` are presented in Figure A.3. Note that it is necessary to explicitly check if the interface is valid for annotation:

- $\mathsf{valid}(I_0) \;=\; \forall m \in \mathtt{dom}(I_0),$ if $mh; = \mathsf{mbody}(m, I_0)$,
  one of the following cases is satisfied:
  $\mathsf{isField}(meth)$, $\mathsf{isWith}(meth, I_0)$ or $\mathsf{isSetter}(meth, I_0)$
- $\mathsf{isField}(I\ m()\,;) \quad = \quad \text{not } \mathsf{special}(m)$
- $\mathsf{isWith}(I'\ \mathtt{\textbf{with\#}}m(I\,x)\,;, I_0) =$
  $I_0 <: I', \mathsf{mbody}(m, I_0) = I\ m()\,;$ and not $\mathsf{special}(m)$
- $\mathsf{isSetter}(I'\ \_m(I\,x)\,;, I_0) \quad =$
  $I_0 <: I', \mathsf{mbody}(m, I_0) = I\ m()\,;$ and not $\mathsf{special}(m)$

That is, we can categorize all abstract methods in a pattern that we know how to implement: it is either a field getter, a with method or a setter.

Moreover, we check that the method `of` is not already defined by the user. In the formalization an existing definition of the `of` method is an error. However, in the prototype (which also needs to account for overloading), the check is more complex as it just checks that an `of` method with the same signature of the one being generated is not already present.

We write **`with#`**m to append m to **`with`**, following the camelCase rule. The first letter of m must be lower-case and is changed to upper-case upon appending. For example **`with#`**`foo=withFoo`. Special names $\mathsf{special}(m)$ are **`with`** and all identifiers of the form **`with#`**m.

The refine function: $\mathsf{refine}(I_0, \overline{meth})$ is defined as follows:

- I_0 $\texttt{with\#}m\texttt{(}I$ $\texttt{\_val);}$ $\in$ $=$
  $\mathsf{refine}(I_0, \overline{meth})$
  $\mathsf{isWith}(\mathsf{mbody}(\texttt{with\#}m, I_0), I_0),\ \texttt{with\#}m \notin \texttt{dom}(\overline{meth})$
- $I_0\ \texttt{\_}m\texttt{(}I\ \texttt{\_val);} \in \mathsf{refine}(I_0, \overline{meth})$ $=$
  $\mathsf{isSetter}(\mathsf{mbody}(\texttt{\_}m, I_0), I_0),\ \texttt{\_}m \notin \texttt{dom}(\overline{meth})$

The methods generated in the interface are **`with-`** and setters. The methods are generated when they are unimplemented in I_0, because the return types need to be refined. To determine whether the methods need to be generated, we check if such **`with-`** or setter methods are required by I_0, but not declared directly in I_0.

The ofMethod function: The function ofMethod generates the method `of`, as an object factory. To avoid boring digressions into well-known ways to find unique names, we assume that all methods with no parameters do not start with an underscore, and we prefix method names with underscores to obtain valid parameter names for `of`.

- $\mathsf{ofMethod}(I_0) =$ $\texttt{static}\ I_0\ \texttt{of(}I_1\ \texttt{\_}m_1\texttt{,}\ \ldots I_n\ \texttt{\_}m_n\texttt{)\{}$
  $\textbf{\texttt{return new}}\ I_0\texttt{()\{}$
  $I_1\ m_1 = \texttt{\_}m_1\texttt{;}\ \ldots I_n\ m_n = \texttt{\_}m_n\texttt{;}$
  $I_1\ m_1\texttt{()\{}\textbf{\texttt{return}}\ m_1\texttt{;\}}\ \ldots I_n\ m_n\texttt{()\{}\textbf{\texttt{return}}\ m_n\texttt{;\}}$
  $\mathsf{withMethod}(I_1, m_1, I_0, \overline{e}_1) \ldots \mathsf{withMethod}(I_n, m_n, I_0, \overline{e}_n)$
  $\mathsf{setterMethod}(I_1, m_1, I_0) \ldots \mathsf{setterMethod}(I_n, m_n, I_0)$
  $\texttt{\};\}}$
  with $I_1\ m_1\texttt{();,}\ldots I_n\ m_n\texttt{();} = \mathsf{fields}(I_0)$
  and $\overline{e}_i = m_1\texttt{,}\ldots\texttt{,}m_{i-1}\texttt{,\_val,}m_{i+1}\texttt{,}\ldots\texttt{,}m_n$

Note that, the function $\mathsf{fields}(I_0)$ denotes all the fields in the current interface:

- $meth \in \mathsf{fields}(I_0)$ $=$
  $\mathsf{isField}(meth)$ and $meth = \mathsf{mbody}(\mathsf{m}^{meth}, I_0)$

For methods inside the interface with the form $I_i\ m_i\texttt{();}$

- m_i is the field name, and has type I_i.
- $m_i\texttt{()}$ is the getter and just returns the current field value.
- if a method $\texttt{with\#}m_i()$ is required, then it is implemented by calling the `of` method using the current value for all the fields except for m_i. Such new value is provided as a parameter. This corresponds to the expressions $\overline{e}_i$.
- $\texttt{\_}m_i\texttt{(}I_i\ \texttt{\_val)}$ is the setter. In our prototype we use name m_i, here we use the underscore to avoid modeling overloading.

The auxiliary functions are defined below. Note that we do not need to check if some header is a subtype of what we would generate, this is ensured by $\mathsf{valid}(I_0)$.

- $\mathsf{withMethod}(I, m, I_0, \overline{e}) \quad =$
  I_0 **`with#`**m`(`I `_val){` **`return`** I_0`.of(`$\overline{e}$`);}`
  with $\mathsf{mbody}($**`with#`**$m, I_0)$ having the form mh;
- $\mathsf{withMethod}(I, m, I_0, \overline{e}) \quad = \quad \emptyset$ otherwise
- $\mathsf{setterMethod}(I, m, I_0) \quad =$
  I_0 `_`m`(`I `_val){` m`= _val;`**`return this`**`;}`
  with $\mathsf{mbody}(\_m, I_0)$ having the form mh;
- $\mathsf{setterMethod}(I, m, I_0) \quad = \quad \emptyset$ otherwise

A.3.2 Other Features

We do not formally model non-fluent setters and the **`with`** method. An informal explanation of how those methods are generated is given next:

- For methods inside the interface with the form `void` m`(`$I\,x$`);`:
  - Check if method $I\,m$`();` exists. If not, generate error (that is, $\mathsf{valid}(I_0)$ is false).
  - Generate the implemented setter method inside `of`:
    `public` `void` m`(`I `_val){` m`=_val;}`
    There is no need to refine the return type for non-fluent setters, thus we do not need to generate the method header in the interface body itself.
- For methods with the form I' **`with`**`(`$I\,x$`);`:
  - I must be an interface type (no classes or primitive types).
  - As before, check that I' is a supertype of the current interface type I_0.
  - Generate implemented **`with`** method inside `of`:
    `public` I_0 **`with`**`(`I `_val){`
      `if``(_val instanceof` I_0`){`**`return`** `(`I_0`)_val;}`
      `return` I_0`.of(`$e_1 \ldots e_n$`);}`
    with $e_i =$`_val.`m_i`()` if I has a m_i`()` method where $m_1 \ldots m_n$ are fields of I_0; otherwise $e_i = m_i$.
  - If needed, as for **`with-`** and setters, generate the method headers with refined return types in the interface.

A.4 Classless Java Lemmas and Theorems

A.4.1 LEMMA 1 and Proof

Lemma 1 (a)**.** *For any expression* $\mathtt{e}$ *under an interface table* $\mathcal{I}$ *IT where* $\Gamma \vdash \mathtt{e} \in I^{\mathcal{I}}$, $\mathcal{I}$ *has* `@ObjOf` *annotation and* $[\![\mathcal{I}]\!] = \mathcal{I}'$, *then under the interface table* $\mathcal{I}'$ *IT,* $\Gamma \vdash \mathtt{e} \in I^{\mathcal{I}}$.

Proof. By induction on the typing rules: by the grammar shown in Figure A.1, there are 6 cases for an arbitrary expression $\mathtt{e}$:

- Variables are typed in the same exact way.
- Field update. The type preservation is ensured by induction.
- A method call (normal, static or super). The corresponding method declaration won't be "removed" by the translation, also the types remain unchanged. The only work `@ObjOf` does is adding a static method `of` to the interface, however, a pre-condition of the translation is $\mathtt{of} \notin \mathtt{dom}(I^{\mathcal{I}})$, so adding `of` method has no way to affect any formerly well typed method call.
- An object creation. Adding the `of` method doesn't introduce unimplemented methods to an interface, moreover, the static method is not inheritable, hence after translation such an object creation still type checks and has the right type by induction.

□

Lemma 1 (b)**.** *For any expression* $\mathtt{e}$ *under an interface table* $\mathcal{I}$ *IT where there is no heir of* $I^{\mathcal{I}}$, $\Gamma \vdash \mathtt{e} \in I^{\mathcal{I}}$, $\mathcal{I}$ *has* `@Obj` *annotation and* $[\![\mathcal{I}]\!] = \mathcal{I}'$, *then under the interface table* $\mathcal{I}'$ *IT,* $\Gamma \vdash \mathtt{e} \in \_ <: I^{\mathcal{I}}$.

Proof. The proof follows the same scheme of Lemma 1 (a), but for the case of method call the return type may be refined with a subtype. This is still ok since we require $\_ <: I^{\mathcal{I}}$. On the other side, this weaker result still allows the application on the method call typing rules, since in the premises the types of the actual parameter are required to be a subtype of the formal one. □

A.4.2 LEMMA 2 and Proof

Lemma 2 (a)**.** *If* $\mathcal{I}$ *has* `@ObjOf` *annotation and* $I^{\mathcal{I}}$ *OK in* $\mathcal{I}$ *IT, then* $[\![\mathcal{I}]\!]$ *OK in* $[\![\mathcal{I}]\!]$ *IT.*

Proof. By the rule (T-INTF) in Figure A.2, we divide the proof into two parts.

Part I. For each default or static method in the domain of $[\![I^{\mathcal{J}}]\!]$, the type of the return value is compatible with the method's return type.

Since $\mathcal{J}$ OK, and by Lemma 1 (a), all the existing default and static methods are well typed in $[\![\mathcal{J}]\!]$, except for the new method `of`. It suffices to prove that it still holds for $\mathsf{ofMethod}(I)$.

By the definition of $\mathsf{ofMethod}(I)$, the return value is an object

$$\texttt{return new}\ I^{\mathcal{J}}\texttt{()\{ ... \}}$$

To prove it is of type $I^{\mathcal{J}}$, we use the typing rule (T-Obj).

