Advanced Object-Oriented Language Mechanisms for Variability Management

Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte

Dissertation

zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

vorgelegt von

Magister der Informatik Vaidas Gasiūnas
geboren in Panevėžys, Litauen

Referent:	Prof. Dr. Mira Mezini
Korreferent:	Prof. Dr. Sophia Drossopoulou
Datum der Einreichung:	25.11.2009
Datum der mündlichen Prüfung:	16.12.2009

Erscheinungsjahr 2010

Darmstadt D17

Abstract

Decomposition of software into components is usually not sufficient to achieve a high-degree of reusability, because a component seldom completely fits to the needs of a particular use, and needs to be adapted to specific requirements and the technical context of that use. Thus, in order to increase reusability of components, they must be made configurable and adaptable, or in other words they must support a certain degree of variation.

Object-oriented techniques, in particular inheritance and subtype polymorphism, facilitate modular variability management. Subtype polymorphism can be used to hide variations of an object behind stable inherfaces and bind them dynamically. Inheritance can modularize unanticipated variations and variations affecting interfaces of objects. The modularized variations can be combined using some form of multiple inheritance. Multi-dispatch of methods enables modularization and dynamic binding of multi-dimensional variation.

Classes are often too small units of modularization. In a lot of cases, a cohesive piece of functionality involves a *group* of related classes. Although mainstream languages provide class grouping mechanisms, such as packages and inner classes in Java, the typical object-oriented techniques, such as inheritance and subtype polymorphism, are not supported at the scope of such class groups. As a result, variations involving multiple classes must be encoded by variations of individual classes. Such encodings compromise type-safety and produce a considerable amount of glue code, which is often error-prone and not stable.

The main statement of this thesis is that by making typical object-oriented techniques available at the scope of a group of classes we can provide a better support for managing variations at that scope.

For the purpose of making inheritance and polymorphism available for a group of classes, we rely on the ideas of *virtual classes* and *family polymorphism.* A large-scale multiple inheritance is enabled by the *propagating mixin composition.* In this thesis we present the first implementation of these ideas for Java, and propose improvements to their semantics, namely a more intuitive linearization algorithm for propagating mixin composition and more flexible path-dependent types. We also introduce *abstract virtual classes*, which increase the advantages of family polymorphism by providing the possibility to describe interfaces for families of classes.

Further, we propose a novel concept of *dependent classes*, which enhances virtual classes in analogous way like multimethods enhance single-dispatch. The multi-dispatch for classes not only enables dispatch of their functionality by multiple constructor parameters, but also generalizes family polymorphism with the possibility to express membership of an object in multiple families. The feasibility of the new concept is validated in two ways. First, we design a concrete language with dependent classes, called DepJ, and implement a type-checker and interpreter for it. Second, we formalize the features of dependent classes in vc^n and DC_C calculi, and verify their soundness and decidability.

A further development of dependent classes, so-called *second-order dependent classes* combines parameterization of classes by objects and by types and provides the advantages of dependent classes for generic classes. In particular, they make it possible to vary the functionality of a generic class with respect to its type parameters, and to define such variations in a modular way. Abstraction from such variations in the client code and their dynamic binding is enabled by representing types as runtime values and supporting dynamic dispatch by such values.

The expected advantages of virtual classes and dependent classes for variation management are validated by a set of variation scenarios. We explore variations at the scope of individual objects, as well as at the scope of a group of objects. We also investigate interactions of different kinds of variations and analyze specific variation scenarios in the context of object-oriented frameworks.

We identify the problems of implementing these scenarios using conventional object-oriented techniques, and show that these problems are resolved by implementations with the advanced techniques. In particular, we show that virtual classes and propagating-mixin composition provide the typical advantages of inheritance for managing variations of a group of objects. Dependent classes provide the typical advantages of multi-dispatch for managing variations of a class. They also generalize the advantages of virtual classes with the possibility to modularize variations of multiple overlapping groups of objects, and provide a better solution for modelling multiple variations of a group of objects.

Zusammenfassung

Die Zerlegung von Software in Komponenten ist oft nicht ausreichend um einen hohen Wiederverwendbarkeitsgrad zu erreichen, weil eine Komponente selten genau den Bedürfnissen einer konkreten Verwendung entspricht und deswegen an die speziellen Anforderungen und den technischen Kontext dieser Verwendung angepasst werden muss. Um folglich die Wiederverwendbarkeit der Komponenten zu erhöhen, müssen sie konfigurierbar und anpassungsfähig gemacht werden, oder, anders gesagt, sie müssen einen bestimmten Grad an Variabilität unterstützen.

Die Modularisierung der Variabilität wird duch die objektorientierten Techniken, inbesondere durch die Vererbung und die Subtyppolymorphie, unterstützt. Die Subtyppolymorphie macht es möglich, die Variationen eines Objekts hinter stabilen Schnittstellen zu verbergen und sie dynamisch zu binden. Die Vererbung ist besonders geeignet zur Modularisierung der unvorhergesehenen Variationen und der Variationen, die die Schnittstellen der Objekte beeinflussen. Die modulariserten Variationen können mittels einer Form der Mehrfach-Vererbung kombiniert werden. Die Multi-Methoden ermöglichen die Modularisierung und das dynamische Binden einer mehrdimensionalen Variation.

Die Klassen sind oft zu kleine Einheiten für Modularisierung. In vielen Fällen betrifft ein logisch zusamenhängender Teil der Funktionalität eine Gruppe von zusamenhängenden Klassen. Obwohl die etablierten Sprachen verschiedene Mechanismen zur Gruppierung von Klassen anbieten, zum Beispiel Packages und innere Klassen in Java, werden auf der Ebene solcher Gruppierungen objektorientierten Techniken, wie die Vererbung und die Subtyppolymorphie, nicht unterstützt. Demzufolge müssen die Variationen, die mehreren Klassen betreffen, durch die Variationen der einzelnen Klassen kodiert werden. Solche Kodierungen beeinträchtigen aber die Typsicherheit und erzeugen eine beträchtliche Menge an Glue-Code.

Die Hauptthese dieser Dissertation ist, dass die Ermöglichung der objektorientierten Techniken auf der Ebene von Gruppen der Klassen eine bessere Unterstützung für die Modularisierung der Variationen auf dieser Ebene erreicht werden kann.

Um die Vererbung und die Subtyppolymorphie für eine Gruppe von Klassen zu ermöglichen, greifen wir auf die Ideen der *virtuellen Klassen* und der *Familienpolymorphie* zurück. Die Verwendung von Mehrfachvererbung im großem Umfang wird durch *propagierende Mixin-Komposition* ermöglicht. In dieser Arbeit zeigen wir die erste Implementierung dieser Ideen für Java und schlagen einige Verbesserungen zu ihrer Semantik vor, nämlich einen intuitiveren Lineasierungsalgorithmus für die propagierende Mixin-Komposition und flexiblere pfadabhängige Typen. Wir führen auch das Konzept der *abstrakten vituellen Klassen* ein, das die Beschreibung der Schittstellen von Klassfamilien ermöglicht und so die Vorteile der Familienpolymorphie steigert.

Außerdem schlagen wir das Konzept der *abhängigen Klassen* vor, der die virtuellen Klassen analog zur Idee des Multi-Dispatch erweitert. Multi-Dispatch von Klassen ermöglicht nicht nur den Dispatch ihrer Funktionalität durch mehrere Konstruktorparameter, sondern erweitert auch die Familien-Polymorphie um die Möglichkeit, die Zugehörigkeit eines Objekts zu mehreren Familien auszudrücken. Die Machbarkeit des neuen Konzepts wurde in zwei Weisen validiert. Erstens haben wir eine konkrete Programmiersprache mit abhängigen Klassen, DepJ, entworfen und implementiert. Zweitens haben wir die Semantik der abhängigen Klassen durch die Kalküle vc^n und DC_C formalisiert, die dann auf Korrektheit und Entscheidbarkeit verifiziert wurden.

Eine Weiterentwicklung der abhängigen Klassen, sogennante *abhängige Klassen zweiter Ordnung*, vereinigen die Parameterisierung der Klassen durch Objekte und durch Typen und stellen die Vorteile der abhängigen Klassen auch für die generischen Klassen bereit. Insbesondere machen sie es möglich, die Variationen der Funktionalität einer generischen Klasse bezüglich ihrer Typparameter in einer modularen Weise zu beschreiben. Um die Abstraktion von solchen Variationen auf der Client-Seite und ihr dynamisches Binden zu ermöglichen, werden Typen als Laufzeitwerte zur Verfügung gestellt, und der dynamische Dispatch mit solchen Werten ermöglicht.

Die erwarteten Vorteile der virtuellen und der abhängigen Klassen bezüglich der Implementierung von Variationen werden durch eine Reihe verschiedenen Variationsszenarien überprüft. Wir untersuchen die Variationen sowohl auf der Ebene der einzelner Objekten als auch auf der Ebene der Gruppen von Objekten. Des Weiteren erforschen wir die Interaktionen zwischen verschiedenen Arten von Variationen und untersuchen spezifische Variationsszenarien im Kontext der objektorientierten Frameworks.

Wir identifizieren die Probleme mit den Implementierungen solcher Szenarien mit den konventionellen objektorientierten Techniken, und zeigen dass diese Probleme durch die fortgeschrittenen Techniken gelöst werden können. Insbesondere zeigen wir dass die virtuellen Klassen und die propagierende Mixin-Komposition die typische Vorteile der Vererbung für die Implementierung von Variationen auf der Ebene einer Gruppe von Objekten bieten. Die abhängigen Klassen stellen die typischen Vorteile des Multi-Dispatch für die Implementierung von Variationen einer Klasse bereit. Sie erweitern auch die Vorteile der virtuellen Klassen um die Möglichkeit, die Variationen von mehreren einander überlappenden Gruppen von Objekten auszudrücken.

Contents

List of Figures

Preface

During my early professional career, I have been involved in the development of various software products and their customizations. I had a chance to observe the difficulties with extending and configuring products for the needs of new customers and gradually transforming products into product lines. Driven by the popular trend of that time, we tried to master the design challenges by refactoring the object-oriented design, applying the classical design patterns and their variations, but with time we found ourselves more and more often using "unclean" techniques, such as conditional compilation and untyped data and so sacrificing modularity and static type-safety. Seeing how monolithic design impedes independent development and how untyped data leads to insidious bugs, I started to ask myself whether we were faced with fundamental design trade-offs or just with the limitations of the technology that we used.

My search for alternatives to the object-oriented programming led me to the early works on aspect-oriented programming and its precursors, such as subject-oriented programming, adaptive programming, and composition filters. These works showed me that there are solutions beyond object-orientation that address the problems that I experienced in practice. Yet, I was not completely satisfied with the existing solutions, because of their complexity or the lack of safety and efficiency. During my master studies I had a chance to meet Bertrand Meyer and learn more about the Eiffel programming language. I was intrigued by the flexibility of extension and composition of classes enabled by multiple inheritance in Eiffel. At some point, I came to the idea that we could have analogous extension and composition mechanisms at a larger scale – at the scope of libraries and frameworks. After my master thesis, I was lucky to find a position in the group of Prof. Mira Mezini, which allowed me to further pursue the idea of using advanced object-oriented mechanisms for flexible modularization of software. This work resulted in the implementation of virtual classes as a part of the CaesarJ programming language, led to the concept of dependent classes, and eventually evolved into this thesis.

The path to the thesis was full of challenges, and I would have hardly made it to the end without the continuous support of my mentors Mira Mezini and Klaus Ostermann. I am very grateful to my mentors for transferring their long-year experience with programming languages and software design, for encouraging me to pursue my new ideas, steering them into the right direction, and helping to turn them into publications. My big thanks also go to my second examiner Sophia Drossopoulou for a thorough review

of this thesis, valuable feedback, and an interesting discussion during the defense. I am very glad that I had a chance to exchange ideas and work together with a lot of brilliant colleagues: Michael Achenbach, Ivica Aracic, Christoph Bockisch, Eric Bodden, Marcel Bruch, Vasian Cepa, Anis Charfi, Tom Dinkelaker, Michael Eichberg, Michael Haupt, Christian Hofer, Sebastian Kanthak, Sven Kloppenburg, Karl Klose, Roman Knöll, Tatjana Korbmacher, Ralf Mitschke, Martin Monperrus, Lucas Satabin, Thorsten Schäfer, Tobias Schuh, Andreas Sewe and Jan Sinschek. My special thanks go to Ivica Aracic for the great teamwork while developing, evaluating, and promoting the CaesarJ programming language, and to Michael Eichberg for proof-reading parts of the thesis. I owe my deepest gratitude to Gudrun Harris who did a lot to ensure continuous funding of my PhD position and helped to cope with various bureaucratic issues. I am also very grateful to my parents for their constant support and encouragement.

Parts of this work were funded by the TOPPrax project (German Ministry of Education and Research grant 01|SC04D) and the AMPLE project (European Commission grant IST-33710). I would like to thank all partners of the project for the fruitful collaboration, interesting discussions, and the patience during the end phase of writing the thesis.

1 Introduction

1.1 Background: Object-Oriented Variation Management

Reuse is one of the major strategies in software engineering for reducing development and maintenance costs. In order to make a part of software reusable, first we must define it as a reusable software component. In a general sense, a reusable component is any identifiable software entity such as a function or a module, which can be used in multiple applications or in multiple contexts within the same application. Decomposition of software into components is usually not sufficient to achieve a high-degree of reusability, because a component seldom completely fits to the needs of a particular use, and needs to be adapted to specific requirements and the technical context of that use. Thus, in order to increase reusability of components, they must be made configurable and adaptable, or in other words they must support a certain degree of variation.

The most straightforward way to support variation in a component is to describe the supported variation in form of parameters to the component and then use conditional statements to vary the functionality of the component depending on the value of the parameters. Concrete examples of this variation technique range from simple function parameters to global variables in an application.

The problem with the straightforward parameterization is that it does not modularize specific variations of a component. Implementations of the variations are mixed with the reusable part of the component as well as with each other. Such monolithic design has several disadvantages. Variation supported by the component is fixed and extending the component with variations requires changing its code. The built-in variations increase the complexity and stability of component and its clients, because they depend on all variation of component, not only on the ones that they actually need.

Object-oriented techniques, in particular inheritance and subtype polymorphism, facilitate modular variability management. Subtype polymorphism makes it possible to outsource variations of an object to other objects and hide them behind stable interfaces. In such a design, the reusable part of an object and its specific variations are separated from each other. Consequently, the design is more stable and extensible, and helps to manage complexity.

Inheritance is a way to describe subtyping in nominally typed object-oriented languages, but it also serves as a useful variation mechanism by itself. Differently from subtype polymorphism, which hides variations behind fixed interfaces, inheritance makes all operations of a class replaceable, which is especially useful for supporting unanticipated variability. Another advantage of inheritance is that variations of a class described its subclasses can affect not only implementation of the class, but also its interface.

Multiple inheritance makes variations outsourced to subclasses composable with each other. An especially elegant solution is proposed by mixin-based inheritance, which proposes to make the functionality described by a subclass reusable, by enabling its composition with different parent classes. Such reusable subclasses, called mixins, provide a very lightweight mechanism for modularization of variations of a class and their composition.

The single dispatch provided by subtype polymorphism in object-oriented languages, is further enhanced by more expressive dynamic dispatch mechanisms, such as multi-dispatch and predicate dispatch. The advanced dispatch mechanisms are interesting from the perspective of variation management, because they modularize more sophisticated structures of variation. While single-dispatch can modularize dependency of functionality of an object on a single variation point, multi-dispatch enables dispatch of a function by multiple parameters and thus can be used to modularize its dependencies on multiple variation points.

1.2 The Thesis in a Nutshell

Units of modularization in object-oriented languages are classes and objects. A class is a bundle of multiple data fields and operations, and thus presents a larger unit modularization unit than individual functions, but nevertheless a significant body of research has raised the concern that classes are a too small unit of modularity [Szy98, ML98, SB98b, Ern03, Ost02]. In a lot of cases a cohesive piece of functionality involves a *group* of related classes.

Mainstream languages provide class grouping mechanisms, e.g., namespaces in C++ [ES95], or packages and inner classes in Java [AG96]. However, inheritance and polymorphism, which are the primary means of object-oriented programming for expressing and modularizing variability, are not supported at the level of such class groups. As a result, variations involving multiple classes cannot be directly expressed and must be encoded by variations of individual classes. As will be demonstrated in this thesis, such encodings compromise type-safety and produce a considerable amount of glue code, which is often error-prone and not stable.

The main statement of this thesis is that by making typical object-oriented techniques available at the scope of groups of objects we can provide a better support for managing variations that affect more than one object. The scope of such variations can range from simple collaborations and composite data structures to the set objects of the entire application.

To validate the statement, we provide the first implementation of virtual classes and a corresponding type system for Java, and analyze its support for variability. Further, we propose dependent classes as a generalization of the idea of virtual classes, and analyze their further contributions to variability management.

1.2.1 Virtual Classes

The idea of *virtual classes* [MMP89] to treat classes as late-bound members of objects provides a conceptually simple and elegant solution for making inheritance and polymorphism available for a group of classes. Like any other members of an object, classes are declared as members of the object by nesting their definitions within the class of the object. Such member classes are considered as virtual, because they can be overridden (or more precisely refined) in subclasses and their access is late-bound, i.e., an access to a class is qualified by a reference to an object and is resolved to the definition of that class within that object.

An inheritance for a family of classes is achieved by declaring these classes as virtual classes of the same enclosing class. We will call the enclosing class as *family class*, and its instances as *family objects*. Then the subclasses of the family class describe extensions of the family of enclosed classes. Due to late-bound access of classes, a refinement to a virtual class is automatically bound to all other classes of the family. Subtype polymorphism is also extended to families of classes, because the interface of a family object consists not only of its operations, but also of its classes.

Variations of groups of classes modularized by subclasses of the family class can be made composable by extending the semantics of mixin-based inheritance to consider virtual classes. Such semantics is proposed in gbeta [Ern99c] and is called *propagating mixin composition*, because mixin composition of family classes is automatically propagated to virtual classes.

The idea of virtual classes of treating classes as members of objects is conceptually simple, but it is challenging technically and theoretically, because late-binding of classes affects not only their instantiation, but also their inheritance relationships and references to them in types. In particular, access to virtual classes in types leads to a form of dependent typing, because types contain terms identifying family objects of classes.

Although the idea of virtual classes is not new, in this thesis we present the first implementation of virtual classes for Java based on the ideas of BETA [MMP89] and gbeta [Ern99c, Ern01]. Since BETA language is very different from mainstream object-oriented languages, integration of virtual classes into a Java-based language raises new language design issues, like support for abstract declarations and constructors. We also introduce a couple of improvements to the core semantics of virtual classes, namely a more intuitive linearization algorithm for propagating mixin composition and more flexible path-dependent types based on our theoretical work on dependent classes.

In order to validate advantages of virtual classes for variation management, we investigate application of virtual classes for different variation scenarios. Besides, the typical scenarios of virtual classes dealing with variations of a group of objects, we also investigate combination of variations at different scopes, combination of dynamic and static variations, as well as specific variation scenarios in the context of object-oriented frameworks.

1.2.2 Dependent Classes

Since virtual classes were introduced as a new kind of object attributes [MMP89], they are based on the perspective of objects as records. As a result, dispatch supported by virtual classes is analogous to single-dispatch of methods and exhibits analogous limitations. From the perspective of variation management, the main limitation of single dispatch is that it supports modularization of dependencies on one variation point only. A virtual class is dispatched by its family object in this way modularizing its dependencies on the variations represented by the family object.

In this thesis, we, however, identify multiple scenarios when an implementation of a class depends on multiple variation points. In the simplest case we have multiple independent variation points influencing the behavior and the interface of an object, e.g., functionality of a table widget depends on multiple variation points, such as the supported selection model, cell coloring strategy, support for resizing of columns and rows, and so on. In more sophisticated cases objects depend on variations bound at different scopes, e.g., an adapter of an application object to a framework may depend both on variations of the framework and the variations of the adapted application object. Such scenarios are not properly supported by virtual classes, because of the limitation of single dispatch.

Although multi-dispatch of methods can be encoded by multiple applications of single dispatch, such encoding is more difficult in the case of classes. The single-dispatch limitation of virtual classes implies not only that instantiation of the class is dispatched by a single family object, but also that in type references we can express membership of a class in one family only. Although encoding of multi-dispatched instantiation of a class using virtual classes is indeed possible, there is no way to encode object's membership in multiple families using the path-dependent types of virtual classes.

Another problem is that by expressing dispatch of virtual classes through nesting we cluster together all classes that depend on the variations of a particular class and so unnecessarily introduce coupling between them. Also, because of such clustering, all virtual classes of the same family are exposed to the clients of each other. The increased coupling makes the design less stable and more difficult to extend.

To address these problems, this thesis proposes a generalization of virtual classes, called *dependent classes*. A dependent class is a class whose definition depends on *arbitrarily many* objects. In a sense, dependent classes can be seen as a combination of virtual classes with multi-dispatch [DG87, Cha92, Sha96, CLCM00].

In analogy to multimethods, dependent classes express dispatch not by nesting, but by explicit parameterization. A dependent class is parameterized by the objects it depends on, and can be specialized by redeclaring the class for more specific types of its parameters. The parameters to the class are bound as constructor parameters during instantiation of the class, and the functionality of the created object is determined by a dynamic dispatch, which collects declarations of the class and its superclasses matching the given constructor parameters. Analogously to virtual classes, the dependency of a class on its parameters affects not only its instantiation, but also its inheritance relationships and references to it in types.

Dependent classes contribute to the variation scenarios, where an object is affected by multiple variation points. We can modularize dependencies of a class on multiple variation points and their interactions by representing the variation points as parameters to the class and moving dependencies on specific parameter values into separate declarations of the class. Dependent classes can also express membership of an object in multiple families, and so express dependencies of the object on variations bound at different scopes. Type-safe interactions of the object with other objects sharing the variations are enabled by dependent typing of the object with respect to these families.

A precise definition of the core semantics of dependent classes is given in the vc^n calculus, which defines the effects of dispatch to different kinds of class uses – instantiation, inheritance, and typing – and type relations of types depending on multiple paths. The calculus is defined in an algorithmic style, which makes it easy to show its decidability. The soundness proof of vc^n is machine-checked in Isabelle/HOL [NPW02].

Although the algorithmic style of vc^n has the advantage of being constructive, and in this way outlining a possible implementation of dependent classes, at the same the algorithmic style makes it difficult to extend the semantics with new expressive power. For this reason, we present an alternative formalization of dependent classes in DC_C calculus, which is more declarative and based on more primitive concepts. The calculus encodes the types of dependent classes by sets of primitive classification and equivalence constraints. The typical type relationships can then be described using a constraint system. The DC_C calculus enhances the semantics of vc^n with support for abstract class and method declarations, supplied with appropriate completeness and uniqueness checks.

Since dependent classes are a generalization of virtual classes, the results of work on vc^n and DC_C can also be reused for virtual classes. Indeed, the type relationships of the vc^n calculus were used for the latest implementation of virtual classes in CaesarJ, because vc^n is simpler than the previously used *vc* calculus [EOC06] and at the same time resolves its limitations on the structure on types and expressions while preserving the algorithmic style. The simplification of type relationships in vc^n is to a high degree a consequence of switching from nested to parametric style, which allows treating references to family objects just as simple field references.

1.3 Contributions of the Thesis

The major contribution of the thesis is the concept of dependent classes, its implementation, formalization and evaluation. The specific contributions in this respect are:

- The thesis introduces a novel concept of dependent classes, as a combination of the ideas of virtual classes and multi-dispatch. Dependent classes resolve the limitations of virtual classes, related to the single-dispatch. They enable modularization of dependencies of classes on multiple variation points. They also extend family polymorphism [Ern01] with the possibility to express membership of an object in multiple families and type-safe interaction with other objects of these families.

- The thesis proposes a concrete programming language for dependent classes, which considers various specific language design issues, such as combination of parametric and nested style, concrete method selection strategies, and so on. A prototype interpreter and a type checker are implemented for the proposed language.

- The thesis presents the idea of second-order dependent classes, which generalizes parameterization by types and by objects and provides a form of parametric polymorphism with a possibility to specialize a class with respect to its parameters.

- The vc^n calculus presented in the thesis gives a precise definition of the syntax and semantics of the core features of dependent classes. Feasibility of the proposed language features are verified by proving soundness and decidability of the calculus.

- The DC_C calculus, presented in the thesis, demonstrates how the specific semantics of dependent classes can be reduced to more primitive concepts: types are reduced to sets of primitive constraints, and specific type relations are replaced by a constraint system. The encoding presents a novel view of dependent classes and related systems, enabling comparison of their expressive power with elements of mathematical logic.
- DC_C calculus enhances the type system of the vc^n calculus with support for abstract method and class declarations, symmetric method dispatch, and arbitrary intersection types. The intersection types are exploited for definition of algorithms for checking completeness and uniqueness of method implementations.
- The thesis presents a systematic evaluation of dependent classes for variation management. Such evaluation not only shows applicability of the proposed mechanisms in concrete practical examples, but also identifies typical problem classes, which benefit from such mechanisms.

The thesis also contributes to practical and theoretical work on virtual classes. The concrete contributions to the research on virtual classes are following:

- The thesis presents the first implementation of virtual classes in CaesarJ, which is the first full-fledged implementation of virtual classes for Java including support type-safe family polymorphism based on path-dependent types [Ern01] and propagating mixin composition [Ern99c]. CaesarJ resolves specific language design issues of integrating virtual classes to Java and implements their translation to Java bytecode.
- The thesis explains an intuitive semantics of virtual classes and propagating mixin composition, and based on the intuitive semantics proposes new linearization rules for propagating mixin composition.
- The vc^n calculus also presents novel theoretical results for virtual classes and virtual types. First, its soundness proof is the first automatically checked soundness proof for a type system with virtual types. Second, the advantage of the calculus over previous formalizations of virtual classes is a better balance between simplicity, decidability and expressive power. For example, the calculus removes limitations of *vc* calculus [EOC06] on the structure of types and expressions and simplifies its type relationships, while at the same time remaining in an algorithmic style.
- The thesis presents a systematic evaluation of virtual classes for variation management. Besides, the typical scenarios of virtual classes dealing with variations of a group of objects, the thesis also presents a lot of novel scenarios of using virtual classes to deal with interactions of static and dynamic variations, variations

at different scopes, and variation scenarios in frameworks. Most of the scenarios benefit from the support for abstract virtual classes, which is a novel contribution of the thesis.

Independently of the contributions to virtual classes, the thesis identifies a series of challenging variation scenarios based on the analysis of design problems in existing Java software. These scenarios can also be interesting for evaluation of other novel language mechanisms, addressing similar problems.

1.4 Structure of the Thesis

Chapter 2 analyzes strengths and limitations of conventional object-oriented techniques for implementation of different kinds of variations. We start from the simplest cases, i.e., single variations of individual objects, and then move to more complicated scenarios involving variations of multiple objects and interactions of multiple variation points. The variations scenarios presented in that chapter are used in further chapters for evaluation of variation support of virtual classes and dependent classes.

Chapter 3 presents the concepts of virtual classes and the general intuition behind the precise semantics of virtual classes and propagating mixin composition. Then we discuss specific semantic aspects of implementation of virtual classes in CaesarJ. In particular, we define a mixin linearization algorithm for virtual classes, explain the semantics of abstract virtual classes and give an informal overview of the type system. The last part of the chapter evaluates virtual classes with respect to support for the identified variation scenarios.

Chapter 4 gives an informal presentation of dependent classes. First, we give a general motivation of dependent classes and informally explain their semantics. Then we discuss specific language design issues and present the design decisions made for the DepJ language. In particular, we discuss the trade-off between the nested and the parametric styles, alternative method selection strategies, support for varying set of parameters, recursive dependent classes, abstract dependent classes, and the unification of dependent classes with parametric polymorphism as it is implemented in DepJ. At the end of the chapter we go through the variation scenarios that were not completely supported by virtual classes, and investigate how they can benefit from dependent classes.

Chapter 5 gives a formal presentation of the semantics of dependent classes, in form of two calculi vc^n and DC_C. The vc^n calculus gives a formalization of the core features of dependent classes in the style of Featherweight Java [IPW99]. We prove decidability of the calculus and present the soundness statement with its major lemmas, while the full proof of soundness is specified using Isabelle/HOL [NPW02] system, where it is automatically checked. Further, the chapter presents DC_C calculus, which enhances the vc^n calculus with abstract methods and classes, and defines the type relationships in form of a constraint system. The chapter also contains the soundness and decidability proofs of the latter calculus.

Related work of dependent classes and our implementation of virtual classes is discussed in 6. Finally we formulate the final conclusions and discuss future work in 7.

2 Implementation of Variability in Object-Oriented Languages

2.1 Introduction

Language support for modular implementation of software with multiple static or dynamic variations is important for taming the complexity of software and for increasing its reusability. It is especially important in the context of software product lines, which achieve a higher degree of reuse through systematic planning and management of variation.

Support for software variability and extensibility has been one of the promises that underlie the popularity of the object-oriented programming paradigm. The support of object-orientation for software variability is usually attributed to subtype polymorphism and inheritance. Subtype polymorphism enables abstraction from concrete types of objects, and so enables hiding variations encapsulated by the objects behind their interfaces. Inheritance is a way to define subtyping in nominally typed languages, but it also presents a variation mechanism by itself, because it enables moving varying parts of a class to its subclasses. Inheritance is especially useful for unanticipated variability because of its support for the Open-Closed principle [Mey97, Mar03]. Concrete scenarios of using subtype polymorphism and inheritance to deal with variations are described by a variety of design patterns [GHJV95], including Strategy, Bridge, Abstract Factory, and Template Method.

In this chapter we take a closer look at the implementation of different kinds of variability using standard object-oriented techniques. Besides simple inheritance and subtype polymorphism, we will also employ multiple inheritance and multi-dispatch in the examples where they are appropriate. We explore a set of variation scenarios differing by various criteria, in particular the scope of a variation and its binding time. We also pay a special attention on different kinds of interactions of variations.

Variation scope is the part of the runtime or static structure of a program affected by a particular variation. It can range from individual objects and functions to various groups of objects, in the extreme case including all objects of an application. The *binding time* of a variation can be static, i.e., fixed during development of the software, or dynamic, i.e., bound at runtime depending on the state of the program. For dynamic variation, we often need to differentiate between the variation bound during creation of objects and the possibility to change variation binding after the objects are created.

The chapter is structured as follows:

- In Sec. 2.2, we start our journey with the simplest case: analyzing the problems related to modularizing individual variable features of individual objects. We will describe the typical ways of modularizing variations using subtype polymorphism and inheritance and analyze their advantages and limitations. We will also explain how variations of functions can be managed using object-oriented techniques.
- Sec. 2.3 analyzes design challenges emerging when several variable orthogonal and non-orthogonal features of individual objects are involved. We analyze the problems of OO designs that exploit different combinations of inheritance and delegation for handling different variations. Also, we discuss the specific advantages of multi-dispatch for dealing with interactions of multiple orthogonal variations.
- In Sec. 2.4, we consider variations affecting multiple objects. Again we analyze both modularization of individual variable features, as well as their combinations. A specific issue emerging when raising the scope of variations is the possibility of interactions of variations bound at different scopes.
- In Sec. 2.5 we investigate variation scenarios in object-oriented frameworks and their instances. Although framework variations can be seen as a special case of group variations, there is a couple specific variation issues related to frameworks. First, a typical framework explicitly supports a set of open variation points, which are intended to be instantiated for specific uses of the framework. Second, a framework is in principle an independently developed large-scale component with its own structure and abstractions, and thus often needs to be adapted to the application-specific variations.

The examples used for illustration of variation scenarios are mostly based on our analysis of design problems in popular Java libraries for graphical user interfaces, such as Swing, SWT and GEF. The survey of the variation scenarios presented in this chapter is limited to the design problems that could be found in the analyzed software. Theoretically we could construct further, even more sophisticated scenarios, but the practical value of analyzing such scenarios would be questionable as long as no real instances of them are known.

2.2 Single Variations of Individual Objects

Object-oriented languages offer two specific mechanisms to handle variations of objects: parameterization by instance variables and inheritance. In this section, we will analyze the advantages and disadvantages of these approaches. In addition to that, the last part of the section explains how the variation mechanisms for objects can be also applied to deal with variations of functions.

2.2.1 Using Object Fields to Model Variability

One can think of the functionality defined in a class as being parameterized by the instance variables of that class. For each independent variation point we can define an instance variable that can be assigned different values depending on the selected variant. The main advantage is support for dynamic variation: instance variables of an object can be set when the object is instantiated and be changed during its lifetime.

There are two different approaches for modeling variability by means of object fields. The varying functionality can be either expressed by conditionals checking the values of the instance variables, or outsourced to other objects referenced by instance variables and used polymorphically. The latter approach is followed in several behavioral design patterns, such Strategy or State [GHJV95]. In the following, we call the object whose variations are outsourced the *master object*, while the objects that encapsulate the implementation of variations are called *helper objects*[1].

An advantage of using instance variables in conjunction with conditional logic is that sophisticated and fine-grained variations can be expressed, because variables can be used in arbitrary conditions inside method bodies. However, this approach does not modularize variations. With each new variation the size and complexity of the class will grow and the class will become less and less maintainable. The implementations of all variations of the class are tangled within the same module and often even within the same methods. Such a design is in fact not really object-oriented.

By outsourcing dynamically varying features to helper objects, the second approach achieves better modularity of the implementation of the dynamically varying features. For example, variations of the clipboard functionality in a table widget can be modularized by moving it to a helper object and providing different implementations of it as shown in Fig. 2.1. Class `Table` declares a field `clipboard` (line 2), to which it delegates clipboard specific operations, e.g., copy (line 12), and event handling, e.g., handling of

[1]This terminology differs from the terminology used in [GHJV95], where "context" is often used to refer to our master object; "strategy" or "state" are different denotations for what we call helper objects.

```
class Table extends Widget {
  TableCB clipboard;
  ...
  void setClipboard(TableCB clipboard) {
      this.clipboard = clipboard;
  }
  void keyPressed(KeyEvent e) {
      super.keyPressed(e);
      clipboard.keyPressed(e);
  }
  void copy() {
      clipboard.copyToClipboard();
  }
  ...
}

abstract class TableCB {
  Table table;
  void keyPressed(KeyEvent e) {
      if (e.getKeyCode() == VK_COPY) {
          copyToClipboard();
      }
      ...
  }
  abstract void copyToClipboard();
  ...
}

class TableAppCB extends TableCB {
  void copyToClipboard() {
      AppClipboard.setText(table.getCellText(currRow, currCol));
  }
  ...
}

class TableSystemCB extends TableCB {
  void copyToClipboard() {
      /* copy to the system clipboard */
  }
  ...
}
```

Figure 2.1: Modularizing variations of the clipboard functionality in table widgets

keyboard (line 9). The type of field clipboard is TableCB, which is an abstract class declaring the methods for the outsourced operations (lines 17-27). These operations can be either abstract or supplied with default implementations. Implementation of the clipboard operations can be varied in the subclasses of TableCB, e.g., TableAppCB (line 29) implements the operations using an application-local clipboard, while TableSystemCB (line 36) provides an alternative implementation using the system clipboard.

As we can see, in this design variations of the clipboard functionality are modularized in separate classes. The design can be extended with new variations of clipboard functionality in a modular way by defining further subclasses of TableCB. Another advantage is support for dynamic variation: the clipboard functionality can be varied at runtime by changing the value of the field clipboard (line 5).

The possibility of outsourcing variations to helper objects is limited in two respects.

First, the outsourced features still leave a "footprint" in the interface and the implementation of the master object. This footprint includes the code for managing the fields that reference helper objects and the methods in the interface of the master object that serve as facades to the outsourced features.

Because of the footprint of the outsourced dynamically varying features, the design of JTable violates both the single-responsibility and the interface segregation principles [Mar03]. Often only a part of the features offered by JTable are needed by particular clients. Part of the optional behavior can be switched off by using flags or providing a dummy implementation of the helper objects that does nothing, e.g., to switch off clipboard functionality we could define a subclass of TableCB with method implementations that do nothing or throw exceptions notifying that an unavailable operations was called. However, in this way we cannot reduce the complexity of the class interface. The interface of the class must support all possible variations of the class.

Ideally, the implementations of basic, optional, and alternative features of a class should be completely separated from each other. This would reduce the complexity of the implementation of the class and would improve its extensibility with new features. Stability of the clients would be improved too, because they would depend only on the functionality that they need.

Second, the modularization works well only as long as we can anticipate variations of the outsourced feature, so that we can define stable interfaces for helper objects abstracting from the variations of that feature. The problem is that when the implementation of a feature is outsourced in a helper object, we need to design a fixed interface that fits all variants of the feature.

For example, consider the interface ListSelectionModel in Fig. 2.2, which is used by JTable for abstracting from the implementation of row and column selection in a table. The

```
interface ListSelectionModel {
   int SINGLE_SELECTION = 0;
   int SINGLE_INTERVAL_SELECTION = 1;
   int MULTIPLE_INTERVAL_SELECTION = 2;

   /** ...
    * In {@code SINGLE_SELECTION} selection mode,
    * this is equivalent to calling {@code setSelectionInterval},
    * and only the second index is used.
    * In {@code SINGLE_INTERVAL_SELECTION} selection mode,
    * this method behaves like {@code setSelectionInterval},
    * unless the given interval is immediately
    * adjacent to or overlaps the existing selection,
    * and can therefore be used to grow the selection.
    * ...
    */
   void addSelectionInterval(int index0, int index1);
   ...
}
```

Figure 2.2: A fragment of ListSelectionModel from Swing library

interface is designed for the most flexible list selection mode, which supports selection of multiple intervals. As a result, the interface is too complicated for simpler selection models, as indicated by the comments of its operations. On the other hand, the design is yet not flexible enough to deal with arbitrary cell range selection in a table widget.

2.2.2 Using Inheritance to Model Variability

A better modularization of structural variations of a class is enabled by inheritance: the common functionality of similar objects can be implemented by a base class and each variation can be implemented by a separate subclass of the base class.

Figure 2.3 shows the modularization of the variations of the selection functionality in table widgets. The class TableBase implements the base functionality of the table widget, such as display of tabular data models. The class TableSel extends the table widget with the functionality that is common for all types of table selection, such as rendering of selected cells. Finally, the classes TableSingleCellSel, TableSingleRowSel, TableRowRangeSel, and TableCellRangeSel implement specific table selection models.

An important advantage of inheritance is that it *can express structural variations of an object*: the available instance variables and operations of an object may be different for different variations. In Fig. 2.3, we introduce different operations and variables for the single cell selection and the single row selection, implemented by the classes TableSingleCellSel and TableSingleRowSel respectively. In this way, we can design the most

```
class TableBase extends Widget {
   TableModel model;
   String getCellText(int row, int col) {
      return model.getCellText(row, col);
   }
   void paintCell(int row, int col, Graphics g) {
      ... getCellText(row, col) ...
   }
   ...
}

abstract class TableSel extends TableBase {
   abstract boolean isCellSelected(int row, int col);
   void paintCell(int row, int col, Graphics g) {
      ... if (isCellSelected(row, col)) ...
   }
   ...
}

class TableSingleCellSel extends TableSel {
   int currRow; int currCol;
   void selectCell(int row, int col) {
      currRow = row; currCol = col;
   }
   boolean isCellSelected(int row, int col) {
      return row == currRow && col == currCol;
   }
   ....
}

class TableSingleRowSel extends TableSel {
   int currRow;
   void selectRow(int row) { currRow = row; }
   boolean isCellSelected(int row, int col) {
      return row == currRow;
   }
   ....
}

class TableRowRangeSel extends TableSel { ... }

class TableCellRangeSel extends TableSel { ... }
```

Figure 2.3: Variations of table selection by inheritance

suitable interface for each type of table selection and do not need to find an interface that fits them all, as in the case when variability is modeled by instance variables.

Furthermore, variation expressed by inheritance can be represented in types of objects. Classes representing specific variations of a table widget can be used as types, e.g., we know that a variable declared with type TableSingleRowSel would always refer to a table supporting single row selection. Representation of variation in types is closely related to the possibility to vary the interface of a class, because it is necessary for a type-safe access of the variation-specific interface.

Another advantage of inheritance is that it reduces the need to anticipate possible variations, because it supports the Open-Closed principle [Mey97, Mar03]: components are closed for change, but open for extension. Indeed, inheritance allows replacing the implementation of an arbitrary method of a class, unless it is explicitly forbidden (e.g., in Java methods can be declared as final). Thus, the developer does not need to plan in advance, which of the methods may vary, and to design an appropriate infrastructure for that purpose. Of course, support for variability in a class may still depend on the granularity of its methods.

The main limitation of inheritance is that it is a static mechanism, unsuitable for modeling dynamic variation. The configuration of a class' implementation may depend on values from the runtime context. For example, various table widget options may come from the dynamic configuration. Depending on the configuration options different compositions of table widget features would have to be instantiated. The mapping from the runtime values to the classes to be instantiated would need to be implemented manually using conditional statements. Although such mapping enables variation binding at the object creation time, it is error-prone and not extensible; when new variants of the class are introduced, the mapping from configuration variables to the subclasses must be updated.

2.2.3 Variation of Functions

So far we were talking about variations of single objects, but variation may also need to be bound at the scope of a particular computation. The conventional approach is to represent a computation as a function, which can use lower level functions as a part of its implementation. In such a setting, the variation scope is delimited by an execution of the function and its subfunctions.

Variation at the scope of a function execution can be expressed by parameters to the function, and the varying functionality would be expressed by conditional statements on the parameter values. The pros and cons of expressing function variations by conditions on parameters are analogous to the ones discussed for object parameterization in Sec. 2.2.1. Although conditionals provide a lot of flexibility for expressing variations, these variations are not modularized, which increases complexity and hinders extensibility. Moreover, the variations of a function must be explicitly passed to the lower level functions in form of parameters.

The functional counterpart of helper objects introduced in Sec. 2.2.1 is functions passed as parameters to other functions. The functions parameterized by other functions are known as higher-order functions. Higher-order functions enable clean separation between generic and varying code, which helps to manage complexity and enables extension with new variants. The problem with higher-order functions is that they support only anticipated variation, which must be exposed by their parameters. If it is not known, which parts of a computation will vary, the most flexible design would be to parameterize the function representing the computation with respect to its all subfunctions. Such design is complicated and produces a lot of glue code.

Again, we can employ inheritance to support unanticipated variations. This can be achieved by declaring the function and its subfunctions as methods of a class. Then variations of the function can be implemented in subclasses of the class, which can selectively override the subfunctions encoded by methods. The advantage of such design is that it makes all parts of a computation replaceable in a type-safe way and with a minimal code overhead.

Another advantage of representing a function by a class is that we can exploit fields for sharing state between subfunctions. In this way we not only avoid the explicit passing of the shared state from function to function, but also make it extensible. In subclasses extending the function, we can introduce new instance variables, which are automatically passed over method calls as the self reference, and can be accessed in a type-safe way in the methods of the subclass.

The conclusion is that in a lot of cases it is advantageous to implement non-trivial computations as classes for a better support of their variations. This means that the variation mechanisms of objects can also be applied to deal with variations of functions. In principle there is no strict distinction between objects and functions. A function can be seen as an object with a somewhat specific lifecycle: while an object is intended to interact with other objects over multiple messages, a typical function is intended to be called once from the outside and return one or more result values. But again, this distinction is not very strict: a computation represented by an object may need to be configured by multiple method calls before executing it; a computation may also need to interact with other application objects in order to provide feedback on the computation status or the possibility to be interrupted.

2.3 Multiple Variations of Individual Objects

In general, the functionality of an abstraction can vary along several dimensions, which might be not completely orthogonal. In this section, we consider issues related to modeling interactions between several variations. We discuss various combinations of variations expressed by inheritance and the outsourcing to helper objects. Then in Sec. 2.3.4 we discuss specific advantages of multi-dispatch for dealing with the combination of variations. For the sake of keeping the discussion simple, mostly two variations will be considered in the examples.

2.3.1 Modeling Variations with Instance Variables

When variations are modeled uniformly by instance variables and conditional logic, interactions between different variations can be programmatically expressed at a very fine-grained level, but also at the cost of losing modularity and, hence, independent extensibility. Also, there is no way one can statically state and check any constraints on the permitted interactions.

Helper objects are an effective means for modeling multiple features of an object with dynamically varying implementations, but with fixed interfaces, as long as there are no interactions between these features. Helper objects cannot be used, however, to modularize code that depends on multiple independent variation points. For example, JTable class in Java Swing library uses the interface TableCellRenderer to abstract from different ways to render table cells. The problem is that cell rendering may depend on other kinds of variations, e.g., on the presence of selection, or drag-and-drop functionality. This is because selected cells and drag-and-drop targets must be rendered in a special way. Such interdependencies of different variations cannot be appropriately modularized using helper objects only.

2.3.2 Modeling Variations with Inheritance

Dependencies between different variations can be expressed by inheritance relationships between the classes modeling corresponding variations. Independent variations can be combined by multiple inheritance, if such a mechanism is supported by the language.

Analogously to the selection model, we can modularize other variations of the table widget: support for resizing of columns, resizing of rows, individual cell coloring, coloring rules and so on. The new features can be implemented in new subclasses of the base table widget as shown in Fig. 2.4.

```
class TableColumnResize extends TableBase { ... }

class TableCellColors extends TableBase {
   Color getCellColor(int row, int col) { ... }
   void paintCell(int row, int col, Graphics g) {
      ... getCellColor(row, col) ...
   }
   ...
}

...

class TableSingleCellSelColumnResize
   extends TableSingleCellSel & TableColumnResize { }

class TableSingleRowSelColumnResize
   extends TableSingleRowSel & TableColumnResize { }

class TableSelCellColor extends TableSel & TableCellColors {
   Color getSelectedCellColor(int row, int col) { ... }
   void paintCell(int row, int col, Graphics g) {
      ... getSelectedCellColor(row, col) ...
   }
   ...
}
...
```

Figure 2.4: Modularizing variations of table by inheritance

Since Java supports only single inheritance, there is no way to combine different variations of the table widget in order to use its different features. This is, however, not a problem in many other object-oriented languages that support some form of multiple inheritance. In Fig. 2.4, we assume that classes can be combined by mixin-composition as, e.g., implemented in CaesarJ [AGMO06, EOC06].