- All field initializations are type correct. By the definition of $\mathsf{ofMethod}(I^{\mathcal{J}})$ in Appendix A.3.1, the fields $\mathtt{m}_1, \ldots, \mathtt{m}_n$ are initialized by `of`'s arguments, and types are compatible.

- All method bodies are well-typed.
  - Typing of the $\mathtt{i}$-th getter $\mathtt{m}_\mathtt{i}$.

    $$\Gamma, \mathtt{m}_\mathtt{i} : I_\mathtt{i}, \texttt{this} : I^{\mathcal{J}} \vdash \mathtt{m}_\mathtt{i} \in I_\mathtt{i}$$

    We know that $I_\mathtt{i} = I^{mh_\mathtt{i}}$ since the $\mathtt{i}$-th getter has its return type the same as the corresponding field $\mathtt{m}_\mathtt{i}$.
  - Typing of the `with-` method of an arbitrary field $\mathtt{m}_\mathtt{i}$. By Appendix A.3.1, if the `with-` method of $\mathtt{m}_\mathtt{i}$ is well-defined, it has the form

    $$I^{\mathcal{J}}\ \texttt{with\#}m_\mathtt{i}\texttt{(}I_\mathtt{i}\ \texttt{\_val)\{ return}\ I^{\mathcal{J}}\texttt{.of(}\overline{e}_\mathtt{i}\texttt{);\}}$$

    $\overline{e}_\mathtt{i}$ is obtained by replacing $\mathtt{m}_\mathtt{i}$ with `_val` in the list of fields, and since they have the same type $I_\mathtt{i}$, the arguments $\overline{e}_\mathtt{i}$ are compatible with $I^{\mathcal{J}}$`.of` method. Hence

    $$\Gamma, \mathtt{m}_1 : I_1 \ldots \mathtt{m}_n : I_n, \texttt{this} : I^{\mathcal{J}},\ \texttt{\_val} : I_\mathtt{i} \vdash I^{\mathcal{J}}\texttt{.of(}\overline{e}_\mathtt{i}\texttt{)} \in I^{\mathcal{J}}$$

    We know that $I^{\mathcal{J}} = I^{mh_\mathtt{i}}$ by the return type of $\texttt{with\#}m_\mathtt{i}$ shown as above.
  - Typing of the $\mathtt{i}$-th setter $\_\mathtt{m}_\mathtt{i}$. If the $\_\mathtt{m}_\mathtt{i}$ method is well-defined, it has the form

    $$I^{\mathcal{J}}\ \texttt{\_}m_\mathtt{i}\texttt{(}I_\mathtt{i}\ \texttt{\_val)\{}\ m_\mathtt{i}\texttt{= \_val;return this;\}}$$

By (T-UPDATE), the assignment "m_i`= _val;`" is correct since m_i and `_val` have the same type I_i, and the return type is decided by **`this`**.

$$\Gamma, \mathtt{\textbf{this}} : I^{\mathcal{J}},\ \mathtt{\_val} : I_i \vdash \mathtt{\textbf{this}} \in I^{\mathcal{J}}$$

We know that $I^{\mathcal{J}} = I^{mh_i}$ by the return type of $\_m_i$ shown as above.

- All method headers are valid with respect to the domain of $I^{\mathcal{J}}$. Namely

$$\mathsf{sigvalid}(mh_1 \ldots mh_n, \mathrm{I})$$

For convenience, we use "*meth* in $\mathsf{ofMethod}(I^{\mathcal{J}})$" to denote that *meth* is one of the implemented methods in the return expression of $\mathsf{ofMethod}(I^{\mathcal{J}})$, namely **`new`** $I^{\mathcal{J}}$`()` `{...}`.

  - For the i-th getter m_i,

$$\begin{array}{rl} & I_i\ m_i\mathtt{()\{...\}}\ \text{ in } \mathsf{ofMethod}(I^{\mathcal{J}}) \\ \text{implies} & I_i\ m_i\mathtt{();} \in \mathsf{fields}(I^{\mathcal{J}}) \\ \text{implies} & I_i\ m_i\mathtt{();} = \mathsf{mbody}(m_i, I^{\mathcal{J}}) \\ \text{implies} & I_i\ m_i\mathtt{();} <: \mathsf{mbody}(m_i, I^{\mathcal{J}}) \end{array}$$

  - For the **`with#`**m_i method,

$$\begin{array}{rl} & I^{\mathcal{J}}\ \mathtt{\textbf{with\#}}m_i(I_i\ \mathtt{\_val)\{...\}}\ \text{ in } \mathsf{ofMethod}(I^{\mathcal{J}}) \\ \text{implies} & \mathsf{mbody}(\mathtt{\textbf{with\#}}m_i, I^{\mathcal{J}}) \text{ is of form } mh; \\ \text{with} & \mathsf{valid}(I^{\mathcal{J}}) \\ \text{implies} & \mathsf{isWith}(\mathsf{mbody}(\mathtt{\textbf{with\#}}m_i, I^{\mathcal{J}}), I^{\mathcal{J}}) \\ \text{implies} & I^{\mathcal{J}}\ \mathtt{\textbf{with\#}}m_i(I_i\ \mathtt{\_val);} <: \mathsf{mbody}(\mathtt{\textbf{with\#}}m_i, I^{\mathcal{J}}) \end{array}$$

– For the $\mathtt{i}$-th setter $\_m_{\mathtt{i}}$,

$$\begin{array}{rl} & I^{\mathcal{J}}\ \_m_{\mathtt{i}}(I_{\mathtt{i}}\ \_\mathtt{val})\{\ldots\}\ \text{in ofMethod}(I^{\mathcal{J}}) \\ \text{implies} & \mathsf{mbody}(\_m_{\mathtt{i}}, I^{\mathcal{J}})\ \text{is of form}\ mh; \\ \text{with} & \mathsf{valid}(I^{\mathcal{J}}) \\ \text{implies} & \mathsf{isSetter}(\mathsf{mbody}(\_m_{\mathtt{i}}, I^{\mathcal{J}}), I^{\mathcal{J}}) \\ \text{implies} & I^{\mathcal{J}}\ \_m_{\mathtt{i}}(I_{\mathtt{i}}\ \_\mathtt{val});\ <:\ \mathsf{mbody}(\_m_{\mathtt{i}}, I^{\mathcal{J}}) \end{array}$$

- All abstract methods in the domain of $I^{\mathcal{J}}$ have been implemented. Namely

$$\mathsf{alldefined}(mh_1 \ldots mh_n, \mathrm{I})$$

Here we simply refer to $\mathsf{valid}(I^{\mathcal{J}})$, since it guarantees each abstract method to satisfy isField, isWith or isSetter. But that object includes all implementations for those cases. A getter $\mathtt{m_i}$ is generated if it satisfies isField; a **`with-`** method is generated for the case isWith, by the definition of withMethod; a setter for isSetter, similarly, by the definition of setterMethod. Hence it is of type $I^{\mathcal{J}}$ by (T-OBJ).

Part II. Next we check that in $[\![\mathcal{J}]\!]$,

$$\mathtt{dom}([\![\mathcal{J}]\!]) = \mathtt{dom}(I_1) \cup \ldots \cup \mathtt{dom}(I_n) \cup \mathtt{dom}(\overline{meth}) \cup \mathtt{dom}(meth')$$

Since $\mathcal{J}$ OK, we have $\mathtt{dom}(\mathcal{J}) = \mathtt{dom}(I_1) \cup \ldots \cup \mathtt{dom}(I_n) \cup \mathtt{dom}(\overline{meth})$, and hence it is equivalent to prove

$$\mathtt{dom}([\![\mathcal{J}]\!]) = \mathtt{dom}(I^{\mathcal{J}}) \cup \mathtt{dom}(meth')$$

This is obvious since a pre-condition of the translation is $\mathtt{of} \notin \mathtt{dom}(I^{\mathcal{J}})$, so $meth'$ doesn't overlap with $\mathtt{dom}(I^{\mathcal{J}})$. The definition of $\mathtt{dom}$ is based on mbody, and here the new domain $\mathtt{dom}([\![\mathcal{J}]\!])$ is only an extension to $\mathtt{dom}(I)$ with the `of` method, namely $meth'$. Also note that after translation, there are still no methods with conflicted names, since the `of` method was previously not in the domain, hence $[\![\mathcal{J}]\!]$ is well-formed, which finishes our proof. □

Lemma 2 (b). *If $\mathcal{J}$ has `@Obj` annotation $I^{\mathcal{J}}$ OK in $\mathcal{J}$ IT and there is no heir of $I^{\mathcal{J}}$, then $[\![\mathcal{J}]\!]$ OK in $[\![\mathcal{J}]\!]$ IT.*

Proof. **Part I.** Similarly to what already argued for Lemma 2 (a), since $\mathcal{J}$ OK, and by Lemma 1 (b), all the existing default and static methods are well typed in $[\![\mathcal{J}]\!]$ IT. The translation function delegates its work to `@ObjOf` in such way that we can refer to Lemma 2 (a) to complete this part. Note that all the methods added (directly) by `@Obj` are abstract, and thus there is no body to typecheck.

Part II. Similar to what we already argued for Lemma 2 (a), but we need to notice that the newly added methods are valid refinements for already present methods in $\texttt{dom}(I^{\mathfrak{I}})$ before the translation. Thus by the last clause of the definition of override(_), mbody(_) is defined on the same method names. □

A.4.3 THEOREM and Proof

Theorem 1 (@ObjOf tuning). *If a given interface table $\mathfrak{I}$ IT is OK where $\mathfrak{I}$ has `@ObjOf`, valid($I^{\mathfrak{I}}$) and $\texttt{of} \notin \texttt{dom}(I^{\mathfrak{I}})$, then the interface table $[\![\mathfrak{I}]\!]$ IT is OK.*

Proof. Lemma 2 (a) already proves that $[\![\mathfrak{I}]\!]$ is OK. On the other hand, for any $\mathfrak{I}' \in \text{IT}\backslash\mathfrak{I}$, by Lemma 1 (a), we know that all its methods are still well-typed, and the generated code in translation of `@ObjOf` is only a static method `of`, which has no way to affect the domain of $\mathfrak{I}'$, so after translation rule (T-INTF) can still be applied, which finishes our proof. □

Theorem 2 (@Obj tuning). *If a given interface table $\mathfrak{I}$ IT is OK where $\mathfrak{I}$ has `@Obj`, valid($I^{\mathfrak{I}}$) and $\texttt{of} \notin \texttt{dom}(I^{\mathfrak{I}})$, and there is no heir of $I^{\mathfrak{I}}$, then the interface table $[\![\mathfrak{I}]\!]$ IT is OK.*

Proof. Similar to what already argued for Theorem 1, we can apply Lemma 2 (b) and Lemma 1 (b). Then we finish by Theorem 1. □

A.5 Code Excerpts From the Maze Game Case Study

A.5.1 Maze Game Code in Java 8

```
/* Defines a base-door with no particular features */
interface TDoor {
    public boolean getLocked();
    public int getDoorMaxCoins();

    default boolean isLocked() {
        return getLocked();
    }
    default int open() {
        if (!isLocked()) {
            out.println("The door has been opened!");
            double rnd = Math.random();
            int cns = (int) (rnd * getDoorMaxCoins()) + 1;
            out.println("You got " + cns + " coins.");
            return cns;
        } else {
            out.println("This door is locked.");
            return -1;
        }
    }
    default int knock() {
        out.print("Door says: ");
        out.print("How you dare, ");
        out.println("I am the one who knocks!");
        int c = (Math.random() < 0.8) ? 0 : 1;
        if (c > 0)
            out.println("Ow! You got a free coin!");
        return c;
    }
}

/* Provides a counter that after a limit releases coins */
interface TCounter {
    public int getCounter();
    public void setCounter(int c);
    public int getLimit();
    public int getCounterMaxCoins();
    default void incrementCounter() {
        setCounter(getCounter() + 1);
    }
    default void decrementCounter() {
        setCounter(getCounter() - 1);
    }
    default boolean hasReachedLimit() {
        return getCounter() >= getLimit();
    }
    default int releaseCoins() {
        double rnd = Math.random();
        int cns = (int) (rnd * getCounterMaxCoins()) + 1;
        out.println("You got " + cns + " coins.");
        return cns;
    }
}

/* Puts together a door and a counter */
interface TKnockDoor extends TDoor, TCounter {
    /** Every know makes the counter increment.
     * If the limit is reached, more coins are released. **/
    default int knock() {
        int coins = TDoor.super.knock();
        incrementCounter();
        if(hasReachedLimit()) {
            out.print("Ohh! A special drop for you!");
            coins += releaseCoins();
        } else {
            //'Lets give a suggestion to the player
            out.print("'Dont challenge me... ");
            int c = getLimit();
            String sug = "never knock a door ";
            sug = sug + "more then " + c + " times.";
            out.println(sug);
        }
        return coins;
    }
}

class KnockDoor implements TKnockDoor {
    /* Fields for the door */
    private boolean locked;
    /* Fields for the counter */
    private int counter;
    private int limit;
    /* Glue Code for TDoor */
    public boolean getLocked()
    {
        return this.locked;
    }
    /* Glue code for TCounter */
    public int getCounter()
    {
        return this.counter;
    }
    public void setCounter(int c)
    {
        this.counter = c;
    }
    public int getLimit()
    {
        return this.limit;
    }
    /* Glue code for coin management */
    public int getDoorMaxCoins()
    {
        return 120;
    }
    public int getCounterMaxCoins()
    {
       return 500;
    }
    /* Constructor */
    public KnockDoor(boolean l, int li) {
       setCounter(0);
       setLocked(l);
       setLimit(li);
    }
    /* Other helpful methods */
    private void setLocked(boolean l)
    {
       this.locked = l;
    }
    private void setLimit(int l)
    {
       this.limit = l;
    }
}
```

A.5.2 Maze Game Code in Classless Java

```
/* Defines a base-door with no particular features */
@Obj interface TDoor {
    public boolean locked();
    public int doorMaxCoins();

    default boolean isLocked() {
       return locked();
    }
    default int open() {
       if (!isLocked()) {
          out.println("The door has been opened!");
          double rnd = Math.random();
          int cns = (int) (rnd * doorMaxCoins()) + 1;
          out.println("You got " + cns + " coins.");
          return cns;
       } else {
          out.println("This door is locked.");
          return -1;
       }
    }
    default int knock() {
        out.print("Door says: ");
        out.print("How you dare, ");
        out.println("I am the one who knocks!");
        int c = (Math.random() < 0.8) ? 0 : 1;
        if (c > 0)
           out.println("Ow! You got a free coin!");
        return c;
    }
}

/* Provides a counter that after a limit releases coins */
@Obj interface TCounter {
    public int counter();
    public void counter(int c);
    public int limit();
    public int counterMaxCoins();
    default void incrementCounter() {
        counter(counter() + 1);
    }
    default void decrementCounter() {
        counter(counter() - 1);
    }
    default boolean hasReachedLimit() {
        return counter() >= limit();
    }
    default int releaseCoins() {
        double rnd = Math.random();
        int cns = (int) (rnd * counterMaxCoins()) + 1;
        out.println("You got " + cns + " coins.");
        return cns;
    }
}

/* Puts together a door and a counter */
@Obj interface TKnockDoor extends TDoor, TCounter {
    /** Every know makes the counter increment.
     * If the limit is reached, more coins are released. **/
    default int knock() {
        int coins = TDoor.super.knock();
        incrementCounter();
        if (hasReachedLimit()) {
           out.print("Ohh! A special drop for you!");
           coins += releaseCoins();
        } else {
           //'Lets give a suggestion to the player
           out.print("'Dont challenge me... ");
```

```
            int c = limit();
            String sug = "never knock a door ";
            sug = sug + "more then " + c + " times.";
            out.println(sug);
        }
        return coins;
    }
}
```

A.6 Proofs for FHJ (Chapter 5)

Lemma 1. *If* $mbody(m, I_d, I_s) = (J, \overline{I_x}\ \overline{x}, I_e\ e_0)$, *then* $\overline{x} : \overline{I_x}, this : J \vdash e_0 : I_0$ *for some* $I_0 <: I_e$.

Proof. By the definition of mbody, the target method m is found in J. By the method typing rule (T-METHOD), there exists some $I_0 <: I_e$ such that $\overline{x} : \overline{I_x}, \mathtt{this} : J \vdash e_0 : I_0$. □

Lemma 2 (Weakening). *If* $\Gamma \vdash e : I$, *then* $\Gamma, x : J \vdash e : I$.

Proof. Straightforward induction. □

Lemma 3 (Method Type Preservation). *If* $mbody(m, J, J) = (K, \overline{I_x}\ \_, I_e\ \_)$, *then for any* $I <: J$, $mbody(m, I, J) = (K', \overline{I_x}\ \_, I_e\ \_)$.

Proof. Since $\mathtt{mbody}(m, J, J)$ is defined, by (T-INTF) we derive that $\mathtt{mbody}(m, I, J)$ is also defined. Suppose that

$$\mathtt{findOrigin}(m, J, J) = \{I_0\}$$

$$\mathtt{findOverride}(m, J, I_0) = \{K\}$$

$$\mathtt{findOrigin}(m, I, J) = \{I_0'\}$$

$$\mathtt{findOverride}(m, I, I_0') = \{K'\}$$

Below we use $I[m \uparrow J]$ to denote the type of method m defined in I that overrides J. We have to prove that $K'[m \uparrow I_0'] = K[m \uparrow I_0]$. Two facts:

- A. By (T-INTF), canOverride ensures that an override between any two original methods preserves the method type. Formally,

$$I_1 <: I_2 \quad \Rightarrow \quad I_1[m \uparrow I_1] = I_2[m \uparrow I_2]$$

- B. By (T-METHOD) and (T-ABSMETHOD), any partial override also preserves method type. Formally,
$$I_1 <: I_2 \;\Rightarrow\; I_1[m \uparrow I_2] = I_2[m \uparrow I_2]$$

By definition of `findOverride`, $K <: I_0, K' <: I_0'$. By Fact B,

$$K[m \uparrow I_0] = I_0[m \uparrow I_0] \quad K'[m \uparrow I_0'] = I_0'[m \uparrow I_0']$$

Hence it suffices to prove that $I_0'[m \uparrow I_0'] = I_0[m \uparrow I_0]$. Actually when calculating `findOrigin`(m, J, J), by the definition of `findOrigin` we know that $I_0 <: J$ and $I_0[m$ `override` $I_0]$ is defined. So when calculating `findOrigin`(m, I, J) with $I <: J$, I_0 should also appear in the set before pruned, since the conditions are again satisfied. But after pruning, only I_0' is obtained, by definition of `prune` it implies $I_0' <: I_0$. By Fact A, the proof is done.

□

Lemma 4 (Term Substitution Preserves Typing). *If* $\Gamma, \overline{x} : \overline{I_x} \vdash e : I$, *and* $\Gamma \vdash \overline{y} : \overline{I_x}$, *then* $\Gamma \vdash [\overline{y}/\overline{x}]e : I$.

Proof. We prove by induction. The expression e has the following cases:

Case Var. Let $e = x$. If $x \notin \overline{x}$, then the substitution does not change anything. Otherwise, since $\overline{y}$ have the same types as $\overline{x}$, it immediately finishes the case.

Case Invk. Let $e = e_0.m(\overline{e})$. By (T-INVK) we can suppose that

$$\Gamma, \overline{x} : \overline{I_x} \vdash e_0 : I_0 \quad \texttt{mbody}(m, I_0, I_0) = (\_, \overline{J}\,\_, I\,\_)$$

$$\Gamma, \overline{x} : \overline{I_x} \vdash \overline{e} : \overline{I_e} \quad \overline{I_e} <: \overline{J} \quad \Gamma, \overline{x} : \overline{I_x} \vdash e : I$$

By induction hypothesis,

$$\Gamma \vdash [\overline{y}/\overline{x}]e_0 : I_0 \quad \Gamma \vdash [\overline{y}/\overline{x}]\overline{e} : \overline{I_e}$$

Again by (T-INVK), $\Gamma \vdash [\overline{y}/\overline{x}]e : I$.

Case New. Straightforward.

Case Anno. Straightforward by induction hypothesis and (T-ANNO).

□

A.6.1 Proof for Theorem 1

Theorem 1 (Subject Reduction). If $\Gamma \vdash e : I$ and $e \to e'$, then $\Gamma \vdash e' : I$.

Proof.