The problem with multiple inheritance is that we must define a class for every combination of the varying features of a class. Such a solution does not scale, because with each new variation point the number of possible combinations of variants grows exponentially.

Fortunately, the functionality of most of variations is independent, as e.g., various table selection types and resizing of table columns. The classes implementing independent variations can be simply composed by mixin-composition without any additional glue code. So, a viable solution could be to implement only classes that define new functionality as part of the library and leave their composition to the users of the table widget.

A library designed in this way is, however, much more difficult to use. Instead of simply configuring a class by assigning values to its fields, the users must know how to define a correct class composition that corresponds to the required configuration. So, the user of the library must know which variations have interactions and which classes resolve these interactions.

For example, a user may wrongly assume that to obtain a table widget with resizable columns, single cell selection, and individual cell coloring, the classes implementing corresponding variations, i.e., TableColumnResize, TableCellColors, and TableSingleCellSel, should be composed. The correct solution would be, however, to compose classes TableColumnResize, TableSelCellColor, and TableSingleCellSel, because the class TableSelCellColor resolves the interactions between cell coloring and table selection, i.e., implements coloring of selected cells.

Besides, if all possible combinations of variations are not predefined in the library, there is no guarantee that the library provides classes for resolving all possible interactions of the intended combinations.

Such a design also hinders the evolution of the library. For example, if during the evolution of the library we identify that column resizing must be handled in a slightly different way for tables with selection, we would like to introduce a new class TableSelColumnResize that handles this variation. Such refactoring can break the code of the users of the library, because their compositions of classes may need to be changed.

As was mentioned in Sec. 2.2.2, variation modeled by inheritance can also be bound dynamically during object instantiation. The mapping from the configuration options to the corresponding implementations of an object requires a conditional logic that instantiates different combinations of classes depending on the values of these options. Not only such code is error-prone and not extensible, but it also does not scale, because it grows exponentially with respect to the number of configuration options and forces to define classes for all concrete configurations.

2.3.3 Combining Inheritance and Helper Objects

As discussed in Sec. 2.2, inheritance and object composition provide different advantages for expressing variation. Object composition is used to implement dynamic variations, but for static variations inheritance is often preferred, because it supports structural variations of classes. Thus, these two mechanisms are often used together to express different variations of an object. In this subsection, we use an example from Java Swing library to analyze the problems that appear in the interaction of these two variation mechanisms.

Object composition supports dynamic variation of an object by outsourcing the varying functionality to helper objects, as described in Sec. 2.2.1. In this way implementation of a logical object is split into multiple objects implementing different parts of its behavior. Such objects tend to be covariant with respect to other variation points of the logical object.

An example that illustrates the need for covariant dependencies between classes and the problems of current mechanisms in this respect concerns the variations of visualization functionality in the Java Swing library. Visual representation of a widget is mainly determined by its type, e.g., a tree has a different representation than a button or a dialog. In addition, the Swing library provides a possibility to change the look-and-feel style of widgets in a running application. Thus visualization of a widget also depends on the selected style.

Figure 2.5 shows an excerpt of a slightly simplified implementation of look-and-feel functionality in the Swing library. The visualization related methods are defined in the strategy interface ComponentUI. The base class for widgets, JComponent, delegates these methods to the strategy object referenced by the variable ui. For each pair of a widget type and a look-and-feel style, there is a class that implements the corresponding visualization functionality. For example, the classes BasicListUI and MotifListUI in Fig. 2.5 implement the visualization functionality of the list widget for the basic look-and-feel style and the Motif toolkit style respectively.

Whenever the look-and-feel style is changed, the widget is notified by calling the method updateUI. The latter uses UIManager to create a visualization implementation – an instance of ComponentUI) – that corresponds to the type of the widget and the current look-and-feel style.

There are several problems with the design of the library.

The first problem concerns the covariant dependency between a widget and its visualization helper. For example, the precise positions of list items depend on the visualization style of the list. Therefore, the interface of list visualization helpers, ListUI, defines additional methods to get information about the locations of the list items (e.g., locationToIndex on line 31). These methods are used in the implementation of JList (line 16). However, the type system of Java cannot specify that instances of JList can be composed only with instances of ListUI. Hence, a type cast is needed to access the additional methods of ListUI.

An analogous problem exists in the opposite direction: widgets pass themselves to visualization objects (see installUI on line 4). However, subclasses of ListUI can work only with instances of JList, hence they must cast the parameter to the expected type (line 38).

Another problem of the library is related to the instantiation of the visualization objects. The class of the visualization helper depends on both the type of the widget to be vi-

```
abstract class JComponent {
    ComponentUI ui;
    String getUIClassID() { return "ComponentUI"; }
    void updateUI() {
        ui = UIManager.getUI(this);
        ui.installUI(this);
    }
    void paint(Graphics g) { ui.paint(g); }
    ...
}

class JList extends JComponent {
    String getUIClassID() { return "ListUI"; }
    ListModel getModel() { ... }
    String getTooltipText(Point pt) {
        int idx = ((ListUI)ui).locationToIndex(pt);
        ...
    }
    ...
}
...

interface ComponentUI {
    void installUI(JComponent c);
    void paint(Graphics g);
    Dimension getPreferredSize(JComponent c);
    ...
}

interface ListUI extends ComponentUI {
    int locationToIndex(Point pt);
    ...
}

class BasicListUI implements ListUI {
    JList list;
    void installUI(JComponent c) {
        list = (JList)c; ...
    }
    void paint(Graphics g) {
        ... list.getModel() ...
    }
    ...
}

class MotifListUI extends BasicListUI {
    void paint(Graphics g) { ... }
    ...
}
```

Figure 2.5: Separation of variations of visualization styles in Swing

```
 1 class UIManager {
 2    static LookAndFeel style;
 3    static ComponentUI getUI(JComponent c) {
 4       String uiCls = style.uiTable.get(c.getUIClassID());
 5       /* load and instantiate uiCls */
 6    }
 7    ...
 8 }
 9
10 abstract class LookAndFeel {
11    Hashtable<String, String> uiTable;
12    abstract void initialize();
13    ...
14 }
15
16 class BasicLookAndFeel extends LookAndFeel {
17    void initialize() {
18       uiTable.put("ListUI", "BasicListUI");
19       uiTable.put("ButtonUI", "BasicButtonUI");
20       ...
21    }
22    ...
23 }
24
25 class MotifLookAndFeel extends BasicLookAndFeel {
26    void initialize() {
27       uiTable.put("ListUI", "MotifListUI");
28       uiTable.put("ButtonUI", "MotifButtonUI");
29       ...
30    }
31    ...
32 }
```

Figure 2.6: Instantiating visualization objects in Swing

sualized and the look-and-feel style selected in the application. Unfortunately, the type system and the dispatch mechanism of Java are not capable of expressing such a dependency. Complex and not type-safe designs based on reflection and factory infrastructure are used as workarounds.

A simplified version of the code of the Swing library responsible for selecting and instantiating visualization classes is shown in Fig. 2.6. The singleton class UIManager keeps track of the currently selected look-and-feel style (referenced by the variable style); its method getUI creates a visualization helper for the given widget and the current style by using Java reflection. Every widget class implements getUIClassID, which returns a textual identifier of the visualization class required by the widget (e.g., see lines 3 and 13 in Fig. 2.5). Further, each look-and-feel style class maintains a table that maps these identifiers to the names of the visualization classes for this style (lines 10-32, Fig. 2.6). Finally, the selected class is instantiated using Java reflection.

```
abstract void paintWidget(JComponent comp, LookAndFeel lf, Graphics g);
void paintWidget(JList comp, BasicLookAndFeel lf, Graphics g) { ... }
void paintWidget(JList comp, MotifLookAndFeel lf, Graphics g) { ... }
void paintWidget(JButton comp, BasicLookAndFeel lf, Graphics g) { ... }
...
```

Figure 2.7: Expressing variations of widget rendering by multi-dispatch on the type of the widget and the type of the look-and-feel style

Such a mechanism is very flexible, because it allows installing new look-and-feel styles at runtime. However, the usage of reflection has several disadvantages with regard to type safety and IDE tool support (e.g., refactoring or dependency analyses).

An alternative solution would use the Abstract Factory design pattern [GHJV95]: the abstract factory would contain methods for the creation of visualization implementations for different types of widgets; there would be a concrete factory for each look-and-feel style implementing these methods correspondingly. A solution based on the factory pattern has its own disadvantages: it makes it difficult to add new types of widgets and introduces undesired interdependencies between widgets, because the abstract factory knows about all widgets and all widgets know about the abstract factory.

2.3.4 Combining Variations with Multi-Dispatch

Multidimensional variation can be expressed by means of method dispatch by multiple parameters, known as *multi-dispatch* or *multimethods*. For example, dependency of widget rendering functionality on the type of widget and the look-and-feel style could be expressed by a multi-dispatched paintWidget method, as shown in Fig. 2.7. The method takes a widget, a look-and-feel style as parameters and has different implementations for different pairs of concrete types of these parameters.[2]

The design with multimethods modularizes dependencies of a method on different combinations of variants of independent variation points, e.g., one variant point representing different widget types, and another one representing different look-and-feel styles. As a result, the method can be extended with respect to new variants of all variation points on which it depends, e.g., method paintWidget can be extended with respect to both new widget types and new look-and-feel styles.

Consistency of the implementation of a multimethod can be ensured by two kinds of static checks. *Completeness checking* (also known as exhaustiveness checking) ensures

[2]The third parameter of the method is not used for dispatch in the example, but we could also specialize the method for subclasses of Graphics, e.g., in order to make use of the graphical operations supported by specific graphical contexts.

that the method is completely implemented for all possible combinations of parameter values. Multi-dispatch also makes it possible to provide various default implementations of the method, e.g., if we define a new look-and-feel style class as a subclass of BasicLookAndFeel, it would inherit all implementations of paintWidget except for the overridden cases. *Uniqueness checking* (also known as ambiguity checking) ensures that for every method call the most specific implementation of the method can be uniquely selected. In this way it is guaranteed, that default implementations of the method are not conflicting. Compleness and uniqueness checking help to prevent accidental errors caused by forgetting to implement a method for certain cases or by giving conflicing implementations. These checks are especially important for ensuring implementation consistency when a method is extended with new variations.

Multi-dispatch of methods is able to express variations of individual methods only. In Fig. 2.7, we simplified the widget visualization functionality, introduced in Sec. 2.3.3, to a single paintWidget method. In the original example, the visualization functionality is implemented by objects containing multiple operations and visualization-related state. The state is necessary for storing the configuration related to widget visualization and various cached computation results. Thus, for a full support of variation of widget visualization we need to express variations of objects rather than individual methods.

Although multimethods cannot directly express dependencies of objects on multiple variations points, they can be used for polymorphic instantiation of objects with respect to multiple parameters. For example, visualization helpers of widgets can be instantiated by a factory method createUI dispatched both by the widget and the look-and-feel style, as shown in Fig. 2.8.

Such design provides the same extensibility properties as instantiation of helpers using reflection, presented in Sec. 2.3.3, at the same time avoiding the problems of reflection. We can extend the library both with new types of widgets and with new types of look-and-feel styles by implementing new subclasses of ComponentUI and instantiating them in corresponding new cases of createUI. Differently from the solution based on reflection, it is statically checked if the instantiated classes exist and are instances of ComponentUI. Moreover, it is statically checked if createUI is completely implemented for all variants of widgets and look-and-feel styles, and if there are no conflicting default implementations of the method.

In general, multi-dispatched factory methods provide a solution for implementing the mapping from the selected configuration to the instantiated classes. As was discussed in Sec. 2.2.2 and 2.3.2, such mapping can also be implemented using conditional statements. The advantage of a multi-dispatched factory method is that it is extensible with respect to new variants, because we can extend the factory method with new cases in a modular way. It is also more reliable than conditional statements, because completeness and

```
abstract ComponentUI createUI(JComponent comp, LookAndFeel lf);
ComponentUI createUI(JList comp, BasicLookAndFeel lf) {
   return new BasicListUI();
}
ComponentUI createUI(JList comp, MotifLookAndFeel lf) {
   return new MotifListUI();
}
ComponentUI createUI(JButton comp, BasicLookAndFeel lf) {
   return new BasicButtonUI();
}
...
abstract class JComponent {
   ComponentUI ui;
   void updateUI() {
      ui = createUI(this, UIManager.style);
      ui.installUI(this);
   }
   void paint(Graphics g) { ui.paint(g); }
   ...
}
```

Figure 2.8: Dispatching instantiation of helper objects for widget visualization both by the type of the widget and the type of the look-and-feel style

uniqueness checking ensure that the method is implemented for all possible cases and detect conflicting default implementations of the method.

Nevertheless, multi-dispatched factory methods do not solve the scalability problem, because we still need to combine variations of an object using multiple inheritance, which means that we still need to define a class for every valid combination of variants and an implementation of the factory method instantiating that class. Also, note that the design of Fig. 2.8 provides a solution for instantiation of helper objects, but not to the typing problems discussed in Sec. 2.3.3.

2.4 Variation of Multiple Objects

So far we considered variations of individual objects, but variations may also affect groups of objects. Examples of such object groupings range from relatively simple data structures such as trees and graphs to sophisticated frameworks and applications. The classes of the objects of such groups can related in different ways: by references to each other in signatures of methods and fields, by instantiation, by inheritance, and even by shared state and dependencies.

Although variations of a group objects can be reduced to combinations of individual variations of each element of the group, encoding such combinations may require additional glue code and preplanning. Moreover, sharing variations among multiple objects may create covariant dependencies between them, which are difficult to express in a type safe way. Variations involving multiple objects also introduce further interesting scenarios for interactions of variations, because an object can be influenced both their individual variations as well as variations of the groups it belongs to.

In the first part of this section we investigate how variations affecting multiple objects can be modularized and composed using simple class inheritance. Then, in Sec. 2.4.3, we discuss variations that are bound at the scope of the entire application and special techniques available for that purpose. In Sec. 2.4.4, we also explain the need to bind variation at multiple scopes and give a concrete example of such situation. Sec. 2.4.5 discusses the ways of dealing with dynamic variations affecting multiple objects.

2.4.1 Variations with Inheritance

As was explained in Sec. 2.2.2, inheritance provides a lot of flexibility for modularizing static variations of an object, because it make it possible to modularize not only variations of implementations of methods, but also variations of object's state and interface. Static variations of a group of classes can be expressed by applying inheritance for each class from the group separately.

For illustration we will use an example of variations of menu structures. A menu is a GUI component consisting of a list of menu items corresponding to different application-specific actions. Menus are usually organized hierarchically: a menu item can be associated with a cascade menu, which pops up when the item is selected. Some of the menu items can be check-boxes, which change their state when selected, or radio-buttons, which implement a choice from multiple alternatives.

A possible implementation of a menu structure is demonstrated in Fig. 2.9. A menu, represented by class Menu (line 27), maintains a list of menu items. Simple menu items are implemented by class MenuItem (line 1), which is subclassed to implement specialized menu items: class CheckMenuItem (line 23) for check-box menu items, class RadioMenuItem (line 25) for radio-button menu items, and CascadeMenuItem (line 15) for menu items that open cascade menus. A CascadeMenuItem contains a reference to an instance of class PopupMenu (line 44), which is a subclass of Menu implementing pop-up menus. MenuBar (line 46) is another subclass of Menu, implementing a menu bar which is usually displayed at the top of a window and serves as the root object of the menu structure.

As we can see, a menu is a complex structure consisting of multiple objects: the menu bar, menu items, and cascade menus. These objects are implemented by different classes, which are organized into inheritance hierarchies to represent variations of the elements of the structure – the subclasses of MenuItem implement different types of menu items,

```
class MenuItem {
    String label; Action action;
    MenuItem(String label, Action action) {
        this.label = label; this.action = action;
    }
    String displayText() {
        return label;
    }
    void draw(Graphics g) {
        ... displayText() ...
    }
    ...
}

class CascadeMenuItem extends MenuItem {
    PopupMenu menu;
    void addItem(MenuItem item) {
        menu.addItem(item);
    }
    ...
}

class CheckMenuItem extends MenuItem { ... }

class RadioMenuItem extends MenuItem { ... }

abstract class Menu {
    List<MenuItem> items;
    MenuItem itemAt(int i) {
        return items.get(i);
    }
    int itemCount() {
        return items.size();
    }
    void addItem(MenuItem item) {
        items.add(item);
    }
    void addAction(String label, Action action) {
        items.add(new MenuItem(label, action));
    }
    ...
}

class PopupMenu extends Menu { ... }

class MenuBar extends Menu { ... }
```

Figure 2.9: Base implementation of menus and menu items

and the subclasses of Menu implements different forms of menu lists. The classes refer to each other in the signatures of their fields (e.g., lines 16 and 28) and methods (e.g., lines 17, 29 and 35).

There is a variety of optional features related to menu functionality: support for accelerator keys for a quick selection of a menu item using specific key stroke, support for multi-lingual text in menu items, support for context help, etc. Variations of menu functionality affect multiple objects constituting the menu structure. Since these objects are implemented by different classes, we need multiple new subclasses to modularize such variations.

In order to understand the problems of modularizing variations affecting a structure of objects, let's analyze an extension of the base menu functionality with support for accelerator keys, presented in Fig. 2.10. An extension of menu items with the new feature is implemented in class MenuItemAccel (line 1), which is declared as a subclass of MenuItem. The extension not only changes the implementation of the existing methods, e.g., adapts method draw to display the accelerator key besides the label of the item (line 13), but also extends the class with new fields and methods: field accelKey (line 2) for storing the key associated to the menu item, method setAccelerator (line 3) for changing this association, and method processKey (line 10) for processing an input key.

When a key is pressed, the action of the menu item associated with that key must be triggered. To implement this functionality, class MenuAccel (line 33) extends Menu with an operation processKey, which takes an input key and propagates it for processing in the items of the menu, by calling the method with the same name. For a simple menu item, this method compares the input key with the key associated with the item and triggers its action if they are equal (line 5). For a cascade menu item, the key processing is additionally propagated to its pop-up menu (line 22).

The presented design suffers from a set of problems.

First, the type declarations do not express covariant dependencies between the objects of a group. The varying functionality of an object may need to access the corresponding varying functionality of another object of the group. This is problematic, because references between objects are typed by invariant types, which provide fixed interfaces. For example, the method processKey in a menu needs to call processKey on its items, accessed using the method itemAt (line 36). The method itemAt is inherited from class Menu, where it was declared with return type MenuItem. Thus, to access the extended functionality of menu items, we must cast the result of itemAt to MenuItemAccel. The design cannot guarantee that such type cast will always be successful, because menu items of MenuAccel are added over the inherited method addItem, which accepts all menu items, both with and without the accelerator functionality.

```
class MenuItemAccel extends MenuItem {
    KeyStroke accelKey;
    boolean processKey(KeyStroke ks) {
        if (accelKey != null && accelKey.equals(ks)) {
            performAction();
            return true;
        }
        return false;
    }
    void setAccelerator(KeyStroke ks) {
        accelKey = ks;
    }
    void draw(Graphics g) {
        super.draw(g);
        displayAccelKey();
    }
    ...
}

class CascadeMenuItemAccel extends CascadeMenuItem & MenuItemAccel {
    boolean processKey(KeyStroke ks) {
        if (((PopupMenuAccel)menu).processKey(ks)) {
             return true;
        }
        return super.processKey(ks);
    }
}

class CheckMenuItemAccel extends CheckMenuItem & MenuItemAccel { ... }

class RadioMenuItemAccel extends RadioMenuItem & MenuItemAccel { ... }

abstract class MenuAccel extends Menu {
    boolean processKey(KeyStroke ks) {
        for (int i1 = 0; i1 < itemCount(); i1++) {
            if (((MenuItemAccel)itemAt(i)).processKey(i1)) {
                return true;
            }
        }
        return false;
    }
    void addAction(String label, Action action) {
        items.add(new MenuItemAccel(label, action));
    }
    ...
}

class PopupMenuAccel extends PopupMenu & MenuAccel { }

class MenuBarAccel extends MenuBar & MenuAccel { }
```

Figure 2.10: Extension of menu functionality with accelerator keys

```
interface MenuContributor {
   void contribute(Menu menu);
}

class FileMenuContrib implements MenuContributor {
   void contribute(Menu menu) {
      CascadeMenuItem openWith = new CascadeMenuItem("Open With");
      menu.addItem(openWith);
      MenuItem openWithTE = new MenuItem("Text Editor", createOpenWithTEAction());
      openWith.addItem(openWithTE);
      ...
      MenuItem readOnly = new CheckMenuItem("Read Only", createReadOnlyAction());
      menu.addItem(readOnly)
      ...
   }
   ...
}
```

Figure 2.11: Menu contribution with operations on files

Second, variations of a group of objects are not combined with the individual variations of its elements. In particular, it is difficult to encode group variations that affect objects independently of their individual variations. For example, MenuItemAccel extends only the functionality of simple menu items with support for accelerator keys, but this functionality must also be available for other types of menu items, such as check-box and radio-button menu items. For this, we must explicitly combine MenuItemAccel with the subclasses of MenuItem implementing specific types of menu items using multiple inheritance (lines 29 and 31).

In a language like Java, which supports only multiple inheritance of interfaces, the code of the extension must be replicated for each subclass. The design with multiple inheritance is not optimal too, because it requires a set of additional class declarations that explicitly combine the extended element class with the classes describing its individual variations. Such design not only produces excessive number of classes, but is also not stable with respect to extensions with new element types, because the developer must not forget to extend the existing variations of the composite with combinations for the new element types.

Third, the code directly instantiating the classes of a group of objects cannot be reused with different variations of the group. For example, MenuItem is instantiated in method addAction of class Menu (line 39, Fig. 2.9). In the extension with support for accelerator keys, we must override this method so that it instantiates the extended version of the class (line 43, Fig. 2.10). In some situations, such overridings can cause a cascade effect: an extension of a class requires extensions of classes that instantiate it, which can again require extensions of further classes that instantiate them.

Fig. 2.11 gives an example of client code instantiating menus. A menu of an application can be built from different reusable pieces, provided by different menu contributors. Fig. 2.11 shows implementation of a menu contributor for operations on files. It implements method contribute, which extends the given menu object with menu items to open files with different text editor, to change the read-only flag of the file, and so on. Since the menu items are created by directly instantiating the respective classes (lines 7, 9, and 12), this piece of code cannot be reused for menus with support for key accelerators, or any other extensions of the menu functionality.

The code of Fig. 2.11 can be made reusable with variations of menu functionality by instantiating menu items indirectly using the Abstract Factory design pattern [GHJV95]. Such design is shown in Fig. 2.12: the class FileMenuContrib instantiates menu items (lines 29, 31, and 34) using respective methods defined in interface MenuFactory (lines 2-4); the class BaseMenuFactory (line 4) implements this interface for menus with base functionality only, and AccelMenuFactory (line 17) for menus with support for accelerator keys. The code of FileMenuContrib can be reused with different variations of menu functionality, by using it with different factory implementations.

The Abstract Factory design pattern enables abstraction from group variations by providing late-bound instantiation of classes. It, however, does not provide an optimal solution. The infrastructure for the design pattern must be manually implemented and maintained. Further, correct usage of the pattern cannot be completely enforced: there is no guarantee that classes are instantiated exclusively over the factory methods, and that only objects instantiated by the same factory are used together. There is also no good solution for managing the reference to the abstract factory. The factory can be implemented as a Singleton [GHJV95] to ensure convenient access to it within the entire application. Such solution is not always possible, however, because an application may need multiple possibly different menu structures. In the latter case, appropriate factory objects must be explicitly passed to the objects that need to instantiate menus and menu items, which introduces additional glue code.

The fact that the type system fails to ensure that only the objects of the same factory are used together is a manifestation of a more general problem. Polymorphism as a means for a client to abstract from the variations of an object does not properly work for a group of objects. A client of a menu structure can abstract from the variations of the menu by using the base classes of menu structure from Fig. 2.9 in combination with the abstract factory. An example of such client is the class FileMenuContrib in Fig. 2.12. The problem is that such polymorphic usage is not type safe, because it is not ensured that only the objects sharing the same group variations are used together.

```
interface MenuFactory {
    MenuItem createMenuItem(String name, Action action);
    CascadeMenuItem createCascadeMenuItem(String name);
    ...
}

class BaseMenuFactory implements MenuFactory {
    MenuItem createMenuItem(String name, Action action) {
        return new MenuItem(name, action);
    }
    CascadeMenuItem createCascadeMenuItem(String name) {
        return new CasadeMenuItem(name);
    }
    ...
}

class AccelMenuFactory implements MenuFactory {
    MenuItemAccel createMenuItem(String name, Action action) {
        return new MenuItemAccel(name, action);
    }
    CascadeMenuItemAccel createCascadeMenuItem(String name) {
        return new CasadeMenuItemAccel(name);
    }
    ...
}

class FileMenuContrib implements MenuContributor {
    void contribute(Menu menu, MenuFactory factory) {
        MenuItem openWith = factory.createCascadeMenuItem("Open With");
        menu.addItem(openWith);
        MenuItem openWithTE = factory.createMenuItem("Text Editor", createOpenWithTEAction());
        openWith.addItem(openWithTE);
        ...
        MenuItem readOnly = factory.createCheckMenuItem("Read Only", createReadOnlyAction());
        menu.addItem(readOnly)
        ...
    }
    ...
}
```

Figure 2.12: Instantiation of menus using the Abstract Factory pattern

```
1  class MenuItemML extends MenuItem {
2     String displayText() {
3        return translateText(label);
4     }
5     ...
6  }
7
8  class CascadeMenuItemML extends CascadeMenuItem & MenuItemML { }
9
10 class CheckMenuItemML extends CheckMenuItem & MenuItemML { }
11
12 class RadioMenuItemML extends RadioMenuItem & MenuItemML { }
```

Figure 2.13: Menus with multi-language support

2.4.2 Combining Variations with Inheritance

Variations modularized by inheritance can be composed by means of multiple inheritance, as was discussed in Sec. 2.3.2. Combining variations of a composite structure are, however, more complicated, because we need to compose multiple classes.

For example, we may need to combine support of accelerator keys in menus with other optional features of menu functionality, such as support for multi-language.

Fig. 2.13 demonstrates an extension of menu functionality with multi-language support: the class MenuItemML inherits from MenuItem, overrides the method displayText (line 2) so that it returns a label translated to the currently selected language; further this extension is made available for specific types of menu items by combining MenuItemML with the respective subclasses of MenuItem (lines 8-12).

In order to have menus with both accelerator keys and multilingual labels, the classes implementing respective extensions must be combined with each other, as shown on lines 1-7 of Fig. 2.14. Since multi-language support extends only the functionality of menu items, we need to combine only the classes implementing menu items, while for menus we can still use MenuAccel and its subclasses from Fig. 2.10. Further, if we use Abstract Factory pattern for enabling abstraction from instantiated classes, as discussed in Sec. 2.4.1, we also have to implement a new factory class with the methods instantiating the classes that combine the variations (lines 9-17).

As we can see, combinations of variations of composite abstractions require a lot of new classes, which further amplifies the problem of combinatorial explosion of classes discussed in Sec. 2.3.2: The number of classes representing combinations of variations of an individual object, which grows exponentially with respect to the number of independent variation points, must be additionally multiplied by the number of classes constituting the group.

```
class MenuItemAccelML extends MenuItemAccel & MenuItemML { }

class CascadeMenuItemAccelML extends CascadeMenuItemAccel & CascadeMenuItemML { }

class CheckMenuItemAccelML extends CheckMenuItemAccel & CheckMenuItemML { }

class RadioMenuItemAccelML extends RadioMenuItemAccel & RadioMenuItemML { }

class AccelMLMenuFactory implements MenuFactory {
   MenuItemAccelML createMenuItem(String name, Action action) {
      return new MenuItemAccelML(name, action);
   }
   CascadeMenuItemAccelML createCascadeMenuItem(String name) {
      return new CasadeMenuItemAccelML(name);
   }
   ...
}
```

Figure 2.14: Combining support for accelerator keys and multi-language

Furthermore, such combinations are not stable with respect to the evolution of the varying functionality. For example, the current implementation of multi-language support affects only menu items. Now if we consider that this functionality is further evolved, it may happen that Menu and its subclasses may also need to be extended to make certain texts in the menu objects multilingual. In such a case, the combination of multi-language and accelerator features will have to be extended to define combinations for the menu classes. Since a developer of a particular variation may be not aware of all its uses, the need to update combinations of features may remain undetected and cause unexpected errors.

2.4.3 Application-Level Variations

Variation at the scope of the entire application can be considered as a special case of a group variation, where the group consists of all application objects. Nevertheless, this special case is interesting, because there are additional techniques for dealing with variations at the scope of the entire application.

Static application-level variations can be managed by decomposing the application code into modules so that code implementing different variations is placed into different modules. In such a design, variation can be bound by selectively including modules to a particular build or a distribution of the application. The possibility of managing variation in this way very much depends on the flexibility of decomposing software into modules supported by a particular programming language. Popular object-oriented languages, such as Java and C++, enforce relatively rigid modularization structure, which

```
 1 class MenuAccel {
 2    static Map<MenuItem, KeyStroke> accelKeys = new HashMap<MenuItem, KeyStroke>();
 3
 4    static void setAccelerator(MenuItem mi, KeyStroke ks) {
 5       accelKeys.put(mi, ks);
 6    }
 7
 8    static boolean processMenuItemKey(MenuItem mi, KeyStroke ks) {
 9       if (mi instanceof CascadeMenuItem) {
10          CascadeMenuItem cmi = (CascadeMenuItem)mi;
11          if (processMenuKey(cmi.menu, ks) {
12             return true;
13          }
14       }
15       if (accelKeys.get(mi) != null && accelKeys.get(mi).equals(ks)) {
16          mi.performAction();
17          return true;
18       }
19       return false;
20    }
21
22    static boolean processMenuKey(Menu menu, KeyStroke ks) {
23       for (int i1 = 0; i1 < menu.itemCount(); i1++) {
24          if (processMenuItemKey(menu.itemAt(i1), ks) {
25             return true;
26          }
27       }
28       return false;
29    }
30    ...
31 }
```

Figure 2.15: Extension of menu functionality with accelerator keys

requires that all functionality of a class is declared within a single module.

In Java, new operations for existing object types can be defined as static methods in newly introduced classes; hash maps can be used to attach new data to existing objects. For example, if all menus of an application are assumed to have the same functionality, the variations of menu functionality can be managed at the level of modules. Fig. 2.15 shows an alternative implementation of the support for menu accelerators. New operations for instances of MenuItem and Menu are defined as static methods taking them as parameters (lines 4, 8 and 22). The accelerator keys are attached to menu items by a corresponding hash map structure (line 2).

Such design does not suffer from the problems discussed in Sec. 2.4.1 and 2.4.2, because the new functionality is defined for existing classes rather than for their subclasses. Since the new operations are available for all menus and menu items, they can be safely applied on the existing relationships between objects, e.g., the calls on lines 11 and lines 24 of Fig. 2.15 avoid type-casts that were necessary at corresponding places of Fig. 2.10. The other two problems are also avoided. Since we do not define new classes to introduce the variations, there is no need to abstract from instantiated classes, or to use multiple inheritance for defining combinations of variations. In must be emphasized though, that this design is suitable only for application-level variations, because the operations and data defined in this way are made available for all menus in the application.

Although new operations can be defined outside classes in form of static methods, the disadvantage is that such methods are not dispatched by dynamic type of the object and, thus, must use conditional statements to handle objects differently depending on their type. As a result, the methods must be modified, when new object types requiring special handling are introduced. The situation when a design provides a good support only for extensions with respect to either new object types or new operations, is known as expression problem. The solutions to the expression problem [Tor04, OZ05a] either give up some type safety, or are based on language extensions that support more flexible modularization. An overview of the techniques for more flexible modularization is given in Sec. 6.2.

Another problem is that it is difficult to replace or refine implementation of existing operations, e.g., changing implementation of MenuItem.draw in order to display the menu accelerators. Analogously, it is also difficult to extend the initialization functionality implemented by constructors. At the level of modules this means that a module must be able to refine a method or constructor defined in another module. In a language such as Java, such refinements can be explicitly supported by some form of callback registries, but they must be planned in advance and require a significant amount of glue code, which in a lot of cases outweigh the benefits of such design.

2.4.4 Variation at Multiple Scopes

As was discussed in this section, variation can be bound at different scopes: for an object or for a group of objects. Since objects may be grouped in different ways, we can also consider cases when objects are grouped in different ways to bind different kinds of variations. Thus, in general, an object can be affected by group variations at different scopes.

For example, applications working with different kinds of documents may contain variations both on the scope of the application as well as on the scope of a document. Application-level variations determine the functionality available in the entire application, while document-level variations determine functionality specific to a document or a document type. Since a document is typically a composite structure, both kinds of variations would affect groups of objects.

As a specific example, consider an application for project planning and tracking. A simple project may consist just of a set of tasks assigned to a team of people. Larger projects may involve multiple teams and consist of multiple phases. Projects may also differ by their planning and tracking methodology, tracked metrics and their measurement units. On the other hand, the application may need to be adapted for company-specific needs, e.g., provide company-specific reports, and be integrated with other software systems of the company, e.g., the employee register or the document management system.

The typical solution for such a scenario is to use a data model that is sophisticated enough to incorporate all kinds of variations, as illustrated in Fig. 2.16. Certain classes, e.g., Project, Task, and Member, relationships between them, and a part of their attributes, e.g., duration and dependencies of tasks, are necessary for all projects. Other classes and attributes are necessary only for specific types of projects or for company-specific integrations with other systems. For example, class Team and its relationships with other classes are necessary for multi-team projects only. Analogously, class Phase and its relationships are necessary for multi-phase projects only. As a part of the integration to the employee register of the company, class Member contains a reference to Employee. Analogously, Task refers to a Document from the document management system, containing description of the task.

In such a design, attributes and operations that are not necessary for a particular project or a particular instance of the application are simply not used. The design suffers from the typical problems of not modularized variations: it is difficult to extend with new variants (e.g., new project types or integrations to other systems), it creates excessive dependencies between different features, and is less stable. For example, since integrations with other subsystems, such as the employee register and the document management system, are not modularized, the enire code is potentially vulnerable to changes in these subsystems.

All these variations can be modularized and composed by means of inheritance, as described in Sec. 2.4.1 and 2.4.2, with analogous advantages and problems. Differences of variation binding scope are not very important when modularizing variation with class inheritance, because variations are managed at level of individual classes anyway, and type-casts are necessary to express covariant dependencies between objects. Nevertheless, in the chapters of on virtual classes and dependent classes we will see that variation binding at different scopes require specific solutions for expressing variations in a type-safe way.

```
class Project {
   boolean isMultiTeam; boolean isMultiPhase; ...
   List <Member> members; List<Task> tasks;
   ...
}

class Task {
  double duration;   List <Task> dependsOn; Member assignedTo;
  Document description; /* used if the document management system available */
  Phase phase;          /* used in a multi-phase project */
  ...
}

class Member {
   String name; double availability ;
   Employee person;     /* used if the employee register available */
   Team team;           /* used in a multi-team project */
   ...
}

class Team {
   List <Member> members; Member teamLead;
   ...
}

class Phase {
   Date startDate;
   ...
}
```

Figure 2.16: An object-oriented project model incorporating different kinds of projects

2.4.5 Dynamic Variations

The techniques of expressing dynamic variation by object composition, described in Sec. 2.2.1, can also be applied for implementing dynamic variations of groups of objects. Variations of a group of objects can be expressed by conditionals on the member variables of the object representing the group, or by delegating certain operations to its helpers. The limitations of such techniques are analogous to the ones discussed in Sec. 2.2.1: variations expressed by conditionals on variables are not modularized, and delegation to a helper object is limited to varying functionality of a fixed set of operations.

Moreover, a single helper object is not always sufficient to capture variation of a group of objects, because the varying functionality may be specific to the individual objects of the group. In such a case, the variation would need to be expressed by a set of helper objects, outsourcing the varying functionality of the invididual objects of the group. This creates additional typing problems.

For illustration, recall the example of variation of widget visualization style, introduced in Sec. 2.3.3. We considered the variation of visualization style from the perspective of individual widgets, but since the look-and-feel style is usually selected for an entire application, it is in fact a variation affecting a group of objects, i.e., all widgets of the application. The variation of the look-and-feel style of an application is modularized by a set of helper objects – one for each widget of the application. Indeed, in the Swing library, the look-and-feel style of an application is changed by traversing its widget structure and replacing the visualization helpers of all widgets so that they implement the currently selected style.

A specific problem of such a design is that it does not guarantee that that a variant is consistently bound in all objects of a group. This can lead to an unsafe interaction among the helper objects implementing the group variation and assuming that all objects of the group are bound to the same variant.

An example of interaction between the visualization helpers is propagation of visualization properties, such as fonts and colors, from composite widgets to their children. Specific look-and-feel styles may introduce specific visualization properties, which also need to be propagated. Fig. 2.17 shows a design of a look-and-feel style, which displays components with different textures. A widget can have its specific background texture or inherit it from its parent. Class TexturedComponentUI, which is the base class for the helper objects implementing visualization styles, maintains the current texture of the widget (line 11).

Access to the texture of the parent widget on line 18 is based on the assumption that the parent widget is also bound to the visualization style supporting textures. Since this assumption is not specified by type references, the visualization helper of the parent

```
abstract class JComponent {
    JComponent parent;
    ComponentUI ui;
    JComponent getParent() { return parent; }
    ComponentUI getUI() { return ui; }
    ...
}
...

abstract class TexturedComponentUI implements ComponentUI {
    Texture backgrTexture;
    JComponent comp;
    Texture backgrTexture() {
        if (backgrTexture != null) {
            return backgrTexture;
        }
        else {
            return ((TexturedComponentUI)comp.getParent().getUI()).backgrTexture();
        }
    }
    void setBackgrTexture(Texture texture) {
        this.backgrTexture = texture;
    }
    ...
}

class TexturedListUI extends TexturedComponentUI implements ListUI {
    void paint(Graphics g) {
        ... g.setPaint(backgrTexture()) ...
    }
    ...
}
```

Figure 2.17: Look-and-feel style with configurable fill textures

retrieved by method getUI needs to be casted to TexturedComponentUI. This type-cast is not safe, because the design does not guarantee that the parent and the child widgets are bound to the same visualization style.

2.5 Variation of Frameworks

Johnson and Foote [JF88] describe an object-oriented framework as a set of classes that embodies an abstract design for solutions to a family of related problems and supports reuse at a larger granularity than classes. The major characteristic of object-oriented frameworks that distinguishes them from simple class libraries is that the control flow is managed by the framework rather than by the application classes. Application-specific behavior is triggered at predetermined extension points in the framework. These extension points can also be seen as open variation points explicitly supported by the framework. They are open, because the set of available variants is not closed by the framework, but is left open for extension with new variants defined for concrete framework instances.

Technically, a framework describes a group of related objects, thus the discussion on implementation of variations of a group of objects is also valid for frameworks. However, there is a couple of specific issues related to variations of frameworks. One challenge is to reuse the code, instantiating a framework for a specific application, with different variations of the framework. As we will see, such reuse is difficult, because a framework is instantiated not just by instantiating its classes, but also by implementing its extension points defined as interfaces or abstract classes. These interfaces and abstract classes can be influenced by variations of the framework. Another challenge is related to the fact that frameworks are developed independently of specific applications. Thus application classes and their variations may need to be adapted to the abstractions of the framework.

For illustration we will use a graphical editing framework. Such a framework can be instantiated to implement a variety of concrete graphical editors, e.g., for UML diagrams, for electrical circuits, or for various other graphical structures. The example is based on analysis of the Eclipse GEF framework and its Logic example [GEF08]. Fig. 2.18 outlines the key classes of such a framework. Class GraphicalEditor is the central class of the framework, which manages other classes of the framework. Further, there is a set of abstract classes that define abstractions for the objects manipulated by a graphical editor: Graphic stands for all graphical objects, ConnectionGraphic for connection objects, NodeGraphic for graphical objects that can be connected, and so on.

```
class GraphicalEditor {
   Graphic graphicAtPoint(Point pt) { ... }
   ...
}

abstract class Graphic {
   GraphicalEditor editor;
   abstract void paint(Graphics g);
   ...
}

abstract class NodeGraphic extends Graphic {
   abstract int connectionCount();
   abstract ConnectionGraphic connectionAt(int i1);
   ...
}

abstract class ConnectionGraphic extends Graphic {
    abstract NodeGraphic sourceNode();
    abstract NodeGraphic targetNode();
    ...
}
```

Figure 2.18: Graphical editor and graphical objects

2.5.1 Instantiating a Framework

A framework is instantiated for specific applications by providing concrete subclasses of its abstractions and implementing the required methods. For example, to instantiate the graphical editing framework, we must define concrete graphical objects by defining concrete subclasses of Graphic, NodeGraphic, and ConnectionGraphic and implementing their abstract methods. For concrete instances of Graphic, the method paint must be implemented to specify the appearance of the graphical object; for instances of NodeGraphic, the methods connectionAt and connectionCount must be implemented to specify the connections of the node; and for instances of ConnectionGraphic, their source and target nodes must be specified by implementing sourceNode and targetNode.

For an illustration of a concrete instance of the framework, consider an implementation of a graphical editor for logic circuits, the model of which is given in Fig. 2.19. A Circuit consists of a set of LogicElements: Various LogicComponents and Wires connecting these logic components. Variations of logic components are implemented by subclasses of LogicComponent: a set of classes, such as OrGate and AndGate, implementing gate components for respective logical operations, and LED for indicating the values of inputs.

In the graphical editor for logical circuits, the objects of the circuit model must be represented by the framework abstractions: logic elements must be represented as nodes,

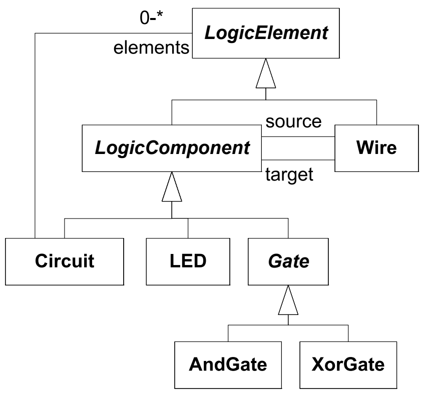

Figure 2.19: Model of logic circuits.

and wires as connections. As shown in Fig. 2.20, this is achieved by subclassing appropriate framework classes for the classes of the model: WireGraphic (line 1) instantiates ConnectionGraphic for wires, while LogicCompGraphic and its subclasses (lines 15, 27, and 41) instantiate NodeGraphic for logic components.

These subclasses serve as *adapters*, adapting existing application objects to the interfaces expected by the framework. For example, class WireGraphic takes a reference to a wire in its constructor, stores it in a field and uses it for implementation of the abstract methods of NodeGraphic, e.g., in the implementation of the method sourceNode (line 9), the source component of the wire, retrieved by method sourceComp, is used to determine the source node of the connection representing the wire.

From each adapter we can navigate to the corresponding application object by accessing the reference to the adaptee, but in a lot of cases we also need navigation in the opposite direction, e.g., we need to know the node object representing a given logic component (line 9) and the connection object representing a wire (line 22). Such navigation could be achieved by introducing respective references in the classes of the circuit model, but it is highly undesirable, because the circuit model can be used independently of the editor, e.g., for some analysis or simulation of the circuits, and thus should not depend on the editing functionality.

In order not to break modularity of existing application classes, it is common to maintain an external mapping from application objects to corresponding framework objects. For example, we can define necessary data structures and methods in the class CircuitEditor (Fig. 2.21). On line 2 we define a mapping from logic components to corresponding graphical objects. Further, we implement method logicCompGraphicFor (line 5), which returns the corresponding adapter for a given logic component: if such adapter already

```
class WireGraphic extends ConnectionGraphic {
  Wire wire;
  WireGraphic(Wire wire, CircuitEditor editor) {
      super(editor);
      this.wire = wire;
  }
  void paint(Graphics g) { ... }
  NodeGraphic sourceNode() {
      return editor.logicCompGraphicFor(wire.sourceComp());
  }
  NodeGraphic targetNode() { ... }
  ...
}

abstract class LogicCompGraphic extends NodeGraphic {
   LogicComponent comp;
   public LogicGraphic(LogicComponent comp, CircuitEditor editor) {
      super(editor);
      this.comp = comp;
   }
   ConnectionGraphic getConnectionAt(int i1) {
      return editor.wireGraphicFor(comp.connectedWires().get(i1));
   }
   ...
}

class LEDGraphic extends LogicCompGraphic {
  public LEDGraphic(LED led, CircuitEditor editor) {
      super(led, editor);
  }
  LED led() {
    return (LED)comp;
  }
  void paint(Graphics g) { ...
      g.drawText(led().getValue()));
      ...
  }
  ...
}

class AndGateGraphic extends LogicCompGraphic {
  public AndGateGraphic(AndGate gate, CircuitEditor editor) {
      super(gate, editor);
  }
  void paint(Graphics g) { ... }
  ...
}

...
```

Figure 2.20: Implementation of graphical objects for circuit editor

```
class CircuitEditor extends GraphicalEditor {
    Map<LogicComponent, LogicCompGraphic> logicAdapters;
    Map<Wire, WireGraphic> wireAdapters;

    LogicCompGraphic logicCompGraphicFor(LogicComponent comp) {
        LogicCompGraphic graphic = logicAdapters.get(comp);
        if (graphic == null) {
            graphic = createLogicGraphic(comp);
            logicAdapters.put(graphic);
        }
        return graphic;
    }
    LogicCompGraphic createLogicGraphic(LogicComponent comp) {
        if (comp instanceof LED) {
            return new LEDGraphic((LED)comp, this);
        }
        else if (comp instanceof AndGate) {
            return new AndGateGraphic((AndGate)comp, this);
        }
        ...
        else {
            throw new InconsistencyException("Unknown type of logic component");
        }
    }
    WireGraphic wireGraphicFor (Wire wire) { ... }
    ...
}
```

Figure 2.21: Implementation of a circuit editor

exists it is taken from the mapping, otherwise it is created and registered in the mapping. Analogously, we implement a mapping from wires to corresponding graphical connections.