Case S-Invk. Let

$$e = ((\mathtt{J})\mathtt{new}\ \mathtt{I}()).\mathtt{m}(\overline{v}) \quad \Gamma \vdash e : \mathtt{I}_e$$

$$e' = (\mathtt{I}_{e_0})[\overline{(\mathtt{I}_x)v}/\overline{x}, (\mathtt{I}_0)\mathtt{new}\ \mathtt{I}()/\mathtt{this}]e_0$$

$$\mathtt{mbody}(\mathtt{m}, \mathtt{I}, \mathtt{J}) = (\mathtt{I}_0, \overline{\mathtt{I}_x}\ \overline{x}, \mathtt{I}_{e_0}\ e_0)$$

Since $\mathtt{mbody}(\mathtt{m}, \mathtt{I}, \mathtt{J})$ is defined, the definition of $\mathtt{mbody}$ ensures that $\mathtt{I} <: \mathtt{J}$. And since e is well-typed, by (T-Invk),

$$\Gamma \vdash \overline{v} : \overline{\mathtt{I}_v} \quad \overline{\mathtt{I}_v} <: \overline{\mathtt{I}_x}$$

By the rules (T-Anno) and (T-New),

$$\Gamma \vdash \overline{(\mathtt{I}_x)v} : \overline{\mathtt{I}_x} \quad \Gamma \vdash (\mathtt{I}_0)\mathtt{new}\ \mathtt{I}() : \mathtt{I}_0$$

On the other hand, by Lemma 5,

$$\overline{x} : \overline{\mathtt{I}_x}, \mathtt{this} : \mathtt{I}_0 \vdash e_0 : \mathtt{I}'_{e_0} \quad \mathtt{I}'_{e_0} <: \mathtt{I}_{e_0}$$

By Lemma 6,

$$\Gamma, \overline{x} : \overline{\mathtt{I}_x}, \mathtt{this} : \mathtt{I}_0 \vdash e_0 : \mathtt{I}'_{e_0}$$

Hence by Lemma 8, the substitution preserves typing, thus

$$\Gamma \vdash [\overline{(\mathtt{I}_x)v}/\overline{x}, (\mathtt{I}_0)\mathtt{new}\ \mathtt{I}()/\mathtt{this}]e_0 : \mathtt{I}'_{e_0}$$

Since $\mathtt{I}'_{e_0} <: \mathtt{I}_{e_0}$, the conditions of (T-Anno) are satisfied, hence $\Gamma \vdash e' : \mathtt{I}_{e_0}$. Now we only need to prove that $\mathtt{I}_{e_0} = \mathtt{I}_e$. Since $\mathtt{I}_{e_0}$ is from $\mathtt{mbody}(\mathtt{m}, \mathtt{I}, \mathtt{J})$, whereas $\mathtt{I}_e$ is from $\mathtt{mbody}(\mathtt{m}, \mathtt{J}, \mathtt{J})$, by the rule (T-Invk) on e. Since $\mathtt{I} <: \mathtt{J}$, by Lemma 7, $\mathtt{I}_{e_0} = \mathtt{I}_e$.

Case C-Receiver. Straightforward induction.

Case C-Args. Straightforward induction.

Case C-StaticType. Immediate by (T-Anno).

Case C-FReduce. Immediate by (T-Anno) and induction.

Case C-AnnoReduce. Immediate by (T-Anno) and transitivity of $<:$.

□

A.6.2 Proof for Theorem 2

Theorem 2 (Progress). Suppose e is a well-typed expression, if e includes $((J)\mathit{new}\ I()).m(\overline{v})$ as a sub-expression, then $\texttt{mbody}(m, I, J) = (I_0, \overline{I_x\ x}, I_e\ e_0)$ and $\#(\overline{x}) = \#(\overline{v})$ for some I_0, $\overline{I_x}$, $\overline{x}$, I_e and e_0.

Proof. Since e is well-typed, by (T-Invk) and (T-Anno) we know that

$$I <: J, \text{and } \texttt{mbody}(m, J, J) \text{ is defined}$$

By (T-Intf), $\texttt{mbody}(m, I, J)$ is also defined, and the type checker ensures the expected number of arguments.

On the other hand, since $I <: J$, by the definition of $\texttt{findOrigin}$,

$$\texttt{findOrigin}(m, I, J) \subseteq \texttt{findOrigin}(m, I, I)$$

By (T-New), $\texttt{canInstantiate}(I) = \mathsf{True}$. By the definition of $\texttt{canInstantiate}$, any $J_0 \in \texttt{findOrigin}(m, I, I)$ satisfies that $\texttt{findOverride}(m, I, J_0)$ contains only one interface, in which the m that overrides J_0 is a concrete method. Therefore $\texttt{mbody}(m, I, J)$ also provides a concrete method, which finishes the proof. □

A.6.3 Proof for Theorem 4

Theorem 4 (Determinacy of One-Step Evaluation). If $t \to t'$ and $t \to t''$, then $t' = t''$.

Proof. The Proof is done by induction on a derivation of $t \to t'$, following the book *TAPL*.

- If the last rule used in the derivation of $t \to t'$ is (S-Invk), then we know that t has the form $((J)\texttt{new}\ I()).m(\overline{v})$ with I, J, m determined. Now it is obvious that the last rule in the derivation of $t \to t''$ should also be (S-Invk) with the same I, J, m. Since $\texttt{mbody}(m, I, J)$ is a *function* that given the same input will calculate the same result, we know the two induction results are the same, thus $t' = t''$ is immediately proved.

- If the last rule used in the derivation of $t \to t'$ is (C-Receiver), then t has the form $e_0.m(\overline{e})$ and $e_0 \to e_0'$. Since e_0 is not a value, the last rule used in $t \to t''$ has to be (C-Receiver) (other rules do not match) too. Assume in the reduction $t \to t''$, $e_0 \to e_0''$, thus $e_0'.m(\overline{e}) = e_0''.m(\overline{e})$. Thus, $t' = t''$ proved.

- If the last rule used in the derivation of $t \to t'$ is (C-StaticType), then t is fixed to be $\texttt{new}\ I()$. The last rule used in $t \to t''$ has to be (C-StaticType), and obviously, $t' = t'' = (I)\texttt{new}\ I()$.

- If the last rule used in the derivation of $t \rightarrow t'$ is (C-FReduce), then t has the form $(I)e$ and $e \rightarrow e'$. The last rule used in $t \rightarrow t''$ cannot be (C-StaticType) because it requires t to be `new` $I()$; it can neither be (C-AnnoReduce) because it requires t to be $(I)((J)$`new` $K())$ where (J)`new` $K()$ is already a value. So the last rule used in $t \rightarrow t''$ can only be (C-FReduce) (other rules do not match). Assume in the reduction $t \rightarrow t''$, $e \rightarrow e''$, and $(I)e \rightarrow (I)e''$. By induction hypothesis, $e' = e''$, thus $t' = t''$ proved.

- If the last rule used in the derivation of $t \rightarrow t'$ is (C-AnnoReduce), then the form of t is fixed to be $(I)((J)$`new` $K())$. Since $(I)((J)$`new` $K())$ is not reducible, the rule (C-FReduce) does not apply. The only rule applies in $t \rightarrow t''$ is (C-AnnoReduce). Thus $t' = t'' = (I)$`new` $K()$ proved.

- If the last rule used in the derivation of $t \rightarrow t'$ is (C-Args), then t has the form $v.m(..., e, ...)$ and $e \rightarrow e'$. The last rule used in $t \rightarrow t''$ cannot be (S-Invk) because it requires all arguments to be values. Thus only (C-Args) applies to $t \rightarrow t''$. Assume in the reduction $t \rightarrow t''$, $e \rightarrow e''$. By induction hypothesis, $e' = e''$, thus $v.m(..., e', ...) = v.m(..., e'', ...)$, thus $t' = t''$ proved.

□

A.7 Proofs for FHJ+ (Chapter 6)

Lemma 5. *If* $mbody(m, I_d, I_s) = (J, \overline{I_x}\ \overline{x}, I_e\ e_0)$, *then* $\overline{x} : \overline{I_x}, \mathit{this} : J \vdash e_0 : I_0$ *for some* $I_0 <: I_e$.

Proof. By the definition of `mbody`, the target method m is found in J. By the method typing rule (T-Method), there exists some $I_0 <: I_e$ such that $\overline{x} : \overline{I_x}, \mathtt{this} : J \vdash e_0 : I_0$. □

Lemma 6 (Weakening). *If* $\Gamma \vdash e : I$, *then* $\Gamma, x : J \vdash e : I$.

Proof. Straightforward induction. □

Lemma 7 (Method Type Preservation). *If* $mbody(m, J, J) = (K, \overline{I_x}\ \_, I_e\ \_)$, *then for any* $I <: J$, $mbody(m, I, J) = (K', \overline{I_x}\ \_, I_e\ \_)$.

Proof. Since $\mathtt{mbody}(m, J, J)$ is defined, by (T-Intf) we derive that $\mathtt{mbody}(m, I, J)$ is also defined. Suppose that

$$\mathtt{findOrigin}(m, J, J) = \{I_0\}$$

$$\texttt{findOverride}(m, J, I_0) = \{K\}$$

$$\texttt{findOrigin}(m, I, J) = \{I_0'\}$$

$$\texttt{findOverride}(m, I, I_0') = \{K'\}$$

Below we use $I[m \uparrow J]$ to denote the type of method m defined in I that overrides J. We have to prove that $K'[m \uparrow I_0'] = K[m \uparrow I_0]$. Two facts:

- A. By (T-Intf), canOverride ensures that an override between any two original methods preserves the method type. Formally,

$$I_1 <: I_2 \quad \Rightarrow \quad I_1[m \uparrow I_1] = I_2[m \uparrow I_2]$$

- B. By (T-Method) and (T-AbsMethod), any partial override also preserves method type. Formally,

$$I_1 <: I_2 \quad \Rightarrow \quad I_1[m \uparrow I_2] = I_2[m \uparrow I_2]$$

By definition of findOverride, $K <: I_0$, $K' <: I_0'$. By Fact B,

$$K[m \uparrow I_0] = I_0[m \uparrow I_0] \quad K'[m \uparrow I_0'] = I_0'[m \uparrow I_0']$$

Hence it suffices to prove that $I_0'[m \uparrow I_0'] = I_0[m \uparrow I_0]$. Actually when calculating $\texttt{findOrigin}(m, J, J)$, by the definition of findOrigin we know that $I_0 <: J$ and $I_0[m \texttt{ override } I_0]$ is defined. So when calculating $\texttt{findOrigin}(m, I, J)$ with $I <: J$, I_0 should also appear in the set before pruned, since the conditions are again satisfied. But after pruning, only I_0' is obtained, by definition of prune it implies $I_0' <: I_0$. By Fact A, the proof is done. □

Lemma 8 (Term Substitution Preserves Typing). *If* $\Gamma, \overline{x} : \overline{I_x} \vdash e : I$, *and* $\Gamma \vdash \overline{y} : \overline{I_x}$, *then* $\Gamma \vdash [\overline{y}/\overline{x}]e : I$.

Proof. We prove by induction. The expression e has the following cases:

Case Var. Let $e = x$.

- If $x \notin \overline{x}$, then the substitution does not change anything.
- Otherwise, since $\overline{y}$ have the same types as $\overline{x}$, it immediately finishes the case.

Case Invk. Let $e = e_0.m(\overline{e})$. By (T-Invk) we can suppose that

$$\Gamma, \overline{x} : \overline{I_x} \vdash e_0 : I_0 \quad \texttt{mbody}(m, I_0, I_0) = (\_, \overline{J}\ \_, I\ \_)$$

$$\Gamma, \overline{x} : \overline{I_x} \vdash \overline{e} : \overline{I_e} \quad \overline{I_e} <: \overline{J} \quad \Gamma, \overline{x} : \overline{I_x} \vdash e : I$$

By induction hypothesis,

$$\Gamma \vdash [\overline{y}/\overline{x}]e_0 : I_0 \quad \Gamma \vdash [\overline{y}/\overline{x}]\overline{e} : \overline{I_e}$$

Again by (T-Invk), $\Gamma \vdash [\overline{y}/\overline{x}]e : I$.

Case New. Let $e = \texttt{new}\ I_0(\overline{e})$. By the rule (T-New), suppose that

$$\texttt{mconstr}(I_0) = \overline{I_x}\ \overline{J.x}$$

$$\Gamma \vdash \overline{e} : \overline{I}$$

$$\overline{I} <: \overline{I_x}$$

$$\Gamma \vdash \texttt{new}\ I_0(\overline{e}) : I_0$$

By induction hypothesis,

$$\Gamma \vdash [\overline{y}/\overline{x}]\overline{e} : \overline{I}$$

By the rule (T-New) again, $\Gamma \vdash \texttt{new}\ I_0([\overline{y}/\overline{x}]\overline{e}) : I_0$, hence $\Gamma \vdash [\overline{y}/\overline{x}]\texttt{new}\ I_0(\overline{e}) : I_0$. Proved.

Case Let. Let $e = I\ x = e_1; e_2$. By the rule (T-Let) we know that

$$\Gamma \vdash e_1 : I_1$$

$$I_1 <: I$$

$$\Gamma, x : I \vdash e_2 : I_2$$

$$\Gamma \vdash I\ x = e_1; e_2 : I_2$$

By induction hypothesis,

$$\Gamma \vdash [\overline{y}/\overline{x}]e_1 : I_1$$

$$\Gamma, x : I \vdash [\overline{y}/\overline{x}]e_2 : I_2$$

Therefore, $\Gamma \vdash [\overline{y}/\overline{x}]I\ x = e_1; e_2 : I_2$, $\Gamma \vdash [\overline{y}/\overline{x}]e : I_2$. Proved.

Case Cast. Straightforward by induction hypothesis and the rule (T-Cast). □

A.7.1 Proof for Theorem 5

Theorem 5 (Subject Reduction). If $\Gamma \vdash e : I$ and $\mu | e \to \mu' | e'$, then $\Gamma \vdash e' : I$.

Proof. The proof is done by induction on the reduction relation.

Case E-Invk. Let

$$e = (J)o.m(\overline{v})$$

$$e' = (I_{e_0})[\overline{(I_x)v}/\overline{x}, (I_0)o/\mathtt{this}]e_0$$

Assume

$$\mu(o) = \mathtt{new}\ I_e(...)$$

$$\mathtt{mbody}(m, I_e, J) = (I_0, \overline{I_x}\ \overline{x}, I_{e_0}\ e_0)$$

Since $\mathtt{mbody}(m, I_e, J)$ is defined, the definition of mbody ensures that $I_e <: J$. And since e is well-typed, by (T-Invk), $\Gamma \vdash \overline{v} : \overline{I_v}$, *where* $\overline{I_v} <: \overline{I_x}$.

By the rule (T-Cast), $\Gamma \vdash \overline{(I_x)v} : \overline{I_x} \quad \Gamma \vdash (I_0)\mathtt{new}\ I_e() : I_0$. On the other hand, by Lemma 5, $\overline{x} : \overline{I_x}, \mathtt{this} : I_0 \vdash e_0 : I'_{e_0}$, *where* $I'_{e_0} <: I_{e_0}$.

By Lemma 6, $\Gamma, \overline{x} : \overline{I_x}, \mathtt{this} : I_0 \vdash e_0 : I'_{e_0}$

Hence by Lemma 8 (substitution preserves typing), $\Gamma \vdash [\overline{(I_x)v}/\overline{x}, (I_0)o/\mathtt{this}]e_0 : I'_{e_0}$

Since $I'_{e_0} <: I_{e_0}$, the conditions of (T-Cast) are satisfied, hence $\Gamma \vdash e' : I_{e_0}$. Now we only need to prove that $I_{e_0} = I$. Since I_{e_0} is from $\mathtt{mbody}(m, I_e, J)$, whereas I is from $\mathtt{mbody}(m, J, J)$, by the rule (T-Invk) on e. Since $I_e <: J$, by Lemma 7, $I_{e_0} = I$. Proved.

Case E-New. Let

$$e = \mathtt{new}\ I(\overline{v})$$

$$e' = (I)o$$

By the rule (T-Constructor), $\Gamma \vdash e : I$. And by the rule (T-Cast), $\Gamma \vdash e' : I$. Proved.

Case E-Getter. Let

$$e = (J)o.f()$$

$$e' = (I_F)o_i$$

and

$$\mu(o) = \mathtt{new}\ I(o_1, ..., o_i, ..., o_n)$$

Here, $J.f$ is the i-th element of $\mathtt{new}\ I(...)$ and $f()$ is a getter with the return type I_F. And by the rule (T-Cast), $\Gamma \vdash (I_F)o_i : I_F$. Proved.

Case E-Setter. Similar as **Case E-Getter**.

Case E-Let. Let

$$e = I\ x = o; e_1$$

$$e' = [(I)o/x]e_1$$

Assume that $\Gamma, x : I \vdash e_1 : I_1$, then by the rule (T-LET), $\Gamma \vdash I\ x = o; e_1 : I_1$. By Lemma 8, $\Gamma \vdash [(I)o/x]e_1 : I_1$. Proved.

Case E-Cast. Let $e = (J)(I)o$ and $e' = (J)o$, then $\Gamma \vdash e : J$ and $\Gamma \vdash e' : J$ immediately from the rule (T-CAST).

Case E-CTX. Let $e = \varepsilon\{e_1\}$ and $e' = \varepsilon\{e_1'\}$. By the induction condition $\mu \mid e_1 \rightarrow \mu' \mid e_1'$ and the hypothesis: if $\Gamma \vdash e_1 : J$ and $\mu \mid e_1 \rightarrow \mu' \mid e_1'$, then $\Gamma \vdash e_1' : J$. Assume that $\Gamma \vdash \varepsilon\{e_1\} : I$, then by Lemma 8, $\Gamma \vdash \varepsilon\{e_1'\} : I$. Proved. □

A.7.2 Proof for Theorem 6

Theorem 6 (Progress). Suppose e is a well-typed expression, if e includes $(J)o.m(\overline{v})$ as a sub-expression, where $\mu(o) = \texttt{new}\ I(...)$, then one of the following conditions holds:

1. $\#\,\overline{v} = 0$, validGetter(m, I, J)
2. $\#\,\overline{v} = 1$, validSetter(m, I, J)
3. $\texttt{mbody}(m, I, J) = (I_0, \overline{I_x}\ \overline{x}, I_e\ e_0)$ and $\#(\overline{x}) = \#(\overline{v})$ for some I_0, $\overline{I_x}$, $\overline{x}$, I_e and e_0.

Proof.
Since e is well-typed and includes $(J)o.m(\overline{v})$ as a sub-expression. Then the sub-expression $(J)o.m(\overline{v})$ can be divided into three cases:

1. m is a getter, validGetter(m, I, J) and getters contain no parameter. And by the rule (T-INVK), the number of arguments is the same to the number of parameters of the method(getter), which is 0.

2. m is a setter, validSetter(m, I, J) and setters contain one and only one parameter. By the rule (T-INVK), the number of arguments is the same to the number of parameters of the method(setter), which is 1.

3. m is neither a getter or setter, by the rule (T-INVK) we know that

   $$I <: J, \text{ and } \texttt{mbody}(m, J, J) \text{ is defined}$$

   By (T-INTF), $\texttt{mbody}(m, I, J)$ is also defined, and the type checker ensures the expected number of arguments.

   On the other hand, since $I <: J$, by the definition of $\texttt{findOrigin}$,

   $$\texttt{findOrigin}(m, I, J) \subseteq \texttt{findOrigin}(m, I, I)$$

By (T-New), assume $\mathtt{mconstr}(I) = \overline{I_x \quad x}$ and MC OK in interface I. $\mathtt{validMC}(I, \overline{I_x \quad x}) = \mathtt{True}$. By the definition of validMC, any $J_0 \in \mathtt{findOrigin}(m, I, I)$ satisfies that $\mathtt{findOverride}(m, I, J_0)$ contains only one interface, in which the m that overrides J_0 is a concrete method. Therefore $\mathtt{mbody}(m, I, J)$ also provides a concrete method, which finishes the proof.

□

A.7.3 Proof for Theorem 8

Theorem 8 (Determinacy of One-Step Evaluation). If $\mu \mid t \to \mu' \mid t'$ and $\mu \mid t \to \mu'' \mid t''$, then $t' = t''$ and $\mu' = \mu''$.

Proof. The Proof is done by induction on a derivation of $\mu \mid t \to \mu' \mid t'$, following the book *TAPL*.

- If the last rule used in the derivation of $\mu \mid t \to \mu' \mid t'$ is (E-Invk), then we know that t has the form $(J)o.m(\overline{v})$ with $\mu(o) = \mathtt{new}\ I(...)$, I, J, m determined and $\neg\mathsf{isField}(m, I)$. Now it is obvious that the last rule in the derivation of $\mu \mid t \to \mu'' \mid t''$ should also be (E-Invk) with the same I, J, m. Since $\mathtt{mbody}(m, I, J)$ is a *function* that given the same input will calculate the same result, we know the two induction results are the same, thus $t' = t''$ is immediately proved. Also from the rule (E-Invk), we know that μ is unchanged, thus $\mu' = \mu'' = \mu$ is also proved.

- If the last rule used in the derivation of $\mu \mid t \to \mu' \mid t'$ is (E-New), then the form of t is fixed to be $\mathtt{new}\ I(\overline{v})$. The only rule applies in $\mu \mid t \to \mu'' \mid t''$ is also (E-New). From the rule (E-New) we know that $t' = t'' = (I)o$ and $\mu' = \mu'' = \mu, o \to \mathtt{new}\ I(\overline{v})$ proved.