2.5.2 Dependency on Application Variations

Variations of application objects propagate into the implementation of adapters. Variations of adapter implementation can be modularized by constructing an inheritance hierarchy of adapter classes, which is parallel to the hierarchy of the adaptees. For example, the implementation of NodeGraphic for logic components is distributed over the inheritance hierarchy of LogicCompGraphic: the functionality common to all types of components, e.g., retrieving connections (line 21 of Fig. 2.20), is implemented directly in LogicCompGraphic, while the functionality specific to component types, e.g., rendering the object (lines 34 and 45), are implemented in respective subclasses.

The hierarchy of adapters, i.e., the framework objects, and the hierarchy of adaptees, i.e., the application objects, need to be related in two ways. First, we need to express that a specific adapter class can be used only with the corresponding type of adaptee. Second, given a particular adaptee object, we must be able to instantiate a suitable adapter for it.

The dependency of the adapter class on the specific type of an adaptee can be expressed by requiring that type of adaptee in the constructor of the adapter. For example, the class LogicCompGraphic is designed to be reusable with any type of logic component, thus it takes an object of type LogicComponent in its constructor (line 17), but class LEDGraphic is dedicated specifically to LED components, thus it accepts only an object of type LED in the constructor (line 28).

More problematic is, however, to specialize the type of the reference to the adaptee. Since LogicCompGraphic works with any type of logic component, it declares its reference to the adaptee, i.e., the field comp, with type LogicComponent (line 16). Adapters for specific logic component types, like LEDGraphic, need to access specific attributes and operations of the respective adaptee types, e.g., LEDGraphic needs to access the current value of the LED component (line 35). For this purpose, it uses a helper method led, which casts comp to class LED. Ideally, we would like to specialize the type of comp to LED, but this is not allowed in Java, because it is unsafe unless we prevent initialization of the field in the superclasses of LED or ensure that all places of initialization are consistently overridden.

In Java, there is also no good solution for instantiation of an adapter object that matches the type of the given adaptee. The straightforward solution is to use conditional statements checking the type of the given adaptee object to decide which adapter class has to be instantiated. Such approach is used in method createLogicGraphic (line 13 of Fig. 2.21), which is responsible for instantiation of a graphical object for the given logic component: the method checks for all possible types of logic components and instantiates the corresponding adapter class.

One problem with such design is that the mapping between adaptees and adapters must be defined manually, and there is no guarantee that the mapping is defined completely. For example, there is no guarantee that the conditional statements inside the method createLogicGraphic check all possible concrete types of logic components. Thus, we must consider the case when none of the checked types match and that case return a null value or throw an exception (line 22).

Another problem is that CircuitEditor is exposed to all variations of logic components. As a result, it must be changed whenever new types of logic components are added or existing types are removed, which is a limitation for extensibility. Moreover, since there is no check for consistency of implementation of the mapping, the developer would be not informed when he or she forgets to update the framework instances after adding new variants in the application model.

2.5.3 Combining Framework Instances and Variations

Framework instances can be seen as variations of the framework functionality, but frameworks may also have variations on their own. Certain functionality of the framework may be optional or alternative. For example, in a graphical editing framework the editing functionality may be optional, allowing to produce a read-only viewer if necessary. Variations of framework may capture various platform dependencies, such as customizations for specific physical environments, operation system services, integrations to other applications, and so on.

It may be necessary to reuse framework instances with different variations of the framework, because these variations may also be necessary in the applications using the framework. For example, an application may need both editors and viewers of a particular graphical model. Such application would make use of the optionality of the editing features in the graphical editing framework. Similarly, an application may reuse variations of platform dependencies in order to support more platforms or be easier to adapt to them.

In Sec. 2.2, we explained how variations of a group of objects can be modularized by subclassing the classes of its members. The same technique can also be applied for modularization of variations of a framework: we can modularize the varying code in subclasses of the framework classes. In order to reuse an instance of a framework with a

```
 1 class GraphicalEditorWithMenus extends GraphicalEditor {
 2   Menu getContextMenu(Point pt) {
 3     Graphic graphic = graphicAtPoint(pt);
 4     return (graphic == null) ? getEditorMenu() :
 5       (GraphicWithMenus)graphic.getContextMenu(pt);
 6   } ...
 7 }
 8
 9 abstract class GraphicWithMenus extends Graphic {
10   abstract Menu getContextMenu(Point pt);
11 }
```

Figure 2.22: Extension of the graphical editor with context menus

framework variation defined in such a way, we must combine the subclasses defined for the framework instance with the subclasses implementing the variation.

For an illustration, consider an optional feature of the graphical editor for support of context menus. The new feature can be implemented in a subclass of GraphicalEditor, e.g., GraphicalEditorWithMenus in Fig. 2.22. We may also need to extend other classes of the framework; e.g., we need to extend Graphic with an abstract method getContextMenu (line 10), so that each graphic object is asked to construct a menu to be shown when the user clicks on that object (line 5).

In the extension with context menus, we can observe the typing problems discussed in Sec. 2.4.1, e.g., on line 5 we need to cast the selected graphical object to GraphicWithMenus for accessing its extended interface. A peculiarity of framework extensions, however, is that they can extend not only the implementation of the framework, but also the requirements to the instances of the framework. For example, in the extension with context menus we extend the requirements to the instances of Graphic, because they must additionally implement the method getContextMenu specifying their context menu.

Fig. 2.23 demonstrates how the framework instance for logic circuits can be extended with support for context menus. For each class instantiating Graphic for a certain type of circuit objects (i.e., each class from Fig. 2.20), we must define a class instantiating GraphicWithMenus for the same type of objects. For example, WireGraphicWithMenus (line 1) instantiates GraphicWithMenus for wires. It inherits from GraphicWithMenus and defines the context menu for wires by implementing getContextMenu. It also inherits from WireGraphic to reuse the instantiation of the base framework functionality for wires. Analogously, we define subclasses of GraphicWithMenus for logic components (lines 5, 9 and 18).

Then we must also define a new class CircuitEditorWithMenus (line 22), which represents the circuit editors with context menus. This class inherits from CircuitEditor, the base circuit editor, and GraphicalEditorWithMenus to combine their functionality. In CircuitEditorWithMenus, we must reimplement the code instantiating the elements of the

```
class WireGraphicWithMenus extends WireGraphic & GraphicWithMenus {
  Menu getContextMenu(Point pt) { ... }
}

abstract class LogicCompGraphicWithMenus extends NodeGraphic & GraphicWithMenus {
    Menu getContextMenu(Point pt) { ... }
}

class LEDGraphicWithMenus extends LEDGraphic & LogicCompGraphicWithMenus {
  Menu getContextMenu(Point pt) {
        Menu menu = super.getContextMenu(pt)
        menu.addAction("Change Value", changeValueAction());
        return menu;
  }
  ...
}

class AndGateGraphicWithMenus extends AndGateGraphic & LogicCompGraphicWithMenus { }

...

class CircuitEditorWithMenus extends CircuitEditor & GraphicalEditorWithMenus {
    LogicCompGraphic createLogicGraphic(LogicComponent comp) {
        if (comp instanceof LED) {
            return new LEDGraphicWithMenus((LED)comp, this);
        }
        else if (logic instanceof AndGate) {
            return new AndGateGraphicWithMenus((AndGate)comp, this);
        }
        ...
        else {
            throw new InconsistencyException("Unknown type of logic component");
        }
    }
    ...
}
```

Figure 2.23: Extension of the circuit editor

framework instance. For example, we redefine the method createLogicGraphic (line 23), responsible for instantiation of an adapter matching the type of the given logic component, so that it instantiates the classes with support for context menus.

The main problem with the design above is that it cannot guarantee that the additional requirements of a framework extension are indeed implemented in the instances of the framework that are used in combination with this extension. For example, the type system would not complain if we forget to instantiate GraphicWithMenus for any of the logic circuit objects, i.e., forget to implement one of the classes from Fig. 2.23. It also does not check whether we consistently update all methods instantiating the adapter classes, e.g., it would not complain if we remove the method createLogicGraphic on line 23 or any of the cases within the method.

In fact, the type system does not prevent us from using the subclasses of Graphic defined for basic circuit editors (Fig. 2.20) in combination with a graphical editor with context menus. However, such use is not safe, because framework extensions assume the extended requirements to the framework instances. For example, the type cast at the line 5 of Fig. 2.22 would fail for the classes from Fig. 2.20, because they do not inherit from GraphicWithMenus.

Another problem is the redundancy in the instantiation code. In the presented design, for every variation of a framework abstraction, e.g., support for menus in graphical objects, we must redefine all classes instantiating that abstraction, even if they do not need to introduce any specific code. For example, the graphical object for AndGate does not need any specific context menu, thus the class AndGateGraphicWithMenus just combines AndGateGraphic with LogicCompGraphicWithMenus without introducing any new functionality. Redefinition of the methods like createLogicGraphic instantiating adapters to the framework can also be seen as a redundancy, because they are completely analogous to the overridden methods, just instantiate the corresponding redefined adapter classes.

In a lot of cases, framework variations can also define default behavior for the affected framework abstractions. For example, GraphicWithMenus could provide a default implementation of getContextMenu, and some of the framework instances may *not* need to specialize this default behavior. However, we would still have to redefine the classes of the framework instances so that they inherit from the extended framework class, e.g., we would still have to redefine all application-specific instances of Graphic to inherit from GraphicWithMenus, which means that we would have a set of classes that do not introduce any new functionality and serve only as a glue code.

2.5.4 Dependency on Multiple Variation Points of a Framework

So far we considered instantiation of individual extension points of a framework. For example, the class Graphic represents the extension point of the framework that enables extensions with new types of graphical objects. In general, a framework can define multiple interfaces or abstract classes representing different extension points. For example, a typical graphical editing framework would provide further variation points, such as available graphical operations or available algorithms for automatic layout. Such extension points are, in principle, open variation points supported by the framework.

Interfaces and abstract classes are suitable for abstracting from functionality depending on single variation points. By declaring abstract method paint within class Graphic, we specify that painting functionality depends on specific types of graphical objects, i.e., on specific instances of the variation point represented by the class Graphic. The type-checker of Java ensures that the method is implemented for all concrete subclasses of Graphic, i.e., for all variants of the variation point.

There can also be functionality depending on combinations of variants of multiple variation points of a framework. For example, a graphical editing framework may support variations of rendering style of graphical objects. In such a design, implementation of rendering depends on two variation points: the type of graphical objects and the rendering style. A framework instance needs to define both concrete types of graphical objects and concrete rendering styles, and then provide implementations of the paint method for all pairs of the variants.

The requirement to provide implementation of a method for the pairs of concrete instances of two variation points of a framework cannot be properly described by simple Java interfaces and abstract classes. In the representation of variation points by interfaces, all abstract methods must be declared within a particular interface. For example, in our graphical framework variation of graphical object is represented by class Graphic, and variation of rendering styles is represented by the class Renderer. We can declare the method paint either within the Graphic or within the Renderer, and pass the other object as a simple parameter to the method. Figure 2.24 illustrates the former case: class Graphic declares method paint and takes a renderer as parameter (See line 2). The graphical editor maintains a reference to the currently selected rendering style (line 12), which is passed when calling the paint method of a graphical object (line 16).

Now let's take a look at the implementation of the rendering functionality for an instance of the framework for logic circuits, shown in Fig. 2.25. In addition to the variations of graphical objects presented in Sec. 2.5.1, we now also have variations of rendering styles. We define two renderers for logic circuits, SchematicRenderer (line 1) and PhysicalRenderer (line 3), enabling switching between the schematic and the physical representation of a

```
abstract class Graphic {
   abstract void paint(Renderer r, Graphics g);
   ...
}

abstract class NodeGraphic extends Graphic { ... }
abstract class ConnectionGraphic extends Graphic { ... }

abstract class Renderer { }

abstract class GraphicalEditor {
    Renderer currentRenderer;

    void paintAll(Graphics g) {
       for (Graphic obj : graphicalObjects()) {
          obj.paint(currentRenderer, g);
       }
    }
    ...
}
```

Figure 2.24: Supporting variations of rendering style in a graphical editing framework

circuit. In the implementations of paint method for the graphical objects of the circuit editor, we must check the type of the given renderer and, depending on the type, paint the objects either in the schematic or in the physical style (lines 6-16 and 24-34).

We can identify two problems in the presented design. First, it does not modularize the implementation of painting with respect to rendering styles, and thus introduces coupling between them and makes extension with new rendering styles more difficult. Second, the design does not enforce that the paint method is implemented for all pairs of graphical objects and rendering styles of a particular framework instance. On one side, the signature of the paint method does not prevent calling it with a graphical object and a rendering style from other framework instances, e.g., we can call paint on an object of WireGraphic with a rendering style from a UML diagram editor, implemented as another instance of the graphical editing framework. Such call would produce a runtime exception in our implementation (lines 14 and 32). On the other side, it is not checked if paint is implemented for all rendering styles supported by the framework instance.

If we choose to declare the paint method within Renderer, we would experience analogous problems with respect to variations of graphical objects.

The design can be improved by implementing paint as a multimethod, dispatched both by the graphical object and the type of the renderer. Such design is outlined in Fig. 2.26. The abstract paint method declared by the framework is intended to be implemented in framework instances for their pairs of graphical objects and rendering styles.

```
class SchematicRenderer extends Renderer { }

class PhysicalRenderer extends Renderer { }

class WireGraphic extends ConnectionGraphic {
    void paint(Renderer r, Graphics g) {
        if (r instanceof SchematicRenderer) {
            ... /* paint schematic wire */
        }
        else if (r instanceof PhysicalRenderer) {
            ... /* paint physical wire */
        }
        else {
            throw InconsistencyError("Unknown rendering style");
        }
    }
}

abstract class LogicCompGraphic extends NodeGraphic {
    ...
}

class LEDGraphic extends LogicCompGraphic {
    void paint(Graphics g) {
        if (r instanceof SchematicRenderer) {
            ... /* paint schematic wire */
        }
        else if (r instanceof PhysicalRenderer) {
            ... /* paint physical wire */
        }
        else {
            throw InconsistencyError("Unknown rendering style");
        }
    }
    ...
}

...
```

Figure 2.25: Circuit editor with variations of rendering style

```
 1 /* Framework */
 2 void paint(Graphic obj, Renderer r, Graphics g) {
 3    throw InconsistencyError("An incompatible pair of a graphical object and a rendering style");
 4 }
 5
 6 class GraphicalEditor {
 7    Renderer currentRenderer;
 8
 9    void paintAll(Graphics g) {
10       for (Graphic obj : graphicalObjects()) {
11          paint(obj, currentRenderer, g);
12       }
13    }
14 }
15
16 /* Framework instance for circuit editor */
17 void paint(WireGraphic obj, SchematicRenderer r, Graphics g) { ... }
18 void paint(WireGraphic obj, PhysicalRenderer r, Graphics g) { ... }
19 void paint(LEDGraphic obj, SchematicRenderer r, Graphics g) { ... }
20 void paint(LEDGraphic obj, PhysicalRenderer r, Graphics g) { ... }
21 ...
```

Figure 2.26: Using multimethods for expressing dependency on multiple variation points of a framework

Although the design solves the problem of modularizing the implementation of methods with respect to the variants of multiple variation points, it still fails to enforce that the methods are completely implemented for the variants of a particular framework instance. According to the signature of the paint method of line 2, the method can be called with an arbitrary pair of a graphical object and a rendering style, e.g., it can still be called with an instance of WireGraphic and a rendering style from a UML diagram editor. Since, we can provide meaningful implementations of the method only for the pairs of variants of the same framework instance, such as graphical circuit objects and circuit rendering styles, in unsupported cases we must still throw a runtime exception. And again, the design does not guarantee that the method is implemented for the pairs of variants of the same framework instance.

2.6 Summary

In this chapter we identified two basic object-oriented techniques for modularizing variation. The first technique uses subtype polymorphism to outsource the varying functionality to helper objects and use them in an abstract way. The second technique uses inheritance to move varying parts of a class to its subclasses. These techniques have specific advantages and limitations:

- Helper objects support dynamic variation binding, but do not permit varying the interface of the objects. The variation must be anticipated and leaves a “footprint” in the core functionality.
- Inheritance can describe variation of the interface of objects and support unanticipated variability, but does not support dynamic variation of object’s functionality. Variation can be bound dynamically during instantiation of an object, using conditional statements to select the class to be instantiated. Such code is, however, error prone and difficult to extend.

We also investigated specific problems of combining the basic techniques for the purpose of modularizing multiple variations of an object and dealing with their interactions:

- An object can have multiple helper objects modularizing its different variations, but such solution is suitable only for completely independent variations.
- Variations modularized by inheritance can be combined using multiple inheritance. The classes defining various combinations of variations can also encapsulate their interactions. In addition to the typical limitations of inheritance, combination of variations by inheritance does not scale with respect to the number of the supported configuration options.
- Variation by helper objects can also be combined with other variations of an object modeled by inheritance. In such a combination, a helper object can be affected by the variations of its master object, which creates a covariant interdependency between the two. Such covariance cannot be expressed in a type-safe way using simple types. It also makes instantiation of helper objects more challenging.

Multidimensional variation at the scope of a function can be expressed by multimethods, which can modularize dependencies of the implementation of the function on combinations of multiple variation points. Multi-dispatched factory methods can implement the mapping from selected configuration to the class implementing that configuration. Such implementation makes the mapping more extensible and less error prone, but still does not solve the problem of its combinatorial explosion with respect to the supported configuration options.

Variations affecting a group objects can be expressed by applying the basic variation techniques for each of object of the group individually. We have seen that such a solution creates two kinds of problems:

- First, it creates more code overhead, because large-scale variations and their compositions must be assembled manually out of primitive variations mechanisms. The code implementing compositions of variations tends to be error-prone and not stable with respect to evolution of the individual variations.

- Second, the designs expressing group variations by variations of individual objects do not ensure that all objects of the group are consistently bound to the same variants. The failure to ensure such consistency makes the interaction between the objects not type-safe, because implementation of the variation for one of the objects may need to access the variation-specific functionality of other objects from the group.

Finally, we have identified specific challenges of variation management in object-oriented frameworks and their instances:

- Variations of a framework can also cause variations of their abstractions, and thus variations of the requirements of the framework to its instances. We were not able to extend the requirements of a framework to instances in a type safe-way using the conventional techniques.
- Frameworks are developed independently of the applications where they are used, thus their instances may need to be adapted to application-specific variations. We have seen that such adaptation code is not extensible with respect to new application-specific variants and contains unsafe covariant dependencies of adapters on their adaptees. Furthermore, it is difficult to reuse the adaptation code with specific variations of the framework.
- Multimethods can be used to modularize functionality of framework instances depending on several extension points of a framework. We could not ensure, however, that such methods are correctly used only with combinations of variants of the same framework instance, and are completely implemented for such combinations.

3 Virtual Classes

3.1 Introduction

A significant body of research has raised the concern that classes are a too small unit of modularity [Szy98, ML98, SB98b, Ern03, Ost02]. In a lot of cases a cohesive piece of functionality involves a *group* of related classes. Mainstream languages provide class grouping mechanisms, e.g., namespaces in C++ [ES95], or packages and inner classes in Java [AG96]. However, inheritance and polymorphism, which are the primary means of object-oriented programming for expressing and modularizing variability, are not supported at the level of such class groups. As a result, variations involving multiple classes cannot be directly expressed and must be encoded by variations of individual classes.

The problems of encoding variations of groups of classes by basic object-oriented mechanisms have been analyzed in depth in Sec. 2.4 and 2.5. We have seen that when related classes are extended individually, the relationships between them are not updated. Thus additional glue code is required to update the inheritance and instantiation relationships between the classes. Since existing type references cannot be updated, unsafe type-casts are required for expressing interaction among the extended classes. We have also seen that even more glue code is required to combine independent variations involving multiple classes, because variations of each class of the group must be composed separately and then again manually linked with the other classes.

These problems can be addressed by supporting object-oriented mechanisms at a larger scope, i.e., by providing inheritance and subtype polymorphism at the scope of groups of interrelated classes. Different terminology has been used in the literature to denote such groups of interrelated classes, such as collaborations [VN96b, Ost02, ML98, LLO03, Her03], layers [SB98b, Ost02], teams [Her03], and families [Ern01]. In this thesis, the notion of a group of interrelated classes corresponds to that of a *class family* [Ern01]. Hence, this term will be used.

Virtual classes, a concept that stems from the programming language Beta [MMP89, MMPN93], provide a conceptually simple, but at the same time powerful solution for large-scale inheritance and polymorphism. The idea is to introduce classes as a new kind of object members and treat them in analogous way as (*virtual*) methods, i.e., to allow overriding them in subclasses and make them late-bound. While conceptually simple, late-binding of classes introduces non-trivial implications both to the operational semantics and the type-system. The semantics of virtual classes have been elaborated and formalized in a more recent work [Ern99b, EOC06].

This chapter presents the implementation of virtual classes in CaesarJ, which is the first full-fledged implementation of virtual classes and related language features, such as propagating mixin composition and path-dependent types, as an extension of Java. The implementation adapts the ideas of BETA and gbeta to a Java-based language, e.g., define the semantics for abstract class and method declarations, constructors, and super-calls.

CaesarJ also contains innovations to the core semantics of virtual classes. The inheritance semantics of virtual classes and propagating mixin composition is based on a new linearization algorithm, which better corresponds to the intuitive semantics of these language features. Implementation of the type relationships in the language are based on the vc^n calculus (cf. Sec. 5.2), which is simpler than the previously used *vc* calculus [EOC06], and at the same time avoids its limitations on the structure of types and expressions.

This chapter is organized as follows. In Sec. 3.2, we motivate virtual classes and explain their concepts. We describe a precise intuitive semantics for virtual classes and propagating mixin composition. In Sec. 3.3, we discuss specific semantic aspects of virtual classes. We define an inheritance graph sorting algorithm that corresponds to the described intuitive semantics, give an overview of the type system implemented in CaesarJ, and define the semantics of abstract virtual classes. In Sec. 3.4 we evaluate virtual classes with respect to the variation scenarios introduced in Chapter 2.

3.2 Virtual Classes in a Nutshell

Virtual classes are inner classes that can be refined in the subclasses of the enclosing class. We will consider virtual classes as members of the objects of the enclosing class.[1] We will call such objects *family objects*, because they represent families of classes available at their members. Analogously we will refer to the enclosing classes as *family classes*. In an inheritance relationship between two family classes we will refer to the subclass as the *heir family* and the superclass as the *parent family*.[2]

This section introduces the idea of virtual classes and the rationale behind their semantics.

[1]As discussed in Sec. 6.1, there are also approaches considering virtual classes as members of classes rather than members of objects.

[2]This is analogous to the inheritance terminology used in Eiffel [Mey97].

3.2.1 Large-Scale Inheritance

In object-oriented languages, the body of a subclass can be seen as a description of a difference to its superclass. The functionality of the subclass is intended to be equivalent to that of the class produced by changing the superclass according to that description: the new members are added and existing members are replaced. In this way extending a class by inheritance is almost[3] as easy and flexible as changing the class directly.

In Java, a subclass can replace the methods of its superclass, but not its inner classes. Declaration of an inner class in a subclass does not have the same effect as changing the corresponding inner class of the superclass. Instead, it is considered as a totally new unrelated class only accidentally having the same name.

The semantics of virtual classes in CaesarJ can be seen as generalization of the principle of treating a subclass as a description of the difference from its superclass for classes with inner classes. However, differently from methods, redeclaration of a virtual class in an heir family is not considered as a replacement for the virtual class with the same name of the parent family, but is again considered as a description of the difference from it. Thus, a redeclared virtual class can be seen as an extension or a refinement of the corresponding virtual class in the parent family. We call the former as a *furtherbinding* and the latter as its *furtherbound*.

For an illustration, recall the example of variation of menu functionality introduced in Sec. 2.4.1: A menu is a composite structure consisting of objects for various kinds of menu items and cascade menus. Each menu structure can be considered as a family of its constituent objects. In order to extend the functionality of these objects together, we group their classes by declaring them as virtual classes of one family class, as shown in Fig. 3.1.

We can see that the classes remain in principle unchanged, compared to the Java classes of Fig. 2.9, except that they are declared with the keyword cclass, which denotes that they are treated by the CaesarJ semantics. In contrast, the classes declared with keyword class are considered as standard Java classes, for which the Java semantics is completely preserved, e.g., inner classes with class keyword are not considered as virtual. In this way full compatibility with Java code is preserved. The inheritance hierarchies of these two kinds of classes are strictly separated: a class declared with the keyword cclass cannot inherit from a class declared with the keyword class, and the other way around.

Fig. 3.2 shows the extension of menus with support for accelerator keys, introduced in Sec. 2.4.1: each menu item can be associated with an accelerator key, which is used as a shortcut to execute the action of the menu item. We define this extension in a new family class MenusAccel, declared as a subclass of BasicMenus.

[3]The granularity of change is limited to replacing members of the class, in particular its methods.

```
cclass BasicMenus {
    cclass MenuItem {
        String label; Action action;
        MenuItem(String label, Action action) {
            this.label = label; this.action = action;
        }
        String displayText() {
            return label;
        }
        void draw(Graphics g) {
            ... displayText() ...
        }
        ...
    }

    cclass CascadeMenuItem extends MenuItem {
        PopupMenu menu;
        void addItem(MenuItem item) {
            menu.addItem(item);
        }
        ...
    }

    cclass CheckMenuItem extends MenuItem { ... }

    cclass RadioMenuItem extends MenuItem { ... }

    abstract cclass Menu {
        List<MenuItem> items;
        MenuItem itemAt(int i) {
            return items.get(i);
        }
        int itemCount() {
            return items.size();
        }
        void addItem(MenuItem item) {
            items.add(item);
        }
        void addAction(String label, Action action) {
            items.add(new MenuItem(label, action));
        }
        ...
    }

    cclass PopupMenu extends Menu { ... }

    cclass MenuBar extends Menu { ... }
}
```

Figure 3.1: Base implementation of menus and menu items with virtual classes

```
cclass MenusAccel extends BasicMenus {
    cclass MenuItem {
        KeyStroke accelKey;
        boolean processKey(KeyStroke ks) {
            if (accelKey != null && accelKey.equals(ks)) {
                performAction();
                return true;
            }
            return false;
        }
        void setAccelerator(KeyStroke ks) {
            accelKey = ks;
        }
        void draw(Graphics g) {
            super.draw(g);
            displayAccelKey();
        }
        ...
    }

    cclass CascadeMenuItem {
        boolean processKey(KeyStroke ks) {
            if (menu.processKey(ks)) {
                return true;
            }
            return super.processKey(ks);
        }
    }

    abstract cclass Menu {
        boolean processKey(KeyStroke ks) {
            for (int i = 0; i1 < itemCount(); i1++) {
                if (itemAt(i).processKey(i1)) {
                    return true;
                }
            }
            return false;
        }
        ...
    }
}
```

Figure 3.2: Extension of menu functionality with accelerator keys

The contents of MenusAccel are analogous to the functionality of accelerator keys presented in Fig. 2.10. MenuItem is extended with fields and methods to maintain the associated accelerator key. Method processKey is introduced in menu items and menus to process the given input key by recursively traversing the menu structure and triggering the action of the menu item with the accelerator matching the key.

As can be seen, the class MenusAccel defines only the functionality related to support for menu accelerators. The overhead of defining an extension compared to a corresponding direct change is very minimal: the only additional code in MenusAccel is the headers of the family class and the refined virtual classes.

According to our intuitive definition of the semantics of virtual classes, the contents of MenusAccel can be seen as a description of a difference from BasicMenus, where the virtual classes of MenusAccel describe differences from the corresponding virtual classes of BasicMenus. The functionality of MenusAccel is equivalent to a class produced by taking BasicMenus and extending its classes MenuItem, CascadeMenuItem and Menu with the members of these classes from MenuAccel, whereas already existing class members are replaced.

Since the functionality of an heir family is equivalent to that of the class obtained by a corresponding invasive change of the parent family, extending a group of classes becomes almost as easy and flexible as changing the classes directly. In this way virtual classes support *Open-Closed Principle* [Mey97, Mar03] at the scale of a group of classes. According the principle, modules should open for extension, but closed for change. In this way they can be reused in other contexts with different requirements without destabilizing the existing clients. From the perspective of variability management, this means that we can close the module implementing the functionality that is common to different clients and implementing client-specific variations as extensions to that module.

3.2.2 Large-Scale Mixin Composition

The idea of a mixin is to make the difference described by a subclass reusable by enabling its composition with different parent classes [BC90, FKF98]. A composition of a mixin with a class produces a subclass of the class, which can be further composed with other mixins. Mixins provide a form of multiple inheritance, because a class can be defined as a composition of multiple mixins.

The idea of mixins can be generalized for groups of classes by treating the difference described by an heir family class composable with different parent families. In this way the extensions of groups of classes can be made composable. For example, the extension of menus with support for accelerators, presented in Fig. 3.2, can be composed with

```
cclass MenusML extends BasicMenus {
   cclass MenuItem {
      String displayText() {
         return translateText(label);
      }
      ...
   }
}
```

Figure 3.3: Menus with multi-language support

```
cclass MenusAccelML extends MenusAccel & MenusML { }
```

Figure 3.4: Menus with support for accelerators and multi-language

other extensions of menu functionality, such as extensions of menus with multi-language support, introduced in Sec. 2.4.2.

Fig. 3.3 shows an implementation of the latter extension with virtual classes analogous to the Java implementation presented in Fig. 2.13. As explained in Sec. 3.2.1, we can consider the extension MenusML as a description of a difference from its superclass, e.g., it specifies that in the class MenuItem of MenusML implementation of the method displayText must be replaced with the new one, which translates the label of the menu.

By treating the differences defined by heir family classes as mixins we can apply them on classes other than their explicitly declared parents. In Fig. 3.4, the extensions described by MenusAccel and MenusML are composed by declaring the class MenusAccelML, which inherits both from MenusAccel and MenusML. In this case MenusAccel is considered as a mixin applied on the class MenusML. The semantics of such composition can be intuitively interpreted as a result of consecutively changing BasicMenus, first by the difference described by MenusML and then by the difference of MenusAccel.

3.2.3 Family Polymorphism

The difference described by a subclass is limited to adding new members and changing the implementation of the methods of its superclass. It is, however, not possible to remove existing members or change their type in an arbitrary way. The reason of this limitation is that inheritance serves not only for the purpose of implementation reuse, but also establishes a *subtyping* relation between the subclass and the superclass.

In general, a type S is a subtype of type T if instances of S can be used wherever instances of type T are expected. Since a piece of code using a variable of type T can also work

```
interface MenuContributor {
   void contribute(Menu menu);
}

class FileMenuContrib implements MenuContributor {
   void contribute(final BasicMenus menus, menus.Menu menu) {
      menus.CascadeMenuItem openWith = menus.new CascadeMenuItem("Open With");
      menu.addItem(openWith);
      menus.MenuItem openWithTE =
         new MenuItem("Text Editor", createOpenWithTEAction());
      openWith.addItem(openWithTextEditor);
      ...
      menus.MenuItem readOnly =
         menus.new CheckMenuItem("Read Only", createReadOnlyAction());
      menu.addItem(readOnly);
      ...
   }
   ...
}
```

Figure 3.5: Menu contribution with operations on files

if the variable refers to an instance of S, this piece of code can be seen as polymorphic with respect to subtypes of T. Therefore, we talk about *subtype polymorphism* in this case.

The main static guarantee in so-called *statically typed* object-oriented languages is that all method calls must be successful, i.e., in each case the method must be found, it must accept the given parameters and return a value of expected type. Thus, a subtype is expected to preserve the interface of its supertype in terms of the available class members and their signatures. Because of that, for a subclass to be a subtype of its superclass it can only add new class members and replace implementations of the existing ones.

With virtual classes the notion of subtyping and polymorphism is extended to families of classes, which is known as *family polymorphism* [Ern01]. The type of a family object determines not only its fields and methods and, but also its virtual classes. Thus, a piece of code using a family object polymorphically, can access and use its virtual classes them without knowing their concrete definitions.

In general, there are two fundamentally different ways of using a class in object-oriented languages: a class serves as a template for instantiating objects and as a type. Family polymorphism affects both of these two forms of using virtual classes. For an illustration, we will use the example of Fig. 2.11 from Sec. 2.4, which demonstrates a client code using menus. Fig. 3.5 shows the design of the example with virtual classes.

First, a virtual class can be accessed as an attribute of a family object for instantiation. This means that the class to be instantiated is not statically known, but determined by a dynamic lookup for that class in the family object. In analogy to late-bound method calls, we can speak about *late-bound instantiation* of classes, because the class to be instantiated is determined only at runtime. In Fig. 3.5, we can see instantiation instructions on lines 7, 10 and 14. In all cases we can see that instantiation is qualified by a reference to the family object menus, which is given a parameter to the method (line 6).

We can see that the instantiation instructions have the same syntax as that of instantiation of inner classes in Java, but they have a different semantics, because the classes to be instantiated are determined by the value menus, more precisely by its dynamic type. For example, if it is a direct instance of BasicMenus then the corresponding virtual classes from Fig. 3.1 would be instantiated, but it can also be an instance of any subclass of BasicMenus, such as MenusAccel from Fig. 3.2. In the latter case the virtual classes of MenusAccel would be instantiated instead.

Second, the classes accessed as attributes of a family object can also be used as types. A type of the form e.C, where e is an expression evaluating to a family object and C is a class name, denotes a set of instances of class C that belong to the family object e. Types including expressions in their definitions are in general known as *dependent types*. Thus, the types of the form e.C are also a form of dependent types. In Fig. 3.5 we can see such types used in declarations of variables on lines 6, 7, 9 and 13, e.g., variable openWith is declared with type menus.MenuItem, which means that it is an instance of the virtual class MenuItem from the family menus.

Within a family class, references to its virtual classes are implicitly qualified by a reference to this family object. For example, the return type of the method itemAt (line 30, Fig. 3.1) is considered as this.out.MenuItem, where out is a reference to the enclosing object (i.e., the family), which means that it returns MenuItems of the same family as the owner of the method.

Dependent typing plays an important role for ensuring type safe extensions of groups of classes. In such extensions we assume that a class is used only in combination with the classes from the same extension. For example, at line 33 of Fig. 3.2, processKey can be called on an instance of MenuItem returned by itemAt, because it is assumed that itemAt returns menu items of the same extension, i.e., instances of this.out.MenuItem with this.out of type MenusAccel. Recall that in the corresponding object-oriented design (See line 36 of Fig. 2.10), a type-cast was required to call processKey, because the type system could not guarantee that only menu items with support for accelerators could be returned by itemAt.

The assumption that only classes from the same extension are used together is specified by the family part of the types, e.g., as was mentioned above, the return type of itemAt is this.out.MenuItem. In the context of the method call on line 33 of Fig. 3.2, we know that the family this.out is of type MenusAccel, and the menu items returned by itemAt are at least instances of MenuAccel.MenuItem, and, thus, have the method processKey.

In contravariant positions, such as method parameters, the family of a type plays a constraining role. For example, method addItem declared on line 36 of Fig. 3.1 expects its argument to be an instance of this.out.MenuItem, i.e., a menu item of the same family. Therefore, on lines 8 and 15 of Fig. 3.5, where the method is called, it is checked if the arguments given to the method are menu items of the same family, which is the case in the example, because both the target of the method call and the arguments are declared as members of the family menus.

3.3 Semantics of Virtual Classes

In this section we take a look at concrete semantic aspects of virtual classes as they are implemented in CaesarJ.

At first, we explain the semantics of mixin composition implemented in CaesarJ, because it serves as a basis for the semantics of inheritance between virtual classes and propagating mixin composition. We define the mixin linearization algorithm implemented in CaesarJ and discuss its properties. We also explain the specific problems with class constructors in the presence of mixin composition and discuss possible solutions.

Next, we explain the inheritance semantics of virtual classes and its relation to mixin-based inheritance. Then we define the precise semantics of inheritance and mixin composition with virtual classes implemented in CaesarJ and explain how it supports the intuitive semantics described in Sec. 3.2.

The last two subsections cover the remaining semantic aspects of virtual classes in CaesarJ. In Sec. 3.3.3, we give an informal overview of the type system implemented in CaesarJ. In Sec. 3.3.4 we explain the benefits of abstract family classes and give an informal definition of the semantics of abstract class and method declarations in CaesarJ.

3.3.1 Mixin Composition

As was shown in Fig. 3.4, CaesarJ classes can be composed with the operator &: The class MenusAccelML composes MenusAccel and MenusML. The composition operator realizes a variant of multiple inheritance that linearizes the superclasses, thereby avoiding

```
cclass Figure {
    int posx, posy;
    void draw(Graphics g) { }
}

cclass Colored extends Figure {
    Color col;
    void draw(Graphics g) {
        g.setColor(col);
        super.draw(g);
    }
}

cclass Text extends Figure {
    String text;
    void draw(Graphics g) {
        g.drawText(posx, posy, text);
        super.draw(g);
    }
}

cclass Rect extends Figure {
    int x2, y2;
    void draw(Graphics g) {
        g.drawRectangle(posx, posy, x2, y2);
        super.draw(g);
    }
}

cclass ColoredRect extends Colored & Rect { }

cclass ColoredText extends Colored & Text { }

cclass ColoredTextRect extends ColoredText & ColoredRect { }
```

Figure 3.6: Figure mixins and their compositions

ambiguities with respect to method dispatch. As we will see in Sec. 3.3.2, mixin composition is not only a useful language feature by itself, but also serves as a basis for implementation of inheritance relationships between virtual classes.

For an illustration of mixin composition semantics in CaesarJ, we will use the example of Fig. 3.6. The example defines a class Figure, a base class for graphical figures with an empty draw method and fields to store the position of the figure, and three subclasses of it: Colored - for colored figures, Text for text figures, and Rect for rectangles. The subclasses override implementation of the draw method and introduce new fields.

These subclasses can be composed by multiple inheritance to produce more specific shapes, e.g., ColoredRect, implementing colored rectangles, can be defined as a subclass of Colored and Rect. Analogously we can define ColoredText. Such compositions can be

$$\begin{aligned}
&\mathit{Mixins}(C) = C\ \mathit{Linearize}([\,\mathit{Mixins}(C') \mid C' \leftarrow C_1 \ldots C_n\,]) \\
&\qquad\qquad\quad \text{where}\ \ \mathit{ClassDef}(C) = \textbf{cclass}\ C\ \textbf{extends}\ C_1\& \ldots \&C_n\ \{\ \ldots\ \}
\end{aligned}$$

$$\begin{aligned}
\mathit{Linearize}(\epsilon) &= \epsilon \\
\mathit{Linearize}(\overline{C}\ \overline{\overline{C}}) &= \mathit{Lin2}(\overline{C}\ \mathit{Linearize}(\overline{\overline{C}}))
\end{aligned}$$

$$\begin{aligned}
\mathit{Lin2}(\epsilon, \epsilon) &= \epsilon \\
\mathit{Lin2}(\overline{C}\ C, \overline{C}'\ C) &= \mathit{Lin2}(\overline{C}, \overline{C}')\ C \\
\mathit{Lin2}(\overline{C}, \overline{C}'\ C) &= \mathit{Lin2}(\overline{C}, \overline{C}')\ C\ \ \text{if}\ \ C \notin \overline{C} \\
\mathit{Lin2}(\overline{C}\ C, \overline{C}') &= \mathit{Lin2}(\overline{C}, \overline{C}')\ C\ \ \text{if}\ \ C \notin \overline{C}' \\
\mathit{Lin2}(\overline{C}\ C\ \overline{C}'', \overline{C}'\ C) &= \mathit{Lin2}(\overline{C}\ \overline{C}'', \overline{C}')\ C \\
&\quad \text{(Note: use first case that matches)}
\end{aligned}$$

Figure 3.7: Linearization of the parents of a given class C

composed in further subclasses, e.g., the class ColoredTextRect for colored rectangles with text can be defined as a subclass of ColoredText and ColoredRect.

We can see that such multiple inheritance can produce ambiguous method implementations, e.g., ColoredRect inherits draw both from Colored and Rect, and repeated inheritance, e.g., ColoredRect inherits from Figure both over Colored and Rect. The ambiguities are resolved by the linearization of the inheritance graph, also known as topological sorting, which also eliminates repeatedly inherited classes.

The linearization algorithm used in CaesarJ is defined in Fig. 3.7. Given a class C, the *Mixins* function computes the list of all ancestors of C, including the class itself. We will refer to this list as the *mixin list* of the class.

In the definitions, we use the following notation:

- $\overline{a}$ is a list $a_1\ a_2\ \ldots\ a_n$, where n is the length of the list.
- The length of the list a is denoted as $|\overline{a}|$.
- ϵ is an empty list.
- $[\,F(x) \mid x \leftarrow \overline{a}\,]$ is the notation for a list comprehension. It evaluates to $F(a_1) \ldots F(a_{|\overline{a}|})$.

The function *Mixins* is defined recursively: the mixin lists of the parents of C are merged using function *Linearize*. The class C itself is included to the beginning of the list. In the definitions we assume the representation of a program as a class table *ClassDef*, i.e., a map from a class name to its declaration in the program. In the definition of *Mixins*, the parents of C are determined by lookup in the class table.

Linearize is defined recursively in terms of a binary linearization function *Lin2*, which defines a composition of two lists. *Lin2* traverses both lists backwards and attaches their elements to the end of the resulting list, which means that an element taken earlier appears closer to the end of the list. If the last elements of both lists are equal they are removed from the both lists and included to the new list only once. Otherwise, the last element of one of the lists is taken if it does not appear in the other list. Note that the order of the rules in Fig. 3.7 defines precedence of rules for the case when multiple rules match. Thus, the elements of the second list are taken first if possible, i.e., in this way they appear closer to the end of the resulting list.

If an element is at the end of both lists or does not appear in one them, putting this element to the end of the resulting list would definitely preserve the order of the both lists. This is, however, not always possible, thus the last rule tells that if none of the previous rules apply the last element of the second list must be placed to the end of the result, and removed from the both lists.

For an illustration, let's compute the mixin list of ColoredTextRect from Fig. 3.6. The parent list of ColoredTextRect is a result of linearizing the parent lists of its direct parents:

$Mixins(\mathsf{ColoredTextRect})$
$= \mathsf{ColoredTextRect}\ Linearize(Mixins(\mathsf{ColoredText})\ Mixins(\mathsf{ColoredRect}))$
$= \mathsf{ColoredTextRect}\ Linearize((\mathsf{ColoredText\ Colored\ Text\ Shape})\ (\mathsf{ColoredRect\ Colored\ Rect\ Shape}))$
$= \mathsf{ColoredTextRect}\ Lin2((\mathsf{ColoredText\ Colored\ Text\ Shape}),\ (\mathsf{ColoredRect\ Colored\ Rect\ Shape}))$

Now we just have to merge two lists:

$Lin2((\mathsf{ColoredText\ Colored\ Text\ Shape}),\ (\mathsf{ColoredRect\ Colored\ Rect\ Shape}))$
$= Lin2((\mathsf{ColoredText\ Colored\ Text}),\ (\mathsf{ColoredRect\ Colored\ Rect}))\ \mathsf{Shape}$
$= Lin2((\mathsf{ColoredText\ Colored\ Text}),\ (\mathsf{ColoredRect\ Colored}))\ \mathsf{Rect\ Shape}$
$= Lin2((\mathsf{ColoredText\ Colored}),\ (\mathsf{ColoredRect\ Colored}))\ \mathsf{Text\ Rect\ Shape}$
$= Lin2(\mathsf{ColoredText},\ \mathsf{ColoredRect})\ \mathsf{Colored\ Text\ Rect\ Shape}$
$= Lin2(\mathsf{ColoredText},\ \epsilon)\ \mathsf{ColoredRect\ Colored\ Text\ Rect\ Shape}$
$= Lin2(\epsilon,\ \epsilon)\ \mathsf{ColoredText\ ColoredRect\ Colored\ Text\ Rect\ Shape}$
$= \mathsf{ColoredText\ ColoredRect\ Colored\ Text\ Rect\ Shape}$

Thus the mixin list of ColoredTextRect is:

$Mixins(\mathsf{ColoredTextRect})$
$= \mathsf{ColoredTextRect\ ColoredText\ ColoredRect\ Colored\ Text\ Rect\ Shape}$

3.3.1.1 Inheritance Semantics

The computed mixin list of a class determines the order of method lookup in the class, and in this way resolves potential ambiguities. For example, ColoredTextRect indirectly inherits draw from four classes: Text, Rect, Colored and Shape. Since, Colored appears first in the mixin list of ColoredTextRect is chosen by the method lookup.