- If the last rule used in the derivation of $\mu \mid t \to \mu' \mid t'$ is (E-Cast), then the form of t is fixed to be $(I)((J)o)$. Since $(J)o$ is not reducible, the only rule applies in $\mu \mid t \to \mu'' \mid t''$ is (E-Cast). Also from the rule (E-Cast) we know that μ is unchanged, thus $t' = t'' = (I)o$ and $\mu' = \mu'' = \mu$ proved.

- If the last rule used in the derivation of $\mu \mid t \to \mu' \mid t'$ is (E-Let), then the form of t is fixed to be $I\ x = o; e$. And the only rule applies in $\mu \mid t \to \mu'' \mid t''$ is (E-Let). Also from the rule (E-Cast) we know that μ is unchanged, thus $t' = t'' = e[(I)o/x]$ and $\mu' = \mu'' = \mu$ proved.

- If the last rule used in the derivation of $\mu \mid t \to \mu' \mid t'$ is (E-GETTER), then the form of t is fixed to be $(J)o.f()$. For the second reduction $\mu \mid t \to \mu'' \mid t''$, the rule (E-SETTER) does not apply because the number of parameters does not match. Also the rule (E-INVK) does not match because the precondition $!\mathit{validGetter}(m, I, J)$ is not satisfied. The only rule applies is the rule (E-GETTER). Also from the rule (E-GETTER) we know that μ is unchanged, thus $t' = t'' = (I_F)o_i$ and $\mu' = \mu'' = \mu$ proved.

- If the last rule used in the derivation of $\mu \mid t \to \mu' \mid t'$ is (E-SETTER), then the form of t is fixed to be $(J)o.SET_f((I')o)$. Then for the second reduction $\mu \mid t \to \mu'' \mid t''$, the rule (E-GETTER) does not apply because the number of parameters does not match. Also the rule (E-INVK) does not match because the precondition $!\mathit{validSetter}(m, I, J)$ is not satisfied. The only rule applies is the rule (E-SETTER). And from the rule (E-SETTER), we know that $t' = t'' = (I)o$. Assume in the first derivation $\mu \mid t \to \mu' \mid t'$, $\mu = \mu_0, o \to \mathtt{new}\ I(o_1, ..., o_i, ..., o_n)$, $\mu' = \mu_0, o \to \mathtt{new}\ I(o_1, ..., o', ..., o_n)$, then in the second derivation $\mu \mid t \to \mu'' \mid t''$, $\mu = \mu_0, o \to \mathtt{new}\ I(o_1, ..., o_i, ..., o_n)$, $\mu'' = \mu_0, o \to \mathtt{new}\ I(o_1, ..., o', ..., o_n) = \mu'$. Proved.

- If the last rule used in the derivation of $\mu \mid t \to \mu' \mid t'$ is (E-CTX), then there are 5 cases (t could take 5 forms):
  - If $t = I\ x = \varepsilon; e$ and the precondition is $\mu \mid e \to \mu' \mid e'$. Then $t' = \varepsilon\{e'\}$ .Then for the second reduction $\mu \mid t \to \mu'' \mid t''$, the rule (E-LET) does not apply because ε is reducible and does not conform to the form of $I\ x = o; e$ in (E-LET). Thus the only rule applies is the rule (E-CTX). Assume the precondition is $\mu \mid e \to \mu'' \mid e''$. By the induction hypothesis, $\mu'' = \mu'$ and $e'' = e'$. Therefore, $t'' = \varepsilon\{e''\} = \varepsilon\{e'\} = t'$. Proved.
  - The proof for the other cases, where $t = \varepsilon.m(\bar{e})$, $t = (J)o.m(\bar{o}, \varepsilon, \bar{e})$ $t = \mathtt{new}\ I(\bar{o}, \varepsilon, \bar{e})$ or $t = (I)\varepsilon$ takes the same approach as the first case.

□

A.8 Code of Office Clerk Example for Chapter 6

A.8.1 Office Clerk Code in FHJ+

```
class Title {}
class Manager extends Title { new(); }
class Director extends Title { new(); }
class Officer extends Title { new(); }
class Point {
   new(Int Point.x, Int Point.y);
   Int x() override Point;
   Void SET_x(Int x) override Point;
   Int y() override Point;
   Void SET_y(Int y) override Point;
}

class Located {
   Point position() override Located;
   Void SET_position(Point x) override Located;
   Int id() override Located;
}

class Employee {
   Title position() override Employee;
   Void SET_position(Title x) override Employee;
   Int age() override Employee;
   Void SET_age(Int age) override Employee;
   String name() override Employee;
   Void SET_name(String name) override Employee;
   Int id() override Employee;
}

class OfficeClerk extends Located, Employee {
   new(String Employee.name, Int Employee.age, Title Employee.
      position, Int OfficeClerk.deskNumber, Point Located.position,
      Int OfficeClerk.id);
   Int id() override OfficeClerk;
   Int deskNumber() override OfficeClerk;
   Void SET_deskNumber(Int x) override OfficeClerk;
   OfficeClerk moveTo(OfficeClerk c) override OfficeClerk {
      return (
         Point p4 = ((Located)c).position();
         ((Located)this).SET_position(p4);
         this.SET_deskNumber(c.deskNumber());
         this
      );
   }
}

OfficeClerk alice = new OfficeClerk("Alice", 27, new Manager(), 1, (
   Point p1 = new Point(2, 3);p1), 23343);
OfficeClerk bob = new OfficeClerk("Bob", 45, new Director(), 2, (
   Point p2 = new Point(2, 4);p2), 45567);
```

```
Title p3 = new Officer();
OfficeClerk a2b = alice.moveTo(bob);
((Employee)a2b).SET_position(p3);
a2b
```

A.8.2 Office Clerk Code in C++

```
#include <iostream>
#include <string>

using namespace std;

class Title {};
class Manager : public Title {};
class Director : public Title {};
class Officer : public Title {};

class Point {
public:
   int x;
   int y;
   Point(int _x, int _y) {
      x = _x;
      y = _y;
   }
};

class Located {
public:
   Point* position;
   int id;
};

class Employee {
public:
   Title* position;
   int age;
   string name;
   int id;
};

class OfficeClerk : public Located, public Employee {
public:
   int deskNumber;
   int id;
   OfficeClerk() {};
   OfficeClerk(string _name, int _age, Title* _employeePosition, int
        _deskNumber, Point* _locatedPosition) {
      this->name = _name;
      this->age = _age;
      this->Employee::position = _employeePosition;
      this->deskNumber = _deskNumber;
      this->Located::position = _locatedPosition;
   }

   OfficeClerk* moveTo(OfficeClerk* c) {
      Point* p4 = ((Located*)c)->position;
      this->Located::position = p4;
      this->deskNumber = c->deskNumber;
      return this;
   }
};

int main(int argc, char const *argv[])
{
   OfficeClerk* alice = new OfficeClerk("Alice", 27, new Manager(),
      1, new Point(2, 3));
   OfficeClerk* bob = new OfficeClerk("Bob", 45, new Director(), 2,
      new Point(2, 4));
   Title* p3 = new Officer();
   OfficeClerk* a2b = alice->moveTo(bob);
   a2b->Employee::position = p3;
   cout << a2b->id << endl;
   return 0;
}
```

A.9 FHJ+ Formalization in MJ Style

Program	P	::=	$\overline{\mathsf{IL}}\ e$
Interfaces	IL	::=	$\texttt{class}\ \mathsf{I}\ \texttt{extends}\ \overline{\mathsf{I}}\ \{\mathsf{MC}?\ \overline{\mathsf{M}}\}$
Constructors	MC	::=	$\texttt{static}\ \mathsf{I}\ \mathsf{m}(\overline{\mathsf{I}_x}\ \overline{\mathsf{J}.x})\ ;$
Methods	M	::=	$\mathsf{I}\ \mathsf{m}(\overline{\mathsf{I}_x}\ \overline{x})\ \texttt{override}\ \mathsf{J}\ \{\texttt{return}\ e;\} \mid \mathsf{I}\ \mathsf{m}(\overline{\mathsf{I}_x}\ \overline{x})\ \texttt{override}\ \mathsf{J}\ ;$
Expressions	e	::=	$x \mid e.\mathsf{m}(\overline{e}) \mid (\mathsf{I})e \mid \mathsf{I}.\mathsf{m}(\overline{e}) \mid \mathsf{I}\ x = e_1; e_2$
Context	Γ	::=	$\overline{x} : \overline{\mathsf{I}}$
Values	v	::=	$(\mathsf{J})\mathsf{I}.\mathsf{m}(\overline{v})$

Figure A.6: Syntax of **FHJ+** formalization in MJ style

Configuration	Config	::=	(H, VS, e, FS)
Scope	VS	::=	is a finite partial function from variables to pairs of expression types and values
Heap	H	::=	is a finite partial function from oids to heap objects
Heap Objects	ho	::=	(I, F)
Map Function	F	::=	is a finite partial function from field names to values
Frame Stack	FS	::=	$F \circ FS$
Frame	F	::=	$CF \mid OF$
Closed Frame	CF	::=	$x \mid e.m(\overline{e}) \mid (I)e \mid I.m(\overline{e}) \mid I\ x = e_1; e_2$ expressions
Open Frame	OF	::=	expressions with holes