The mixin list also determines the lookup for super calls of methods. Differently from Java the target of super calls are not fixed for a class, but depends on a concrete composition of this class with other classes. A super call within a method of class C on an instance of class C' is resolved by looking up the method in the mixin list of C' starting from the class listed after C. For example, when Colored.draw is executed on an instance of ColoredTextRect, the super call on line 10 is bound to Text.draw, because Text is the next class after Colored in the mixin list of ColoredTextRect implementing draw. Analogously the super call in Text.draw would call Rect.draw, and the super call in the latter would call Shape.draw. As a result, a call to ColoredTextRect.draw would set the color of the figure, draw the text and then the rectangle.

The described form of inheritance is called *mixin composition*, because the semantics of method lookup is analogous to the semantics of mixins [BC90]. *A mixin* is a class parameterized by its parent class, i.e., it can be combined with various parents to produce different classes. We will write A[B] to denote mixin A combined with class B. In such combination the methods of A override the methods of B, and B serves as the targets of the super calls in A.

Every CaesarJ *class declaration* can be used in two ways: as a class, which can be used for instantiation of objects, and as a mixin which can be composed with other mixins to produce different classes. When considering a class declaration as a mixin, its declared parents specify the upper bound of the actual parents, with which the mixin can be combined, e.g., since Text is declared with the parent Figure, it can be composed only with subtypes Figure; analogously, ColoredRect can be composed with classes, which are subtypes of both Colored and Rect.

The semantics of a class is equivalent to the combination of all its ancestors as mixins in the order specified by the computed mixin list, e.g., to produce the ColoredTextRect first we combine Rect with Shape as a parent (i.e., the parent of Rect remains as it is), then the resulting class is used as a parent of Text, the result thereof is used as parent of Colored and so on. Note that according to the computed mixin list, each mixin is combined with a class which is a subtype of the declared upper bound, e.g., in our example Text is combined with Rect[Shape], which is a subtype of Shape, or ColoredRect is combined with Colored[Text[Rect[Shape], which is a subtype of both Colored and Rect.

3.3.1.2 Constructors

Class constructors present a specific problem for mixin composition. The situation is different from methods, because while a subclass preserves the methods of the superclass and their signatures, this does not hold for constructors (e.g., in Java and C++). Constructors are not inherited, and a subclass can define totally different constructors than its superclass. Preservation of constructors is not required, because they are not relevant for subtyping: a constructor cannot be called after an object is constructed and, thus, is not a part of the interface of the object.

The fact that constructors are not preserved in subclasses is a problem for super constructor calls in mixins, because there is no guarantee that they will be preserved in the classes with which a mixin can be combined. For example, for the Figure and its subclasses could have typical constructors, which initialize their instance variables with the values of the corresponding constructor parameters, as shown in Fig. 3.6. The constructor of Colored on line 10 takes the coordinates and the color of the figure as parameters, and calls the super constructor with the coordinates, as expected by the constructor of its superclass Figure. In the class ColoredText, which inherits from the combination of Colored and Text, the super reference in Colored is bound to Text, which does not define a constructor matching the super call of line 11, but instead additionally requires a parameter to initialize the text.

CaesarJ addresses this problem by treating constructors as methods, which means that a subclass inherits constructors of its superclass and cannot hide them, only override their implementation. It means that in the example of Fig. 3.8 Colored and Text inherit the constructor of Figure, i.e., the constructor taking only coordinates of the figure as parameters is still available in these classes. ColoredText then inherits the constructors of all three classes, and the super calls of lines 11 and 19 are still bound to the constructor of Figure, because it is not overridden in any of the subclasses.

An alternative solution to the problem can be found in Scala [OZ05b]. It disallows defining constructors in the classes that can be used as mixins (declared with keyword trait in Scala). For the example of Fig. 3.8 this would mean that since Colored is used as a mixin its constructor it could not have a constructor. Instead, ColoredText should define a constructor which calls the constructor of Text and additionally takes care of initialization of the fields inherited from Colored.

Both CaesarJ and Scala solutions are not optimal. In CaesarJ constructors are degraded to a more handy syntax for analogous initialization methods, which could be called after an object is constructed, while in Scala constructors in mixins are simply disallowed.

```
cclass Figure {
    int posx, posy;
    Figure(int x, int y) {
        this.posx = x; this.posy = y;
    }
}

cclass Colored extends Figure {
    Color col;
    Colored(int x, int y, Color col) {
        super(x, y);
        this.col = col;
    }
}

cclass Text extends Figure {
    String text;
    Text(int x, int y, String txt) {
        super(x, y);
        this.text = txt;
    }
}

cclass ColoredText extends Colored & Text { }
```

Figure 3.8: Constructors for figure classes

3.3.1.3 Properties of the Linearization Algorithm and Alternatives

The presented linearization algorithm is a relaxed version of the *C3 linearization* algorithm [BCH+96, Ern99c]. The strict version of the C3 algorithm corresponds to the definition of *Lin2* in Fig. 3.7 without the last rule. The strict version guarantees that the partial order between classes defined by their order in the extends clause is preserved, e.g., Colored would appear before Rect in the mixin list of any subclass of ColoredRect, because Colored is declared before Rect in its extends clause (line 30, Fig. 3.6).

In some cases the order of parents can be different in different classes, e.g., in Fig. 3.9 both C and D inherit from A and B, but in a different order. Because of that, the strict C3 algorithm would fail for class E, which combines C and D, but it would be accepted by the relaxed algorithm which would favor the order defined in the later mixin, i.e. in D.

The strict linearization algorithm is problematic in larger inheritance hierarchies and especially in the cases when classes are developed independently, because it is difficult to ensure that the order of parents is consistent in all classes. In particular, we experienced this problem in our experiments of feature-oriented programming with virtual classes [GA07].

```
cclass A { }
cclass B { }
cclass C extends A & B { }
cclass D extends B & A { }
cclass E extends C & D { }
cclass F extends D & A & B { }
```

Figure 3.9: An example of conflicting order of parents

If there are no conflicts, the relaxed algorithm works in the same way as the strict one, i.e., preserves the order declared in the extends clause. Also, the relaxed algorithm *always* preserves the partial order defined by inheritance: a class always precedes its parents in the mixin lists of its all subclasses.

The linearization algorithm has been designed so that the ordering of mixins in a class can be controlled by the programmer of a subclass, in a similar spirit as when the programmer of a subclass can decide to override a method from any of its parents. For example, class F of Fig. 3.9 overrides the order of inheritance from A and B defined in its parent class D. This is achieved by redeclaring inheritance from A and B directly in F in the desired order. Since A and B appear after D in parent list of F their ordering has a precedence, over the ordering inherited from D.

Other linearization algorithms are also possible, e.g., a slightly different linearization algorithm is used in Scala [OZ05b], which is simpler, but does not necessarily preserve the order of parents also in the cases when there are no ordering conflicts, e.g., the linearization of ColoredTextRect in an analogous design with Scala traits would be

ColoredTextRect ColoredText Text ColoredRect Colored Rect Shape

because the mixins inherited from ColoredRect would appear after any of the additional mixins from ColoredText, which means that Text would appear before Colored in the list, which contradicts to the order of parents defined in ColoredText. As a result, the text would be drawn before setting the color of the figure.

Nevertheless, the linearization algorithm of Scala also supports overriding the ordering of parents in subclasses as explained above, which enables fixing the order of inheriting from Colored and Text in ColoredTextRect. Besides, the linearization algorithm of Scala has an advantage for mapping to a single inheritance hierarchy, because the first parent class can be reused as it is when constructing analogous inheritance chain. In this way more code duplication can be avoided, when translating mixin composition to Java bytecode.

3.3.2 Inheritance with Virtual Classes

3.3.2.1 Explicit and Implicit Inheritance

Virtual classes can be related by *explicit inheritance* relationships with other virtual classes, e.g., CascadeMenuItem on line 16 of Fig. 3.1 is declared as a subclass of MenuItem. In addition, a virtual class *implicitly inherits* from its furtherbound, i.e. the virtual classes with the same name from the parent family, if one exists. For example, CascadeMenuItem declared in MenusAccel (line 21 of Fig. 3.2) implicitly inherits from the CascadeMenuItem of BasicMenus.

The explicit inheritance relationships between virtual classes are preserved, e.g., CascadeMenuItem in MenusAccel still inherits from MenuItem, even if this inheritance relationship is not redeclared on line 21 of Fig. 3.2. The inheritance relationships between virtual classes are, however, updated to refer to the newest version of the class. This means that CascadeMenuItem in MenusAccel inherits from the new version of MenuItem in MenusAccel, declared on line 2 of Fig. 3.2.

Inheritance relationships are also updated in the inherited virtual classes that are not explicitly redeclared in the heir family. For example, MenusAccel does not redeclare CheckMenuItem and RadioMenuItem, but these classes are automatically updated to inherit from the furtherbinding of MenuItem. Analogously PopupMenu and MenuBar are updated to inherit from the furtherbinding of Menu. As a result, even the classes not explicitly redefined in an heir family are not necessarily identical to their counterparts in the parent family.

In the case of virtual classes, the desired semantics of multiple inheritance, resulting from combination of implicit and explicit inheritance, can be clearly derived from the intended intuitive semantics. As was discussed in Sec. 3.2.1, the contents of the heir family class are seen as a description of change to the parent family. According to such view, the changes to a parent of a virtual class A can never replace functionality defined directly for A in the parent family – the furtherbinding of the parent of A only replaces the old version of the parent.

Such replacement of parents corresponds to the semantics of the mixins. In a virtual class inheriting both from its furtherbound A and an explicit parent B, the furtherbound is considered as a mixin combined with the parent class, i.e., A[B]. For example, MenusAccel.CheckMenuItem can be seen as inheriting from BasicMenus.CheckMenuItem[MenusAccel.MenuItem]. In this case, BasicMenus.CheckMenuItem is used as a mixin, while MenusAccel.MenuItem is used as a class.

$$\begin{aligned}
&\mathit{Mixins}(\mathsf{G}) = \mathsf{G}\\
&\mathit{Mixins}(P.C) = [\,\mathit{Defs}(P.C') \mid C' \leftarrow \mathit{Ancestors}(P.C)\,]\\
&\mathit{Ancestors}(P.C) = C\ \mathit{Linearize}([\,\mathit{Ancestors}(P.C') \mid C' \leftarrow \mathit{Parents}(P.C)\,])\\
&\mathit{Parents}(P.C) = \mathit{Linearize}([\,\overline{C} \mid P' \leftarrow \mathit{Defs}(P.C),\\
&\qquad \mathit{ClassDef}(P'.C) = \textbf{cclass}\ C\ \textbf{extends}\ C_1\&\ldots\&C_n\ \{\ \ldots\ \}\,])\\
&\mathit{Defs}(P.C) = [\,P'.C \mid P' \leftarrow \mathit{Mixins}(P),\ \mathit{ClassDef}(P'.C) \neq \bot\,]
\end{aligned}$$

Figure 3.10: Linearization of the inheritance graph of a given class C

3.3.2.2 Linearization of Inheritance Graph for Virtual Classes

As was discussed above, the semantics of inheritance relationships between virtual classes can be defined in terms of mixins. Thus, we can reuse the mixin composition semantics defined in Sec. 3.3.1 for definition of semantics of virtual classes. The computation of mixin list defined in Fig. 3.7 needs to be updated to incorporate the specific relationships of virtual classes. First, the mixin list must include the furtherbounds of a virtual class in addition to its (explicit) parents. Second, the algorithm must take to account that the parents of a virtual class can be inherited from its furtherbounds.

Linearization algorithm for virtual classes is given in Fig. 3.10. In the new definitions a class is identified not by simple name C, but by *a path* of class names separated by dots ($P = \mathsf{G}.C_1.C_2\ldots.C_n$), which specifies the location of the class in the nested structure of classes, starting with a special class name G representing the global scope, e.g., $\mathsf{G}.C_1.C_2$ is a virtual class with name C_2 in a family class with name C_1. The class table *ClassDef* now takes a path as a parameter.

The form of paths implies recursive class nesting structure, i.e., it allows virtual classes to act as family classes containing deeper virtual classes. The algorithm treats all virtual classes in a uniform way independently from their depth of nesting. The global scope G is introduced for convenience of treating top-level classes as if they were virtual classes of G. The imaginary class G does not have any inheritance relationships, thus the mixin list of G contains only G.

The process of collecting the inherited mixins is distributed over multiple functions: *Defs*, *Parents*, *Ancestors* and *Mixins*. $\mathit{Defs}(P.C)$ collects visible definitions of the class C in the context of P. In principle it is the definition of C in P itself and in its furtherbounds. $\mathit{Parents}(P.C)$ collects the *names* of the declared parents of C from all its definitions in P. $\mathit{Ancestors}(P.C)$ traverses the inheritance graphs defined by *Parents* in order to collect the *names* of all ancestors of C in P, including C itself. Finally, *Mixins* collects all definitions of all ancestors of C in the context of P.

Each of the functions defines a different aspect of linearization of the inheritance links of a virtual class. According to the definition of $\mathit{Mixins}(P.C)$ the furtherbounds of each ancestor of C are kept together in its mixin list. In particular, it means that the furtherbounds of C override the mixins inherited its explicit parents, which corresponds to the intuitive semantics of virtual classes as was explained above.

The internal order of each furtherbound group is determined by the mixin linearization of the enclosing class P as can be seen from the definition of *Defs*. It also determines the order of collecting the parents of a virtual class from its furtherbounds in *Parents*. The graph of ancestors of the class, based on the collected parents, is then sorted using the original linearization algorithm presented in Sec. 3.3.1, i.e., the function *Linearize* as it is defined in Fig. 3.7.

Note that such linearization corresponds to the intuition of considering an heir family as a description of difference from its parent, and its mixin list as the order in which these differences are applied. Indeed, the result of the mixin composition defined by the linearization of a virtual class $C_1.C_2$ defined in Fig. 3.10 is equivalent to first merging the ancestor families of C_1 in the order determined by the algorithm of Fig. 3.7, and then in the resulting family class merging the ancestors of C_2 again by the algorithm of Fig. 3.7.

The presented linearization algorithm for virtual classes is different from the algorithm used in the formalization of virtual classes in [EOC06] and early implementations of CaesarJ [AGMO06], which are slightly simpler and do not support the described intuition, e.g., it is not guaranteed that the furtherbounds of a virtual class appear before the mixins inherited from its explicit parents in the linearization of the class.

3.3.3 Dependent Types

Since the type system of virtual classes is subsumed by the type system of dependent classes, defined in Sec. 5.2, in this section we will give only a informal description of types implemented in CaesarJ.

As was explained in Sec. 3.2.3, virtual classes, accessed over polymorphic references to family objects, can be used as types. Such type consists of a family expression and a class name, and represents the set of objects that belong to that family and are instances of the virtual class. For example, the type of the variable `menu1`, declared at line 11 of Fig. 3.11 is `menus1.PopupMenu`, where `menus1` is a family expression, and `Menu` is a class

```
 1 class Test {
 2    menus.Menu buildMenu(final BasicMenus menus) {
 3       menus.Menu menu = menus.new MenuBar("MyMenu");
 4       menu.addItem(menus.new CascadeMenuItem("MyItem");
 5       return menu;
 6    }
 7
 8    void test() {
 9       final BasicMenus menus1 = new MenusAccel();
10       final BasicMenus menus2 = new MenusML();
11       menus1.Menu menu1 = menus1.new PopupMenu("Menu1"); // ok
12       menus2.Menu menu2 = menu1; // error
13
14       final MenusAccel menus3 = new MenusAccel();
15       menus3.Menu menu3 = menus3.new PopupMenu("Menu3");
16       menu3.processKey(...);
17
18       menus1.Menu menu4 = buildMenu(menus1);
19       buildMenu(menus3).processKey(...);
20
21       BasicMenus.MenuItem menu5 = menu4.itemAt(0); // ok
22       menu4.addItem(menu5); // error
23    }
24 }
```

Figure 3.11: An example of dependent types

from that family. The type specifies that the family of menu1 is equal to the value of the variable menus1, and it is a direct or indirect instance of the class Menu.

The subtype relation between such types requires that their family expressions refer to the same family and the virtual classes are known to be subclasses of each other in the context of the family, e.g., the initialization of the variable menu1 is correct, because it is of type menus1.Menu and is assigned an expression of type menus1.PopupMenu. In contrast, the assignment on line 12 is not allowed, because a variable of type menus2.Menu cannot be assigned a value of type menus1.Menu, because menus1 and menus2 are not guaranteed to refer to the same family object at runtime.

As we can see subtyping relation requires checking equivalence of the family expressions. Equivalence of two arbitrary expressions is an undecidable problem, because expressions could include method calls implementing arbitrary, potentially non-terminating computations. Therefore, the form of family expressions is constrained to so-called to paths, which start from this or some other variable and navigate over various instance fields. Such types are also usually called path-dependent types [OCRZ03, EOC06] in order to distinguish them from other forms of dependent types.

Path-dependent types are syntactically expressed as lists of names separated by dots, which is also the form of types supported in Java. CAESARJ simply extends the set of possible names to be used in a type to the names of various variables. A dependent type in CaesarJ is in principle any sequence of names representing a valid Java variable access followed by one or more class names. In particular, this means that a family expression is any valid combination of accesses to local variables, method parameters, static fields, instance fields, the this object, and the enclosing objects. Access to this and the enclosing objects is determined implicitly.

The interface of a dependently typed object depends on the static type of the family expression. For example, variable menu3 of type menus3.Menu is known to be an instance of MenusAccel.Menu, because menus3 is declared with the type MenusAccel. Thus, we can call method processKey, declared in MenusAccel.Menu on menu3. The static type of a family expression, involving the reference to this or a method argument, depends on the context of using the type. Types depending on this, and the role of such types for type safe extensions of groups of classes were explained in Sec. 3.2.3.

Fig. 3.11 demonstrates an example of a type depending on a method argument. Method buildMenu on line 2 is declared with the return type menus.Menu, where menus is the method argument. Hence, the type of the returned menu depends on the type of the given family. For example, on line 18 the method is called with menus1, thus the returned object is known to be of type menus1.Menu and we can safely assign it to a variable of this type. In the next line, the method is called with menus3 as a parameter. Since menus3 is of type MenusAccel, we know that the returned object is of type MenusAccel.Menu and, thus, has method processKey.

Static references to virtual classes can also be used as types. For example, variable menu5 is declared with type BasicMenus.MenuItem on line 21. Such type does not specify the family of the object, and only tells that its family must be of type BasicMenus. Initialization of the variable is correct, because the expression menu4.itemAt(0) is of type menus1.MenuItem, and menus1 is an instance of BasicMenus. The static type BasicMenus.MenuItem provides the same interface as the type menus1.MenuItem, but nevertheless it is a weaker type, because it does not specify the family, e.g., the call on line 22 is not permitted, because the method expects an object from the family menus1, but the family of menu5 is not known.

Implementation of the type-system in CaesarJ is based on the vc^n calculus, presented in Sec. 5.2[4]. Since the calculus is defined in an algorithmic style, its relationships are almost directly implemented in the compiler. More specifically, the compiler takes over the path normalization and typing (cf. Sec. 5.2.3), the type substitution relationships (cf. Sec. 5.5), the subtyping relationship (cf. Sec. 5.2.5), and the expression typing (cf. Sec. 5.2.6). The calculus, however, abstracts from concrete method selection strategies.

[4]The calculus is defined not for virtual classes directly, but for dependent classes, which generalize virtual classes with dispatch by multiple parameters as will be explained in the next chapter.

For this purpose CaesarJ relies on the propagating mixin composition semantics defined in Sec. 3.3.2.

The type system of older versions of CaesarJ relied on the definitions of a variant of the *vc* calculus [EOC06]. Ad-hoc solutions were used to resolve the limitations of *vc* with respect to the structure of types and expressions (See Sec. 6.1.3 for more details on the limitations). Soundness of these ad-hoc solutions was not verified. By moving to the definitions of the vc^n calculus in the new implementation of compiler, we substantially simplified implementation of the type system and gave it a more solid theoretical basis, because the limitations of *vc* were resolved directly by the vc^n calculus. Unlike the ad-hoc improvements to *vc*, the soundness of vc^n is proven, and the proof is automatically checked using the Isabelle/HOL system [NPW02].

3.3.4 Abstract Classes

The benefits of polymorphism can be increased by using abstract family classes. A separate interface concept is not necessary because we do not have the single inheritance bottleneck. An abstract family class can describe an interface of a family of objects. Such interface can specify the classes available in the family, the signatures of their methods and constructors, and even the subtype relationships between them. We will call such interfaces *family interfaces*.

For example, Fig. 3.12 shows an abstract family class IMenuStruct, which defines the public interface of the BasicMenus (Fig. 3.1) sufficient for constructing menus and accessing their structure. It declares virtual classes of BasicMenus, but with abstract declarations of methods and constructors. The methods irrelevant for clients constructing menus, e.g., draw, are omitted.

Such abstract family class can be used for hiding implementation of menus from its clients. For example, class FileMenuContrib from Fig. 3.5, contributing menu items for working with files to a given menu, can be decoupled from menu implementation by using IMenuStruct as shown in Fig. 3.12. Such decoupling makes the design more stable and enables reuse of the client FileMenuContrib with different implementations of the interface.

As in Java, declaring a class as abstract means that it cannot be instantiated. According to this rule, we cannot create instances of the class IMenuStruct. The meaning of declaring virtual classes as concrete or abstract is analogous: only concrete virtual classes can be instantiated. Note that since virtual classes are instantiated polymorphically, we can declare them as concrete in abstract family classes, in order to enable their polymorphic instantiation. For example, all virtual classes in IMenuStruct except Menu are declared as

```
abstract cclass IMenuStruct {
    cclass MenuItem {
        abstract MenuItem(String label, Action action);
        abstract void setAction(Action action);
        ...
    }

    cclass CascadeMenuItem extends MenuItem {
        abstract CascadeMenuItem(String label);
        abstract void addItem(MenuItem item);
        ...
    }

    cclass CheckMenuItem extends MenuItem { ... }

    cclass RadioMenuItem extends MenuItem { ... }

    abstract cclass Menu {
        abstract MenuItem itemAt(int i);
        abstract int itemCount();
        abstract void addItem(MenuItem item);
        abstract void addAction(String label, Action action);
        ...
    }

    cclass PopupMenu extends Menu { ... }

    cclass MenuBar extends Menu { ... }
}

cclass BasicMenus extends IMenuStruct { ... }
```

Figure 3.12: Interface describing menu structure, and an example of using it

```
1 class FileMenuContrib implements MenuContributor {
2    void contribute(final IMenuStruct menus, menus.Menu menu) {
3       menus.CascadeMenuItem openWith = menus.new CascadeMenuItem("Open With");
4       menu.addItem(openWith);
5       ...
6    }
7    ...
8 }
```

Figure 3.13: Abstraction from implementation of menus

concrete, which means that they can be instantiated over a polymorphic family reference of type IMenuStruct.

We can use *abstract constructors* to specify constructors with parameters for classes without giving their implementation. For example, CascadeMenuItem is instantiated over a variable menus of type IMenuStruct on line 3 of Fig. 3.13. It calls the constructor declared as line 9 of Fig. 3.12, which takes a menu label as a parameter.

Note that in a family interface we can also declare inheritance relationships between virtual classes. In this way the subclasses inherit the interface of the superclasses, e.g., method itemAt declared for Menu on line 19 of Fig. 3.12 is also available for its subclasses PopupMenu (line 26) and CascadeMenu (line 28).

Inheritance links within a family interface also establish necessary subtype relations, e.g., to type check the call on line 4 of Fig. 3.13, we must know that the type of openWith, i.e., menus.CascadeMenuItem, is a subtype of the argument type expected by menu.addItem, i.e., menus.MenuItem. Because of that, it is necessary to know that CascadeMenuItem is a subclass of MenuItem already in the family interface, i.e., in IMenuStruct.

In Java, a class containing an abstract method must be declared as abstract, but as can be seen in Fig. 3.12 concrete virtual classes can contain abstract methods. In CaesarJ this rule is weakened: *a method can be abstract when at least one of its enclosing classes is abstract.* According to the rule abstract methods are allowed in both concrete virtual classes of abstract family classes and abstract virtual classes of concrete family classes, e.g. Menu of BasicMenus in Fig. 3.1. This rule is sufficient to ensure that abstract methods will never be called, because it excludes the possibility of direct instances of the class declaring the method.

The completeness checking must recursively traverse all concrete classes and their concrete virtual classes and ensure that all their methods and constructors are implemented. Note that completeness check of a family class must consider not only the concrete virtual classes declared in it, but must also recheck the ones implicitly inherited from the

```
cclass Figures {
   abstract cclass Figure {
      int posx, posy;
   }
}

cclass FiguresDraw extends Figures {
   abstract cclass Figure {
      abstract void draw(Graphics g);
   }
}

cclass Rects extends Figures {
   cclass Rect extends Figure {
      int x2, y2;
   }
}

cclass MyApp extends Rects & FiguresDraw {
   /* error: Rect.draw() not implemented */
}
```

Figure 3.14: Example of completeness checking

parent families, because of the additional methods that could be inherited from the updated parents.

For illustration, consider Fig. 3.14, which shows an example of independent extensions of an abstract virtual class with new methods and new subclasses. Family Figures declares an abstract virtual class Figure. In its heir family FiguresDraw, the class Figure is extended with an abstract method draw. Another heir family Rects defines Rect as a concrete subclass of Figure. All three family classes – Figures, FiguresDraw and Rects – are completely implemented, and thus can be declared as concrete. However, the family class MyApp, which combines FiguresDraw and Rects is not completely implemented, because it contains a concrete virtual class Rect, which inherits an unimplemented method draw from Figure. Such error can be resolved by giving an implementation of Rect.draw in MyApp, or declaring the family class as abstract.

3.4 Variation Management with Virtual Classes

In this section we evaluate the advantages of virtual classes for the variation scenarios presented in Chapter 2. As was discussed in Sec. 3.2, virtual classes provide inheritance and polymorphism for a group of classes and so improve modularization of variations involving multiple objects. Indeed, we used the examples of such variations presented in Sec. 2.4. The capability of virtual classes to relate variations of multiple classes is also useful for combination of static and dynamic variation discussed in Sec. 2.3.3, and for modularization of variations of frameworks presented in Sec. 2.5.

3.4.1 Variations of Multiple Objects

As a mechanism for inheritance of groups of classes, virtual classes solve the problems of modeling variations of groups of objects by inheritance that were identified in Sec. 2.4 and illustrated by an example of menu structures, consisting of multiple menu items and cascade menus (See Fig. 2.9 - 2.14). Since we used this example for motivation and illustration of virtual classes (See Fig. 3.1 - 3.4), we don't explain this example in detail again, but only discuss generalization of this design of any group of objects and analyze it with respect to the problems identified in Sec. 2.4.

In Sec. 3.4.1.1, we investigate advantages of expressing individual group variations with virtual classes. Then in Sec. 3.4.1.2 we analyze composition of such variations by means of propagating mixin-composition. As explained in Sec. 3.4.1.3, the same techniques can be applied for flexible modularization of application-level variations. Then, in Sec. 3.4.1.4, we discuss the possibilities of combining variations at different hierarchically related scopes. Finally, in Sec. 3.4.1.5, we analyze applicability of virtual classes for expressing dynamic variations of a group of objects.

3.4.1.1 Single Group Variations

In general, a group of objects can be implemented as a family class with virtual classes implementing its members, as was illustrated for menu structures in Fig. 3.1. Variations of a composite structure can be implemented in heir family classes, e.g., MenusAccel of Fig. 3.2 and MenusML of Fig. 3.3 implement optional functionality for respectively menu accelerators and multi-language support. The varying functionality can be composed by mixin composition (with propagating semantics) as illustrated in Fig. 3.4

While the inheritance graph of family classes can be used for modeling variations of an object group, individual variations of the group's members can be modeled by inheritance relationships between virtual classes: common parent virtual classes capture commonalities of member object and are used as types for describing polymorphic relations between them. As we can see the inheritance relations directly express combination of variations at the scope of members with variations at the scope of the entire composite.

Let's analyze this design with respect to the problems identified in Sec. 2.4.1.

The first problem that we observed in the object-oriented design was that the type system did not express the fact that the members of a group share its variations. The references between the member objects were typed by their base classes, thus varying functionality of an object required type casts to access the corresponding varying functionality of related objects.

In the design with virtual classes this problem is solved by dependently typed references between the objects of a group. In the type of a reference between objects we can specify that they belong to the same family, and thus share its variations. As a result, the objects of a family can safely access varying functionality of each other. For example, all references to MenuItem in Menu of Fig. 3.1 are implicitly resolved to this.out.MenuItem, which means that Menu works only with MenuItem's from the same family. As a result, the class Menu in MenuAccel of Fig. 3.2 can safely call methods of MenuItem, specific to this variation, on its menu items, e.g., method processKey can be called without a type cast on line 33.

The second problem was related to combination of group variations with individual variations of its members. In the object-oriented design both kinds of variations were modeled by inheritance hierarchies of the member classes. We assumed some form of multiple inheritance in order to enable composition of variations along the two dimensions. The problem was that manual composition of variations produces excessive number of classes and is not stable with respect to extensions with new variants of member objects.

The advantage of the design with virtual classes is that variations of a group and its members are automatically composed. Since variations of the group are modeled by inheritance between family classes, their effect to member classes are expressed by furtherbindings of these classes. At the same time individual variations of objects are modeled by explicit inheritance between virtual classes. As was defined in Sec. 3.3.2, a furtherbinding of a virtual class is automatically combined with its explicit subclasses. As a result, the variations of an object with respect to its group, expressed by the furtherbindings of its class, are automatically composed with the individual variations of the object, expressed by explicit subclasses. For example, the support for accelerators implemented by attributes and methods of MenuItem in MenuAccel of Fig. 3.2 are automatically propagated to its subclasses, i.e., CascadeMenuItem, CheckMenuItem and RadioMenuItem, even if the latter two classes are not declared in MenuAccel. In this way we avoid redundant class declarations and improve the stability of the design, because if a new type of menu items is introduced it will also automatically inherit the support for acceleration.

Interactions between group variations and individual variations of its members can be resolved in the furtherbindings of the virtual classes representing the individual variations. For example, the accelerator functionality defined for menu items in the furtherbinding of MenuItem in MenuAccel can be specialized for specific menu items. In the example, we specialize implementation of method processKey for CascadeMenuItem in Fig. 3.2 so that accelerator key processing is propagated to cascade menus.

The third problem of the object-oriented design was that the code directly instantiating the classes of a group was not reusable with variations of the group, because these variations were implemented as new classes. As was explained, the problem can be alleviated by Abstract Factory design pattern, which in principle encodes late-bound instantiation of classes dispatched by the dynamic type of a concrete factory.

Virtual classes support late-bound instantiation directly, which is an advantage in several respects.

First, the glue code for supporting late-bound instantiation is avoided. Note, that the number of factory methods required to encode late-bound instantiation of virtual classes is not always proportional to the number of virtual class declarations, because furtherbinding of a virtual class propagates into subclasses and thus require redefinition of factory methods of these subclasses.

Second, the need for late-bound instantiation does not need to be preplanned and prepared in advance, because all virtual classes are late-bound instantiated. Of course, grouping classes into families requires certain preplanning too, but it is comparable with the decision of declaring methods in a class rather than externally, which makes them overridable. Since making an inner class extensible does not require additional coding, all inner classes are made extensible without additional cost, in this way supporting the Open-Closed principle.

Finally, directly supported late-bound instantiation of classes is safer than their encoding with factory methods. As was mentioned in Sec. 2.4.1 factory methods can be evaded by instantiating the classes directly. Also there is no guarantee that only objects instantiated by the same factory are used together. These problems do not exist for virtual classes, because their late-bound instantiation cannot be evaded, and dependent typing ensures that only objects of the same family are used together.

Note, that while solving the problems of modeling group variations by inheritance, we preserve its advantages. Variations of a group of objects are modularized in separate classes, which reduces interdependencies within the system and enables non-invasive extensions of classes with new variations. Moreover, we can flexibly modularize variations of both the implementation and the interface of the objects, and statically bound variants can be reflected in types of families and its objects.

3.4.1.2 Combining Group Variations

So far we analyzed the advantages of virtual classes for modularizing individual variations of a group of objects. Now let's take a look at the additional problems that are specific to combination of such variations.

In the object-oriented design presented in Sec. 2.4.2, variations of a group of objects were combined by combining each of their member classes separately by means of multiple inheritance, and defining a new concrete factory class with methods to instantiate the combined classes.

Propagating mixin composition significantly simplifies composition of independent group variations, because it automatically composes corresponding variations of all their member classes. As can be seen in Fig. 3.4, the optional variations of support for menu accelerators and multi-language can be composed by simply declaring a family class which inherits from both MenuAccel and MenuML.

Compared to automatic composition enabled by propagating mixin composition, manual composition of variations not only requires much more glue code (see Fig. 2.14 for comparison), but is also more error-prone, because it is not statically checked if all member classes are consistently combined and the respective factory methods are redefined. This is especially a problem with respect to evolution of the base implementation of a composite structure and its variations, because there is no guarantee that the compositions of these variations are consistently updated to reflect the changes introduced by the evolution.

In the example of Fig. 3.4, combination of variation is completely automatic, because these variations are orthogonal, but in other cases interactions between variations can be resolved by redefining classes and methods in the family class defining the composition.

3.4.1.3 Application-Level Variations

Application-level variations can be also modularized by virtual classes as a special case of group variations. As was discussed in Sec. 2.4.3, static application level variations can be managed at the level of modules, thereby avoiding the problems of expressing group variations by class inheritance identified in Sec. 2.4.1 and 2.4.2. The main problem with such approach is the rigid modularization structure of conventional programming languages, which requires all fields and methods of a class to be declared within one module.

Limitations to decomposition into modules can be lifted by using family classes as modules. Dependencies between modules are then modeled by inheritance relations between the corresponding family classes. The actual classes that implement application objects are declared as virtual classes. Such a module system gives more flexibility for modularization of varying functionality of an application. Since application classes are declared as virtual classes they can be freely extended with new fields, operations and even new inheritance relations. Variation is bound by declaring a family class, which inherits from the family classes encapsulating the variations to be included in the specific application instance. In principle, such family class can be seen as a root module of the application instance.

The module system based on family classes resolves the limitations of Java modules, stated in Sec. 2.4.3. First, the design supports extensions with both new operations and new object types. An heir family can extend the parent family both with new virtual classes and with new operations for existing virtual classes. Second, a family class encapsulating a particular variation of an application can specialize implementation of existing methods of application classes by overriding them in their furtherbindings.

At the same time, the advantages of managing variations by modules outlined in Sec. 2.4.3 are preserved, because the design with virtual classes also resolves the problems of expressing group variations by inheritance, as was explained in Sec. 3.4.1.1 and 3.4.1.2. In other words, due to virtual classes the advantages of managing application-level variations at the level of modules are made available for any object group variations.

3.4.1.4 Variation at Multiple Scopes

In Sec. 2.4.4, we indicated that variations can affect different groups of objects, and an object can be affected by variations at different scopes. As was explained in Sec. 3.4.1.1 group variations can be expressed by inheritance at the level of family classes. An instance of a virtual class belongs to the group identified by its family object and varies together with other objects of the family. For an object to be affected by variations of different object groups, it must be associated with multiple families.

Class families can be organized hierarchically. A family class can be a virtual class and in this way belong to a larger family. A virtual class of a virtual class belongs to two families: the family identified by the instance of the direct enclosing class, as well as to the family of the latter family. In this way we can express variations at multiple hierarchically organized scopes with all the advantages of type safety and composition of variation explained in Sec. 3.4.1.1 and 3.4.1.2.

For illustration, consider the example of project management data model from Sec. 2.4.4. A project is a complex structure consisting of project members, tasks and other project related objects. The objects of a project are affected by variations of the project and application-level variations. Since all objects of a project are also application objects, these two groupings can be organized by hierarchical nesting of virtual classes. Application-level variations can be modeled by an inheritance hierarchy of the outermost family classes, while variations of projects are then modeled by inheritance hierarchy of the virtual classes. At the same time the virtual classes representing projects can be family classes with virtual classes implementing different project objects.

```
cclass ProjectModel {
   cclass Project {
      cclass Task {
         double duration; List<Task> dependsOn; Member assignedTo;
         ...
      }
      cclass Member {
         String name; double availability;
         String displayName() { return name; }
         ...
      }
   }
}

cclass MultiTeamProjectModel extends ProjectModel {
   cclass MultiTeamProject extends Project {
      cclass Team {
         List<Members> members; Member teamLead;
         ...
      }
      cclass Member {
         Team team;
         ...
      }
   }
}

cclass EmployeeRegister {
   cclass Employee {
      String employeeID; List<String> name; Department deptm;
      String fullName() { ... }
      ...
   }
   cclass Department { ... }
   ...
}

cclass ProjectEmployees extends ProjectModel & EmployeeRegister {
   cclass Project {
      cclass Member {
         Employee empl;
         String displayName() { return empl.fullName(); }
      }
   }
}

cclass MyApp extends ProjectEmployees & MultiTeamProjectModel & ... { }
```

Figure 3.15: Modeling variations of project models with multi-level virtual classes

Such design is sketched in Fig. 3.15. The outermost classes modularize variations at the level of an application, i.e., they represent various features that can be included or not included in particular instances of the application. The ProjectModel on line 1 contains the definition of basic project model, which is necessary for all instances of the application. It contains a virtual class Project with deeper virtual classes describing the objects of the project and their relationships. This data model contains only the entities, attributes and relationships that are necessary for all projects.

The structure of ProjectModel supports two kinds of extensions of the project model.

First, we can introduce a new type of projects by defining a subclass of Project and extending its virtual classes. For example, class MultiTeamProjectModel on line 15 encapsulates an extension of ProjectModel with support for multi-team project. It introduces a new virtual class MultiTeamProject declared as a subclass of Project and extending the project model with new entities and relationships specific to multi-team projects. In particular, it introduces a new virtual class Team and refines Member to relate members with teams.

Second, we can define extensions common to all types of projects by refining Project and its internal virtual classes. For example, class ProjectEmployees on line 38 encapsulates integration of the application to the employee register of the company. It refines virtual class Project and the deeper virtual class Member. The furtherbinding of Member declares a reference to an instance of Employee, coming from the employee register model[5], and overrides some of the methods of the furtherbound to exploit the integration. Such extension affects all kinds of projects, because the classes describing specific types of projects implicitly inherit from the furtherbinding of Project.

As a result, variations of deep virtual classes, such as Member and Task, can be bound at different scopes. For example, in the class MyApp on line 47, which defines an application including both ProjectEmployees and MultiTeamProjectModel, the attribute empl will be available for the members of all projects, while the attribute team will be available only for the members of multi-team projects. Such different scoping is achieved by different combinations of furtherbinding and explicit inheritance between virtual classes.

As was explained In Sec. 3.4.1.4, family classes can be used as extensible modules, which can extend classes with new members and override implementation of existing methods. In addition to that, by declaring family classes as virtual classes, we can refine the existing virtual classes of a particular family class as well as extend the family with new virtual classes. For example, in the module ProjectEmployees we were able to extend class Member, which is a virtual class of Project already declared in the module ProjectModel. We could also introduce new virtual classes to Project, e.g., we could introduce InternalMember

[5]In this example, the employee register model is also encapsulated within a family class EmployeeRegister (line 28) and is used by inheriting from it.

and ExternalMember as subclasses of Member to differentiate between company employees and external members of a project.

3.4.1.5 Dynamic Variations

In Sec. 2.4.5 we discussed the ways of modularizing dynamic variations of multiple objects. In simple cases dynamic variations can be outsourced to a shared helper object. In more sophisticated cases, the dynamic variation can be expressed by multiple helper objects serving for different objects of the group. An example of the latter scenario is dynamic variation of look-and-feel style in an application. Such variation affects all application's widgets and can be expressed as a combination of dynamic variations of individual widgets, which in turn are expressed by means of helper objects.

As was discussed in Sec. 2.4.5, the problem with such a design is that it does not guarantee that the variation is consistently bound in all objects of the group. This can lead to type-safety problems in case of interactions between helper objects. The problem was illustrated with an example of a look-and-feel style rendering widgets with specific background textures. If a texture is not set for a widget it is inherited from the parent of the widget. Such functionality is not type-safe, because it assumes that the widget and its parent are bound to the same look-and-feel style, but this assumption is not statically enforced by the design presented in Sec. 2.4.5.

Covariant dependencies between helper objects implementing the same group variation can be expressed by grouping them into a family. In our example, we can group the visualization helpers by the look-and-feel style that they implement. Then we need to specify that all widgets of an application are bound to the visualization helpers from the same family.

A possible design is presented in Fig. 3.16. Since interfaces of visualization helpers vary for different look-and-feel styles we implement them by a class UI, declared as a virtual class of LookAndFeel. In subclasses of LookAndFeel for specific look-and-feel styles, virtual class UI is refined is an implementation of widget visualization in the respective style. The look-and-feel style, rendering widgets with different textures, is implemented by class TexturedLookAndFeel. Its UI provides operations to get and set the background texture of a widget. The currently selected look-and-feel style family (line 48) is stored in a static field UIManager.style (line 42). If the variation were bound not for entire application, but for a narrower group of objects, the reference to the family would be kept in the object representing the group.

Differently from the design presented in Sec. 2.4.5, a widget does not contain a direct reference its visualization helper. The reason for not maintaining direct references from widgets to their helpers is that we cannot relate their types to the currently selected

```
abstract cclass LookAndFeel {
    cclass UI {
        JComponent comp;
        ...
    }

    Map<JComponent, UI> uiMap = new HashMap<JComponent, UI>();

    UI getUI(JComponent comp) {
        UI ui = uiMap.get(comp);
        if (ui == null) {
            ui = createUI(comp);
            uiMap.put(comp, ui);
        }
        return ui;
    }
    ...
}

abstract cclass TexturedLookAndFeel extends LookAndFeel {
    cclass UI {
        Texture backgrTexture;

        Texture backgrTexture() {
            if (backgrTexture != null) {
                    return backgrTexture;
                }
                else {
                    return getUI(comp.parent).backgrTexture();
                }
            }
            void setBackgrTexture(Texture texture) {
                this.backgrTexture = texture;
            }
            ...
        }
        ...
    }
}

class UIManager {
    static LookAndFeel style;
    ...
}

abstract class JComponent {
    JComponent parent;
    LookAndFeel.UI getUI() { return UIManager.style.getUI(this); }
    ...
}
```

Figure 3.16: Look-and-feel style with configurable fill textures

family of look-and-feel style, because only immutable references to family objects can be used in types. In order to specify that a widget refers to a helper from the family of the currently selected look-and-feel style, i.e., of type UIManager.style.UI, we would need to declare the field UIManager.style as final, because only then it could be used in types. The problem is that such declaration would prohibit dynamic variation of look-and-feel style, because we wouldn't be able to change the value of UIManager.style.

Therefore, in Fig. 3.16 we present a more sophisticated design. Instead of direct references from widgets to helpers, LookAndFeel maintains a map from widgets to their helpers (line 7) and provides a method getUI (line 9) to return the helper associated to the widget. If such helper is not available yet, it is created on-demand. A widget uses the method getUI to access its helper from the currently selected look-and-feel style family (line 48), which is stored in a static field UIManager.style (line 42).

The presented design enables type-safe interaction between visualization helpers of different widgets. For example, line 29 demonstrates implementation of inheritance of texture from the parent widget. The helper of the parent widget is retrieved by method getUI(), which returns an instance of UI from the same family, thus, we can safely access its method backgrTexture.

As we can see, although we managed to preserve flexibility of the design while making it type-safe, we had to introduce more glue code and to make sacrifices to the runtime efficiency (replace direct object references by lookups in a hash table). Such complications could be avoided by supporting type-safe mutable references to family objects, which are however not supported in CaesarJ and other type systems with virtual types [OCRZ03, EOC06, NQM06, CDNW07, Hut06], and thus remain a topic for future research.

3.4.2 Interaction of Inheritance and Helper Objects

Virtual classes are useful for dealing with interactions between static variations, expressed by inheritance, and dynamic variations outsourced to helper objects. As was explained in Sec. 2.3.3, outsourcing dynamic variations to helper objects causes splitting of a logical object into multiple physical objects, corresponding to the main object and its helpers. These physical objects tend to vary together with respect to other variations of the logical objects. In Sec. 2.3.3, it was demonstrated that such covariant dependencies cannot be properly expressed by conventional object-oriented features. In Sec. 3.4.2.1, we demonstrate how such dependencies can be expressed by declaring helper objects as virtual classes. In Sec. 3.4.2.2 we show how the initial design can be extended to make it extensible with respect to the both dimensions of variation. In Sec. 3.4.2.3 we discuss limitations of virtual classes, when dealing with dynamic variations of multiple objects.

3.4.2.1 Expressing Dependencies of Helper Objects on their Owners

The covariant dependencies between the physical objects that belong to the same logical object can be naturally expressed by considering the logical object as a family of its physical objects. Such family would be represented by the class of the main object, which would contain virtual classes representing its helpers.

For illustration, we will use the example of dynamic variation of visualization styles of a widget introduced in Sec. 2.3.3. In the design with virtual classes presented in Fig. 3.17, the base class of widgets JComponent is declared as a family class with the helper classes for visualization as its virtual classes. The abstract class UI describes the interface of visualization functionality, and its concrete subclasses, such as BasicUI and MotifUI, provide implementations of the interface for different visualization styles. Visualization related functionality of a widget is delegated to its field ui of type UI. Dynamic variation of visualization style is achieved by assigning different values to ui.

The advantage of declaring helpers as virtual classes is that this makes it possible to extend their interface in a type safe way for the subclasses of the main object. For example, in JList, we can extend UI with new operations that are specific to list widgets and depend on the selected visualization style, e.g., method locationToIndex defines the mapping from given point to the list item index. The concrete subclasses of UI are refined in JList to implement the new methods as well as to specialize implementations of the methods declared by JComponent.UI.

The design with virtual classes provides a more precise typing of references between the main object and its helpers. Since helper classes are declared as virtual classes, references to them are dependently typed with respect to the type of the main object, which makes it possible to access the extended of a helper in a type-safe way. For example, ui field is of type this.UI, which means that it depends on the type of the widget. Thus, in the context of JList we can safely call specific methods of JList.UI on ui, e.g., method locationToIndex on line 25.

The reference from a helper object to its owner is managed automatically as the reference to the enclosing family this.out. This reference is also dependently typed and provides the interface of the enclosing family class. In this way instances of JList.UI can safely access the interface of JList, e.g., method getModel on line 36.

```
abstract cclass JComponent {
    UI ui;
    void setUI(UI ui) { this.ui = ui; }
    void paint(Graphics g) { ui.paint(g); }
    ...
    abstract cclass UI {
        abstract void paint(Graphics g);
        abstract Dimension getPreferredSize(JComponent c);
        ...
    }
    cclass BasicUI extends UI {
        void paint(Graphics g) { ... }
        ...
    }
    cclass MotifUI extends BasicUI {
        void paint(Graphics g) { ... }
        ...
    }
    ...
}

cclass JList extends JComponent {
    ListModel getModel() { ... }
    String getTooltipText(Point pt) {
        int idx = ui.locationToIndex(pt);
        ...
    }
    ...
    abstract cclass UI {
        abstract int locationToIndex(Point pt);
        ...
    }
    cclass BasicUI {
        int locationToIndex(Point pt) { ... }
        void paint(Graphics g) {
            ... getModel() ...
        }
        ...
    }
    cclass MotifUI {
        int locationToIndex(Point pt) { ... }
        void paint(Graphics g) { ... }
        ...
    }
    ...
}

class Test {
    void test() {
        JComponent comp = new JList();
        comp.setUI(comp.new BasicUI());
    }
}
```

Figure 3.17: Dynamic variation of visualization styles with virtual classes.

3.4.2.2 Decoupling Variations of Helper Objects

The problem with the design presented in the previous subsection is that helper classes are nested within the main class of the logical object. Such structure introduces undesired coupling, because the main class and all its dynamic variations are declared within the same module. As a result, it is not possible to extend the main class with new variants in a non-invasive way, and all clients of the main class are exposed to all its variants, even if they don't use them.

The problem is caused by the limitation that a family class cannot be extended with new virtual classes. This is, however, true only for family classes that are not virtual classes themselves. As was demonstrated in Sec. 3.4.1.4, such extensibility limitations can be resolved by using outermost family classes as extensible modules with the actual application classes declared as their virtual classes.

Figure 3.18 demonstrates a refactored version of the example using outermost family classes for modules. Instead of declaring JComponent and JList as outermost classes, we nest them within classes representing their modules, i.e., JComponentModule and JListModule respectively. In these modules we declare them only with virtual classes UI and BasicUI, which define the interface of the varying visualization functionality and its default implementation. Implementations of specific visualization styles such as MotifUI are moved to other modules, e.g. module JComponentMotifUI extends JComponent with virtual class MotifUI, which is then specialized for JList in the module JListMotifUI.

The refactored design solves the problems of the previous design with virtual classes. Widget classes and their clients are made independent from implementation of specific visualization styles, because they are moved to more specific modules. The design also supports non-invasive extensions with new visualization styles. Note that the design equally well supports extensions with new types of widgets and new visualization styles, because in new modules we can introduce both new subclasses for both JComponent and JComponent.UI.

3.4.2.3 Variations of Multiple Objects

So far we analyzed combination of dynamic and static variations of individual objects. This design can be also used to modularize variations of a group of objects by declaring the helper classes as additional virtual classes of the family class representing the group. Such a design, however, does not provide an optimal solution to the scenario, when dynamic variations need to be expressed by multiple helpers. This scenario was presented and its typing problems were discussed in Sec. 2.4.5. For illustration we used the same example of dynamic variations of look-and-feel styles, but we focused on another aspect

```
cclass JComponentModule {
    abstract cclass JComponent {
        UI ui;
        ...
        abstract cclass UI {
            abstract void paint(Graphics g);
            ...
        }
        cclass BasicUI extends UI {
            void paint(Graphics g) { ... }
            ...
        }
    }
}

cclass JListModule extends JComponentModule {
    cclass JList extends JComponent {
        ...
        abstract cclass UI {
            abstract int locationToIndex(Point pt);
            ...
        }
        cclass BasicUI {
            int locationToIndex(Point pt) { ... }
            void paint(Graphics g) { ... }
            ...
        }
    }
}

cclass JComponentMotifUI extends JComponentModule {
    cclass JComponent {
        cclass MotifUI extends BasicUI {
            void paint(Graphics g) { ... }
            ...
        }
    }
}

cclass JListMotifUI extends JListModule & JComponentMotifUI {
    cclass JList {
        cclass MotifUI {
            int locationToIndex(Point pt) { ... }
            void paint(Graphics g) { ... }
            ...
        }
        ...
    }
}
```

Figure 3.18: Separation of UI Functionality in Swing

of it, namely the problem of type-safe interaction between the helpers of different widgets, which is based on assumption that all widgets in an application are bound to the same look-and-feel style.

In Sec. 3.4.1.5 we presented a solution to the problem of interaction between helpers implementing the same dynamic group variation, based on grouping the helper objects into a family, and in this way enabling type-safe covariant dependencies between them. In that solution, we however ignored interaction between the helper objects and variations of widgets.

Although we have presented solutions both to the problem of expressing dynamic variations of group of objects by multiple collaborating helpers, and to the problem of expressing interaction between static variation of objects and their helper objects, these two solutions are not compatible with each other and, therefore, cannot be combined to solve the both problems at once. The problem is that the solutions assign helper objects to different families. On one hand, we group helpers into a family representing the dynamic variation, which is the look-and-feel style in our example, in order to enable covariant dependencies between them. On the other hand, we assign a helper to the family represented by its owner object, which is the widget in our example, in order to express covariance between the helper and its owner.

In Sec. 3.4.1.4 we have seen an example of objects assigned to different families, that represent variations at different scopes. These families were organized hierarchically, i.e., one family was declared as a member of another one. Such hierarchical organization is, however, not possible in our example, because neither the family represented by a widget is completely included to the family of a look-and-feel style, nor the other way around: a widget can be used with different look-and-feel styles, and a look-and-feel style affects multiple widgets.

Consequently, we must choose for one of the ways of grouping helper objects into families to be explicitly represented in design. Since the other way of grouping is not explicit in design, covariant dependencies between the objects of these groups must be still expressed by unsafe type-casts. For example, if we assign a visualization helper to the family of its widget, then it cannot communicate with the helpers of other widgets implementing the same look-and-feel style in a type-safe way. Analogously, if we assign a helper to the family of the look-and-feel style that it implements, then it cannot be specialized for specific types of the widget in a type-safe way.

3.4.3 Variation of Frameworks

In this section we analyze advantages of virtual classes for managing variations of frameworks, based on the scenarios and problems discussed in Sec. 2.5. Frameworks can be

```
abstract cclass GraphicalEditor {
   Graphic graphicAtPoint(Point pt) { ... }

   abstract cclass Graphic {
      abstract void paint(Graphics g);
      ...
   }

   abstract cclass NodeGraphic extends Graphic {
      abstract int connectionCount();
      abstract ConnectionGraphic connectionAt(int i1);
      ...
   }

   abstract cclass ConnectionGraphic extends Graphic {
      abstract NodeGraphic sourceNode();
      abstract NodeGraphic targetNode();
      ...
   }

   ...
}
```

Figure 3.19: Graphical editor and graphical objects

seen as a special case of composite structures, thus the analysis of modeling variations of composite structures presented in Sec. 3.4.1 is also valid for frameworks. The distinguishing feature of a framework, however, is that it explicitly abstracts from certain variations, which are intended to be provided in application-specific instances of the framework. Thus, in the following we will take a look at the advantages of framework instantiation with virtual classes and specific variation scenarios related to that.

3.4.3.1 Combining Framework Instances and Variations

Since a framework consists of a group of interacting objects, the technique of modularizing group variations, explained in Sec. 3.4.1, can be also exploited for modularizing static variations of a framework. This means that the framework and its variations are represented by a hierarchy of family classes with virtual classes implementing different framework objects and their individual variations. In such design, abstract virtual classes can be used to describe explicitly supported extension points of the framework.

For example, the family class GraphicalEditor in Fig. 3.19 encapsulates implementation of the base functionality of the graphical editing framework, which we used for illustration in Sec. 2.5. The framework explicitly supports extensions with specific types of interactive graphical objects by declaring abstract virtual classes Graphic, NodeGraphic

```
1  abstract cclass GraphicalEditorWithMenus extends GraphicalEditor {
2    Menu getContextMenu(Point pt) {
3      Graphic graphic = graphicAtPoint(pt);
4      return (graphic == null) ? getEditorMenu() : graphic.getContextMenu(pt);
5    }
6    ...
7
8    abstract cclass Graphic {
9       abstract Menu getContextMenu(Point pt);
10   }
11 }
```

Figure 3.20: Extension of the graphical editor with context menus

and ConnectionGraphic, concrete implementations of which must be supplied in application specific instances of the framework. The abstract methods of the virtual classes, such as paint and connectionCount, specify the operations expected to be supplied in the framework instances.

In the variations of a framework modularized by heir families, we can not only extend implementation of the framework, but also the requirements of the framework to its instances. This is achieved by extending the abstract virtual classes of the framework, describing interfaces of its extension points, with new abstract methods. For example, in class GraphicalEditorWithMenus of Fig. 3.20, which extends graphical editing framework with support for context menus, virtual class Graphic is extended with a new abstract method getContextMenu. In this way instances of the framework are additionally required to supply the context menus for their graphical objects.

Compared to the plain object-oriented design presented in Fig. 2.22, the advantage of the design with virtual classes is that the new expected operations can be accessed in a type-safe way. E.g., on line 4 we don't need a type-cast to call getContextMenu on a graphic object returned by graphicAtPoint. In principle, the design ensures that only a framework is used only with objects that fullfill the extended requirements.

Now let's take a look at the corresponding design of framework instances. A framework is instantiated by giving implementations to its extension points. Since the extension points are declared as abstract virtual classes, we implement them by defining their concrete subclasses. The instantiation of the framework for graphical editing logic circuits is shown in Fig. 3.21. The framework instance itself is represented by a family class CircuitEditor subclassing GraphicalEditor. Within the family class we define different concrete subclasses of NodeGraphic and ConnectionGraphic, which implement graphical objects for different elements and wires of a logic circuit. Completeness checking of virtual classes ensures that implementations of all methods expected by the framework, i.e., declared as abstract for NodeGraphic and ConnectionGraphic, are given.

```
cclass CircuitEditor extends GraphicalEditor {
    ...
    cclass WireGraphic extends ConnectionGraphic {
        Wire wire;
        WireGraphic(Wire wire) {
            this.wire = wire;
        }
        void paint(Graphics g) { ... }
        NodeGraphic sourceNode() {
            return logicCompGraphicFor(wire.sourceComp());
        }
        NodeGraphic targetNode() { ... }
        ...
    }

    abstract cclass LogicCompGraphic extends NodeGraphic {
        LogicComponent comp;
        LogicGraphic(LogicComponent comp) {
            this.comp = comp;
        }
        ConnectionGraphic getConnectionAt(int i1) {
            return wireGraphicFor(comp.connectedWires().get(i1));
        }
        ...
    }

    cclass LEDGraphic extends LogicCompGraphic {
        LEDGraphic(LED led) {
            super(led);
        }
        void paint(Graphics g) { ... }
        ...
    }
    ...
}
```

Figure 3.21: Implementation of graphical objects for circuit editor

```
cclass CircuitEditorWithMenus extends CircuitEditor & GraphicalEditorWithMenus {
  cclass WireGraphic {
    Menu getContextMenu(Point pt) { ... }
  }

  abstract cclass LogicCompGraphic {
    Menu getContextMenu(Point pt) { ... }
  }

  cclass LEDGraphicWithMenus {
    Menu getContextMenu(Point pt) {
      Menu menu = super.getContextMenu(pt)
      menu.addAction("Change Value", changeValueAction());
      return menu;
    }
  }
  ...
}
```

Figure 3.22: Extension of the circuit editor with context menus

Framework instances defined in such a way can be combined with framework variations by declaring a family class which inherits both from the family class implementing the instance and the family classes implementing the selected variations of the framework. For instance, the class CircuitEditorWithMenus, shown in Fig. 3.22, combines framework instance for logic circuits with the framework extension with context menus by inheriting from CircuitEditor and GraphicalEditorWithMenus.

In such a combination, the virtual classes of the framework instance are combined with furtherbindings of framework classes from the family classes implementing framework variations. For example, in CircuitEditorWithMenus the furtherbinding of Graphic defined in GraphicalEditorWithMenus is inherited by the concrete subclasses of Graphic coming from CircuitEditor.

If a framework variation does not declare new requirements to the framework instances, its combination with framework instances can be completely automatically resolved by propagating mixin-composition. This is, however, not the case for the extension of the graphical editing with context menus, because it introduces a new abstract method getContextMenu to Graphic. In such cases the additional requirements must be fulfilled in the composing class. Therefore, in CircuitEditorWithMenus the subclasses of Graphic declared in CircuitEditor are extended with corresponding implementations of the method getContextMenu. Otherwise, the missing methods would be detected by the type checker.

3.4.3.2 Dependency on Application Variations

The objects of a framework instance often act as adapters from existing application objects to the interface expected by the framework. Implementation of the adapters can depend on the variations of the adaptees. As was explained in Sec. 2.5.2, such dependency results in a parallel inheritance hierarchies of adapters and adaptees, which need to be related in two ways: First, for a given adaptee we must instantiate the corresponding adapter class. Second, we need to express the covariant dependency in the types of the references between adapters and adaptees. An example of such dependency is the implementation of graphical objects for different types of logic components.

In Sec. 2.5.2 we instantiated adapters by a factory method, which takes an adaptee and by means of conditions on the type of the adaptee decides which adapter class has to be instantiated. One problem with such design is that the factory method is exposed to all variants of the adaptees, which creates undesired dependencies on these variants and limits extensibility with new variants. A natural solution to this problem would be a dispatch of the method by the type of the adaptee. In a language like Java, however, this would require an invasive change to the adaptee class and would create an undesired dependency of the implementation of the adaptee on its adapters to specific frameworks, e.g., the data model of a logic circuit would be made dependent on its editing functionality.

The problem of expressing dependency of adapters on adaptees without changing the adaptee classes can be solved by employing the technique of using family classes as extensible modules described in Sec. 3.4.1.3. By declaring the adaptee classes as virtual classes, we can extend them with new late-bound factory methods for instantiation of corresponding adapters. By using the same technique we can also make framework classes extensible with new virtual classes implementing adapters for new adaptees.

Such a design is illustrated in Fig. 3.23. The external family classes are again considered as extensible modules. Assuming that the circuit model is open for extension with new types of logic components, in MCircuitModel we declare only the base structure of circuit model. The classes representing the framework and its instances, such as GraphicalEditor and CircuitEditor are nested within the corresponding modules MGraphicalEditor and MCircuitEditor, which makes the editor classes extensible with new virtual classes. The module MCircuitEditor contains the part of CircuitEditor independent from specific logic components. Extensions with specific component types are introduced in separate modules. For instance, Fig. 3.24 shows an extension of the data model and the framework instance with LED components: the module MLEDComponents extends data model with class LED, and MCircuitEditorLED extends circuit editor with an adapter class for the new component type.

```
cclass MCircuitModel {
   abstract cclass LogicComponent { ... }
   cclass Wire { ... }
   ...
}

cclass MCircuitEditor extends MCircuitModel & MGraphicalEditor {
   cclass CircuitEditor extends GraphicalEditor {
      Map<LogicComponent, LogicCompGraphic> logicAdapters
         = new HashMap<LogicComponent, LogicCompGraphic>();

      LogicCompGraphic logicCompGraphicFor(LogicComponent comp) {
         LogicCompGraphic graphic = logicAdapters.get(comp);
         if (graphic == null) {
            graphic = comp.createGraphic(this);
            logicAdapters.put(graphic);
         }
         return graphic;
      }

      abstract cclass LogicCompGraphic extends NodeGraphic {
          LogicComponent comp;
          LogicCompGraphic(LogicComponent comp) {
             this.comp = comp;
          }
          ...
      }

      cclass WireGraphic extends ConnectionGraphic {
         NodeGraphic sourceNode() {
            return logicCompGraphicFor(wire.sourceComp());
         }
         ...
      }
      ...
   }
   abstract cclass LogicComponent {
      abstract editor.LogicCompGraphic createGraphic(final CircuitEditor editor);
   }
}
```

Figure 3.23: Circuit editor supporting extensions with new logic component types

```
cclass MLEDComponents extends MCircuitModel {
    cclass LED extends LogicComponent {
        int getValue();
        ...
    }
}

cclass MCircuitEditorLED extends MLEDComponents & MCircuitEditor {
    cclass CircuitEditor extends GraphicalEditor {
        class LEDGraphic extends LogicCompGraphic {
            Color valueColor;
            public LEDGraphic(LED led) {
                super(led);
            }
            LED led() {
                return (LED)comp;
            }
            void paint(Graphics g) { ...
                g.setColor(valueColor);
                g.drawText(led().getValue());
                ...
            }
            void setValueColor(Color c) {
                valueColor = c;
            }
        }

        public void highlightValue(LED led) {
            ((LEDGraphic)logicCompGraphicFor(led)).setValueColor(HIGHLIGHT_COLOR);
        }
    }

    cclass LED {
        editor.LogicCompGraphic createGraphic(final CircuitEditor editor) {
            return editor.new LEDGraphic(this);
    }
}
```

Figure 3.24: Extension of the circuit editor with LED components

In addition to that, MCircuitEditor and MCircuitEditorLED extend the circuit model classes with factory methods instantiating the corresponding adapter classes. In MCircuitEditor we declare an abstract method createGraphic for LogicComponent (line 38 of Fig. 3.23), which takes a reference to an editor and creates an appropriate adapter in that editor. Then in module MCircuitEditorLED we give implementation of this method for LED components, i.e., for class LED the method instantiates the adapter class LEDGraphic (line 35 of Fig. 3.24).

Note that instantiation of adapter classes is dispatched by the type of the given editor. The factory method also exploits dependent types to guarantee that the returned adapter object belongs to the given editor. This ensures type safe usage of the method within the framework, e.g., no type casts are needed to call the method and use its result on line 15 of Fig. 3.23 within the method logicCompGraphicFor implementing on-demand creation of adapters for logic components.

In fact, instantiation of the adapter classes is double-dispatched by the type of the adaptee and by the type of the framework. E.g., implementation of createGraphic is dispatched by the type of the logic component, but since the adapter classes are instantiated with the given editor as a receiver, they are also dispatched by the type of the editor. Such design supports both extensions with new adaptee types, as well as with new variations of the framework. Moreover, the factory methods can be completely reused with framework variations, e.g., the factory methods defined for CircuitEditor can be reused for its subclass CircuitEditorWithMenus extending the framework instance with support for context menus. Type-safety of such reuse is ensured by dependent typing of createGraphic.

Although instantiation of adapter classes is double-dispatched in the presented design, the references to the adapters are still dependently typed only with respect to the framework, but not with respect to the adaptees. As a result, interaction between the corresponding adapter and adaptee types remains unsafe.

For example, in LEDGraphic we need to cast the reference to the adaptee to LED (line 16 of Fig. 3.24), in order to provide access to its specific interface, e.g., to access the LED value on line 20.

The method logicCompGraphicFor for retrieving the adapter of a given adaptee (line 12, Fig. 3.23) is not precisely typed either. Although the type of the adapter returned by the method corresponds to the type of the adaptee, this property is not specified by the signature of the method. For example, LEDGraphic provides a method setValueColor (line 23, Fig. 3.24) to configure the color of the displayed LED value. This method is used for highlighting the value of the given LED component on line 29 of Fig. 3.24. The method is called on the adapter of the LED component returned by logicCompGraphicFor. The returned adapter must be cast to LEDGraphic, in order to access the method.

```
 1 abstract cclass GraphicalEditor {
 2    Graphic graphicAtPoint(Point pt) { ... }
 3
 4    abstract cclass Graphic {
 5       abstract void paint(Renderer r, Graphics g);
 6       ...
 7    }
 8
 9    abstract cclass ConnectionGraphic extends Graphic { ... }
10    abstract cclass NodeGraphic extends Graphic { ... }
11
12    abstract cclass Renderer { }
13    ...
14 }
```

Figure 3.25: Supporting variations of rendering style in a graphical editing framework

Covariance between adapters and adaptees could be expressed by nesting the adapter class within the class of the adaptee. This is, however, not possible in the presented example, because the adapter classes are already nested within framework classes in order to express their dependency on the variations of the framework.

3.4.3.3 Dependency on Multiple Variation Points of a Framework

A framework can provide multiple variation points, and, as was demonstrated in Sec. 2.5.4, certain functionality of framework instances may depend on the variants of multiple variation points. For illustration, we analyzed rendering functionality in a graphical editing framework, which varies with respect to both graphical object types and rendering styles.

In Sec. 2.5.4, we have identified two problems when dealing with such variations in Java. First, we were unable to modularize implementation of a method with respect to variants of multiple variation points. Second, we could not specify that the method can be used only with combinations variants of the same framework instance, and must be implemented for these combinations. We have also analyzed a design with multimethods, which could solve the first problem, but not the second one.

Both problems can be solved with virtual classes, based on the design of a framework and its instances introduced in Sec. 3.4.3.1. In such a design, we represent the variation points of a framework by abstract virtual classes, which are subclassed in the instances of the framework. Thus, in order to support variations of rendering styles, we additionally introduce an abstract virtual class Renderer to the GraphicalEditor, as shown in Fig. 3.25. Analogously to the object-oriented design of Fig. 2.24, we declare method paint within class Graphic with a parameter of type Renderer (line 5).

```
1  cclass CircuitEditor extends GraphicalEditor {
2     ...
3
4     cclass WireGraphic extends ConnectionGraphic {
5        void paint(Renderer r, Graphics g) { r.paintWire(this, g); }
6        ...
7     }
8
9     abstract cclass LogicCompGraphic extends NodeGraphic { ... }
10
11    cclass LEDGraphic extends LogicCompGraphic {
12       void paint(Renderer r, Graphics g) { r.paintLED(this, g); }
13       ...
14    }
15
16    ...
17
18    abstract cclass Renderer {
19       abstract void paintWire(WireGraphic wire, Graphics g);
20       abstract void paintLED(LEDGraphic led, Graphics g);
21       ...
22    }
23
24    cclass SchematicRenderer extends Renderer {
25       void paintWire(WireGraphic wire, Graphics g) { ... }
26       void paintLED(LEDGraphic led, Graphics g) { ... }
27       ...
28    }
29
30    cclass PhysicalRenderer extends Renderer {
31       void paintWire(WireGraphic wire, Graphics g) { ... }
32       void paintLED(LEDGraphic led, Graphics g) { ... }
33       ...
34    }
35 }
```

Figure 3.26: Circuit editor with variations of rendering style

Although the design with virtual classes looks very similar to the object-oriented design of Fig. 2.24, there is one important difference. The declaration of method paint on line 5 specifies that the method takes not any instance of Renderer as a parameter, but only a renderer of the same framework instance. This is because the type Renderer is implicitly interpreted as this.out.Renderer, i.e., the renderer is from the family of the graphical object referenced by this. Consequently, the method paint can be called only with pairs of objects from the same framework instance and must be implemented for such pairs only.

Now let's analyze how the more precise typing can be exploited in the implementation of a framework instance. Figure 3.26 demonstrates the implementation of the framework instance for our example of circuit editors. Like in Sec. 2.5.4, we implement two rendering styles for circuits, schematic and physical, represented by respective concrete subclasses of Renderer: SchematicRenderer on line 24 and PhysicalRenderer on line 30.

The double-dispatch of method paint is encoded by two single-dispatch steps. First, in the Renderer class we declare abstract methods for specific graphical object types of the framework instance (lines 19-21). Second, we provide implementations of the methods for every concrete renderer class (lines 25-27 and 31-33). Finally, in every concrete subclass of Graphic of the framework instance, we forward implementation of paint to the corresponding method of the given renderer object (lines 5 and 12). As a result, every call to paint is first dispatched by the type of the graphical object, and then by the type of the renderer.

In the presented design implementation of the method paint is modularized with respect to the both variation points. The framework instance implemented by class CircuitEditor can be extended both with new rendering styles and new types of graphical objects. To extend with a new rendering style, we just need to define a new concrete subclass of Renderer. An extension with new type of graphical object is a bit more sophisticated: besides introducing a new subclass of Graphic representing the new object type, we must also extend Renderer and its subclasses with a new method for painting the new type of objects, which can be done in a modular way, because Renderer and its subclasses are virtual classes and thus can be extended with new methods within subclasses of CircuitEditor.

Moreover, type-checking of virtual classes ensures that methods depending on multiple variation points are implemented for all combinations of their variants of the same framework instance. The design presented in Fig. 3.25 and Fig. 3.26 ensures that paint is implemented for all valid combinations of concrete graphical object types and rendering styles. Since only combinations of objects from the same family, i.e., the same framework instance, are valid, there is no need to implement the method for unrelated graphical objects and rendering styles.

Note that the encoding of double-dispatch of method paint by single-dispatch relies on the possibility to extend virtual classes with new methods. Class Renderer, originally declared as empty in the framework (line 12 of Fig. 3.25), in the circuit editor is extended with abstract methods for painting graphical circuit objects (lines 19-21 of Fig. 3.26), and these methods can be safely called in the implementations of paint (lines 5 and 12 of Fig. 3.26), because the parameter r is specified to be a renderer from the same family. An analogous encoding of double-dispatch by single-dispatch in pure Java would require type-casts on the renderer parameter, which would invalidate the advantages of the presented design with respect to ensuring correct usage and completeness of the method.

3.4.4 Summary

In this section we have showed that virtual classes completely or partially solve the majority of the variation management problems identified in Chapter 2:

- Virtual classes directly support static variations affecting groups of objects. Propagating mixin-composition enables automatic composition of such variations. The design supports type-safe interactions between the objects of the group. It also minimizes the amount of the glue code required for modularization and composition of variations, which also contributes to reliability and stability of the code. Virtual classes at different levels of nesting can be used to combine variations at multiple hierarchically related scopes.

- Virtual classes also improve the design of dynamic group variations, outsourced to the helper objects of the group's members. Due to virtual classes, the helper objects implementing the same variation can be grouped into a family, which enables type-safe communication between them.

- The interaction between variations described by helper objects and the variations of the object modeled by inheritance is resolved as a special case of group variation. The group in that case consists of the master object and its helpers. The resulting design enables type-safe covariant interdependencies between the master object and its helpers. Extensibility with new variants of both kinds of variations is achieved by using family classes to model extensible modules.

- Framework variations can also be handled as a special case of group variations. Abstract virtual classes play an important role in such design, because they enable type-safe extension of framework abstractions. In this way variations of a framework can extend requirements to the framework instances in a type-safe way.

- Virtual classes solve some of the problems related to the adaptation of a framework to application-specific variations. First, the adaptation code is made reusable with variations of the framework. Second, by using family classes as modules, we enable modular extension of the framework instance with respect to new types of application objects.

- Functionality of a framework instance depending on several extension points of the framework can be encapsulated within the family class of that framework. Due to virtual classes, the multi-dispatched methods implementing such functionality can be encoded by single-dispatch in a modular way. The design ensures that the method is used only with combinations of variants of the same framework instance, and is completely implemented for such combinations.

Virtual classes do not solve all of the problems identified in Chapter 2, however. In particular, the following problems still remain:

- Virtual classes do not provide any improvement for modeling multi-dimensional variations of an object. Such variations are still modeled by multiple inheritance, which suffers from the problem of combinatorial explosion of classes. On the contrary, propagating-mixin composition applies an analogous technique for combining group variations, thereby inheriting the problems of that technique.
- Virtual classes fail to provide a type-safe solution for combining dynamic group variations, expressed by helper objects, with individual variations of the objects of the group. On the one hand, for a type-safe dependency of a helper on the variations of its master, it must be assigned to the family of the latter. On the other hand, to enable type-safe interaction among helpers implementing the same group variation, they must be grouped into the family representing that variation. Virtual classes cannot express the two groupings into families simultaneously.
- Virtual classes do not solve all problems of adapting a framework instance to application-specific variations. In particular, the covariant dependency of adapters on adaptees is not expressed by the design. The problem is that, since adapters are already declared as virtual classes of the framework instance, they cannot be additionally declared as virtual classes of their adaptees.

Besides, we needed relatively complicated designs to solve the stated extensibility and typing problems in some of the scenarios:

- We employed multiple levels of nesting to enable modular extensions with new types of helpers in Sec. 3.4.2.2, and new types of adaptees in Sec. 3.4.3.2. Although such design solves the extensibility problem, it forces to nest otherwise external classes. Such nesting complicates use of the classes and must be preplanned.
- Instantiation of adapters in Sec. 3.4.3.2 and modularization of functionality depending on multiple extension points of a framework in Sec. 3.4.3.3 are based on encoding of multi-dispatch by single-dispatch. Although virtual classes improve modularity of such encodings, they are still more complicated and less intuitive than the direct implementation of multi-dispatch.
- In Sec. 3.4.1.5, the type-safe interaction between helper objects implementing the same group variation was achieved at the cost of replacing direct references from master objects to their helpers with a corresponding hash-table. The resulting design is more complicated and gives up some runtime efficiency.

As we can see, most of the remaining problems are related to implementation of multi-dimensional variability. In the next chapter we will introduce the concept of dependent classes, which combines the ideas of virtual classes and multi-dispatch, and show how the new language features address the remaining problems.

4 Dependent Classes

4.1 Introduction

Virtual classes and related mechanisms have proved useful in various situations: To define families of collaborating objects [Ern01, MO02], to develop large-scale extensible components [NCM04, AGMO06, OZ05b, NQM06], to address the "expression problem" - the possibility to extend both the set of data structures and the set of operations [EOC06, Ern03], and to modularize features that involve multiple classes [MO04, AGMO06].

The downside is, however, that virtual classes must be nested within other classes. Nesting requires clustering together all classes that depend on the variations of a particular class, which may unnecessarily introduce coupling between them. It also limits extensibility: When new classes need to be modeled as depending on the variations of an existing class, one must modify that class and its subclasses to include new nested declarations.

Furthermore, nesting limits expression of variability since the interface and the implementation of a virtual class can only depend on its *single* enclosing object. There are, however, various application scenarios, where classes that depend on multiple objects are needed. In Chapter 2, we have seen several variation scenarios, where a class depends on multiple variation points: the multiple variation points of a table widget; editor objects that depend both on the editor and the data model objects that they represent; or the renderers of widgets that depend the widgets, but also on the selected look-and-feel style. We were not able to solve all problems of these scenarios with virtual classes, because the dispatch semantics and the type system of virtual classes provide direct support for dependency with respect to one variation point only.

To address these problems, we propose a generalization of virtual classes, which we call *dependent classes*. A dependent class is a class whose structure depends on *arbitrarily many* objects; this dependency is expressed explicitly over class parameters, rather than by nesting. In a sense, dependent classes can be seen as a combination of virtual classes with multi-dispatch [DG87, Cha92, Sha96, CLCM00]. The notion of path-dependent types, as employed by most proposals for static type systems for virtual classes to keep track of the dependencies between a type and the "family" or "enclosing" object, is generalized as well, so that types can depend on an arbitrary number of objects described by paths.

This chapter is organized as follows. In Sec. 4.2 we present the rationale behind the concepts of dependent classes and their semantics. In Sec. 4.3 we discuss a set of specific language design issues related to dependent classes and present the decisions made for the DepJ language. In Sec. 4.4 we analyze application of dependent classes for different variation scenarios with a special focus on their additional value with respect to virtual classes.

4.2 Dependent Classes in a Nutshell

Dependent classes applies the idea of multi-dispatch dispatch to classes. We start this section by reviewing the rationale of dynamic dispatch in the approaches supporting multi-dispatch of methods. Then we discuss the meaning of dynamic dispatch for classes and show how it can be extended to multi-dispatch. Finally, we discuss the implications of dependent classes to family polymorphism and to the dynamic dispatch of methods.

4.2.1 Dynamic Dispatch

Object-oriented languages are traditionally based on a view of objects as records containing data fields and methods. A method call can be conceived as a dynamic lookup of the method by its name in the record representing the receiver object. The actual implementation of method lookup varies in specific languages, e.g., in class-based languages objects store only a reference to the method table of its class, but they still share analogous perception of objects.

Another view is to see object-orientation as a form of decomposition of functions depending on the types of given parameters. Methods with the same name are seen as implementations of different cases of a function. The method to be executed for a specific function call is determined by dynamic dispatch, which selects the most specific case matching the given arguments. From this perspective, dynamic method lookup in the receiver object is a specific case of dynamic dispatch, namely dispatch by the type of the first argument to the function only. Such perception of object-orientation leads to more powerful forms of dynamic dispatch, such as multimethods [DG87, Cha92, Sha96, CLCM00], which can be dispatched by the types of multiple arguments, and predicate dispatch [EKC98, Mil04], which enables dispatch by logical conditions on parameter values.

Single dispatch, resulting from the representation of objects as records, is suitable for modularization of code depending on individual variation points. For example, in a graphical editing application we can identify variation points such as supported types of shape and available editing operations. We can use single dispatch to modularize rendering of specific shapes, or reactions to user input depending on the selected editing operation. However, there can also be functionality depending on multiple variation points. For example, a reaction to a mouse click may depend not on only the selected editing operation, but also on the type of the shape at the mouse pointer. Such dependencies on multiple variation points can be naturally expressed by multi-dispatch, e.g., reaction to a mouse can be implemented by a function dispatched both by the current editing operation and the shape at the mouse pointer. Although multi-dispatch can be encoded by multiple applications of single dispatch, such encodings are less concise and more error prone.

Another limitation of the representation of an object as a record is that it implies that every operation on the object is an indispensable part of it and is logically related with other operations and data fields of the object. Such a perspective is reflected in the design of the mainstream object-oriented languages, which require declaring all fields and methods of an object in one place, e.g., within its class. However, in many cases there are a lot of different unrelated functionalities working with the same objects and depending on their variations. For example, in a graphical editing application there is a lot of unrelated functionality depending on the type of a shape, such as shape rendering, persistence of shapes, various geometrical operations on shapes and so on. The requirement to nest all fields and operations of the same object within one class introduces unnecessary coupling between otherwise unrelated operations and hinders extensions with new functionality.

From the perspective of dispatch, a method is not an inherent part of an object, but rather an implementation of one of the functions depending on the object type. Therefore, some languages that support more sophisticated forms of dispatch separate declarations of methods from declarations of classes (or objects) [DG87, Cha92, Sha96, EKC98]. Instead of nesting a method within a class, the method takes an instance of the class as a parameter. As a result, methods working with the same object can be defined independently from each other, and new methods can be introduced without changing existing modules.

4.2.2 Dispatch of Classes

Madsen and Möller-Pedersen introduced virtual classes were introduced byas a new kind of object attributes [MMP89]. In this sense, they are seen as an extension based on the record-based representation of objects. In accordance to this perspective, a virtual class is defined by nesting it within the class of the owner object and is considered as a part of the object. This perception is also compatible with the rationale of virtual classes, presented in Sec. 3.2. Virtual classes were considered as integral parts of larger software units – families of classes – and were used as a means for expressing large-scale inheritance and polymorphism.

Virtual classes can also be considered from the perspective of dispatch. In analogy to the perception of methods with the same name as implementations of different cases of a function, virtual classes with the same name can be seen as implementations of different cases of a class for different types of constructor parameters. Dispatch of classes is a way to express dependency of implementation and interface of objects on their constructor parameters. While in case of methods we talk only about variations of their implementations, variations of an object can be more versatile. In addition to variation of implementation of object's methods, we may need to vary the structure of the object, namely the set of its data fields, the set of its methods and its inheritance relations.

For illustration of dispatch of classes, recall the example of a graphical editor for logic circuits, introduced in Sec. 2.5. Interface and implementation of the graphical objects for rendering and editing of a circuit component depends on the type of that component. Virtual classes can express this dependency by nesting definition of the graphical object within the classes implementing different types of logic components as shown in Fig. 4.1. The class LogicComponent, representing the base class of all logic components, contains a virtual class LogicCompGraphic, implementing graphical representation and editing of the component.

Variations of the graphical object with respect to the component type are implemented in furtherbindings of the virtual class in the respective subclasses of LogicComponent, e.g., LogicCompGraphic within LED contains rendering and editing functionality specific to LED components. In furtherbindings of a class we can modularize variations of its method implementations, e.g., implementation of paint, as well as variations of its structure, i.e., the set of its methods and fields, e.g., the graphical objects of LED additionally have field valueColor and method setValueColor. The inheritance relations of the class can also be varied, e.g., Subcircuit is represented as a composite graphical object, and thus its furtherbinding of LogicCompGraphic additionally inherits from CompositeGraphic.

Dispatch of classes implemented by virtual classes provides the characteristic advantages of dispatch outlined in Sec. 4.2.1. Dependencies of a class on specific variants, e.g., specific circuit component types, are separated from each other and from the code generic with respect to that variation point, e.g., editing functionality generic with respect to circuit component types. Such modularization results in better stability and extensibility as was explained in Sec. 4.2.1.

On the other side, virtual classes exhibit limitations analogous to the ones identified for single dispatch of methods in Sec. 4.2.1. Virtual classes express dispatch by one con-

```
abstract cclass LogicComponent {
    ...
    cclass LogicCompGraphic extends NodeGraphic {
        CircuitEditor editor;
        LogicCompGraphic(CircuitEditor editor) { super(editor); }
        ConnectionGraphic getConnectionAt(int i1) { ... }
        ...
    }
}

cclass LED extends LogicComponent {
    int value;
    int getValue() { return value; }
    ...
    cclass LogicCompGraphic {
        Color valueColor;
        void paint(Graphics g) { ... }
        void setValueColor(Color c) { ... }
        ...
    }
}

cclass Subcircuit extends LogicComponent {
    ...
    cclass LogicCompGraphic extends CompositeGraphic {
        Graphic getChildAt(int i1) { ... }
        ...
    }
}
```

Figure 4.1: Expressing the dependency of graphical objects on circuit component types through nesting

structor parameter only, i.e., the family object given as the qualifier of an instantiation expression. As a result, only dependencies on one variation point are modularized by such dispatch. For example, the graphical object for a logical component depends not only on the type of the logical component, but also on the variations of editor functionality, represented by inheritance hierarchy of the editor classes. The reference to the editor is given as an additional constructor parameter to LogicCompGraphic in Fig. 4.1, but dispatch with respect to this parameter is not supported. In fact, variations of graphical objects with respect to the editor can be expressed by declaring them as virtual classes of the editor classes, as was demonstrated in Sec. 3.4.3, but then they cannot be nested within the circuit component classes to express dependency on the variations of the components.

Another problem is that by expressing dependency through nesting we introduce coupling between methods and classes depending on the same variation points. For example, nesting of LogicCompGraphic within LogicComponent and it subclasses introduce highly undesired coupling between the data model and its presentation. The clients of the data model classes are also exposed to its presentation and editing functionality, even if they do not need them.

4.2.3 Multi-Dispatch of Classes

To address the limitations of virtual classes, we propose a generalization of virtual classes, which we call *dependent classes*. A dependent class is a class whose structure depends on *arbitrarily many* objects; this dependency is expressed explicitly over class parameters, rather than by nesting. In a sense, dependent classes can be seen as a combination of virtual classes with multi-dispatch.

Figure 4.2 shows the implementation of LogicCompGraphic, the class of the graphical objects for logic components, as a dependent class. Instead of nesting declarations of the class within the classes of logic components, we declare that the class takes a logic component as a parameter. Such parameters are bound like conventional constructor parameters during instantiation of the class. They are also available during the lifetime of an object as immutable fields. For example, LogicCompGraphic takes two constructor parameters, a reference to the logic component it represents and a reference to the editor instance. These references can be accessed from the object over the fields comp and editor.

Dispatch of a class is expressed through multiple declarations of the class for different types of parameters. In Fig. 4.2 we can see three declarations of LogicCompGraphic. The declaration of LogicCompGraphic with comp of type LogicComponent defines functionality common to the graphical object of all logic components. The declaration of

```
class CircuitEditor extends GraphicalEditor { ... }

class LogicCompGraphic(LogicComponent comp, CircuitEditor editor) extends NodeGraphic {
   ConnectionGraphic(editor) getConnectionAt(int i1) { ... }
   ...
}

class LogicCompGraphic(LED comp, CircuitEditor editor) {
   Color valueColor;
   void paint(Graphics g) { ... comp.getValue(); ... }
   void setValueColor(Color c) { ... }
   ...
}

class LogicCompGraphic(Subcircuit comp, CircuitEditor editor) extends CompositeGraphic {
   Graphic(editor) getChildAt(int i1) { ... }
   ...
}
```

Figure 4.2: Dependency of graphical objects on circuit component types via dependent classes

LogicCompGraphic with comp of type LED contains the functionality specific to graphical objects of LED components. It provides a specific implementation of paint method, as well as introduces new fields and methods, such as valueColor and setValueColor. Like in case of virtual classes, refinements of a dependent class can also introduce new inheritance relations, e.g., LogicCompGraphic for Subcircuit additionally inherits from CompositeGraphic and implements its methods.

In an analogous way, dependent classes can also be used to express dependency of graphical objects on the variations of their editors. For example, in an extension of a graphical editor with support for context menus, its graphical objects must additionally specify their context menus. Such a design is shown in Fig. 4.3. Class Graphic, the base class of all graphical objects, is parameterized by a reference to the editor. To extend the interface of graphical objects with support for context menus, the declaration of Graphic with editor of type GraphicalEditorWithMenus additionally declares a method getContextMenu (line 27), which is intended to be implemented in concrete subclasses.

Differently from virtual classes, dependent classes can be dispatched by all constructor parameters. For example, the graphical objects for circuit components depend both on the types of the components, as well as on the variations of the editor. In Fig. 4.2 we have seen specializations of the class LogicCompGraphic with respect to the constructor parameter representing the circuit component. Analogously we can specialize the class by the second constructor parameter representing the editor as shown in Fig. 4.4. Class LogicCompGraphic with editor of type CircuitEditorWithMenus (line 3) defines the specific functionality of instances of the class in editors with support for context menus.

```
class GraphicalEditor {
    ...
}

abstract class Graphic(GraphicalEditor editor) {
    abstract void paint(Graphics g);
    ...
}

abstract class NodeGraphic(GraphicalEditor editor) extends Graphic {
    abstract ConnectionGraphic(editor) getConnectionAt(int i1);
    ...
}

abstract class CompositeGraphic(GraphicalEditor editor) extends Graphic {
    abstract Graphic(editor) getChildAt(int i1);
    ...
}

abstract class ConnectionGraphic(GraphicalEditor editor) extends Graphic { ... }

class GraphicalEditorWithMenus extends GraphicalEditor {
    ...
}

abstract class Graphic(GraphicalEditorWithMenus editor) {
    abstract Menu getContextMenu(Point pt);
}
```

Figure 4.3: Expressing dependency of graphical objects on variations of editor functionality

```
class CircuitEditorWithMenus extends CircuitEditor, GraphicalEditorWithMenus { ... }

class LogicCompGraphic(LogicComponent comp, CircuitEditorWithMenus editor) {
    Menu getContextMenu(Point pt) { ... }
}

class LogicCompGraphic(LED comp, CircuitEditorWithMenus editor) {
    Menu getContextMenu(Point pt) { ... }
}
```

Figure 4.4: Circuit editor with support for context menus

```
 1 class CircuitEditor extends GraphicalEditor {
 2     LogicCompGraphic(comp, this) graphicForComp(LogicComponent comp) {
 3         LogicCompGraphic(comp, this) graphic = logicAdapters.getVal(comp);
 4         if (graphic == null) {
 5             graphic = new LogicCompGraphic(comp, this);
 6             logicAdapters.putVal(comp, this);
 7         }
 8         return graphic;
 9     }
10     void highlightValue(LED led) {
11         graphicForComp(led).setValueColor(HIGHLIGHT_COLOR);
12     }
13     ...
14 }
```

Figure 4.5: Using the dependent class LogicCompGraphic in the circuit editor

In this class declaration we specify a default context menu for circuit components by implementing the method getContextMenu.

A dependent class can also be specialized by multiple parameters at once. For example, LogicCompGraphic on line 7 of Fig. 4.4, declared with comp of type LED and editor of type CircuitEditorWithMenus, contains the functionality specific to graphical objects representing LED components in editors with context menu. It provides a specific implementation of getContextMenu for LED components, overriding the default implementation of the method.

Due to the double role of classes as templates for object instantiation and object types, the effect of class dispatch is also twofold: a class can be dispatched both dynamically during its instantiation and statically for determining interfaces of types based on the class. In the following, we will take a closer look at the semantics of the different aspects of using dependent classes.

Instantiation and Dynamic Dispatch. During object creation, class dispatch can be seen as collecting all declarations of a class matching the given constructor parameters. The created object inherits the fields and methods of all these declarations. This kind of dispatch depends on the runtime values of the constructor parameters. Note that differently from method dispatch, which selects a unique method implementation to be executed, the created object inherits all class declarations collected by class dispatch.