Figure A.7: Configuration

$$(\textsc{T-Var})\ \frac{\vdash \Delta\ \mathsf{OK} \qquad \Delta \vdash \Gamma\ \mathsf{OK}}{\Delta;\Gamma, x : I \vdash x : I}$$

$$(\textsc{T-Invk})\ \frac{\begin{array}{c}\Delta;\Gamma \vdash e_0 : I_0 \\ \texttt{mbody}(m, I_0, I_0) = (K, \overline{I_x}\ \overline{x}, I\ \_) \qquad \Delta;\Gamma \vdash \overline{e} : \overline{I} \qquad \overline{I} <: \overline{I_x}\end{array}}{\Delta,\Gamma \vdash e_0.m(\overline{e}) : I}$$

$$(\textsc{T-StaticInvk})\ \frac{\texttt{mbody}(m, I_0, I_0) = (K, \overline{I_x}\ \overline{x}, I\ \_) \qquad \Delta;\Gamma \vdash \overline{e} : \overline{I} \qquad \overline{I} <: \overline{I_x}}{\Delta,\Gamma \vdash I_0.m(\overline{e}) : I}$$

$$(\textsc{T-Cast})\ \frac{\Delta,\Gamma \vdash \Gamma e : I \qquad I <: J}{\Delta,\Gamma \vdash (J)e : J}$$

$$(\textsc{T-Method})\ \frac{\begin{array}{c}I <: J \qquad \texttt{findOrigin}(m, I, J) = \{J\} \qquad \texttt{mbody}(m, J, J) = (K, \overline{I_x}\ \overline{x}, I_e\ \_) \\ \Delta;\Gamma, \overline{x} : \overline{I_x}, \texttt{this} : I \vdash e_0 : I_0 \qquad I_0 <: I_e\end{array}}{I_e\ m(\overline{I_x}\ \overline{x})\ \texttt{override}\ J\ \{\texttt{return}\ e_0;\}\ \mathsf{OK\ IN}\ I}$$

$$(\textsc{T-AbsMethod})\ \frac{\begin{array}{c}I <: J \quad \texttt{findOrigin}(m, I, J) = \{J\} \\ \texttt{mbody}(m, J, J) = (K, \overline{I_x}\ \overline{x}, I_e\ \_)\end{array}}{I_e\ m(\overline{I_x}\ \overline{x})\ \texttt{override}\ J\ ;\ \mathsf{OK\ IN}\ I}$$

$$(\textsc{T-MC})\ \frac{I_0 \equiv I \qquad \texttt{validMC}(I, \overline{I_x}, \overline{J.x})}{\texttt{static}\ I_0\ m(\overline{I_x}\ \overline{x})\ ;\ \mathsf{OK\ IN}\ I}$$

$$(\textsc{T-Intf})\ \frac{\begin{array}{c}MC\ \mathsf{OK\ IN}\ I \qquad \overline{M}\ \mathsf{OK\ IN}\ I \\ \forall J >: I \text{ and } m, \texttt{mbody}(m, J, J) \text{ is defined} \Rightarrow \texttt{mbody}(m, I, J) \text{ is defined} \\ \forall J >: I \text{ and } m, I[m\ \texttt{override}\ I] \text{ and } J[m\ \texttt{override}\ J] \text{ defined} \Rightarrow \texttt{canOverride}(m, I, J)\end{array}}{\texttt{class}\ I\ \texttt{extends}\ \overline{I}\ \{MC?\ \overline{M}\}\ \mathsf{OK}}$$

$$(\textsc{T-Prog})\ \frac{\forall I \in \overline{IL}, I\ \mathsf{OK} \qquad \Delta;\Gamma \vdash e : J}{\overline{IL}\ e\ \mathsf{OK}}$$

$$(\textsc{T-Let})\ \frac{\Delta;\Gamma \vdash e_1 : I_1 \qquad I_1 <: I \qquad \Delta;\Gamma, x : I \vdash e_2 : I_2}{\Delta;\Gamma \vdash I\ x = e_1; e_2 : I_2}$$

Figure A.8: Typing rules of **FHJ+** formalization in MJ style

$$\text{(E-VarAccess) } (H, BS \circ VS, x, FS) \to (H, BS \circ VS, v, FS)$$

$$\text{where } BS(x) = v$$

$$\text{(EC-Let)}(H, VS, I\ x = e_1; e_2, FS) \to (H, VS, e_1, (I\ x = \bullet; e_2) \circ FS)$$

$$\text{(E-Let)}(H, BS \circ VS, I\ x = v; e, FS) \to (H, BS' \circ VS, e, (\mathrm{returnLet}\ e) \circ FS) \text{ where } BS' = \{x \to v\} \circ BS$$

$$\text{(E-Return)}(H, BS \circ VS, v, (\mathrm{return}\ \bullet) \circ FS) \to (H, VS, v, FS)$$

$$\text{(E-ReturnLet)}(H, BS \circ VS, v, (\mathrm{returnLet}\ \bullet) \circ FS) \to (H, BS' \circ VS, v, FS)$$

$$\text{where } BS' = \mathtt{tail}(BS)$$

$$\text{(E-Invk)}(H, VS, (J)o.m(\overline{v}), FS) \to (H, VS', (I_e)e_0, (\mathrm{return}\ \bullet) \circ FS)$$

$$\text{where } m \text{ is not a getter } (I_0.m \text{ does not exist in } I.of), H(o) = I[\overline{o}], VS' = \{\mathtt{this} \to$$

$$(I_0)o, \overline{x \to v}\} \circ VS, \mathtt{mbody}(m, I, J) = (I_0, \overline{I_x}\ \overline{x}, I_e\ e_0)$$

$$\text{(E-InvkGetter)}(H, VS, (J)o.f(), FS) \to (H, VS, (I_F)o_i, FS)$$

$$\text{where } H(o) = I[o_1, ..., o_i, ...], \mathtt{mbody}(m, I, J) =$$

$$(I_0, \overline{I_x}\ \overline{x}, I_e\ e_0), (I_F\ I_0.f) \text{ is the i-th element of } I.of.$$

$$\text{(EC-Invk1)}(H, VS, e.m(\overline{e}), FS) \to (H, VS, e, (\bullet.m(\overline{e})) \circ FS)$$

$$\text{(EC-Invk2)}(H, VS, v.m(v_1, ..., v_{i-1}, e_i, ..., e_n), FS) \to (H, VS, e_i, (v.m(v_1, ..., v_{i-1}, \bullet, ..., e_n)) \circ FS)$$

$$\text{(E-Cast)}(H, VS, (J)((I)o), FS) \to (H, VS, (J)o, FS)$$

$$\text{(EC-Cast)}(H, VS, (J)e, FS) \to (H, VS, e, ((J)\bullet) \circ FS)$$

$$\text{(EC-of) } (H, VS, I.of(v_1, ..., v_{i-1}, e_i, ..., e_n), FS) \to$$

$$(H, VS, e_i, I.of(v_1, ..., v_i - 1, \bullet, ..., e_n) \circ FS)$$

$$\text{(E-of)}(H, VS, I.of(\overline{v}), FS) \to (H', VS, (I)o, FS)$$

$$\text{where } \overline{v} = \overline{(I)o}, H' = H[o \to I[\overline{o}]],\ \text{fresh } o \notin \mathrm{dom}(H)$$

Figure A.9: Semantics of **FHJ+** formalization in MJ style

$$(\text{E-S}\textsc{et})(H, VS, (I)o.SET\_f((I')o'), FS) \rightarrow (H', VS, (K)o, FS)$$

$$\text{where } H(o) = K[o_1, ..., o_n], H' = H[o \rightarrow$$

$$K[o_1, ..., o_{i-1}, o', o_{i+1}, ...o_n]], \text{f is the } i^{th} \text{ of } K'\text{s constructor.}$$

$$(\text{EC-S}\textsc{et1})(H, VS, e.SET\_f(e'), FS) \rightarrow (H, VS, e, (\bullet.SET\_f(e')) \circ FS)$$

$$(\text{EC-S}\textsc{et2})(H, VS, v.SET\_f(e'), FS) \rightarrow (H, VS, e', (v.SET\_f(\bullet)) \circ FS)$$

$$(\text{EC-F}\textsc{ill})(H, VS, v, F \circ FS) \rightarrow (H, VS, F(v), FS)$$

Figure A.10: Semantics of **FHJ+** formalization in MJ style (continued)

Bibliography

[Dar, 2016] (2016). Dart programming language. https://www.dartlang.org.

[Allen et al., 2003] Allen, E. E., Bannet, J., and Cartwright, R. (2003). A first-class approach to genericity. In *OOPSLA*, pages 96–114.

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

[Ancona and Zucca, 2002] Ancona, D. and Zucca, E. (2002). A calculus of module systems. *J. Funct. Program.*, 12(2):91–132.

[Bettini et al., 2013] Bettini, L., Damiani, F., Schaefer, I., and Strocco, F. (2013). Traitrecordj: A programming language with traits and records. *Sci. Comput. Program.*, 78(5):521–541.

[Bierman et al., 2003] Bierman, G., Parkinson, M., and Pitts, A. (2003). Mj: An imperative core calculus for java and java with effects. Technical report, University of Cambridge, Computer Laboratory.

[Bono et al., 2012] Bono, V., Kusmierek, J., and Mulatero, M. (2012). Magda: A new language for modularity. In *ECOOP*, volume 7313, pages 560–588.

[Bono et al., 2014] Bono, V., Mensa, E., and Naddeo, M. (2014). Trait-oriented programming in java 8. In *PPPJ*, pages 181–186.

[Bracha et al., 2008] Bracha, G., Ahe, P., Bykov, V., Kashai, Y., and Miranda, E. (2008). The newspeak programming platform.

[Bracha and Cook, 1990] Bracha, G. and Cook, W. R. (1990). Mixin-based inheritance. In *OOPSLA/ECOOP*, pages 303–311.

[Bracha et al., 1998] Bracha, G., Odersky, M., Stoutamire, D., and Wadler, P. (1998). Making the future safe for the past: Adding genericity to the java programming language. In *OOPSLA*, pages 183–200.

[Bruce, 1994] Bruce, K. B. (1994). A paradigmatic object-oriented programming language: Design, static typing and semantics. *J. Funct. Program.*, 4(2):127–206.

[Bruce et al., 1998] Bruce, K. B., Odersky, M., and Wadler, P. (1998). A statically safe alternative to virtual types. In *ECOOP*, volume 1445, pages 523–549.

[Canning et al., 1989] Canning, P. S., Cook, W. R., Hill, W. L., Olthoff, W. G., and Mitchell, J. C. (1989). F-bounded polymorphism for object-oriented programming. In *FPCA*, pages 273–280.

[Cardelli, 1988] Cardelli, L. (1988). A semantics of multiple inheritance. *Inf. Comput.*, 76(2/3):138–164.

[Chambers and Leavens, 1995] Chambers, C. and Leavens, G. T. (1995). Typechecking and modules for multimethods. *ACM Trans. Program. Lang. Syst.*, 17(6):805–843.

[Chambers et al., 1991] Chambers, C., Ungar, D. M., Chang, B., and Hölzle, U. (1991). Parents are shared parts of objects: Inheritance and encapsulation in SELF. *Lisp and Symbolic Computation*, 4(3):207–222.

[Clifton et al., 2000] Clifton, C., Leavens, G. T., Chambers, C., and Millstein, T. D. (2000). Multijava: modular open classes and symmetric multiple dispatch for java. In *OOPSLA*, pages 130–145.

[Cook, 1990] Cook, W. R. (1990). Object-oriented programming versus abstract data types. In *REX Workshop*, volume 489, pages 151–178.

[Cremet et al., 2006] Cremet, V., Garillot, F., Lenglet, S., and Odersky, M. (2006). A core calculus for scala type checking. In *MFCS*, volume 4162, pages 1–23.

[Dahl and Nygaard, 1966] Dahl, O. and Nygaard, K. (1966). SIMULA - an algol-based simulation language. *Commun. ACM*, 9(9):671–678.