For example, the definition of the instance of LogicCompGraphic created within method graphicForComp on line 5 of Fig. 4.5 depends on the values of comp and this, i.e., the logic component given as a parameter to the method and the enclosing editor instance. If comp is an instance of LED, then the constructed object inherits the declaration of LogicCompGraphic for LED (declared on line 8 of Fig. 4.2), its methods and fields. Analogously, if this is of type CircuitEditorWithMenus, then the created instance of LogicCompGraphic would inherit declaration of LogicCompGraphic with editor of type CircuitEditorWithMenus (line 3 of Fig. 4.4). The declaration of LogicCompGraphic for LED and CircuitEditorWithMenus (line 7 of Fig. 4.4) would be inherited by the constructed object, if both comp and this are of respective types.

Types and Static Dispatch. In addition to the dispatch for the construction of objects, dependent classes also support parameterization and dispatch of types. Like in the case of virtual classes, a type can be a path or a class type (See Sec. 3.3.3). A path is a singleton type containing only the object referenced by the path expression. A class type can be parameterized by types of parameters of that class.[1] For example, method graphicForComp on line 2 takes a circuit component and returns a graphical object for the given component in the current editor. This contract of the method is specified by its return type LogicCompGraphic(comp, this), which tells that the returned object is not only an instance of LogicCompGraphic, but also that its first parameter (component) is equal to the value of comp given to the method, and its second parameter (editor) is equal to this, i.e., the current editor.

Parameterization of types by paths allows abstracting from the variations represented by the paths without losing precision. The method graphicForComp is generic with respect to variations of the editor and the types of logic components, but typing precision is not lost, because it is dependently typed with respect to the polymorphic variables this and comp, which hide the respective variations. For a specific method call, the return type will be specialized based on the types of the parameter bindings of the method call. For example, the call to graphicForComp on line 11 binds argument comp to expression led of type LED, while the type of the target object remains the same, i.e., this. The type of the method call is obtained, by substituting its parameters with specific bindings, i.e., its type would be LogicCompGraphic(LED, this).

In order to determine the interface of an instance of a type based on a dependent class, we must perform static dispatch of the class for the given parameters, i.e., we must collect all declarations of the class matching the statically known types of the parameters. The interface provided by the type is the sum of all these declarations. For example, to determine the interface of the object returned by the method call graphicForComp(led) on line 11, we must perform dispatch of its type LogicCompGraphic(LED, this) for this of type CircuitEditor, i.e., we must collect all declarations of the class LogicCompGraphic matching comp of type LED and editor of type CircuitEditor. Since the declaration of LogicCompGraphic on line 8 of Fig. 4.5 matches the type parameters, the methods and

[1]Formal definition of types and type relations is given in Sec. 5.2.

```
abstract class Graphic(GraphicalEditor editor) {
    Dimension size;
    void setSize(Dimension size) { this.size = size; update(); }
    void update() { repaint(); }
    ...
}

...

class GraphicalEditorWithLayout extends GraphicalEditor {
    ...
}

abstract class CompositeGraphic(GraphicalEditorWithLayout editor) {
    void layoutChildren() { ... }
    void update() { layoutChildren(); repaint(); }
    ...
}

class CircuitEditorWithLayout extends CircuitEditor, GraphicalEditorWithLayout {
    void updateLayout(Subcircuit sc) {
        graphicForComp(sc).layoutChildren();
    }
    ...
}
```

Figure 4.6: Circuit editor with support for layout

fields of this declaration are available on the returned object, which allows us to safely call setValueColor on the returned object.

Inheritance and Its Role in Dispatch. The third way of using a class, besides instantiation and describing types, is inheriting from it. Inheritance affects both the dynamic and the static dispatch of classes, introduced above. During object construction, the constructed object inherits not only the declarations of the directly instantiated class, but also the matching declarations of its superclasses. Analogously, the interface of a type based on a dependent class also includes matching declarations of the superclasses of the class.

Like in case of virtual classes, a dependent class inherits variations of its superclasses. For example, on line 15 of Fig. 4.2, LogicCompGraphic is declared as subclass of CompositeGraphic for a component of type Subcircuit and an editor of type GraphicalEditor. The class CompositeGraphic is also declared as a dependent class of an editor (See line 15) of Fig. 4.3). Thus, the dispatch by the editor parameter is propagated from LogicCompGraphic to CompositeGraphic.

For illustration of the effect of inheritance on class dispatch, consider the example of Fig. 4.6, which shows an extension of graphical editors with support for automatic layout within composite graphical objects. The declaration of class CompositeGraphic on line 14 extends the class for an editor of type GraphicalEditorWithLayout with layout functionality. It adds method layoutChildren to perform layout of the children, and overrides method update to recompute layout as a part of the update.[2] This specialization of CompositeGraphic is implicitly propagated to subclasses of CompositeGraphic. For example, it is inherited by graphical objects for subcircuit components in a circuit editor with layout support, i.e., by instances of type LogicCompGraphic(CircuitEditorWithLayout, Subcircuit).

Such implicit inheritance is achieved by propagation of (static and dynamic) class dispatch of classes over inheritance relations, e.g., an instance of type LogicCompGraphic(CircuitEditorWithLayout, Subcircuit) matches the declaration of LogicCompGraphic of line 15 of Fig. 4.2 and, therefore, is also an instance of CompositeGraphic. Because of propagation of dispatch over inheritance, we also collect declarations of CompositeGraphic matching editor of type CircuitEditorWithLayout and so determine that the object inherits methods from the declaration of CompositeGraphic of line 14 of Fig. 4.6. As a result, the update call on line 3 in the context of this of dynamic type LogicCompGraphic(CircuitEditorWithLayout, Subcircuit) would be bound to the implementation of update from line 16. Also, we can safely call method layoutChildren on an instance of LogicCompGraphic(CircuitEditorWithLayout, Subcircuit) on line 22.

4.2.4 Multiple Families

In Sec. 3.2.3, we presented the concept of family polymorphism, as the idea of the polymorphic usage of families of classes, rather than individual objects. More precisely, polymorphic references to family objects can be used to access classes in types and instantiation expressions. One effect of the polymorphic access of classes is the static and dynamic dispatch of their interfaces and implementations, which we discussed in the previous two subsections. Another effect of family polymorphism is the possibility to relate objects from the same family, and in this way enable type-safe covariant dependencies between such objects.

From this perspective, dependent classes generalize virtual classes with the possibility to express membership of an object in multiple families. Each parameter of an object of a dependent class represents one of the families of the object. For instance, the two parameters of the class LogicCompGraphic from Fig. 4.2 expresses membership of its instances, i.e., graphical objects for logic components, in two families: the editor and the logic component from the data model.

[2]The method update is originally declared in Graphic for GraphicalEditor on line 4 and is called after changing the graphical object, e.g., after changing its size on line 3.

```
class LogicCompGraphicAttrib(LogicComponent comp) {
    Color baseColor;
    ...
}
class LogicCompGraphicAttrib(LED comp) {
    Color valueColor;
    ...
}
...

class LogicCompGraphic(LogicComponent comp, CircuitEditor editor) extends NodeGraphic {
    void saveGraphicAttrib(LogicCompGraphicAttrib(comp) attrib) { ... }
    void restoreGraphicAttrib(LogicCompGraphicAttrib(comp) attrib) { ... }
    ...
}

class LogicCompGraphic(LED comp, CircuitEditor editor) {
    Color valueColor;
    ...
    void restoreGraphicAttrib(LogicCompGraphicAttrib(comp) attrib) {
        ... valueColor = attrib.valueColor; ...
    }
    ...
}
```

Figure 4.7: Covariant dependency between graphical objects and graphical attributes of logic components

Each parameter can be used to express covariant relation with other objects of the respective family. In case of an instance of LogicCompGraphic, its parameter editor can be used to relate the object with other graphical objects of the editor. For example, the return type of the method getConnectionAt on line 4 is ConnectionGraphic(editor), which tells that the connections returned by the method are from the same editor as the instance of LogicCompGraphic. In this way, we ensure that only graphical objects from the same editor are interconnected and all of them provide the interface expected by that editor, e.g., in a graphical editor with context menus, we know that all its graphical objects implement method getContextMenu.

Analogously, the parameter comp of an instance of LogicCompGraphic relates it with the family represented by the logical component. The family of the logical component includes not only the component and its representation in the circuit editor, but also other objects depending on the logic component: representations of the logic component in various application-specific analyses, simulations and views. A logic component can also have representations in other subsystems of the data model, e.g., in a physical layout of the circuit. By considering the logic component as one of the families of such objects, we can enable covariant dependencies between them.

For illustration of a covariant dependency between objects from the family of a logic component consider the example of Fig. 4.7. The graphical objects visualizing logic components can be configured by various graphical attributes, such as colors and fonts. In order to preserve the graphical attributes associated to the logic components between multiple sessions of a circuit editor, we define class LogicCompGraphicAttrib (lines 1-9) with variables to store the attributes. The definition of this class depends on the type of the logic component, e.g., LED components additionally need the color for the text displaying their values (line 6).

Class LogicCompGraphic provides methods to save and restore its graphical attributes (lines 12 and 13). In the signatures of these methods we can exploit the class parameter comp as a family to require an object of type LogicCompGraphicAttrib(comp) as a parameter to the methods, which ensures that the graphical object receives a correct set of graphical attributes to be saved or restored. For instance, in the LogicCompGraphic working with a LED component, we can be sure that the parameter to method restoreGraphicAttrib provides the color for its value (line 21).

4.2.5 Dependent Classes and Multimethods

Since dependent classes and multimethods share similar semantics for dispatching functionality, the question arises as to how they interact in terms of language design. As will be demonstrated below, multi-dispatch of classes indirectly supports multi-dispatch of methods. Furthermore, dependent classes enhance multimethods by supporting more precise definition of method signatures. In particular, types of methods parameters can be related with each other by means of dependent typing, this way constraining the set parameter combinations to be handled by the method.

4.2.5.1 Encoding Method Dispatch by Class Dispatch

By nesting a method within a class declaration, we specify method implementation for the instances of the class. Since a dependent class can have multiple declarations, more precise is to say that a method nested within a class declaration is applicable to the instances of the class matching the parameters of the class declaration. As we can see, applicability of a method is determined by applicability of the enclosing a class declaration. Consequently, the expressivity of class dispatch is implicitly available for method dispatch.

In particular, multi-dispatch of classes can be used to encode multi-dispatch of methods. The listing on the left-hand side of Fig. 4.8 shows a typical example of multimethods. Method intersect takes two shapes and determines whether they intersect. The implementation of the method depends on the type of both shapes, e.g., the intersection of

```
abstract class Shape { }
class Circle extends Shape { ... }
class Rect extends Shape { ... }

abstract boolean intersect(Shape sh1, Shape sh2);
boolean intersect(Rect sh1, Rect sh2) { ... }
boolean intersect(Circle sh1, Rect sh2) { ... }
boolean intersect(Rect sh1, Circle sh2) { ... }
boolean intersect(Circle sh1, Circle sh2) { ... }
```

```
class Intersect(Shape sh1, Shape sh2) {
    abstract boolean execute();
}
class Intersect(Rect sh1, Rect sh2) {
    boolean execute() { ... }
}
class Intersect(Circle sh1, Rect sh2) {
    boolean execute() { ... }
}
class Intersect(Rect sh1, Circle sh2) {
    boolean execute() { ... }
}
class Intersect(Circle sh1, Circle sh2) {
    boolean execute() { ... }
}
```

Figure 4.8: Encoding multimethods by dependent classes

two circles is determined by a different algorithm than intersection of two rectangles. Different case implementations of the method can be naturally modularized by multi-dispatch.

Such modularization can be encoded by dependent classes as shown on the right side of Fig. 4.8. We declare dependent class Intersect taking two shapes as parameters, and implement the intersection test as execute method of the class. The method does not take any parameters, but instead uses the values of the class parameters as input. To check whether two shapes intersect, we need to instantiate Intersect with these shapes as the parameters and call execute on the constructed object. The implementation of the method execute for different cases can be modularized within declarations of Intersect for different types of shapes. In principle, we encode multi-dispatch of methods by a single dispatch over a more sophisticated subtype relation. Method execute takes only one parameter, i.e., an instance of Intersect, but is dispatched not by the class of the instance, but by the types of the fields of that class.

Although multimethods can be encoded by dependent classes, this introduces certain syntactical overhead. Therefore, in our DepJ language we support direct declarations of multimethods. The listing of the left-hand side of Fig. 4.8 is also valid DepJ program and provides analogous dispatch semantics as the listing on the right. The relation between nested and external declarations of classes and methods will be discussed in more detail in Sec. 4.3.1.

```
/* Framework */
abstract class Renderer(GraphicalEditor editor) { }

abstract void paint(Graphic(GraphicalEditor) obj, Renderer(obj.editor) r, ...);

/* Framework instance for circuit editor */
class SchematicRenderer(CircuitEditor editor) extends Renderer { }
class PhysicalRenderer(CircuitEditor editor) extends Renderer { }

void paint(WireGraphic(Wire,CircuitEditor) obj, SchematicRenderer(obj.editor) r, ...) { ... }
void paint(WireGraphic(Wire,CircuitEditor) obj, PhysicalRenderer(obj.editor) r, ...) { ... }
void paint(LogicCompGraphic(LED,CircuitEditor) obj, SchematicRenderer(obj.editor) r, ...) { ... }
void paint(LogicCompGraphic(LED,CircuitEditor) obj, PhysicalRenderer(obj.editor) r, ...) { ... }
void paint(LogicCompGraphic(Subcircuit,CircuitEditor) obj, SchematicRenderer(obj.editor) r, ...) { ... }
void paint(LogicCompGraphic(Subcircuit,CircuitEditor) obj, PhysicalRenderer(obj.editor) r, ...) { ... }
...

/* Framework instance for UML Diagram Editor */
class UMLDiagramEditor extends GraphicalEditor { ... }
class UMLClassNode(UMLDiagramEditor editor) extends GraphicalNode { ... }
...
class UMLOverviewRenderer(UMLDiagramEditor editor) extends Renderer { }
...
void paint(UMLClassNode(UMLDiagramEditor) obj, UMLOverviewRenderer(obj.editor) r, ...) { ... }
...
```

Figure 4.9: paint as a double-dispatched method

4.2.5.2 Enhancing Method Dispatch

Dependent classes enhance multimethods in two ways. First, the type-system of dependent classes supports describing more precise types of objects, defining the types of their fields and relating objects from the same family. The more precise object types can be used for a more precise typing of multimethods and so constraining the combinations of concrete parameters, for which the method needs to be implemented. Second, the more expressive types and the subtype relation between them also extend the expressivity of dynamic dispatch, e.g., we can dispatch a method not only by the objects given as their immediate parameters, but also by the values of the fields of these objects.

For illustration, recall the multimethod paint, introduced in Sec. 2.5.4. The implementation of paint depends on the type of the graphical object and on the type of the selected rendering style. Although multi-dispatch of the method made it possible to modularize its implementation with respect to both variation points, we were unable to specify that the method can be called only on pairs of graphical objects and rendering styles of the same framework instance, and only these cases need to be implemented.

With dependent classes we can define the type of the paint method more precisely. Recall, that in Fig. 4.3 we defined Graphic, the base class of graphical objects, as a dependent class parameterized by a graphical editor. Analogously, on line 2 of Fig. 4.9 we also declare the class Renderer, the base class of different renderers, as a dependent class of a graphical editor. Now by means of dependent typing we can define a precise signature of the paint method on line 4: the method takes any graphic objects as the first parameter and a renderer from the same editor as a second parameter.

Constraints between the parameters of multimethods imposed by dependent typing also affect completeness checking of the method. The method needs to be implemented not for arbitrary combinations of instances of Graphic and Renderer, but only for the ones declared for the same concrete framework instance. An example of such a framework instance is class CircuitEditor, which is declared as a concrete subclass of GraphicalEditor in Fig. 4.2. The concrete graphical objects of CircuitEditor are instances of WireGraphic and LogicCompGraphic, also declared in Fig. 4.2. Lines 7-8 of Fig. 4.9 declare two concrete renderer classes for CircuitEditor: SchematicRenderer and PhysicalRenderer. Now we must implement paint for pairs of concrete graphical objects and concrete renderers or CircuitEditor, which is done on lines 10-16 of Fig. 4.9.

Analogously, we must implement the paint method for combinations of concrete graphical object and renderer classes of other framework instances, e.g., for UML diagram editor, as shown on lines 19-25. But we don't need to implement the method for combinations of graphical objects and renderers from different framework instances, e.g., paint does not need to be implemented for pairs of WireGraphic and UMLOverviewRenderer.

The implementation of paint in Fig. 4.9, also illustrates the extended dispatch semantics supported by dependent classes. On lines 12-15, paint is dispatched not only by the classes of its immediate parameters obj and r, but also by the class of obj.comp, i.e., by the logical component.

4.3 Language Design

In Sec. 4.2 we explained the idea of dependent classes and the core of their semantics, such as class dispatch and types depending on multiple families. A formal definition of the core semantics of dependent classes is given in Chapter 5. There are, however, many different ways how these ideas can be implemented in a concrete programming language. In this section, we will discuss various specific language design issues, and present the design decisions made for DepJ, our experimental language implementing dependent classes.

```
 1 class Tree {
 2     Node root;
 3     class Node {
 4         String data;
 5         List<Node> children;
 6         void addChild(String data) {
 7             Node child = new Node();
 8             child.data = data;
 9             children.add(child);
10         }
11     }
12 }
```

```
 1 class Tree {
 2     Node(this) root;
 3 }
 4 class Node(Tree out) {
 5     String data;
 6     List<Node(out)> children;
 7 }
 8 void addChild(Node(Tree) out, String name) {
 9     Node(out.out) child = new Node(out.out);
10     child.data = data;
11     out.children.add(child);
12 }
```

Figure 4.10: Equivalent code in the nested and the parametric styles

4.3.1 Combining Nesting and Parameterization

Dispatch of methods and classes can be expressed by nesting or by parameterization. As was explained in Sec. 4.2, the parametric style provides more flexibility for expressing dispatch and supports more flexible source code organization. Nevertheless, in certain cases the nested style is more intuitive. Namely, when the members are interrelated and are inherent to the abstraction implemented by the class. For example, it is quite natural to declare class Node as a nested class of Tree, because nodes are inherent elements of a tree structure.

Nesting also helps to make the code more concise. Figure 4.10 shows the same example in the nested and the parametric styles. On the left-hand side, class Node nested within Tree, and method addChild is nested within Node. On the right, Node and addChild are externalized by adding the additional field out to pass the enclosing object of the nested style. As we can see, the code in the parametric style side is more verbose. It requires an extra parameter for each externalized class or method. Moreover, this extra parameter needs to be bound explicitly in type declarations (lines 2, 6 and 9 on the right-hand side) and during construction of classes (line 9).

To recap, both nested and parametric styles have their advantages, and in DepJ we decided to support both of them. Nevertheless, the nested style is supported just as a special syntax, which is translated to an equivalent code in the parametric style before further semantic analysis and execution of a program. In Fig. 4.10 we deliberately chose name out for the extra parameter, because this name is also taken by the automatic translation from the nested to the parametric style. As a result, the code on the right-hand side demonstrates the result of automatic translation of the left-hand side performed by the DepJ interpreter.

In general, the translation of the nested style to the parametric style affects the sig-

```
abstract class SelType { }
class SingleCellSel extends SelType { }
...

abstract class Bool { }
class True extends Bool { }
class False extends Bool { }

class Table(SelType sel, Bool colResize, Bool cellColors) extends Widget { ... }
class Table(SingleCellSel sel, Bool colResize, Bool cellColors) { ... }
class Table(SelType sel, True colResize, Bool cellColors) { ... }
...
```

Figure 4.11: Table widget as a dependent class with fixed parameters

natures of class and method declarations, class references in types, method calls and class instantiation expressions. Nested class and method declarations are extended with an additional out parameter of the type corresponding to the signature of the enclosing class. In an instantiation expression, the instantiated class is searched in the context of class declarations enclosing the expression. If a matching class declaration is found in an enclosing class C, the instantiation expression is extended with a parameter out bound to the reference to enclosing instance of C. For example, instantiation of Node on line 7 on the left-hand side of Fig. 4.10, is extended with a binding of its parameter out to out.out, which in this case refers to the enclosing instance of Tree. Method calls and class references in types are extended in an analogous way.

4.3.2 Varying Set of Fields

In the examples of Sec. 4.2, every dependent class is parameterized by a fixed set of parameters, i.e., all declarations of a particular dependent class have the same set of parameters. For example, all declarations of LogicCompGraphic from Fig. 4.2 and Fig. 4.4 have parameters comp and editor. Such design expresses that the class varies with respect to a fixed set of variation points, e.g., LogicCompGraphic varies with respect to its data model object and its owner editor.

Fixing variation points of a class is not optimal in all scenarios. In certain cases we may not know all variation points of a class in advance, and thus we would like to leave this open for new variation points. For instance, in Sec. 2.2.2 and 2.3.2 we used an example of table widgets, which have numerous, often unrelated variations, such as the supported selection model, support for column resizing, different cell coloring models and so on. We can represent each variation point of the table widget by a parameter of the table widget class and modularize implementation specific to certain variants or combinations of the variants in the declarations of the class with corresponding parameter types.

```
 1 class Table extends Widget { ... }
 2
 3 class Table(SelType sel) {
 4    abstract boolean isCellSelected(int row, int col);
 5    ...
 6 }
 7 class Table(SingleCellSel sel) {
 8    boolean isCellSelected(int row, int col) { ... }
 9    ...
10 }
11 ...
12
13 class Table(Bool colResize) { }
14 class Table(True colResize) { ... }
15 class Table(False colResize) { ... }
16
17 class Table(Bool cellColors) { }
18 ...
19
20
21 void test() {
22    Table(sel:SingleCellSel) table = new Table(sel=new SingleCellSel(), colResize = new True());
23 }
```

Figure 4.12: Table widget as a dependent class with variable parameters

For an illustration, Fig. 4.11 demonstrates table widget implemented as a dependent class with three parameters, sel, colResize and cellColors, representing variations of selection type, support for column resizing, and support for cell coloring. The declaration of the class with the most general types of parameters on line 4.11 contains the functionality common to all instances of table widgets. The declaration of Table with parameter sel of type SingleCellSel on line 10 contains functionality specific to tables with single selection, and so on.

Such a design will be elaborated in Sec. 4.4.1.1 and 4.4.1.2, where we will also analyze its advantages and disadvantages. At this point, it is important just to note that by fixing the set of parameters of a class we fix the set of its variation points. Also, the requirement to have the same set of parameters in all declarations of a class, can introduce dependencies between otherwise unrelated variations of the class. For example, cell coloring is completely unrelated to column resizing, but the class declaration implementing cell coloring (line 10, Fig. 4.11) is still parameterized by colResize and, thus, depends on the fact that the widget supports variations of column resizing. Introduction of a new variation point to the table widgets, e.g., support for row resizing, would require introducing a new parameter to all existing declarations of the class, even if the content of these declarations does not need to be changed.

In order to completely decouple unrelated variations of a class and to enable modular extensions of the class with new variation points, DepJ allows declaring a dependent class with different sets of parameters. As shown in Fig. 4.12, we can first declare class Table without any parameters, containing the functionality common to all table widgets. Then we can gradually extend the class with its variation points and their variants. For example, the declaration of Table with parameter sel of type SelType states that the class supports variations of selection models and contains the functionality of all table widgets supporting some form of selection. The functionality of specific selection models is then implemented in declarations of Table with more specific types of sel, e.g., Table with sel of type SingleCellSel implements support for single selection. Analogously, we introduce support for column resizing by declarations of Table with field colResize (lines 13-15) and so on.

Supporting class declarations with different sets of parameters also requires different syntax for parameter binding in types and instantiation expressions. So far we have seen position-based parameter binding, i.e., parameter values (or types) are bound according to their position in the expression (or the type). This form of binding assumes that it is possible to uniquely determine the parameters by their position. This assumption can be in general violated in case of declarations of a class with different sets of parameters.

For this reason, DepJ also supports name-based binding of parameters. In instantiation expressions we specify constructor parameters by a list of pairs of parameter names and parameter values. Analogously, in types we can specify the types of object parameters by a list of pairs of parameter names and parameter types. For example, line 22 of Fig. 4.12 demonstrates instantiation of Table with parameters sel of value new SingleCellSel() and colResize of value new True(). The object is assigned to a variable of type Table(sel:SingleCellSel), which specifies that it is a Table with the parameter sel of type SingleCellSel.

The name-based binding enables partial binding of parameters, i.e., we can specify only the values or types of the parameters relevant to a particular situation. On line 22 we instantiate a table widget with single selection and column resizing. For this instantiation expression, we do not need to know that the class supports other variations by further parameters. Such an instantiation expression can match only the class declarations with the given parameters. Declarations of the class with parameters not listed in the instantiation expression, e.g., cellColors, would not match and would not be inherited by the created object. Analogously, for the variable table at the same line, we specify only the type of its parameter sel. Again, the type matches only the class declarations with the specified fields, i.e., the declarations of Table of lines 1, 3 and 7.

A further advantage of partial parameter binding is that it supports modular extensions of a class with new parameters without affecting existing clients of the class. Existing clients would further specify only the parameters of the class that they know and can remain completely unaware of the new parameters of the class. If variations of class

```
1  abstract class Graphic(GraphicalEditor editor) {
2     List<Handle> getEditHandles() { return resizingHandles(); } ...
3  }
4
5  abstract class ChildGraphic(GraphicalEditor editor, Graphic(editor) parent) extends Graphic { ... }
6
7  abstract class ImmutableGraphic(GraphicalEditor editor) extends Graphic {
8     List<Handle> getEditHandles() { return noHandles(); } ...
9  }
10
11 abstract class ChildGraphic(GraphicalEditor editor, ImmutableGraphic(editor) parent)
12    extends ImmutableGraphic { }
```

Figure 4.13: Dependency of child graphical objects on their parents

parameters were not supported, an alternative solution would be to introduce new class parameters in new subclasses of the class and combine such classes by multiple inheritance. This would lead to explosion of class names, because we would need a new class name for every independently introduced parameter and for every combination of them. In contrast, declarations of the same class are always considered as composable, which is taken into account during completeness and uniqueness checking of methods taking instances of the class as parameters.

4.3.3 Recursive Dependencies

In the examples so far there is a clear distinction between dependent classes and the classes on which they depend, e.g., graphical objects may depend on the editor and the application classes, but not vice versa. This, however, must not always be the case, and a dependent class may depend on its own instances. An interesting application of such recursive dependencies is expressing context dependencies in recursive data structures, e.g., described by the Composite pattern [GHJV95].

For example, in composite graphics the appearance and behavior of nested objects may depend on the type of their enclosing objects. Such dependencies can be supported by the design of our graphical editor as shown in Fig. 4.13. The class ChildGraphic, declared on line 5, is a special subclass of Graphic to represent graphical objects that are children of composites. The class ChildGraphic depends not only on the editor, but also on its parent composite, which is referenced by the parameter parent.

Fig. 4.13 demonstrates recursive propagation of behavior from composites to their children. The class ImmutableGraphic (line 7) implements immutable graphical objects, e.g., ImmutableGraphic overrides getEditHandles to specify that immutable objects do not provide handles for editing. To express that all children of an immutable object are also immutable, independent of their level of nesting, one can refine ChildGraphic for parents of type ImmutableGraphic (line 11). The refined ChildGraphic is declared as a subclass ImmutableGraphic. This declaration has a recursive effect and specifies that all (direct and indirect) children of an immutable object are immutable too, e.g., they would inherit the implementation of getEditHandles from ImmutableGraphic.

4.3.4 Abstract Dependent Classes

Dependent classes can be declared as abstract with the meaning and implications analogous to the abstract virtual classes, presented in Sec. 3.3.4. The classes declared as abstract cannot be directly instantiated. They serve only for the purpose of generalization over concrete classes in type specifications and for defining their commonalities.

Each concrete declaration of a class serves as a constructor and each instantiation expression must match at least one constructor. For example, the concrete definition of LogicCompGraphic in Fig. 4.2, line 3, specifies that LogicCompGraphic can be instantiated for any circuit editor and any logic element. Since dependent classes are instantiated polymorphically, it makes sense to have concrete declarations of a class that do not implement all methods. For instance, in Fig. 4.2, the concrete class LogicCompGraphic(LogicComponent, CircuitEditor) does not implement the method paint inherited from Graphic. This is fine, as long as all methods are implemented for all possible instances of LogicCompGraphic. This is the case, since each instantiation expression of LogicCompGraphic will match at least one of its declarations for some concrete subclass of LogicCompGraphic, and the methods in these declarations are fully implemented.

4.3.5 Dispatch Strategy

When multiple implementations of a method are applicable for certain parameters, method dispatch must decide which of them to execute. Dispatch is *unique* if for all valid method parameters, the dispatch strategy can unambiguously decide which of the matching method implementations should be executed. The general rule is that the method with the most specific parameters is selected. Such a rule is sufficient to uniquely select a method in case of single dispatch and single inheritance.

Ambiguities can, however, occur in languages supporting multiple inheritance or multi-dispatch, i.e., there can be multiple method implementations given certain parameters and none of them is more specific than the others. In case of multiple inheritance, a class can inherit an implementation of a method from multiple parents. In case of multi-dispatch, two matching methods can be incomparable, because one of them is more specific by the first parameter and the other one more specific by the second parameter.

There are in principle two strategies to deal with such ambiguities, known as *symmetric* and *asymmetric* dispatch [CLCM00]. In case of symmetric dispatch the ambiguities are considered as errors, which are detected either statically or at runtime. The strategy of asymmetric dispatch is to define additional rules to decide which of the methods should be selected. For example, the method implementation ambiguities resulting from multiple inheritance can be resolved by linearization of class inheritance hierarchy, presented in Sec. 3.3.1. Ambiguities resulting from multi-dispatch can be resolved by asymmetric treatment of method parameters: if one method is more specific than the other by the first parameter, the types of consecutive parameters are ignored; if the types of the first parameter are equivalent, then the methods are compared by the second parameter, and so on.

In case of dependent classes, dispatch ambiguities can occur both because of multiple inheritance and because of dispatch by multiple parameters, and we need to choose between symmetric and asymmetric dispatch strategies to deal with them. Both strategies have their trade-offs.

The main problem with asymmetric dispatch is to deal with ambiguities resulting from the multiple inheritance including the implicit and explicit inheritance relationships between dependent classes. In order to resolve ambiguities resulting from multiple inheritance automatically, we would need to define a total order between class declarations inherited by an object. In other words, we would need to define a linearization algorithm, analogous to the one for virtual classes defined in Sec. 3.3.2.

In case of the intuition behind the dependent classes, it is, however, difficult to find intuitive rules for linearization. The proposed linearization of virtual classes was based on the intuition to view inheritance between family classes as describing of changes to groups of classes, and mixin composition as the consecutive application of such changes. This intuition is not suitable for dependent classes, because in the context of class dispatch, inheritance is viewed more as a relation for classifying objects rather than a specification of a change. Although we could define ordering rules based on the order of parameters and the order in which inheritance relations are introduced, such ordering would appear rather accidental from the perspective of the developer, hiding rather than solving the problems.

Therefore, in DepJ, we follow a *symmetric* dispatch strategy, which is independent of syntactic ordering of parameters, class declarations and other syntactic structures; method selection is based only on specificity of its parameters. The cases, when there is no unique most specific method matching the given parameters are considered as errors. Note that symmetric dispatch is applied at the level of methods and not at the level of classes, i.e., it is not required that for every created object, there is a most specific matching class declaration. It is sufficient to ensure that the created object does not inherit ambiguous method implementations. Symmetric dispatch in the context of dependent classes, however, poses challenges on method completeness and uniqueness analysis, which will be discussed in Sec. 4.3.7.

4.3.6 Method Refinements

The advantage of symmetric method dispatch is that it is based on a simple and intuitive semantics, which helps to avoid unexpected errors. On the other hand, symmetric dispatch does not provide the same flexibly for describing composable extensions of methods, as the mixin composition of virtual classes discussed in Sec. 3.3.2.

Method dispatch resulting from mixin composition is a form of asymetric dispatch, because it automatically resolves method ambiguities caused by multiple inheritance through asymetric treatment of superclasses. The linearization of the inheritance graph determines a total order between class declarations inherited by a certain object, which also defines a total order between inherited implementations of every method. This order determines the most specific method, and its linearity guarantees that the most-specific method can be always uniquely selected. Besides that, the total order allows defining a *super-call* to a method as the call to the next implementation of the method according the order.

Consistent usage of super calls in all overridings of a method produces a chain of calls through all implementations of the method inherited by an object, and results in their effective composition. Such a technique makes it possible to decompose variations of classes at a fine level of granularity: we can decompose variations of individual methods by moving parts of them to subclasses and connecting them with each other through super calls. For example, in Fig. 3.6 we used mixin composition with super calls to decompose the implementation of method `draw` by different variations of figures. We decomposed the drawing functionality of a figure by the variations of its shape (circle, rectangle, etc.) as well as by variations of its properties (colored, with text, etc.). There are a lot of situations in practice where such decomposition of methods is useful. In particular it is useful for methods implementing initialization of objects, reactions to various external events or state changes, methods returning composable result values, and so on.

The composition of methods through polymorphic super calls relies on the linearization of the inheritance graph of a class, which is not available for dependent classes in DepJ. Without linearization we can support only static (super) calls to specific implementations of a method, which is also problematic in DepJ, because it is difficult to identify specific implementations of a method in the context of more advanced forms of dispatch. For example, specific implementations of method paint from Fig. 4.9 could be identified only by their relatively complicated signature.

Therefore, in addition to symmetric dispatch, DepJ also introduces special syntax for so-called method refinements. A method declared with the keyword refine is considered as a refinement of the method with the same name. A method can contain multiple refinements for different types of parameters. At a method call, a list of refinements of the method matching the given parameter values is collected, and the first method of the list is executed. A method refinement can include super calls, denoted by the keyword super, to the next method refinement from the list. In the last method refinement from the list, the super call refers to the method implementation selected by the symmetric dispatch.

The symmetric dispatch considers only normal method implementations, i.e., the ones declared without keyword refine. Method refinements are not considered as self-sufficient method implementations, but only as decorations of the method implementation selected by the symmetric dispatch. Thus, the type-checker ensures that symmetric dispatch will be successful at each method call, independently of the refinements for that method call.

Figure 4.14 illustrates method refinements by an example of event handling in a table widget. Method mousePressed declared in Widget (line 2) is called whenever a mouse is pressed on a widget. The default implementation of the method in Widget sets the focus on the widget. The implementation of the method for tables depends on their variations. Such dependencies are modularized by decomposing the method into multiple refinements. The refinement of mousePressed for tables with single cell selection (line 11), selects the cell containing the mouse pointer. Analogously, in tables with single row selection, method mousePressed selects the row containing the mouse (line 21). In tables supporting column resizing, mousePressed is refined to start column resizing under appropriate conditions (line 31), and in tables supporting row resizing, mousePressed may begin resizing of rows (line 41).

Each refinement of mousePressed makes a super call at the beginning of the body. As a result, the implementation of mousePressed for a specific table widget is composed of all refinements of the method inherited by the object. For example, a call to mousePressed on a table with single row selection and column resizing would trigger execution of the method refinements of lines 21 and 31 and subsequently the default implementation (line 2).

Unlike mixin composition semantics, the order of executing matching method refinements is not completely determined. The matching refinements are sorted only with respect to specificity of their parameter types. Such sorting criterion defines only partial order

```
abstract class Widget {
    void mousePressed(Point pt) {
        setFocus();
    }
    ...
}

class Table extends Widget { ... }

class Table(SingleCellSel sel) {
    refine void mousePressed(Point pt) {
        super;
        if (withinCell(pt) && canBeSelected(cellAtPoint(pt))) {
            selectCell(cellAtPoint(pt))
        }
    }
    ...
}

class Table(SingleRowSel sel) {
    refine void mousePressed(Point pt) {
        super;
        if (withinCell(pt) && canBeSelected(rowAtPoint(pt)) {
            selectRow(rowAtPoint(pt))
        }
    }
    ...
}

class Table(True colResize) {
    refine void mousePressed(Point pt) {
        super;
        if (withinColumnHeader(pt) && betweenColumns(pt)) {
            startColumnResizing(columnLeftFromPoint(pt));
        }
    }
    ...
}

class Table(True rowResize) {
    refine void mousePressed(Point pt) {
        super;
        if (withinRowHeader(pt) && betweenRows(pt)) {
            startRowResizing(rowAbovePoint(pt));
        }
    }
    ...
}

...
```

Figure 4.14: Modularizing the mouse click handler with respect to variations of table widgets

between method refinements, though. For example, the order of execution of mousePressed from lines 21 and 31 is undetermined, because none of them is more specific than the other. Thus, the usage of method refinements is appropriate only in the cases when the order of their execution is not important.

4.3.7 Checking Method Completeness and Uniqueness

Type-safety in the presence of abstract methods and symmetric dispatch relies on completeness and uniqueness. Completeness requires that a method is implemented for all valid combinations of parameter values. Uniqueness requires that for a given parameter set, there is an implementation that is more specific than the others. In languages with single dispatch, completeness is ensured by checking that every abstract method is implemented in all concrete subclasses. Uniqueness is checked only if multiple inheritance is supported. In that case, it is again sufficient to check every class for a possible ambiguity of its methods. Completeness of a multimethod is ensured by checking that the method is implemented for all combinations of concrete implementations of parameter types. The same combinations of classes are also considered for checking method uniqueness.

If we want to apply an analogous strategy for checking a method in the presence of dependent classes, we must take into account that the signatures of a method declaration and its implementations may contain types that specify not only the classes of their parameters, but also the types of their (direct and indirect) fields. This means that the method must be checked for all applicable combinations of constructors of its parameters and their fields.

For example, completeness checking of the paint method from Fig. 4.9 defined for a pair of instances of Graphic and Renderer, must collect their concrete subclasses, such as WireGraphic and LogicCompGraphic for Graphic or SchematicRenderer and PhysicalRenderer for Renderer. It is, however, not sufficient to check that the method is implemented for combinations of these classes. We may also need to analyze what concrete parameters these classes can take. For example, although on line 3 of Fig. 4.2 we declared LogicCompGraphic with parameter types LogicComponent and CircuitEditor, we do not implement paint for a graphical object of type LogicCompGraphic(LogicComponent, CircuitEditor) in Fig. 4.9: Since LogicComponent is an abstract class it is sufficient to implement the method for its concrete subclasses, such as LED and Circuit. Thus, the completeness analysis must combine the constructor LogicCompGraphic(LogicComponent, CircuitEditor) with all possible constructors of its parameters.

The set of combinations of constructors cannot be constructed in a brute force way for two reasons. First, constructors can be combined recursively, as discussed in Sec. 4.3.3, which can lead to an infinite number of possible combinations. Second, the set can be constrained by path-dependent types; the latter limit the scope of completeness checks to instances of the same framework. For example, the method paint from Fig. 4.9, needs to be implemented only for concrete graphical objects and renderers of the same framework instance.

Therefore, method checking algorithms must work with finite sets of types approximating the possibly infinite set of concrete parameter types. In Sec. 5.5.4 we will show how such sets can be constructed and define algorithms for checking method completeness and uniqueness in terms of these sets.

4.3.8 Modules

DepJ programs can be flexibly decomposed into modules. Declarations of a class or a method can be distributed over multiple modules. Each module starts with a header specifying the name of the module and the list of modules, on which it depends. For example, Fig. 4.15 shows a possible decomposition of the table widget example, based on the design of Fig. 4.12. Declarations of Table are distributed into multiple modules, which implement different variations of the table widget and can be used independently from each other. The module TableCore implements the base functionality of table widgets, independent from specific variations, while modules TableSel and TableColResize contain declarations of table widget implementing support for selection and column resizing respectively.

An execution of the DepJ interpreter is parameterized by a *main module*. The module and its dependencies are loaded and type-checked. Then the method with the name main is searched within the loaded modules and executed. Only the declarations of classes and methods from the modules that are directly or indirectly used by the main module are considered by the dynamic dispatch.

Each module is checked in isolation with respect to its dependencies. Only class and method declarations visible within the module, i.e., declared within the module and the used modules, are considered for type checking of the module. In particular, each method call must match a method declaration visible within the module, and each instantiation expression must match a visible class constructor.

As explained in Sec. 4.2.3, the interface of an expression typed by a dependent class is determined by a static dispatch, collecting the declarations of a class matching the statically known types of the class parameters. In the presence of modules, the static dispatch collects only class declarations visible in the module, which ensures that the code depends only in the definitions visible in the modules. By reducing the set of

```
module Widget;

abstract class Widget {
    abstract void paint(Graphics g);
    ...
}
```

```
module TableCore uses Widget

class Table extends Widget {
    void paint(Graphics g) { ... }
    ...
}
```

```
module TableSel uses TableCore

abstract class SelType { }
class SingleSel extends SelType { }
...

class Table(SelType sel) {
    abstract boolean isCellSelected(int row, int col);
    ...
}

class Table(SingleCellSel sel) {
    boolean isCellSelected(int row, int col) { ... }
    ...
}
...
```

```
module TableColResize uses TableCore, Bool

class Table(Bool colResize) { }
class Table(True colResize) { ... }
class Table(False colResize) { ... }
```

```
module TableTest uses TableSel, TableColResize

void main() {
    Table(sel:SingleCellSel) table = new Table(sel=new SingleCellSel(), colResize = new True());
}
```

Figure 4.15: Decomposition of table widget functionality into modules

collected class declarations we do not compromise type-safety, because in this way the type-checker can become only more conservative.

On the contrary, dynamic dispatch of a method call or class instantiation expression considers not only the declarations visible within the module where the expression is declared, but also the declarations visible within the main module. In this way, the set of declarations considered by a dynamic dispatch of an expression can be larger than the set of the declarations available for type checking of the expression.

The module system of DepJ guarantees monotonicity of expression typing: an expression that is well-typed within a module M is also well-typed within a module N directly or indirectly using module M; the type of an expression computed in the context of the module M is preserved in the module N. The monotonicity relies on the fact that all declarations visible in M are also visible in N. Thus every method call, constructor call, or field access that is valid in M will also be valid in N.

Method completeness and uniqueness checking is also performed with respect to the declarations visible within a module. A module can be marked as abstract when it is not completely implemented with respect to the visible declarations. In this case, completeness and uniqueness checking is skipped. An abstract module cannot be used as the main module for program execution.

Method completeness and uniqueness checking is not monotonic, though: if method completeness and uniqueness is guaranteed with respect to the declarations visible in the module M, these properties are not necessarily preserved within the dependent module N. N can introduce new cases for an abstract method declared in M, which can be caused by new class declarations declared in N or imported from other modules. Thus, completeness and uniqueness of method implementations must be rechecked within the context of each module. Nevertheless, each module declared as non-abstract must be self-consistent.

The modularity of method completeness and uniqueness checking in DepJ is comparable to the modularity of type-checking of classes in languages with multiple inheritance. Each superclass of a class A must be self-consistent, but there can be conflicting method declarations inherited from different superclasses, thus the inherited declarations must be rechecked for consistency in A, even if they were already checked in the context where they were declared.

An even closer analogy exists between the modules of DepJ and the extensible modules modeled by family classes, which were explained in Sec. 3.4.1.3. A composition of two complete family classes may produce an incomplete family class, thus, completeness of every family class must be rechecked with respect to all inherited virtual classes and their members. An example of such a situation was given in Fig. 3.14.

4.3.9 Second-Order Dependent Classes

Since dependent classes are parameterized classes, it raises the question how they relate to generics [AFM97, BOSW98, BML97, CS98, OW97]. Both dependent classes and generics have certain advantages and cannot completely encode each other.

First, unlike generics, dependent classes are parameterized by objects and not by types. Although the parameters of dependent classes can also be used in types, they are interpreted as singleton types, usually used to represent membership of objects in a family. On one hand, the singleton types refer to individual objects and thus cannot be used as a replacement for generic parameters, which abstract from arbitrary object types. On the other hand, generics cannot express membership in families represented by objects, because it does not provide any ways to refer to objects in types.

Second, the interface and the implementation of a dependent class can depend on the type of its parameters, which is expressed by giving different declarations of a class for different parameter types. A generic class has a single declaration, which means that its definition is identical for all possible parameter bindings.

Finally, dependent classes and generic classes differ by the binding time of their parameters: instances of dependent classes are bound to values computed at runtime, while the parameters of generic classes must be bound statically, and only a completely parameterized generic class can be instantiated. The parameters bound to an object of a dependent class are available as dynamically accessible fields of the object. In this sense, dependent classes are closer to first-class genericity [CS98], which also makes the class parameters available for dynamic use, in particular for dynamic type checks and type casting.

The advantages of generics and dependent classes can be combined, by enabling the parameterization of dependent classes by types. In order to preserve all advantages of dependent classes, in particular the dynamic dispatch, the types must be made available as runtime values, which can be referenced by variables and passed polymorphically during the instantiation of dependent classes. The types of such variables and dependent class parameters are in principle types of types, which we will call *second-order types*[3]. The types of objects, introduced in Sec. 4.2.3, will be considered then as *first-order types*. Analogously, we will refer to the dependent classes parameterized and dispatched by types, as *second-order dependent classes*.

The benefit of second-order dependent classes is twofold. First, they make it possible to combine generic types with path-dependent types, and hence to combine instances of generic classes and dependent classes in a type-safe manner. This makes it possible to use

[3]We decided not use the term *kind*, because it is associated to a specific form of second-order typing in functional languages.

conventional collection classes such as lists and arrays for storing instances of dependent classes. Second, by replacing generic classes by second-order dependent classes, we enable their specialization for specific types of parameters. For example, we can specify that lists on comparable objects additionally support sorting, which is not available for lists of arbitrary objects.

In Sec. 4.3.9.1 we will take a look at the second-order types supported in DepJ and relationships between them. In Sec. 4.3.9.2 we will take a look at an example of dispatch by types and analyze its benefits. In Sec. 4.3.9.3 we will discuss combination of parameterization by types and dependent typing and analyze the limitations of such combination in DepJ.

4.3.9.1 Second-Order Types

Parameterization of classes by types in DepJ is supported by allowing types as parameters of classes. The parameters taking types as values are typed by types of types, which we call *second-order types*. The language supports three kinds of second-order types corresponding to different sets of first-order types: the set of all first-order types, a singleton set containing only the given type, and a set of types limited by an upper bound, i.e., including all subtypes of the given bound.[4]

For illustration, consider the implementation of a list collection shown in Fig. 4.16. The given implementation is based on the typical encoding of lists in functional languages: A list is an empty list (Nil) or a list node consisting of a pair of a value and a list (Cons val list). The listings on the left and on the right contain equivalent implementations of list, but in a slightly different syntax.

At first, let's focus on the listing on the left of Fig. 4.16, which is based on the primary syntax of dependent classes. The set of all (first-order) types is denoted by keyword type. For example, on line 1 the class List is declared with parameter T of type type, which means the class can be parameterized by any first-order type. The same holds for its subclasses, Nil (line 3) and Cons (line 5).[5]

A type of the form $[<: t]$ represents the set of all subtypes of the type t, while a type of the form $[=t]$ represents the set containing only the type t. For example, on line 15 we declare variable lst1 with type List([=String]), which means that the parameter T of the list is known to be of type [=String], i.e., a singleton second-order type containing

[4]The language can be extended with further forms of second-order types, e.g., also supporting lower bounds, but in the current implementation we limited ourselves to the most common cases, which are sufficient validating the general concept.

[5]We follow the naming convention to capitalize variables and fields that take types as values. This convention is not imposed by the language.

```
1  abstract class List(type T) { }
2
3  class Nil(type T) extends List { }
4
5  class Cons(type T) extends List {
6      T^ head;
7      List(T) tail;
8  }
9
10 List(list.T) add(List(type) list, list.T^ val) {
11     return new Cons(list.T).init(val, list);
12 }
13
14 List(type) test() {
15     List([=String]) lst1 = new Nil([String]);
16     List([<:String]) lst2 = lst1;
17     lst1 = lst1.add("a").add("b").add("c");
18     // lst2.add("d"); // typing error
19     return lst1;
20 }
```

```
1  abstract class List(type T) { }
2
3  class Nil(type T) extends List { }
4
5  class Cons(type T) extends List {
6      T^ head;
7      List<T> tail;
8  }
9
10 List<list.T> add(List<?> list, list.T^ val) {
11     return new Cons<list.T>.init(val, list);
12 }
13
14 List<?> test() {
15     List<String> lst1 = new Nil<String>;
16     List<? <: String> lst2 = lst1;
17     lst1 = lst1.add("a").add("b").add("c");
18     // lst2.add("d"); // typing error
19     return lst1;
20 }
```

Figure 4.16: A second-order dependent class for lists

String as the only instance. In other words, we declare that lst1.T is equal to the type String. In contrast, type List([<:String]) declares that the precise value of its parameter T is not statically known – we just know that it is a subtype of String. Thus, we cannot add string values to a variable of this type on line 18.

Type [=String] is a subtype of [<:String], because the set of instances of [=String] consisting of only String is a subset of the set of instances of [<:String], i.e., the set of subtypes of String. By analogous consideration, although String is a subtype of Object, [=String] is not a subtype of [=Object], but [<:String] is a subtype of [<:Object]. According to the general subtyping rules of dependent classes (See Sec. 5.2.5), List(t) is a subtype of List(t'), iff t is a subtype of t'. Consequently, List([=String]) is a subtype of List([<:String]) and List([<:Object]), but not a subtype of List([=Object]). For example, the assignment on line 16 is type-safe.

Second-order types also produce *second-order paths*, i.e., paths pointing to values that are not objects, but types. So far we used paths to represent singleton types with the object referenced by the path as the only instance. The same usage of paths is also possible for second-order paths: A second-order path represents a second-order singleton type having the type referenced by the path as the only instance. For example, on line 7 we declare the variable tail of type List(T), which is the same as List(this.T). Hence, the type of tail.T is this.T, which means that tail contains the same type of elements as this.

Another way to use a second-order path is to declare instances of the type referenced by the path. To differentiate between the two usages of a path, we use a different syntax for the latter case. Namely, we write $p\hat{}$ to declare instances of the type referenced by the path p. This means that in a list we must write T^ or this.T^ to declare types of variables referencing elements of the list, e.g., the type of variable head declared on line 6. Declaring head with type T instead would mean that it is a variable referencing a type, and its value is always equal to the value of the field T.

Fields and variables typed by second-order types can be assigned values representing types. A type value can be constructed by a special expression $[t]$, where t is any type valid in the context. For example, on line 15 class Nil is instantiated as an empty list of strings by binding its parameter T to the expression [String], which evaluates to type String. If a type depends on identifiers, they are substituted during evaluation of expression. Thus, type values do not depend on this or local variables. Instead, they may contain references to the corresponding objects in the heap.

Type values can be also a result of evaluating any other expression, e.g., a result of an access to a field referencing a type. For example, in the instantiation expression of line 11 the parameter T of the created object is bound to the result of evaluation of list.T. If the parameter list of method add is a list of strings, then list.T will evaluate to String and the created object would be again a list of strings.

The listing on the right of Fig. 4.16 shows an alternative syntax supported in DepJ, which is more intuitive to Java developers, and therefore we will use it in most of examples. While in the original syntax we specify all class parameters within simple brackets, alternatively we can specify type parameters in angled brackets using Java-like syntax. In the instantiation expressions, e.g., on lines 11 and 15 of the listing on the right, we can use angled brackets to bind type parameters. We can also use angled brackets to specify the types of type parameters of a class using a Java-like syntax: symbol ? is used instead of type, syntax ? $<: t$ replaces $[<: t]$, and instead of writing $[= t]$ we simply give the type t. For example, on line 15, we declare a list of strings using List<String>; type List<? <: String> on line 16 refers to a list, whose element type is at least String; and the return type of test on line 14 is List<?>, i.e., an arbitrary list.

The example demonstrates that second-order dependent classes can be used like generic classes in Java with similar capabilities. Note that these capabilities intensively reuse the base syntax and semantics of dependent classes. The structure of generic types is based on the structure of class types, which already support specification of the types of parameters. References to generic parameters are encoded by path types. We also reuse path equivalence and typing (cf. Sec. 5.2.3) for determining equivalence and upper bounds of the generic parameters. The existing subtyping rules of dependent classes (cf. Sec. 5.2.5) can be used to compare generic types with each other and with the generic parameters.

```
 1 abstract class Serializable {
 2     abstract void serialize(Stream output);
 3 }
 4
 5 abstract class List([<:Serializable] T) extends Serializable { }
 6
 7 class Nil([<:Serializable] T) {
 8     void serialize(Stream output) {
 9         output.writeString("[Nil]");
10     }
11 }
12
13 class Cons([<:Serializable] T) {
14     void serialize(Stream output) {
15         output.writeString("[Cons]");
16         head.serialize(output);
17         tail.serialize(output);
18     }
19 }
```

Figure 4.17: Serialization of lists with serializable elements

4.3.9.2 Dispatch by Types

Second-order dependent classes not only encode generic types, but also enable static and dynamic dispatch by the type parameters. A generic class can be specialized for specific types of its type parameters. For example, the generic definition of a list from Fig. 4.16 can be specialized for specific types of list elements. Fig. 4.17 shows the definition of serialization for lists with serializable elements. Serializable objects are instances of an abstract class Serializable declared on line 1 and implement its method serialize. On line 5 of Fig. 4.17 we refine the generic declaration of List for the case when its elements are instances of Serializable. In this case, List is declared to be a subclass of Serializable and its concrete subclasses must implement serialize. In particular, implementations serialize are given for corresponding refinements of Nil and Cons (lines 8 and 14).

The type system determines that the variables of class Cons, head and tail, are both instances of Serializable, and thus we can safely call method serialize on them on lines 16 and 17. Since tail is of type List(T) (See declaration on line 7 of Fig. 4.16), its inheritance from Serializable is determined by the static dispatch of class List by the statically known type of T in this context, which is [<:Serializable].

Dynamic dispatch of classes by type parameters makes it possible to instantiate them without static knowledge of their type parameters. For example, lines 1-9 of Fig. 4.18 show a method concatenating two lists. The method is implemented for arbitrary lists assuming only that the lists work with the same types of elements. The result of the method is again a list with the same type of elements as the argument lists. Thus,

```
 1 abstract List<l1.T> concat(List<?> l1, List<l1.T> l2);
 2
 3 List<l1.T> concat(Nil<?> l1, List<l1.T> l2) {
 4    return l2;
 5 }
 6
 7 List<l1.T> concat(Cons<?> l1, List<l1.T> l2) {
 8    return new Cons<l1.T>.init(l1.head, concat(l1.tail, l2));
 9 }
10
11 class Person extends Serializable {
12    void serialize(Stream output) {
13       output.writeString("Person");
14    }
15 }
16
17 void testSerialize(Stream output) {
18    List<Serializable> list1 = new Nil<Person>.add(new Person());
19    List<Serializable> list2 = new Nil<Person>.add(new Person());
20    concat(list1, list2).serialize(output);
21 }
```

Figure 4.18: Generic list concatenation function, instantiating appropriate lists by means of dynamic dispatch

concatenation of two lists with serializable elements on line 20 is again a list with serializable elements, and also an instance of Serializable. The call of concat on line 20 creates a serializable list due to the dynamic dispatch of the new expression on line 8. The class declarations inherited by the created object are determined by the dynamic value of l1.T. Hence if it evaluates to Serializable or a subtype thereof, the constructed object inherits declarations of Cons and List for the element type [<:Serializable].

In a design with Java generics, we would have to define explicit subclasses of Nil and Cons constraining the element type to a subclass of Serializable and implementing the interface Serializable. The problem with explicit subclasses compared to specialization by parameter types is that the generic functionality of lists would ignore the specialization. Although an analogous concat method in Java could be called with serializable lists of parameters, it would still instantiate the class Cons and return a list without support for serialization. We could solve the instantiation problem by passing a factory object that instantiates the correct list classes as an additional parameter to concat. Yet, the return type of concat would be still a simple list without support for serialization, and the call to serialize on line 20 would require an explicit type cast to the Serializable interface.

```
class Map(type K, type V) {
   class Entry {
      K^ key; V^ val;
   }
   List<Entry> entries;
   void put(K^ key, V^ val) { ... }
   V^ get(K^ key) { ... }
   ...
}
```

Figure 4.19: Simple implementation of a map collection in DepJ

4.3.9.3 Parameterization by Dependent Types and its Limitations

As was demonstrated in this section, second-order dependent classes can be used to implement typical generic classes, such as lists and other collections. The advantage of uniform encoding of parameterization of classes by objects and types is that we can freely mix the two kinds of parameterization. The most useful combination scenario is to define of a collection of objects of a certain family via parameterization of collection classes by dependent types. In DepJ a type can depend on this, on its fields (including references to the enclosing objects), on method parameters and on immutable local variables. Such types can be used as parameters to generic classes. For example, on line 6 in Fig. 4.10 we declared a list of nodes of the same tree by the type expression List<Node(out)>. In that type expression we instantiate the generic definition of lists with the dependent type dependent type Node(out), which is in turn based on parameterization of class Node by expression out.

It is, however, difficult to express interdependencies between type parameters of the same class. For illustration, consider a collection implementing a table from keys to values that can be parameterized by the type of the keys and the type of the values. Figure 4.19 shows a typical design of the map collection in DepJ. Class Map is parameterized by the type of its keys K and the type of its values V, and provides methods put and get to associate a value to a key and to get the value associated to the given key. The simplest implementation of a map is based on a list of pairs of keys and values, e.g., using the List class of Fig. 4.16.

As we will see in Sec. 4.4, in certain cases the type of a value assigned to a key may depend on the type of the key. A simple example illustrating the problem is a map taking trees as keys and relating each tree to one of its nodes. In such a map, if we call get method with tree t it should return a node of type Node(t). There is no way to define such a map by parameterization of Map from Fig. 4.19, because there is no way to express a dependency of type V on type K. We can derive type Map<Tree, Node(Tree)>, but such a type allows assigning an arbitrary node to an arbitrary tree.

```
 1 abstract class DependentMap(type K) {
 2    abstract class Entry(K^ key) {
 3    }
 4    List<Entry(K^)> entries;
 5    void putEntry(K^ key, Entry(key) entry) { ... }
 6    Entry(key) getEntry(K^ key) { ... }
 7    ...
 8 }
 9
10 class TreeNodeMap([=Tree] K) extends DependentMap {
11    class Entry(K^ key) {
12       Node(key) val;
13    }
14    void putVal(K^ key, Node(key) val) {
15       Entry(key) entry = new Entry(key);
16       entry.val = val;
17       putEntry(key, entry);
18    }
19    Node(key) getVal(K^ key) {
20       return getEntry(key).val;
21    }
22 }
```

Figure 4.20: Encoding dependency of values on keys in a map structure

This said, a type safe implementation of such a map structure is possible, as shown in Fig. 4.20. The reusable part of such a map is extracted to class DependentMap (lines 1-8). As we can see, the class is parameterized only by the type of keys, and the class of its entries Entry is declared as a dependent class of a key. Instead of assigning a value to a key, the map relates a key to a respective entry.

Concrete subclasses of DependentMap are expected to specialize the type of the key and specify the type of the value to be kept in the map entries. The class TreeNodeMap, defined on lines 10-21, specializes DependentMap for a map from trees to their nodes. It constrains the type of the keys K to Tree and extends the entries of the map with a variable val of type Node(key), which defines the value assigned to the key[6]. The key of each entry is a Tree, and the value of the entry is a node of that tree. We also define methods putVal and getVal, which serve as convenient wrappers of putEntry or getEntry allowing clients to work directly with the values of the map, rather than its entries.

Note that the class Entry is a class depending on two objects. Dependency on the key K^ is declared by explicit parameterization, while dependency on the map class is expressed by nesting. Dependency on the key enables declaring a variable for storing the value with a type depending on the key, such as Node(key). Dependency on the map enables a type-safe access of the new variable on lines 16 and 20.

[6]Types K^ and Tree are equivalent in the context TreeNodeMap, and we could replace all occurrences of K^ to Tree.

As we can see, a type-safe implementation of a map with values depending on keys is possible, but less of its functionality can be reused. A class defining a map with other types of keys and values would differ from TreeNodeMap only by the constraining type of K and the type of values. For example, the implementation of a map from trees to respective lists of their nodes could be obtained by replacing all occurrences of Node(key) by List<Node(key)>.

We are not able to abstract from occurrences of Node(key) on lines 12, 14 and 19, because in all the cases, key is an identifier which is local to a method or an instance of Entry and is not available at the scope of the class representing the map. Making the code of TreeNodeMap reusable, would require parameterization of the class by a type function, i.e., a function taking a key object as a parameter and returning the type of the value corresponding to the key. Extending DepJ with some form of type functions is an interesting topic for future research.

4.4 Variation Management with Dependent Classes

Since dependent classes are a generalization of virtual classes, all designs with virtual classes presented in Sec. 3.4 are also supported by dependent classes. The specific advantage of dependent classes is that they can better express dependencies on multiple variation points. The presentation of this section is divided into three parts: variations on single objects, variations on multiple objects, and framework-specific variation scenarios.

4.4.1 Variations of Individual Objects

With respect to modeling variability of individual objects, dependent classes provide an interesting combination of the advantages of instance variables and inheritance. Like with instance variables, the variations of a dependent class can be explicitly represented by values and can be bound dynamically during instantiation of an object. Like with inheritance and unlike with instance variables, refinements of a dependent class for specific field types can be modularized in separate declarations and are represented in types. For illustration we will use the examples of variations of table widgets, introduced in Sec. 2.2.2.

4.4.1.1 Single Variations

To modularize the variations of a class, the class can be declared as a dependent class, parameterized by its variation points. Figure 4.21 shows how dependent classes can be used to modularize the variations of the table selection type. For representing the variations of the selection type we introduce an abstract class SelType with subclasses for each specific selection type: SingleCellSel for single cell selection, SingleRowSel for single row selection, and so on.

The base functionality of tables is implemented in the declaration of Table with no fields. Further declarations of Table declare the field sel, which specifies the selection type in the table widget. The declaration of Table with sel of type SelType specifies the implementation of any table widget supporting selection. The declarations with more specific types of sel implement tables for specific selection types. The variation declared in this way can be bound during construction of a table widget. Line 48 in Fig. 4.21 shows an instantiation of Table with single cell selection.

Since we can specify fields as part of the types of objects, the types can reflect specific variants bound to the objects. For example, on line 48, the constructed object is assigned to the variable tbl of type Table(sel: SingleCellSel), which states that tbl is a table with field sel of type SingleCellSel. This enables a type-safe call to the operation selectCell, which is available only for tables with single cell selection. If we did not specify any field types for tbl, only the operations of Table without fields would be available.

The design with dependent classes preserves all advantages of class inheritance over instance fields, which were identified in Sec. 2.2.2. Like the design based class inheritance, the design with dependent classes makes it possible to modularize variations, affecting not only the implementations of methods, but also the interface of the class, i.e., the set of available fields and methods. Variation can be reflected by the type of a specific instance of the class by specifying the types of the field representing the variation. Finally, the support for variation does not require any specific glue code unlike the design with helper objects.

An important advantage of variation management with dependent classes over inheritance, is that the selected variant is represented by an explicit value, which can be bound dynamically during object instantiation. For example, the selection mode in application's table widgets can be made a configurable user preference, which can be bound dynamically during creation of application tables. Such a scenario is outlined in Fig. 4.22. The class ApplicationPrefs contains various application preferences, stored in static fields of the class. The table selection mode preference is represented by a field of type SelType on line 2. When instantiating a concrete table in the application we bind its selection mode to the value of the preference field, as shown on line 10.

As we can see, the code instantiating the table is unaware of concrete selection modes and binds the preferred selection mode polymorphically. In a design that modularizes the variations of the table widget by inheritance, we would need a conditional statement to decide which table class should be instantiated, depending on the selected value. Note

```
abstract class SelType { }
class SingleCellSel extends SelType { }
class SingleRowSel extends SelType { }
...

class Table extends Widget {
    TableModel model;
    String getCellText(int row, int col) {
        return model.getCellText(row, col);
    }
    void paintCell(int row, int col, Graphics g) {
        ... getCellText(row, col) ...
    }
    ...
}

class Table(SelType sel) {
    abstract boolean isCellSelected(int row, int col);
    void paintCell(int row, int col, Graphics g) {
        ... if (isCellSelected(row, col)) ...
    }
    ...
}

class Table(SingleCellSel sel) {
    int currRow; int currCol;
    void selectCell(int row, int col) {
        currRow = row; currCol = col;
    }
    boolean isCellSelected(int row, int col) {
        return row == currRow && col == currCol;
    }
    ...
}

class Table(SingleRowSel sel) {
    int currRow;
    void selectRow(int row) { currRow = row; }
    boolean isCellSelected(int row, int col) {
        return row == currRow;
    }
    ...
}

...

void test1() {
    Table(sel:SingleCellSel) tbl = new Table(sel=new SingleCellSel);
    tbl.selectCell(2, 3);
}
```

Figure 4.21: Variants of selection functionality with dependent classes

```
1  class ApplicationPrefs {
2     static SelType tableSelMode;
3     ...
4  }
5
6  class CustomerTable extends Table { ... }
7
8  class UIController {
9     void openCustomerTable() {
10       CustomerTable table = new CustomerTable(sel=ApplicationPrefs.tableSelMode);
11       ...
12    }
13    ...
14 }
```

Figure 4.22: Polymorphic binding of variation with dependent classes

that on line 10, we instantiate not the class Table, but its application specific subclass CustomerTable. Since variations of a dependent class are inherited by its subclasses, the dynamic binding of the selection mode is also available for subclasses. In case of variation binding by conditional statements, this code would have to be repeated for every subclass of Table.

Representation of variants by explicit values has the advantage that it makes it possible to separate the code managing the configuration, from the code depending on the configuration. For example, the user interface for inspecting and changing application preferences is independent of any specific functionality of table widgets. It depends only on the class SelType and its subclasses, but not on the class Table. The design also makes the variation bound to a particular object accessible as the value of the corresponding field after the object is created.

Nevertheless, dependent classes do not resolve all limitations of inheritance.

Like with inheritance and unlike with variation modeling by means of conditional statements not all sophisticated and fine-grained variations can be modularized into separate declarations of dependent classes. The types of the fields of a dependent class declaration can be seen as a condition on the fields that specifies when the declaration matches. This condition is always a conjunction of atomic conditions, which test the types of the fields (or the types of the fields of the fields). All other conditions that, for example, involve disjunction and negation are not supported.

Another limitation of dependent classes is that their fields are immutable. The immutability of class fields is a necessary condition for the possibility to use them in types, and so enable type-safe access to the variation-specific interface. Therefore, a variation point that is represented by class fields must be bound during instantiation of objects

```
class Table(Bool colResize) { }

class Table(True colResize) { ... }

class Table(Bool cellColors) { }

class Table(True cellColors) {
    Color getCellColor(int row, int col) { ... }
    void paintCell(int row, int col, Graphics g) {
        ... getCellColor(row, col) ...
    }
    ...
}

class Table(SelType sel, True cellColors) {
    Color getSelectedCellColor(int row, int col) { ... }
    void paintCell(int row, int col, Graphics g) {
        ... getSelectedCellColor(row, col) ...
    }
    ...
}
...

void test2() {
    Table tbl = new Table(sel=ApplicationPrefs.tableSelMode,
                          colResize=ApplicationPrefs.allowColumnResizing,
                          cellColors=ApplicationPrefs.useColoredCells);
}
```

Figure 4.23: Other table variations with dependent classes

and cannot be changed afterwards. On the contrary, instance variables can be changed during the life-cycle of an object.

4.4.1.2 Combining Multiple Variations

In the previous subsection, we considered modeling a single varying feature of tables, the selection type, with dependent classes. Other variations of the table widgets can be modularized in further declarations of Table with new fields, as demonstrated in Fig. 4.23. For simple optional variations we can use fields of type Bool with subclasses True and False. For example, table widgets with resizable columns can be implemented in the declaration of Table with field colResize of type True, while cell coloring can be introduced in Table with field cellColors of type True.

Interactions between features can be implemented in declarations that depend on types of multiple fields. For example, coloring of selected cells can be implemented in a declaration of Table with fields sel of type TableSel and cellColors of type True. In this way, we can decompose the functionality of Table in the presence of multiple varying features analogously to the decomposition that we achieved with inheritance in Fig. 2.3 and 2.4.

The presented design has the same advantages and limitations that were discussed for individual variations of classes in Sec. 4.4.1.1. The design preserves all advantages of the design with class inheritance, and additionally enjoys the advantages of representing variation by explicit values and their polymorphic binding. On the other hand, expressivity of dispatch is more limited than in case of conditionals. With the dispatch supported by dependent classes we can express dependency of code on individual variants and their intersections. It is not possible to express dependencies on more sophisticated conditions on selected variants. Another limitation is that variation must be bound during construction of an object and cannot be changed afterwards.

The dispatch from the values representing variants to the corresponding implementations of a class is an advantage for the clients of the class. From the perspective of the clients, using the class and its variations is as easy as in the solution with instance variables: Clients specify the desired configuration of the objects by instantiating them with corresponding field values; the corresponding implementation of the object is derived automatically by collecting and combining all declarations of the class that match the given field values. For example, line 25 in Fig. 4.23 illustrates the instantiation of Table with variations bound according to user preferences, managed by the class ApplicationPrefs. The functionality of the instantiated object is derived automatically according to the runtime values received from the preferences.

Compared to multiple inheritance, dependent classes provide a better layer of abstraction between the implementations and the clients of classes with variations. The clients of a dependent class only need to know what fields are available for the configuration of the class, but they are unaware of the concrete implementations of these variations and how they must be composed. The provider of the dependent class can freely add or remove its declarations that specialize it for specific variants or resolve interactions of multiple variants. For example, the client instantiating a table on line 25 does not need to be aware that there is a declaration of Table on line 15 resolving the interaction between selection functionality and individual cell coloring. The provider of the Table can add and remove such declaration without affecting type-safety of the clients.

Another important advantage is that it is automatically checked whether all possible compositions of the declarations of a dependent class are consistent. In particular, it is checked whether all abstract methods are implemented and whether all potential method implementation conflicts are resolved. To enforce such checks in a design based on multiple inheritance, we would need to define all possible compositions as explicit classes, which would lead to an explosion of the number of classes. Such an approach would also be error-prone, because the developer must ensure that all possible combinations are defined. Moreover, he or she must not forget to update the combinations when new variations are introduced to the class during evolution.

The mapping from field values to the corresponding class variations is achieved automatically by the dispatch mechanism of dependent classes. In this way we avoid manual implementation of the dispatch function and its extensibility problems. We also do not need a separate class declaration for each possible combination of independent variations: all declarations of a dependent class matching the constructor parameters are combined automatically.

Name-based binding is used instead of the more conventional position-based binding for the constructor parameters on line 25 of Fig. 4.23. Advantages of name-based binding were discussed in Sec. 4.3.2. From the perspective of variability management, it is especially important that name-based binding enables modular extension with new parameters and, thus, new variation points to a class.

4.4.1.3 Interaction of Inheritance and Helper Objects

In Sec. 3.4.2 we demonstrated how virtual classes help to deal with interactions of variations expressed by inheritance and helper objects. The covariant dependency between an object and its helper can be expressed in a type-safe way by declaring the helper class as a virtual class of the owner class. The problem is that nesting helper classes within the owner classes exposes the helpers and their variations to the clients of the owner classes and complicates extensions with new variations of helpers.

As was shown in Sec. 3.4.2, these problems can be alleviated by introducing an additional level of nesting, but a more natural design can be achieved by expressing dependency of helpers on the owner classes by parameterization instead of nesting. Moreover, we can completely hide the helper classes from the clients of the owner classes by representing the variations of helpers by explicit values.

Figure 4.24 illustrates such a design for the example of dynamic variations of look-and-feel style of widgets, used in Sec. 3.4.2. The visualization functionality is implemented as a class `UI`, which depends on two objects: the widget (field `comp`) and the look-and-feel style (field `lf`). The implementations of the visualizations for specific widget and look-and-feel types can be modularized in declarations of `UI` with corresponding field types.

In principle, the design is very similar to the design with virtual classes of Fig. 3.17 and preserves its advantages, i.e., it allows extending the interface of visualization helpers for specific types of widgets and ensures type-safe interaction between specific widgets and their helpers. The new design is, however, different in two ways.

```
abstract class JComponent {
    UI(this, LookAndFeel) ui;
    void setLF(LookAndFeel lf) { this.ui = new UI(this, lf); }
    void paint(Graphics g) { ui.paint(g); }
    ...
}

class JList extends JComponent {
    ListModel getModel() { ... }
    String getTooltipText(Point pt) {
        int idx = ui.locationToIndex(pt);
        ...
    }
    ...
}

abstract class LookAndFeel { }
class BasicLF extends LookAndFeel { }
class MotifLF extends BasicLF { }

class UI(JComponent comp, LookAndFeel lf) {
    abstract void paint(Graphics g);
    abstract Dimension getPreferredSize(JComponent c);
    ...
}

class UI(JComponent comp, BasicLF lf) {
    void paint(Graphics g) { ... }
    ...
}

class UI(JComponent comp, MotifLF lf) {
    void paint(Graphics g) { ... }
    ...
}

class UI(JList comp, LookAndFeel lf) {
    abstract int locationToIndex(Point pt);
    ...
}

class UI(JList comp, BasicLF lf) {
  int locationToIndex(Point pt) { ... }
  void paint(Graphics g) {
      ... comp.getModel() ...
  }
  ...
}

class UI(JList comp, MotifLF lf) {
    int locationToIndex(Point pt) { ... }
    void paint(Graphics g) { ... }
    ...
}
```

Figure 4.24: Dynamic variation of visualization style with dependent classes

```
1 class Test {
2    void test() {
3       JComponent comp = new JList();
4       comp.setLF(new MotifLF());
5    }
6 }
```

Figure 4.25: Dynamic variation of visualization style with dependent classes

First, the dependency of visualization helpers on widget types is expressed by parameterization rather than by nesting. As a result, the visualization helpers are declared outside the widget classes, and can be declared in separate modules, in this way hiding them from the clients of the widgets. Such syntax also directly supports modular extensions with new types of helper objects, i.e., new look-and-feel styles.

Second, visualization helpers are dispatched by an additional parameter, which is the value representing the selected look-and-feel style. This design decision is based on the idea of representation of variation by explicit values, introduced in Sec. 4.4.1.1. As a result, clients of a class can configure the dynamic variation, without being exposed to the helper classes.

Fig. 4.25 shows an example of client code configuring the look-and-feel style of a widget. On line 4, widget is bound to the Motif look-and-feel style by calling the method setLF of the widget with the value identifying the look-and-feel style. Differently from the design presented on lines 48-53 of Fig. 3.17, the client code does not need to instantiate the helper class directly, and in this way *be exposed to its definition.* Instead, the helper is instantiated within the method setLF on line 3 using double dispatch by the widget type and the look-and-feel style.

Representation of the look-and-feel style variation by explicit values enables propagation a variation binding to other objects. For example, a composite widget needs to propagate the chosen look-and-feel style to its children.

4.4.2 Variation of Multiple Objects

As demonstrated in Sec. 4.2.4, dependent classes can express membership of objects in multiple families that do not need to be hierarchically related. In Sec. 4.4.2.1 we will analyze this design from the perspective of variation at multiple scopes. In Sec. 4.4.1.2, we will demonstrate how the design of modularizing class variations by dispatch presented in Sec. 4.4.1.2 can be extended for a group of objects. Finally, in Sec. 4.4.2.3 we demonstrate how dependent classes remove the limitations of virtual classes for relating dynamic variations of multiple objects, which were discussed in Sec. 3.4.2.3.

4.4.2.1 Variation at Overlapping Scopes

As was demonstrated in Sec. 3.4.1.4, virtual classes can define variations at different scopes, but these scopes must be organized hierarchically, i.e., completely include each other. This is because an instance of a virtual class can be *directly* assigned to one family only, while membership in other families can be expressed only indirectly as families of families. An instance of a dependent class can be directly assigned to multiple families. These families do not need to be hierarchically related, which makes it possible to modularize variations bound at multiple overlapping scopes.

Membership of objects in multiple families was demonstrated in Sec. 4.2.4. In that example, we had variations at the scope of a graphical editor, which affect the objects implementing the editor functionality, and at the scope of a logic component, which affect various objects depending on it. These scopes overlap, because the graphical object of a logic component in a circuit editor belongs to the variation scopes of both the logic component and the editor. The scopes do not include each other, because an editor and a logic component can be created independently from each other.

The advantages of such a design are analogous to the ones discussed for group variation with virtual classes in Sec. 3.4.1.1, i.e., it provides the typical advantages of variation inheritance and resolves the specific problems of the variations involving groups of objects. Unlike virtual classes, dependent classes resolve the problems of group variations at multiple scopes. First, we can express type-safe covariant dependencies of the object on other objects from multiple families. Second, variations of the object with respect to multiple families are automatically combined with each other and with the individual variations of the object. Third, the object can be instantiated independently from the specific group variations by which it is affected.

4.4.2.2 Modeling Group Variations by Family Parameters

In this subsection we will demonstrate, how the advantages of modeling variations by class dispatch discussed in Sec. 4.4.1.1 and 4.4.1.2 can be extended to model variations of multiple objects.

In the designs with virtual classes presented in Sec. 3.4.1 and in the examples of dependent classes of Sec. 4.2 we represented variations by inheritance hierarchies of family classes, e.g., hierarchies of subclasses of BasicMenus and GraphicalEditor. As explained in Sec. 3.4.1.2, combinations of group variations can be expressed by propagating mixin-composition, which is a form of explicit multiple inheritance extended with semantics to compose virtual classes. This design suffers from the problems of variation management with multiple inheritance that were identified in Sec. 2.3.2: explosion of classes covering

```
 1 class Menus {
 2    class MenuItem { ... }
 3    class CascadeMenuItem extends MenuItem {
 4       PopupMenu menu = new PopupMenu();
 5       ...
 6    }
 7    class CheckMenuItem extends MenuItem { ... }
 8    class RadioMenuItem extends MenuItem { ... }
 9    abstract class Menu { ... }
10    class PopupMenu extends Menu { ... }
11    class MenuBar extends Menu { ... }
12 }
13
14 class Menus(Bool withAccel) { }
15 class Menus(True withAccel) {
16    class MenuItem { ... }
17    class CascadeMenuItem {
18       refine boolean processKey(KeyStroke ks) {
19          if (menu.processKey(ks)) {
20             return true;
21          }
22          return super;
23       }
24    }
25    abstract class Menu {
26       boolean processKey(KeyStroke ks) { ... }
27    }
28 }
29
30 class Menus(Bool withML) { }
31 class Menus(True withML) {
32    class MenuItem { ... }
33 }
```

Figure 4.26: Modeling variations of menus with dependent classes

all possible combinations of variants; explosion and error-proneness of code implementing dynamic variation binding by means of conditional statements.

An alternative to modeling variations of groups of objects by explicit inheritance hierarchies of family classes is to model them by parameters of the family class. For example, variations of menu structures can be modeled by parameters of class Menus as shown in Fig. 4.26. The declaration of the class Menus of line 1 contains the base functionality of menus, equivalent to that of BaseMenus from Fig. 3.1. On lines 14 and 30, we declare that Menus can be parameterized by boolean parameters withAccel and withML, specifying that the menu structure can be configured with support for key accelerators and multilingual texts. Then, the declarations of Menus on lines 15 and 31 with the respective parameters of type True contain implementations of support for key accelerators and support for multi-language, which are equivalent to the implementations of the class MenusAccel from Fig. 3.2 and the class MenusML from Fig. 3.3. [7]

The inner classes of Menus, implementing different types of menu items and menus, are dependent classes that take an instance of Menus as a single parameter and are dispatched by its fields, such as withAccel and withML. The nested style allows us to express this dispatch in a more concise way: the out parameter referencing to the enclosing object does not need to be explicitly declared in all inner classes and is implicitly passed in the references between inner classes, e.g., in the type reference and instantiation of PopupMenu on line 4.

The effect of propagating mixin composition, which we used to compose variations of family classes, is replaced by the dispatch semantics of dependent classes. For example, if we consider the instantiation of PopupMenu on line 4, the functionality of the created menu object is determined by the declarations of PopupMenu and its superclasses matching the given out parameter. For example, if this expression is evaluated with out of type Menus(withAccel:True,withML:True), its dispatch would collect the class declarations Menu(Menus) (line 9), PopupMenu(Menus) (line 10) and PopupMenu(Menus(withAcell:True)) (line 25).

Analogously, static dispatch enables type-safe interaction between objects depending on the same variations of the family object. For example, the call to processKey on variable menu on line 19 is enabled by a static dispatch of the variable PopupMenu(out) by the statically known type of out in the context of the call. Since out in this context is of type Menus(withAccel:True), we know that the interface of the variable includes the declaration of PopupMenu from line 25, and thus provides the method processKey.

The presented design has the advantages analogous to the ones discussed in Sec. 4.4.1.1 and 4.4.1.1 with the difference that the variations are bound at the scope of families of objects rather than individual objects. Representation of variants by explicit values enables dynamic binding of variations during object instantiation, and makes it possible to separate the code managing configuration from the code depending on it. In case of multiple variation points, the presented design also avoids explosion of class declarations necessary to describe all combinations variants, but at the same time statically ensures that all supported combinations are consistent.

[7] Since the variations of the example are optional, we represent them by boolean values. Selection from a (possibly open) list of variants can be represented by instances of a specific abstract class with concrete subclasses representing the variants. Such scenario was illustrated by variations of selection type in table widgets in Fig. 4.21.

4.4.2.3 Dynamic Variations

In Sec. 2.4.5 we analyzed how dynamic variations of a group of objects can be modularized by outsourcing them to helper objects. For instance, the dynamic variation of a look-and-feel style in an application can be implemented by a set of helper objects implementing the visualization of the application's widgets in that style. Sec. 3.4.1.5 presented a design with virtual classes that supports type-safe covariant dependencies between the helper objects, which was achieved by grouping them into a family. Such a design is not optimal however, because it does not allow helpers to depend on variations of their owner objects, as discussed in Sec. 3.4.2.3.

The problem is that virtual classes can express membership of an object in one family only and hence support covariant references only to other objects of that family. In Sec. 3.4.1.5 we grouped helpers into families representing the variations that they modularize, e.g., the visualization helpers were grouped by the look-and-feel style that they implement. In Sec. 3.4.2.1, we declared helpers as virtual classes of their owner objects in order to express their dependency on the variations of the owners, e.g., visualization helpers were declared as virtual classes of the widgets that they represent. Thus, we grouped helpers in different ways to solve different variation problems. Yes, we could not express both dependencies at the same time.

This limitation is resolved by dependent classes, because they enable membership of an object in multiple groups and covariant dependencies on other objects of these groups simultaneously. Consequently, a helper object can be assigned to both the family of its owner and the family of the variant it implements. In case of the visualization helpers, we can assign them to both the families of their owner widgets and the families of the look-and-feel styles that they implement.

In fact, such a design was already presented in Fig. 4.24 of Sec. 4.4.1.3, where we discussed the advantages of dependent classes for combining static variations expressed by inheritance with dynamic variations expressed by helper objects. The dispatch of helper objects on the look-and-feel style was introduced for the purpose of dynamic binding of look-and-feel style variation in a widget, but it can also be exploited for expressing covariant dependencies between multiple helper objects implementing the same look-and-feel style.

An example of the interaction between visualization helpers, introduced in Sec. 2.4.5, concerns the propagation of graphical attributes specific to a look-and-feel style from parent widgets to their children. For example, in a look-and-feel style supporting the display of widgets with different textures, the background texture of a widget needs to be propagated to its children. Figure 4.27 shows the implementation of such a propagation based on the design with dependent classes. The declaration of visualization helper class

```
 1 class TexturedLF extends LookAndFeel { }
 2
 3 class UI(JComponent comp, TexturedLF lf) {
 4    Texture backgrTexture;
 5
 6    Texture backgrTexture() {
 7       if (backgrTexture != null) {
 8          return backgrTexture;
 9       }
10       else {
11          return lf.getUI(comp.parent).backgrTexture();
12       }
13    }
14    ...
15 }
```

Figure 4.27: Look-and-feel style with configurable fill textures

UI for the "textured" look-and-feel style on line 3 contains a variable backgrTexture to store the selected background texture and a method with the same name, which returns the value of the variable or the background texture inherited from the parent widget if the variable is null.

The key point in Fig. 4.27 is the access to the background texture of the parent widget on line 11. The visualization helper of the parent widget is obtained by calling getUI with the current look-and-feel style and the parent widget. Implementation of this method is presented in Fig. 4.28, where it is declared as a method of the class LookAndFeel. The signature of the method (line 11) declares that the method takes a widget as a parameter and returns an object of type UI(comp, this), i.e., the visualization helper of the given widget for the enclosing look-and-feel style. Therefore, we can be sure that the call to the method on line 11 of Fig. 4.27 returns an object of type UI(JComponent, TexturedLF), which provides method backgrTexture in its interface.

According to the implementation of class LookAndFeel of Fig. 4.28, a look-and-feel style maintains a table from widgets to the visualization helpers implementing rendering of the widgets in that look-and-feel style. The method getUI takes a widget and returns the visualization helper for that widget from the table if it finds one, otherwise it creates a corresponding helper and saves it in the table. The implementation of the table is based on the design presented in Sec. 4.3.9.3, where we use a combination of parameterization by types with dependent typing, supported by the second-order dependent classes.

A map from widgets to their helpers is necessary, because we cannot keep direct references from widgets to their helpers without violating our design goals. As was explained in Sec. 3.4.1.5, by keeping references from widgets to their visualization helpers and at the same time allowing to change the look-and-feel style during the lifetime of the

```
abstract class LookAndFeel {
    UIMap([=JComponent] key) extends DependentMap {
        class Entry(K^ key) {
            UI(key, this.out.out) val;
        }
        ...
    }

    UIMap<JComponent> uiMap = new UIMap<JComponent>;

    UI(comp, this) getUI(JComponent comp) {
        UI(comp, this) ui = uiMap.getVal(comp);
        if (ui == null) {
            ui = new UI(comp, this);
            uiMap.putVal(comp, ui);
        }
        return ui;
    }
    ...
}
```

Figure 4.28: Look-and-feel style with configurable fill textures based on the design with dependent classes

```
abstract class JComponent {
    UI(this, LookAndFeel) getUI() { return UIManager.style.getUI(this); }
    ...
}

class JList extends JComponent {
    String getTooltipText(Point pt) {
      ... int idx = getUI().locationToIndex(pt); ...
    }
    ...
}

class UI(JList comp, LookAndFeel lf) {
    abstract int locationToIndex(Point pt);
    ...
}
```

Figure 4.29: Interaction between widgets and visualization helpers

widgets, we cannot statically ensure that all related widgets refer to the same look-and-feel style and so support type-safe interaction between their helpers. Thus, for true dynamic customization of look-and-feel, the design presented Fig. 4.24 must be modified so that widgets retrieve their helpers using the getUI method of the currently selected look-and-feel style.

The modified design is shown in Fig. 4.29. In the base class of widgets JComponent we declare method getUI (line 2), which returns the current visualization helper of the widget. Here we assume that the currently selected widget is stored in a static variable UIManager.style, but it could also be retrieved in other ways. The return type of the method is UI(this,LookAndFeel), which ensures that the returned helper implements the interface expected by the widget. For example, on line 8 a list widget can safely call locationToIndex on the object returned by getUI(), because in this context the returned helper is at least an instance of UI(JList,LookAndFeel) and, thus, matches the declaration of UI of line 13, which declares the method.

To summarize, the presented design combines the advantages of the two designs with virtual classes of Sec. 3.4.1.5 and Sec. 3.4.2.1: the design supports type-safe covariant interactions both between a helper and its owner and between multiple helpers implementing the same dynamic variation. Thus, the design provides a type-safe solution for combining dynamic group variations with individual varations of the objects of the group.

4.4.3 Variation of Frameworks

In Sec. 3.4.3, we analyzed advantages of virtual classes for managing variations of frameworks, based on the scenarios and problems discussed in Sec. 2.5. Our running example of a graphical editing framework and its variation scenarios were already used in Sec. 4.2 to introduce the concepts of dependent classes. In this section we will analyze the specific benefits of dependent classes in these scenarios.

Since the problems of combining the variations of a framework with its instances are solved by virtual classes (cf. Sec. 3.4.3.1), there are no obvious further advantages of dependent classes specific in this scenario. It must be pointed out, however, that the design of modeling variations of group of objects by parameters of the family, which was presented in Sec. 4.4.2.2, can be also be applied for expressing variations of a framework with analogous consequences. For example, instead of defining subclasses of GraphicalEditor like CircuitEditorWithMenus from Fig. 4.4 and GraphicalEditorWithLayout from Fig. 4.6 we can extend the class GraphicalEditor with parameters withMenus and withLayout to GraphicalEditor. Again, such design avoids the explosion of classes covering all possible combinations of variants, while still performing automatic consistency checks for all valid combinations. It also provides a dynamic binding mechanism from selected variants to the corresponding implementation of the framework, and so makes an analogous manual dispatch redundant.

The multi-dispatch provided by dependent classes can be exploited in framework instances for combining their dependencies on application-specific variations with the dependencies on the variations of the framework, which is the topic of Sec. 4.4.3.1. Finally, in Sec. 4.4.3.2 we will analyze how the combination of multi-dispatch with dependent typing, can be used for implementing dependencies on multipe variation points of a framework.

4.4.3.1 Dependency on Application Variations

In Sec. 2.5.2, we discussed the problems of expressing dependency of framework instances on application-specific variations. For illustration, we considered the instantiation of the graphical editing framework for implementation of application-specific editors. The objects of the framework instance representing the objects of an application-specific data model depend on both the variations of the editor and the variations of the data model objects. E.g., a graphical object for a logic component depends both on the variations of the logic component and the variations of the editor. In other words, the objects of the framework instance serve as adapters from the application objects to the abstractions expected by the framework. Such adapters depend both on the variations of the adaptees and the variations of the abstractions.

With virtual classes we could directly express variation with respect to one dimension only. The graphical object of a logic component, i.e., the adapter, can either belong to the family of the editor, i.e., the framework, or to the family of the logic component, i.e., the adaptee, but not to both families. As was demonstrated in Sec. 3.4.3.2, if the adapter classes are declared as virtual classes of the framework, it is difficult to express their dependency with respect to the adaptees. There two problems. First, given an adaptee object, we must instantiate an appropriate adapter; second, we need to express the covariant dependency between the type of the adapter and the type of the adaptee. In Sec. 3.4.3.2, we showed a solution for the instantiation problem, which is based on encoding double-dispatch by single-dispatch and introducing an additional level of class nesting to support extensions of application classes with the methods used by the dispatch. The typing problem could not be solved at all.

Dependent classes solve the problems by supporting membership of an adapter both in the family of the framework and in the family of the adaptee. The adapters can be implemented by a dependent class parameterized both by the framework and the adaptee. For example, the graphical objects for logic components can be implemented by a dependent class LogicCompGraphic, parameterized by the editor and the logic component, as was shown in Fig. 4.2 and 4.4.

The parameterization of an adapter by the framework instance provides advantages analogous to the design with virtual classes: the adapter can inherit from an abstract class of the framework and be modularly refined to implement additional requirements of specific framework variations. For example, LogicCompGraphic inherits from a class of the framework NodeGraphic (line 3 of Fig. 4.2), which is a subclass of a more general framework abstraction Graphic. In the variant of the graphical editor supporting context menus, its abstraction Graphic is extended with a new abstract method getContextMenu (line 27 of Fig. 4.3). Due to the dependency of LogicCompGraphic on the editor we can modularly extend the implementation of the adapter with an implementation of getContextMenu (Fig. 4.4) for a circuit editor with context menus.