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

[Duponcheel, 1995] Duponcheel, L. (1995). Using catamorphisms, subtypes and monad transformers for writing modular functional interpreters. Technical report, Utrecht University.

[Ellis and Stroustrup, 1990] Ellis, M. A. and Stroustrup, B. (1990). *The Annotated C++ Reference Manual*. Addison-Wesley.

[Ernst, 2001] Ernst, E. (2001). Family polymorphism. In *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326.

[Ernst, 2004] Ernst, E. (2004). The expression problem, Scandinavian style. In *MASPEGHI*.

[Ernst et al., 2006] Ernst, E., Ostermann, K., and Cook, W. R. (2006). A virtual class calculus. In *POPL*, pages 270–282.

[Fenton et al., 2017] Fenton, S., Fenton, and Spearing (2017). *Pro TypeScript*. Springer.

[Flatt et al., 1998] Flatt, M., Krishnamurthi, S., and Felleisen, M. (1998). Classes and mixins. In *POPL*, pages 171–183.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley.

[Garrigue, 1998] Garrigue, J. (1998). Programming with polymorphic variants. In *ML Workshop*, volume 13.

[Goetz and Field, 2012] Goetz, B. and Field, R. (2012). Featherweight defenders: A formal model for virtual extension methods in java. http://cr.openjdk.java.net/~briangoetz/lambda/featherweight-defenders.pdf.

[Goldberg and Robson, 1983] Goldberg, A. and Robson, D. (1983). *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley.

[Hendler, 1986] Hendler, J. (1986). Enhancement for multiple-inheritance. *SIGPLAN Notices*, 21(10):98–106.

[Igarashi et al., 2001] Igarashi, A., Pierce, B. C., and Wadler, P. (2001). Featherweight java: a minimal core calculus for java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450.

[Kamina and Tamai, 2007] Kamina, T. and Tamai, T. (2007). Lightweight scalable components. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering*, GPCE '07.

[Kamina and Tamai, 2008] Kamina, T. and Tamai, T. (2008). Lightweight dependent classes. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering*, GPCE '08.

[Krishnamurthi et al., 1998] Krishnamurthi, S., Felleisen, M., and Friedman, D. P. (1998). Synthesizing object-oriented and functional design to promote re-use. In *ECOOP*, volume 1445, pages 91–113.

[Kusmierek and Bono, 2007] Kusmierek, J. D. M. and Bono, V. (2007). Hygienic methods ñ introducing hygjava. *Journal of Object Technology*, 6(9):209–229.

[Lagorio and Servetto, 2011] Lagorio, G. and Servetto, M. (2011). Strong exception-safety for checked and unchecked exceptions. *Journal of Object Technology*, 10:1: 1–20.

[Lagorio et al., 2009] Lagorio, G., Servetto, M., and Zucca, E. (2009). Featherweight jigsaw: A minimal core calculus for modular composition of classes. In *ECOOP*, volume 5653, pages 244–268.

[Limberghen and Mens, 1996] Limberghen, M. V. and Mens, T. (1996). Encapsulation and composition as orthogonal operators on mixins: a solution to multiple inheritance problems. *Object Oriented Systems*, 3:1–30.

[Liquori and Spiwack, 2008] Liquori, L. and Spiwack, A. (2008). Feathertrait: A modest extension of featherweight java. *ACM Trans. Program. Lang. Syst.*, 30(2):11:1–11:32.

[Löh and Hinze, 2006] Löh, A. and Hinze, R. (2006). Open data types and open functions. In *PPDP*, pages 133–144.

[Madsen and Moller-Pedersen, 1989] Madsen, O. L. and Moller-Pedersen, B. (1989). Virtual classes: a powerful mechanism in object-oriented programming. In *OOPSLA '89*.

[Malayeri and Aldrich, 2009] Malayeri, D. and Aldrich, J. (2009). CZ: multiple inheritance without diamonds. In *OOPSLA*, pages 21–40.

[McDirmid et al., 2001] McDirmid, S., Flatt, M., and Hsieh, W. C. (2001). Jiazzi: New-age components for old-fashioned java. In *OOPSLA*, pages 211–222.

[Meyer, 1987] Meyer, B. (1987). Eiffel: programming for reusability and extendibility. *SIGPLAN Notices*, 22(2):85–94.

[Microsoft, 2003] Microsoft (2003). Csharp explicit interface member implementations document. https://msdn.microsoft.com/en-us/library/aa664591(v=vs.71).aspx.

[Neildo, 2011] Neildo (2011). Project lombok: Creating custom transformations. http://notatube.blogspot.hk/2010/12/project-lombok-creating-custom.html.

[Nystrom et al., 2004] Nystrom, N., Chong, S., and Myers, A. C. (2004). Scalable extensibility via nested inheritance. In *OOPSLA*, pages 99–115.

[Nystrom et al., 2006] Nystrom, N., Qi, X., and Myers, A. C. (2006). J&: nested intersection for scalable software composition. In *OOPSLA*, pages 21–36.

[Odersky et al., 2004] Odersky, M., Altherr, P., Cremet, V., Emir, B., Maneth, S., Micheloud, S., Mihaylov, N., Schinz, M., Stenman, E., and Zenger, M. (2004). An overview of the scala programming language. Technical report, EPFL.

[Oliveira, 2009] Oliveira, B. C. d. S. (2009). Modular visitor components. In *ECOOP*, volume 5653 of *Lecture Notes in Computer Science*, pages 269–293.

[Oliveira and Cook, 2012] Oliveira, B. C. d. S. and Cook, W. R. (2012). Extensibility for the masses - practical extensibility with object algebras. In *ECOOP*, volume 7313, pages 2–27.

[Oliveira et al., 2006] Oliveira, B. C. d. S., Hinze, R., and Löh, A. (2006). Extensible and modular generics for the masses. In *Trends in Functional Programming*, volume 7, pages 199–216.

[Pierce, 2002] Pierce, B. C. (2002). *Types and programming languages*. MIT Press.

[Ramalingam and Srinivasan, 1997] Ramalingam, G. and Srinivasan, H. (1997). A member lookup algorithm for C++. In *PLDI*, pages 18–30.

[Ramananandro, 2012] Ramananandro, T. (2012). *Mechanized Formal Semantics and Verified Compilation for C++ Objects. (Les objets en C++ : sémantique formelle mécanisée et compilation vérifiée)*. PhD thesis, Paris Diderot University, France.

[Reppy and Turon, 2006] Reppy, J. and Turon, A. (2006). A foundation for trait-based metaprogramming. In *FOOL/WOOD*.

[Reppy and Turon, 2007] Reppy, J. H. and Turon, A. (2007). Metaprogramming with traits. In *ECOOP*, volume 4609, pages 373–398.

[Reynolds, 1975] Reynolds, J. C. (1975). User-defined types and procedural data structures as complementary approaches to data abstraction. In *New Directions in Algorithmic Languages*.

[Saito and Igarashi, 2008] Saito, C. and Igarashi, A. (2008). The essence of lightweight family polymorphism. *Journal of Object Technology*, 7(5):67–99.

[Saito and Igarashi, 2013] Saito, C. and Igarashi, A. (2013). Matching mytype to subtyping. *Sci. Comput. Program.*, 78(7):933–952.

[Saito et al., 2008] Saito, C., Igarashi, A., and Viroli, M. (2008). Lightweight family polymorphism. *Journal of Functional Programming*, 18.

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

[Schärli et al., 2003] Schärli, N., Ducasse, S., Nierstrasz, O., and Black, A. P. (2003). Traits: Composable units of behaviour. In *ECOOP*, volume 2743, pages 248–274.

[Singh, 1995] Singh, G. B. (1995). Single versus multiple inheritance in object oriented programming. *OOPS Messenger*, 6(1):30–39.

[Snyder, 1986] Snyder, A. (1986). Encapsulation and inheritance in object-oriented programming languages. pages 38–45.

[Stroustrup, 1989] Stroustrup, B. (1989). Multiple inheritance for C++. *Computing Systems*, 2(4):367–395.

[Stroustrup, 1994] Stroustrup, B. (1994). *The design and evolution of C++*. Pearson Education India.

[Stroustrup, 1995] Stroustrup, B. (1995). *The C++ programming language*. Pearson Education India.

[Swierstra, 2008] Swierstra, W. (2008). Data types à la carte. *J. Funct. Program.*, 18(4):423–436.

[Taivalsaari, 1996] Taivalsaari, A. (1996). On the notion of inheritance. *ACM Comput. Surv.*, 28(3):438–479.

[Torgersen, 2004] Torgersen, M. (2004). The expression problem revisited. In *ECOOP*, volume 3086, pages 123–143.

[Ungar and Smith, 1987a] Ungar, D. M. and Smith, R. B. (1987a). Self: The power of simplicity. In *OOPSLA*, pages 227–242.

[Ungar and Smith, 1987b] Ungar, D. M. and Smith, R. B. (1987b). Self: The power of simplicity. In *OOPSLA*, pages 227–242.

[Wadler, 1998] Wadler, P. (1998). The Expression Problem. Email. Discussion on the Java Genericity mailing list.

[Wang and Oliveira, 2016] Wang, Y. and Oliveira, B. C. d. S. (2016). The expression problem, trivially! In *MODULARITY*, pages 37–41.

[Wang et al., 2016] Wang, Y., Zhang, H., Oliveira, B. C. d. S., and Servetto, M. (2016). Classless java. In *GPCE*, pages 14–24.

[Wasserrab et al., 2006] Wasserrab, D., Nipkow, T., Snelting, G., and Tip, F. (2006). An operational semantics and type safety prooffor multiple inheritance in C++. In *OOPSLA*, pages 345–362.

[Wehr and Thiemann, 2011] Wehr, S. and Thiemann, P. (2011). Javagi: The interaction of type classes with interfaces and inheritance. *ACM Trans. Program. Lang. Syst.*, 33(4):12:1–12:83.

[Wright and Felleisen, 1994] Wright, A. and Felleisen, M. (1994). A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94.

[Würthinger et al., 2013] Würthinger, T., Wimmer, C., Wöß, A., Stadler, L., Duboscq, G., Humer, C., Richards, G., Simon, D., and Wolczko, M. (2013). One VM to rule them all. In *Onward!*, pages 187–204.

[Zenger and Odersky, 2001] Zenger, M. and Odersky, M. (2001). Extensible algebraic datatypes with defaults. In *ICFP*, pages 241–252.

[Zenger and Odersky, 2005] Zenger, M. and Odersky, M. (2005). Independently extensible solutions to the expression problem. In *FOOL*, volume 12.

[Zwitserloot and Spilker, 2016] Zwitserloot, R. and Spilker, R. (2016). Project lombok. http://projectlombok.org.