The parameterization of an adapter by its adaptee expresses the dependency of the adapter on the adaptee, and solves the problems that remained in the design with virtual classes. Instead of creating an explicit inheritance hierarchy of adapters parallel to the hierarchy of adaptees, like it was done in the design with virtual classes of Fig. 3.21, we have one adapter class with multiple refinements for specific types of adaptees. For example, the adapter class LogicCompGraphic is initially declared for the base adaptee class LogicComponent on line 3 of Fig. 4.2 and is further refined for its subclasses, e.g., for LED on line 8 of Fig. 4.2.

The parameterization of adapters by adaptees solves the problem of adapter instantiation. Instantiation of an adapter corresponding to a given adaptee is as easy as simply passing the adaptee object to the constructor of the adapter class. The appropriate implementation of adapter is then automatically determined by dynamic dispatch semantics of dependent classes. For example, LogicCompGraphic is instantiated on demand for the given logic component in the method graphicForComp of the circuit editor (line 5 of Fig. 4.5). Due to polymorphic instantiation, the implementation of graphicForComp and its clients remain independent of the specific types of logic components. Compared to the solution with virtual classes, the solution with dependent classes is simple and concise.

The parameterization of adapters by adaptees also solves the problem of relating their types.

First, in refinements of adapters for specific types of adaptees, the reference from an adapter to its adaptee is automatically specialized, which enables type-safe access to the specific interface of the adaptee. E.g., on line 10 of Fig. 4.2 the implementation of LogicCompGraphic can safely access getValue of the LED component through its adaptee parameter.

```
 1 class CircuitEditor extends GraphicalEditor {
 2    LogicCompGraphicMap([=LogicComponent] key) extends DependentMap {
 3       class Entry(K^ key) {
 4            LogicCompGraphic(key, this.out.out) val;
 5       }
 6       ...
 7    }
 8
 9    LogicCompGraphicMap<LogicComponent> logicAdapters
10       = new LogicCompGraphicMap<LogicComponent>();
11
12    LogicCompGraphic(comp, this) logicCompGraphicFor(LogicComponent comp) {
13       LogicCompGraphic graphic = logicAdapters.getVal(comp);
14       if (graphic == null) {
15          graphic = new LogicCompGraphic(comp, this);
16          logicAdapters.put(graphic);
17       }
18       return graphic;
19    }
20    ...
21 }
```

Figure 4.30: Storing mapping from circuit components to their graphical objects

Second, we can use dependent typing to relate the type of an adapter to the type of the adaptee, and in this way enable type-safe access to the specific functionality of the adapter. For example, the return type of the method graphicForComp on line 2 of Fig. 4.5 is LogicCompGraphic(comp, this), where comp refers to the logic component given as a parameter. As a result, the interface of the returned adapter depends on the type of the given adaptee. For example, since we know that graphicForComp on line 11 of Fig. 4.5 is called at least with an object of type LED, we can safely access method setValueColor, which is available only for the graphical objects of LED components (this method is declared in LogicCompGraphic for LED on line 11 of Fig. 4.2).

In Fig. 4.5, we did not show how the data structure storing the table from adaptees to adapters is implemented. In Java we would implement such a table using some concrete subclass of Map, which can be parameterized by the type of the keys and the type of the values. Generic collections can be encoded by second-order dependent classes, as was demonstrated in Sec. 4.3.9, e.g., Fig. 4.19 shows a simple implementation of Java's Map. Usage of Map in our example would be not type-safe, however, because it would not support expressing the dependency of the values, i.e., the adapters, on the keys, i.e., the adaptees, to which they are assigned.

For a precise typing of the table that avoids type-casts, we need to follow the design presented in Sec. 4.20. Definition of this data structure is shown in Fig. 4.30, where we define a specific subclass of DependentMap from Fig. 4.20 and specify the dependency of the table's values on their keys on line 4: The type of the values stored in the map entries are LogicCompGraphic(key, this.out.out), where key is the key of the entry (which must be a logic component), and this.out.out is a reference to the enclosing instance of CircuitEditor. In this way we define that the map entry maps a logic component, given as a key, to its graphical object in the enclosing editor.

4.4.3.2 Dependency on Multiple Variation Points of a Framework

In Sec. 2.5.4 we analyzed the scenario when functionality of framework instances depends on multiple variation points of the framework. For example, a graphical editor can explicitly support two variation points: variations of graphical objects, represented by subclasses of Graphic, and variations of rendering styles, represented by subclasses of Renderer. The actual rendering functionality depends on both variation points, because we need to implement rendering for concrete pairs of graphical objects and rendering styles. More precisely, the functionality depending on multiple variation points of a framework must be implemented only for combinations of variants of the same framework instance, e.g., rendering must be implemented for graphical objects and rendering styles of the same editor instance only.

In Sec. 2.5.4 we analyzed the implementation of such scenario using multimethods: a multimethod takes the objects representing the variants of the framework as parameters and is dispatched by them. That is, rendering functionality can be implemented by a multimethod dispatched by a graphical object and a rendering style. As was discussed in Sec. 2.5.4, in a simple object-oriented type system we are not able to give a precise type to such a multimethod that ensures that the method (a) is called only with combinations of variants of the same framework instance and (b) is completely implemented for such cases. In Sec. 3.4.3.3, it was demonstrated how the set of combinations can be constrained using virtual classes and dependent typing. Representing variations points of a framework by abstract virtual classes made it possible to relate their instances so that they belong to the same framework instance.

The design with virtual classes encodes multi-dispatch of methods by single dispatch, though. Dependent classes enable a direct combination of multimethods and dependent typing, as was explained in Sec. 4.2.5. In fact, the discussion of that section was illustrated by the example of variations of rendering functionality in a graphical editing framework. The method paint of Fig. 4.9 takes a graphical object and a renderer as parameters and is dispatched by the types of both of them. We use family types for describing the precise type of the method: The signature of the method on line 4 declares that the method takes a graphical object of any graphical editor and a rendering style of the same editor as the graphical object. In this way we ensure that the method is called only with variants of graphical objects and rendering styles.

At the same time we can exploit dispatch by both parameters to modularize implementations of the method for different combinations of variants. Most importantly, we can modularize the implementations of the method for different framework instances. The ability to extend to the method with respect to new cases of both parameters is essential for the possibility to define new instances of the framework, because a framework instance is expected to introduce new variants to both variation points. For example, a concrete instance of a graphical editor framework is expected to define both concrete types of graphical objects and concrete rendering styles.

Completeness checking of the method ensures that the method is fully implemented for all valid combinations of parameters. Such checking is performed incrementally at the scope of each module and its dependencies, as was described in Sec. 4.3.8. To ensure that completeness of the method is checked in a modular way for a particular framework instance, we must ensure that the variants of that framework instance are declared or imported within a single module.

The definition of the dispatch of method paint in Fig. 4.9 is relatively verbose, but it could be made more compact if we combine multi-dispatch with nesting. Figure 4.31 demonstrates a design analogous to the one of Fig. 4.9, but with class and method declarations nested within the classes representing the framework and its instances. Within the framework class GraphicalEditor (lines 1-8) we declare the abstract classes representing the variation points of the framework and the method paint dispatched by these variation points. Within a class instantiating the framework, such as CircuitEditor (lines 10-23), we declare concrete variants of the variation points, i.e., concrete subclasses of Renderer and Graphic, and implementation of the paint method for combinations of these classes.

The resulting program code is more concise, because nesting allows hiding all references to the framework instances, which are then derived automatically. First, all nested class and method declarations are implicitly extended with a parameter out referencing the framework instance. This means that paint also takes a reference to the framework instance as the first parameter, which is passed automatically if the method is called within the scope of a framework class. Second, all types referencing the nested classes are extended with a binding of the out parameter to the enclosing framework instance. For example, the signature of paint declared on line 6 is implicitly extended to paint(GraphicalEditor out, Graphic(out) obj, Renderer(out) r).

Note that the implementation of paint of Fig. 4.31 is actually dispatched by three parameters: the framework instance, the graphical object and the rendering style. Again, the dependent typing relates the graphical object and the rendering style to the framework instance, i.e., the first parameter of the method, and in this way limits the set of parameter combinations for which the method needs to be implemented.

```
abstract class GraphicalEditor {
   abstract class Renderer { ... }
   abstract class Graphic { ... }
   ...

   abstract void paint(Graphic obj, Renderer r, ...);
   ...
}

class CircuitEditor extends GraphicalEditor {
   class SchematicRenderer extends Renderer { }
   class PhysicalRenderer extends Renderer { }

   class WireGraphic(Wire wire) extends ConnectionGraphic { ... }
   class LogicCompGraphic(LogicComponent comp) extends NodeGraphic { ... }
   ...

   void paint(WireGraphic(Wire) obj, SchematicRenderer r, ...) { ... }
   void paint(WireGraphic(Wire) obj, PhysicalRenderer r, ...) { ... }
   void paint(LogicCompGraphic(LED) obj, SchematicRenderer r, ...) { ... }
   void paint(LogicCompGraphic(LED) obj, PhysicalRenderer r, ...) { ... }
   ...
}
...
```

Figure 4.31: Implementing paint as nested multi-dispatched method

4.4.4 Summary

In this section we have investigated the specific advantages of dependent classes for the variation scenarios identified in Chapter 2. We have showed that dependent classes solve most of the problems that remained in the designs with virtual classes, and make some of the latter designs simpler and more intuitive. In particular, we have showed the following contributions of dependent classes to the variation scenarios:

- Dependent classes provide an improvement over multiple inheritance for modeling multi-dimensional variations of an object. They ensure consistency of all possible combinations of the variations and provide a dynamic mapping from configuration options to the corresponding implementations. At the same time they avoid combinatorial explosion of class declarations.

- Dependent classes provide a simpler solution for combining variations expressed by helper objects with other variations of the master object. By expressing dependency of helpers on their masters in a parametric style rather than by nesting, we directly support extension of the design with new types of the helpers. Besides, the double parameterization of a helper object by its master and the value representing the implemented variant allows hiding the helpers from the clients of the master object.

- Implementation of a helper object as a dependent class parameterized by its master object and the implemented variant enables type-safe covariant dependency of the helper on its master as well as type-safe communication with other helper objects implementing the same variant. Consequently, we obtain a type-safe solution for combining dynamic group variations with individual variations of the group's objects.
- Dependent classes improve the design of adapting framework instances to application-specific variations. The design with dependent classes preserves the advantages of the design with virtual classes, and improves it in two respects. First, it expresses the type dependency of adapters on their adaptees, which enables type-safe access to the variation-specific functionality of the adapters and the adaptees. Second, dependent classes directly support double-dispatched instantiation of adapters, and modular extensions with new types of adaptees, thereby avoiding the complexity of the corresponding solution with virtual classes.
- Dependent classes provide an improved solution to the functionality of a framework instance depending on several extension points of the framework. They enable precise typing for the multimethods implementing such functionality, which ensures that methods are correctly used and completely implemented. Unlike the solution with virtual classes, the multi-dispatch of the method is expressed directly, which makes the design simpler and more intuitive.

5 Semantics of Dependent Classes

5.1 Introduction

In Chapter 4 we introduced the concept of dependent classes and their semantics. The semantics of dependent classes involves various non-trivial elements. Method calls and class instantiation expressions rely on dynamic dispatch over multiple parameters and the types of their fields. However, most of the complexity lies in the type system, which integrates static dispatch with dependent typing. For example, dependent typing introduces term equivalence relation, and inheritance relationships based on static dispatch.

For a better understanding of the semantic elements of dependent classes and their interaction, we define the vc^n calculus, which captures the core semantics of dependent classes, including static and dynamic dispatch and dependent typing based on paths. The calculus is carefully designed to ensure that the static dispatch and static normalization of terms in types defines a proper abstraction over dynamic dispatch and evaluation of expressions. At the same time, the expressiveness of the language must be balanced against decidability of the type system, which is especially difficult to guarantee in the presence of dependent typing. For this reason, the vc^n calculus was from the very beginning designed in an algorithmic style, so that the proof of its decidability is straightforward.

The soundness proof of vc^n is machine-checked in Isabelle/HOL [NPW02]. Since soundness proofs for languages with path-dependent types are usually either quite sketchy or quite complex[1], it is hard to make sure that the proof is free of bugs. Indeed, we have discovered various bugs and unnecessary well-formedness conditions that we probably would not have discovered with a hand-written proof. The formal definitions and the proof in Isabelle/HOL can be downloaded at [GMO06].

Although the algorithmic style of vc^n has the advantage of being constructive, and in this way outlines a possible implementation of dependent classes, at the same time it makes it difficult to extend the semantics with new expressive power. In particular,

[1]For example, the soundness proof of vc [EOC06] is 12 double-column pages long; the print-out of the vc^n proof is about 70 pages.

we experienced difficulties while trying to extend the calculus with support for abstract declarations and corresponding completeness and uniqueness checks. The definition of these checks required testing disjointness of types and computation of their common lower bounds. The new relationships produced new forms of types, which invalidated assumptions of the algorithmic design of normalization and subtyping relations in vc^n.

For this reason, we designed a new calculus with the goal of being more declarative and based on more primitive concepts than vc^n. In the new DC_C calculus, we were able to encode the types of dependent classes by sets of primitive classification and equivalence constraints. The representation of types as sets of constraints leads to a simpler and more intuitive calculus. The type system is reduced to a constraint system with axioms encoding the rules of subtyping and term equivalence. The intersection of types can be simply defined as the union of the corresponding sets of constraints. Constraints can be directly interpreted at runtime and used for method dispatch. As a result, the bulk of the operational semantics can be encoded within the same constraint system. Specific type substitution relationships of vc^n calculus, can be expressed by simple variable substitution and union of constraint sets.

DC_C provides a novel view on the semantics of dependent classes and related type systems. The expressive power of the calculus is mostly based on the axioms of the constraint system. Such an encoding clearly exposes limitations of the calculus, and shows natural directions for extending it. The major obstacle for adding new features is ensuring decidability of necessary operations on constraints. Not surprisingly, we devote a large section of this chapter to the decidability of the constraint system of DC_C. The practical outcome of the DC_C calculus is the enhancement of the semantics of vc^n with support for abstract class and method declarations and necessary completeness and uniqueness algorithms.

The chapter is structured as follows. In Sec. 5.2 we define the core semantics of dependent classes in the form of the vc^n calculus and present its properties. In Sec. 5.3 we give the definition of the DC_C calculus, and prove its soundness in Sec. 5.4. Sec. 5.5 is devoted to decidability issues of the DC_C calculus. In particular, we prove decidability of its constraint system and expression typing; then we define an algorithm for method completeness and uniqueness checking and formulate conditions, under which it terminates.

The style and notation that we will use for formal definitions in this chapter is similar to the one of Featherweight Java [IPW99]:

- A bar above a metavariable denotes a list: $\overline{f}$ stands for $f_1, \ldots, f_k$ for some natural number $k \geq 0$. If $k = 0$ then the list is empty, denoted by ϵ.
- Following common convention, $\overline{t}\ \overline{f}$ represents a list of pairs $t_1\ f_1 \cdots t_k\ f_k$.

- The list notation is also used to denote repeated application to all members of a list; for example, $\Gamma \vdash \overline{e} : \overline{t}$ denotes the conjunction of all $\Gamma \vdash e_i : t_i$ for each list index i. To keep the notation lightweight we assume a globally available program P.
- $\langle a; b \rangle$ is a pair of terms a and b.
- $a_{\{\!\{x \mapsto b\}\!\}}$ is a term obtained by substituting all free occurrences of x in term a with term b.
- The equality $=$ symbol is used to denote syntactic equivalence of terms, including α-renaming where appropriate.
- $a \sqsubset b$ for any terms a,b, denotes that a is a (strict) syntactical subterm of b.

5.2 vc^n Calculus

In this section, we present a formal calculus that precisely describes the dynamic and static semantics of dependent classes. This calculus is called vc^n to indicate that it generalizes the previous formalization of virtual classes in *vc* calculus [EOC06] by introducing more dispatch dimensions.

The calculus is focused only on the core aspects of dependent classes: dynamic and static dispatch, as well as path-dependent types, allowing to relate variations of multiple objects. The core semantics of dependent classes is already complicated enough, thus it was important to keep the calculus as small as possible. Therefore, like Featherweight Java [IPW99], we define the calculus in a functional style to avoid the complexity associated with mutable state and its typing. vc^n also follows the encoding of methods by classes introduced in *vc* [EOC06]. This further simplifies the calculus by reusing class instantiation expressions and class dispatch for method calls and method dispatch, respectively.

The resulting calculus is much smaller than the DepJ, which we used for examples. In Sec. 5.2.1 we present the syntax of the calculus and explain its differences to DepJ. The operational semantics of the calculus is defined in the "small-step" style. We managed to reduce the operational semantics to two computation rules and two congruence rules, which are presented in Sec. 5.2.2.

The major complexity of the calculus lies in its type system, the presentation of which is divided in four sections. Sec. 5.2.3 defines the type equivence relation, which is based on normalization of terms in dependent types. Sec. 5.2.4 defines substitution relations for dependent types. Sec. 5.2.5 defines mutually recursive subtyping and matching relations,

Syntax:

$$\begin{array}{lcl} P & ::= & \overline{D} \\ D & ::= & C(\overline{f} : \overline{t}) : t \textbf{ extends } \overline{C}\ \{e\} \\ p, q & ::= & \overline{f} \\ t, u & ::= & p \mid C(\overline{f} : \overline{t}) \mid v \\ e & ::= & \textbf{this} \mid e.f \mid \textbf{new } C(\overline{f} = \overline{e}) \mid v \\ v & ::= & C(\overline{f} = \overline{v}) \end{array}$$

C – class names
f – field names

Context:

$$\Gamma \ ::= \ C(\overline{f} : \overline{t}) \mid \varnothing$$

Figure 5.1: Syntax

which are pivotal to the semantics of dependent classes. Finally, Sec. 5.2.6 defines expression typing and remaining type-checking rules.

In Sec. 5.2.7 we present the properties of the calculus. We state the soundness theorem and the major lemmas, outlining more specific properties of the relations of the calculus supporting the soundness. The section does not include the complete proof of soundness and only gives a reference to the automatically checked proof definitions using Isabelle/HOL. In that subsection, we also demonstrate decidability of the calculus and formulate the completeness property for path normalization.

The vc^n calculus is deliberately left open for variations on specific dispatch strategies. Therefore, in Sec. 5.2.8 we discuss different strategies for selecting the most specific matching declarations, and explain how the calculus could be specialized to support them.

5.2.1 Syntax

The syntax of vc^n is defined on the left-hand side of Fig. 5.1. We have made a few design decisions to keep the calculus simple in order to focus just on the core semantics of dependent classes, and to ease the soundness proofs. For the informal explanation of the concepts and the examples in Chapter 4 we used the syntax of DepJ language that is close to the syntax of Java. Besides, we used various language features that are not interesting from a semantics perspective, but are useful for practical programming. For example, various predefined types, such as `int`, `double`, `String`, and operations on them are not available in the formal calculus. In the following, we explain the formal syntax of vc^n and its differences to DepJ.

A program P in vc^n consists of multiple class declarations. A class declaration, D, starts with a class name (note that the `class` keyword is skipped), followed by a list of field declarations and the return type of the class constructor. The list of declared fields also specifies the list of constructor parameters. A class can have an arbitrary number of super-classes specified in its **extends** clause. The body of a class declaration contains its constructor expression, which is called when the class is instantiated.

There is no special syntactical category to encode methods. As usual in formal accounts of virtual classes [OCRZ03, EOC06], we use the syntax of class declarations to encode both classes and methods. A method declaration is encoded in vc^n by a class declaration: Method parameters are encoded as constructor parameters, the return type as the constructor return type and the implementation as the constructor expression. For "normal" classes, i.e., those that do not encode methods, we assume the expression this to be the default constructor body, i.e., the constructor simply returns the constructed object. The default return type is the empty path ϵ – the path pointing to this. Method calls are encoded as constructor calls. Multimethods can hence be encoded by using class declarations as methods.

In the calculus all declarations are at the top level, which means that it is not possible to nest methods within class declarations. However, the nested style can be easily translated to the parametric style as was explained in Sec. 4.3.1.

A *path*, referred to by p or q, is a sequence of fields; it refers to the object that is reached by navigating over the fields in the sequence starting from **this**. As a special case, the empty path ϵ refers to **this**.

A type, referred to by t or u, can be a class type $C(\overline{f} : \overline{t})$, a path p, or a value v. The *class type* $C(\overline{f} : \overline{t})$ represents all objects of C and its subclasses, whose fields f_i have values compatible with the respective types t_i. The only instance of a path type is the object referenced by the path. A *value type*, v, has the value v as its only member.

Types that contain paths are called *relative types*, as they are defined only relative to some object (referred to by **this**). On the contrary, *absolute types* are combinations of class and value types. For instance, Vector(s: v1.s) is a relative type, whereas Vector(s: 3DSpace) is an absolute type.

Most type relations of the calculus are defined relative to a typing context Γ, which is either empty ($\varnothing$) or defines the type of **this**. Relative types make sense only in a non-empty context, while an empty context can be used with absolute types. During static type checking (program well-formedness), the context will always be non-empty, whereas during runtime checking (typing intermediate expressions during evaluation) the expression **this** does not occur (it is replaced by a value) and the context will always be empty.

The calculus requires that in a class type the types of all fields that are available in this class are specified (this requirement is enforced in the well-formedness rules). In the informal language we allow omitting some of the field type annotations. In this case, the field types from the declaration of the class are assumed. In case of multiple class declarations, we assume the types of the most general declaration of the class as the default type of that field. An alternative solution would be to mark one of the class declarations which contains the default field types explicitly with some default keyword.

Like Featherweight Java [IPW99], vc^n supports only functional style object-oriented programming. Classes have only immutable fields that are at the same time their constructor parameters. The body of a constructor is hence an expression, rather than a list of statements.

An expression e can be **this**, a field access, a class constructor call, or a value v. We use the new keyword to mark constructor calls and distinguish them from values. In the formal syntax, constructor calls take parameters by name, rather than by position. This makes it easier (in fact: trivial) to define the mapping from constructor parameters to field names, which would otherwise be cumbersome in the presence of multiple inheritance.

A value v is a class name together with values for its fields. Values can be used both as expressions and as types, but they are not part of the written syntax: They occur as expressions only in intermediate programs during rewriting and as types of intermediate programs containing values (we use a small-step operational semantics).

In DepJ we specified certain classes and methods as abstract. The informal meaning for a class being abstract is that it cannot be instantiated, while an abstract method has no implementation, but it can be called. However, abstract dependent classes are not formalized in vc^n. When converting the examples to the calculus, abstract methods must be encoded as concrete methods with some default implementation, e.g., recursively calling the method with the same parameters.

Figure 5.2 illustrates encoding of the DepJ syntax in the calculus by a fragment of the graphical editor example from Sec. 4.2.3. The listing at the top shows dependent classes NodeGraphic and LogicCompGraphic with declarations of getConnectionAt in the syntax of DepJ. The listing below shows encoding of these class and method declarations in vc^n. The implementations of the getConnectionAt method are taken out from the class declarations and declared at the top level. Their implicit this parameter is encoded by an explicit out parameter with an appropriate type. The declarations of NodeGraphic and LogicCompGraphic are extended with the default constructor and the default return type. The declaration of NodeGraphic also receives an extends clause with an empty list of parents. The abstract getConnectionAt method of NodeGraphic is replaced by a method with a default implementation. The method calls in the implementation of getConnectionAt for LogicCompGraphic are replaced by corresponding constructor calls. The calculus supports only name-based bindings of parameters, thus all position-based parameter bindings in types and expressions are replaced with equivalent name-based bindings.

```
class NodeGraphic(GraphicalEditor editor) {
    abstract ConnectionGraphic(editor) getConnectionAt(int i1);
}

class LogicCompGraphic(LogicComponent comp, CircuitEditor editor) extends NodeGraphic {
    ConnectionGraphic(editor) getConnectionAt(int i1) {
        return editor.wireGraphicFor(...);
    }
}
```

```
NodeGraphic(editor: GraphicalEditor) extends ϵ : ϵ { this }

getConnectionAt(out: NodeGraphic(editor: GraphicalEditor), i1: Int)
    : ConnectionGraphic(editor: out.editor) {
    new getConnectionAt(out = out)
}

LogicCompGraphic(comp: LogicComponent, editor: CircuitEditor) extends NodeGraphic : ϵ { this }

getConnectionAt(out: LogicCompGraphic(comp: LogicComponent, editor: CircuitEditor), i1: Int)
    : ConnectionGraphic(editor: out.editor) {
    new wireGraphicFor(out = out.editor, wire = ...)
}
```

Figure 5.2: An example in DepJ syntax and its encoding in vc^n calculus

5.2.2 Operational Semantics

The operational semantics in small-step style is given on the right-hand side of Fig. 5.3. There are only two computation rules: field access (Red-Field) and constructor call (Red-New). The other two reduction rules are just congruence rules.

Field access applied to a value, $C(\ldots f_i = v_i \ldots)$, is resolved by looking up the value of the field. The reduction of a constructor call uses the Select relation to select a declaration of class C for the given parameter values $\overline{v}$.

The Select relation is responsible for selecting one of the declarations that match the given parameter values. The set of matching declarations is defined by the Match relation, which will be discussed in Sec. 5.2.5. Intuitively, a declaration matches a given set of parameter values to a constructor call, if the types of these parameter values are more specific than the corresponding field types of the declaration.

Once a matching declaration is selected, the evaluation proceeds with the expression of the selected class declaration e, whereby **this** in e is replaced by the value of the constructed object.

The definition of Select determines the dispatch strategy. The non-deterministic strategy [CDNW07] in Fig. 5.3 (any matching declaration can be selected) is the most general

Computation:

$$C(\ldots f_i = v_i \ldots).f_i \rightarrow v_i \quad (\textsc{Red-Field}) \qquad \frac{\ldots \{e\} \in \mathrm{Select}(C(\overline{f} = \overline{v}))}{\textbf{new } C(\overline{f} = \overline{v}) \rightarrow e_{\{\mathbf{this} \mapsto C(\overline{f}=\overline{v})\}}} \; (\textsc{Red-New})$$

Congruence:

$$\frac{e \rightarrow e'}{e.f \rightarrow e'.f} \; (\textsc{RedC-Field}) \qquad \frac{e \rightarrow e'}{\textbf{new } C(...f = e...) \rightarrow \textbf{new } C(...f = e'...)} \; (\textsc{RedC-New})$$

Select Declaration:

$$\frac{\epsilon \vdash D \in \mathrm{Match}(C(\overline{f} : \overline{v}))}{D \in \mathrm{Select}(C(\overline{f} = \overline{v}))} \quad (\textsc{Select})$$

Figure 5.3: Operational semantics

definition that is sufficient to prove soundness of the calculus. We have proved that any definition for Select that fulfills the following condition makes the type system sound: Whenever a well-formed type has any matching declarations, then the selection for this type succeeds and selects one of the matching declarations. Different choices in the design space of the dispatch mechanism will be discussed in Sec. 5.2.8.

5.2.3 Path Normalization and Type Equivalence

To determine whether two dependent types are equivalent, it is necessary to define an equivalence relation on those kinds of expressions that types may depend on. Type systems that allow types to depend on arbitrary, possibly non-terminating, expressions are often undecidable. Types in vc^n may only depend on path expressions, and for the latter a decidable equivalence relation can be defined, as shown in Fig. 5.4.

Two paths are equivalent (rule $\simeq$-Path) if they have the same *normal form*; the definitions for type equivalence (rules $\simeq$-Value, $\simeq$-Class, and $\simeq$-Path) just propagate path equivalence to the type level.

Intuitively, a path is in a normal form, if neither the path itself, nor any part of it, are declared as aliases of other paths. Accordingly, path normalization can be seen as the process of eliminating alias paths. If a path $p.f$ is an alias of path p' then the normal form of $p.f$ is the same as the one of p' ($\rightsquigarrow$-Field2). Otherwise, we only need to

Path bound:

$$C(\overline{f}:\overline{t}) \vdash \epsilon \prec C(\overline{f}:\overline{t}) \quad (\prec\text{-}\textsc{This}) \qquad \frac{\begin{array}{c}\Gamma \vdash p \rightsquigarrow p' \\ \Gamma \vdash p' \prec C(\overline{f}:\overline{t})\end{array}}{\Gamma \vdash p.f_i \prec t_i} \quad (\prec\text{-}\textsc{Field})$$

Path normalization:

$$\Gamma \vdash \epsilon \rightsquigarrow \epsilon \; (\rightsquigarrow\text{-}\textsc{This}) \qquad \frac{\begin{array}{c}\Gamma \vdash p.f \prec C(\overline{f}:\overline{t}) \\ \Gamma \vdash p \rightsquigarrow p'\end{array}}{\Gamma \vdash p.f \rightsquigarrow p'.f} \quad (\rightsquigarrow\text{-}\textsc{Field1}) \qquad \frac{\begin{array}{c}\Gamma \vdash p.f \prec p' \\ \Gamma \vdash p' \rightsquigarrow p''\end{array}}{\Gamma \vdash p.f \rightsquigarrow p''} \; (\rightsquigarrow\text{-}\textsc{Field2})$$

Type Equivalence:

$$\frac{\begin{array}{c}\Gamma \vdash p \rightsquigarrow p'' \\ \Gamma \vdash p' \rightsquigarrow p''\end{array}}{\Gamma \vdash p \simeq p'} \; (\simeq\text{-}\textsc{Path}) \qquad \frac{\forall i.\; t_i \simeq t'_i}{\Gamma \vdash C(\overline{f}:\overline{t}) \simeq C(\overline{f}:\overline{t'})} \quad (\simeq\text{-}\textsc{Class}) \qquad \Gamma \vdash v \simeq v \; (\simeq\text{-}\textsc{Value})$$

Figure 5.4: Path- and type equivalence

normalize the prefix p ($\rightsquigarrow$-Field1). As a special case (not covered by $p.f$), the empty path normalizes to itself ($\rightsquigarrow$-This).

To determine if a path is an alias of another path we use the relation $\Gamma \vdash p \prec t$, which computes *the bound of the path*[2] based on the types of the fields declared in the context Γ. The relation is by a recursion on the structure of the path. The bound of the empty path, i.e., the self-reference, is the type given in the context ($\prec$-This). To compute the bound of $p.f$, we first normalize p to p' and then take the bound of p'. The bound of a normal path is always a class type, thus we can determine the bound of $p.f$ by looking up the type of field f in the bound of p'.

We have proved the following theorem which characterizes the meaning of path equivalence: *Two paths are equivalent if and only if they are indistinguishable by the operational semantics in all extensions of the program.* The "only if" direction is required for type soundness; the "if" direction is a completeness property which states that the path normalization is optimal, i.e., any bigger path equivalence relation would be unsound. We give a formal statement of the completeness property in Sec. 5.2.7.2.

[2]In several places we call the relation as *path typing.*

Precise substitution:

$$[p']_p = p.p' \quad ([\cdot]\text{-}\textsc{Path}) \qquad\qquad [\epsilon]_v = v \quad ([\cdot]\text{-}\textsc{ValueThis})$$

$$\frac{\forall i.\ [t_i]_t = t'_i}{[C(\overline{f} : \overline{t})]_t = C(\overline{f} : \overline{t'})} \ ([\cdot]\text{-}\textsc{Class}) \qquad\qquad \frac{[p]_{t_i} = t'}{[f_i.p]_{C(\overline{f}:\overline{t})} = t'} \ ([\cdot]\text{-}\textsc{ClassField})$$

$$[v]_t = v \quad ([\cdot]\text{-}\textsc{Value}) \qquad\qquad \frac{[p]_{v_i} = t'}{[f_i.p]_{C(\overline{f}=\overline{v})} = t'} \ ([\cdot]\text{-}\textsc{ValueField})$$

Imprecise substitution:

$$\frac{[t']_t = t''}{\lceil t' \rceil_t = t''} \ (\lceil\cdot\rceil\text{-}\textsc{Weaken}) \qquad\qquad \frac{\lceil p \rceil_{t_i} = t'}{\lceil f_i.p \rceil_{C(\overline{f}:\overline{t})} = t'} \ (\lceil\cdot\rceil\text{-}\textsc{ClassField})$$

$$\frac{\forall i.\ \lceil t_i \rceil_t = t'_i}{\lceil C(\overline{f} : \overline{t}) \rceil_t = C(\overline{f} : \overline{t'})} \ (\lceil\cdot\rceil\text{-}\textsc{Class}) \qquad\qquad \lceil \epsilon \rceil_{C(\overline{f}:\overline{t})} = C(\overline{f} : \overline{t}) \ (\lceil\cdot\rceil\text{-}\textsc{ClassThis})$$

Figure 5.5: Substitution of the self-reference in types

5.2.4 Substitution in Types

In vc^n, path-dependent types are parameterized by the self-reference, which appears at the beginning of all paths (represented by the empty path). In a type declaration, the self-reference refers to an instance of the enclosing class, but in a specific use of the type we can have more specific knowledge about the self-reference. For example, in the type t of some field f, the self-reference refers to the owner of the field. In a field access expression $e.f$, we know that the owner of the field is e and thus the type of the self-reference in t is at least the statically known type of e.

Fig. 5.5 defines two relationships[3] $[t]_u = t'$ and $\lceil t \rceil_u = t'$ defining substitution of the self-reference in t by its statically known type u. Since values of vc^n are also available as types, the relationships define substitution of the self-reference in type by a value as a special case.

These two relationships are very similar. According to $\lceil\cdot\rceil$-Weaken, if $[t]_u$ is defined, then $[t]_u = \lceil t \rceil_u$. The difference between the two relationships is that $[t]_u$ preserves the precision of t, while $\lceil t \rceil_u$ additionally covers the cases when the precision is lost, i.e.,

[3]We have proved that a choice of u and t uniquely determines t'. Therefore, we will also use the relationships as functions $[t]_u$ and $\lceil t \rceil_u$.

it allows substituting occurences of paths (singleton types) by their statically known bounds.

Preservation of type precision is critical in so-called contravariant positions. For example, during the type-checking of a constructor call, we must check if the types of the given parameters match the expected types of the constructor parameters. It is important to preserve the precision of the expected parameter types when specializing them for the context of that constructor (replacing their self-references by the type of the constructed object). Otherwise, the type-checking would be weakened and become unsound. In so-called covariant positions, such as types of fields or return type of constructors, loss of precision is safe. In such cases, usage of the imprecise substitution is preferred, because it is less restrictive and, therefore, enables accepting more type-safe programs.

In class types substitution is simply propagated into the types of fields ($[\cdot]$-Class and $\lceil\cdot\rceil$-Class), and value types remain unchanged ($[\cdot]$-Value). The interesting case is substitution in paths, which is defined by the remaining rules:

- Substitution of the self-reference by a path in a path leads to the concatenation of these two paths ($[\cdot]$-Path). Such concatenation corresponds to the lexicographical substitution of the self-reference.
- Substitution of the self-reference by a value in a path corresponds to the evaluation of that path starting from that value ($[\cdot]$-ValueThis and $[\cdot]$-ValueField).
- Substitution of the self-reference by a class type in a path is the most complicated case. According to rules $[\cdot]$-ClassField and $\lceil\cdot\rceil$-ClassField, we use the path to navigate within the structure of the class type. In the precise substitution, such navigation must terminate by a path or a value (i.e., one of the rules $[\cdot]$-Path, $[\cdot]$-ValueThis and $[\cdot]$-ValueField), which ensures that the result of path substitution is a path or a value. The imprecise substitution defines an additional termination rule $\lceil\cdot\rceil$-ClassThis, which permits replacing the self-reference by the given class type and so weakening the type.

5.2.5 Subtyping

The subtyping rules are shown in Fig. 5.6. Subtyping has deliberately been defined in an algorithmic style in order to demonstrate decidability. In particular, there is no subsumption or transitivity rule; rather, transitivity follows as a lemma.

Two equivalent types are subtypes of each other (<:-Equiv). This is the only rule that accepts a path or a value as a supertype: The value itself is the only subtype of a value type and a subtype of a path must be an equivalent path.

Subtyping:

$$\frac{\Gamma \vdash t \simeq t'}{\Gamma \vdash t <: t'} \quad (<:\text{-}\textsc{Equiv})$$

$$\frac{\Gamma \vdash p \rightsquigarrow p' \qquad \Gamma \vdash p' \prec C'(\overline{f'} : \overline{t'}) \quad \Gamma \vdash C'(\overline{f'} : \overline{t'}) \text{ OK} \quad \Gamma \vdash C'(\overline{f'} : p'.\overline{f'}) <: C(\overline{f} : \overline{t})}{\Gamma \vdash p <: C(\overline{f} : \overline{t})} \quad (<:\text{-}\textsc{PathClass})$$

$$\frac{\forall i.\ \exists j.\ f'_j = f_i \ \wedge\ \Gamma \vdash t'_j <: t_i \quad C \in \text{Parents}(\Gamma, C', \overline{f'} : \overline{t'}, \varnothing) \quad \Gamma \vdash C'(\overline{f'} : \overline{t'}) \text{ OK}}{\Gamma \vdash C'(\overline{f'} : \overline{t'}) <: C(\overline{f} : \overline{t})} \quad (<:\text{-}\textsc{Class})$$

$$\frac{\Gamma \vdash C'(\overline{f'} : \overline{v'}) <: C(\overline{f} : \overline{t})}{\Gamma \vdash C'(\overline{f'} = \overline{v'}) <: C(\overline{f} : \overline{t})} \quad (<:\text{-}\textsc{ValueClass})$$

Parents:

$$\begin{array}{c}\text{Parents}(\Gamma, C, \overline{f} : \overline{t}, S) = \{C\} \cup (\bigcup_{C' \in S' \setminus S} \text{Parents}(\Gamma, C', \overline{f} : \overline{t}, S \cup \{C\})) \\ \text{where } \ S' = \{C''_i \mid \Gamma \vdash \ldots \textbf{extends } \overline{C''} \ldots \in \text{Match}(\text{MakeType}(C, \overline{f} : \overline{t}))\}\end{array} \quad (\textsc{Parents-Def})$$

Matching declarations:

$$\frac{D \in P \qquad C(\overline{f} : \overline{t'}) = \text{Sig}(D) \quad \forall i.\ \exists t''.\ [t'_i]_{C(\overline{f}:\overline{t})} = t'' \ \wedge\ \Gamma \vdash t'' \text{ OK} \ \wedge\ \Gamma \vdash t_i <: t''}{\Gamma \vdash D \in \text{Match}(C(\overline{f} : \overline{t}))} \quad (\textsc{Match})$$

Auxilliary definitions:

$$\text{Sig}(C(\overline{f} : \overline{t}) : u \textbf{ extends } \overline{C}\{e\}) = C(\overline{f} : \overline{t}) \quad (\textsc{Sig})$$

$$\frac{C(\overline{f} : \overline{t}) \ldots \in P \quad \forall i, j.\ i \neq j \Rightarrow f_i \neq f_j}{\text{Fields}(C) = \overline{f}} \quad (\textsc{Fields})$$

$$\frac{\overline{f'} = \text{Fields}(C) \qquad \overline{f'} \subseteq \overline{f} \quad \forall i, j.\ (f_i = f'_j) \Rightarrow (t_i = t'_j)}{\text{MakeType}(C, \overline{f} : \overline{t}) = C(\overline{f'} : \overline{t'})} \quad (\textsc{MakeType})$$

Figure 5.6: Subtyping

```
Tree : ϵ { this }
Node(t: Tree) : ϵ { this }

parentof(n1: Node(t:Tree), n2: Node(t:n1.t)) : Bool { False }
test(n: Node(t:Tree)) : Bool { parentof(n1 = n, n2 = n) }
test2() : Bool { parentof(n1 = Node(t = Tree()), n2 = Node(t = Tree())) }
```

Figure 5.7: An example of trees in vc^n

The comparison of a value type with a class type (<:-ValueClass) is defined in terms of comparing two class types by replacing the value with the most specific class type that is compatible with it.

The comparison of a path p with a class type (<:-PathClass) is reduced to a class type comparison. For this purpose, the most specific class type that is a supertype of p must be constructed.

A class type for p could be computed as the bound of the normalized p. However, this type is too weak to type-check many interesting programs. For illustration, consider the example of Fig. 5.7. A Tree consists of multiple Node's, which refer to their tree by field t. The function parentof tests if one node is a parent of another node in the tree. The function test calls parentof to check if a node is a parent of itself. If we use the path bound for subtyping, this call would not pass the type checker. The actual type of the second argument passed to parentof is n, while its formal type, specialized for the constructor call, is Node(t:n.t). The bound of n in the context of the call is Node(t:Tree) (as specified in the declaration of test), which is not a subtype of the formal type Node(t: n.t). Hence we need another strategy for constructing the most specific class type that is a supertype of p.

For this reason, the type of fields of $C'(\overline{f'} : \overline{t'})$ are further specialized. We know that each field f' actually has the type $p'.f'$. Hence, by substituting each $\overline{t'_i}$ for $p'.\overline{f'_i}$, a more specific class type is constructed, which is still a supertype of p. The resulting type is finally compared with $C(\overline{f} : \overline{t})$. In the example, Node(t:Tree) (the bound of o) is specialized to Node(t:n.t).

Let us now focus on the rule for comparing class types (<:-Class) in Fig. 5.6. For a class type $C'(\overline{f'} : \overline{t'})$ to be a subtype of class type $C(\overline{f} : \overline{t})$, it must be well-formed in Γ (to be defined later), the types of the corresponding fields must be more specific, and C must be a *parent* of $C'(\overline{f'} : \overline{t'})$.

The function $\text{Parents}(\Gamma, C, \overline{f} : \overline{t}, S)$ determines all parents of class C relative to field types $\overline{f} : \overline{t}$ in the context Γ. The function MakeType constructs a valid type from a class C and field types $\overline{f} : \overline{t}$ by selecting only those fields that are declared for class C. The idea of $\text{Parents}(\Gamma, C, \overline{f} : \overline{t}, S)$ is to collect superclasses from all the declarations of C that match the given field types, and then recursively collect the parents of these superclasses. Further, a class C is also considered a parent of itself.

The last parameter of the Parents function is an accumulator that remembers the classes already visited by the algorithm. By checking that no class is visited twice, termination is ensured even in the presence of cyclic inheritance relations. Since the parents of a class are relative to the types of its fields, it can happen that for some type t it holds that $\Gamma \vdash C(f : t) <: C'(f : t)$, for another type t' we have $\Gamma \vdash C'(f : t') <: C(f : t')$ and for yet another type t'' both relations may hold simultaneously.

The relation Match defines the set of class declarations that match a class type $C(\overline{f} : \overline{t})$. It is used to determine which declarations contribute to a given class type. Intuitively, a declaration D matches a class type, if the type is of the same class as the declaration and the type is compatible with the type of **this** assumed by the declaration. A declaration assumes that the type of **this** is at least as specific as the signature of the declaration (see Fig. 5.6 for the definition of Sig). For a type $C(\overline{f} : \overline{t})$ to be compatible with the signature $C(\overline{f} : \overline{t'})$, the types $\overline{t}$ must be more specific than the types $\overline{t'}$.

The self-reference in the declared types $\overline{t'}$ must be replaced with the type used for matching, i.e., $C(\overline{f} : \overline{t})$. The resulting types, $\overline{t''}$, should be supertypes of the corresponding types from $\overline{t}$. The precise substitution guarantees that the substituted types $\overline{t''}$ preserve the precision of $\overline{t'}$. To illustrate why the precise substitution is critical for matching, consider a call to parentof with two arbitrary tree nodes on line 6 in Fig. 5.7. Type checking the call involves matching the type u that describes a parentof of two arbitrary nodes, against D - the declaration of node, which expects two nodes from the same editor:

$$\begin{array}{lcl} u & = & \mathsf{parentof(n1{:}\ Node(t{:}Tree),\ n2{:}\ Node(t{:}Tree))} \\ \mathrm{Sig}(D) & = & \mathsf{parentof(n1{:}\ Node(t{:}Tree),\ n2{:}\ Node(t{:}n1.t))} \end{array}$$

The declaration D should not match u, because its signature is more specific than u; it requires that n2.t = n1.t, which cannot be ensured by assuming **this** to be u. D is not included into the set of matching declarations of u, because the precise substitution $[\mathsf{Node(t{:}n1.t)}]_u$ fails. If the imprecise substitution were used instead, D would incorrectly match, because $\lceil \mathsf{Node(t{:}n1.t)} \rceil_u = \mathsf{Node(t{:}Tree)}$.

Since Match is also used in the operational semantics (Select rule in Fig. 5.3), the question raises how much the operational semantics depends on the type system. The answer is that only a small subset of the typing rules is actually needed in the operational semantics. This is because the type to match in Fig. 5.3 is always a value type (a value used as type). Substitution of the self-references in a type with a value always produces an absolute type that does not contain paths. This means that for the operational semantics we just need to compare values with absolute types and never have to deal with paths.

5.2.6 Expression Typing and Well-Formedness

Fig. 5.8 specifies type assignment for expressions. The type of **this** is the empty path ϵ (Type-This). As usual for a small-step semantics, we also need a typing rule for values. The type of a value is the corresponding value type (Type-Value). Since values occur only during execution, this rule is only used for runtime type checking, i.e., for the preservation theorem (Sec. 5.2.7.1).

Typing a field access expression (Type-Field) is performed by first computing the type t of the prefix e and then substituting it for the self-references in the type of f. The imprecise substitution is used in this case, because it is less restrictive and it is safe to weaken the type of an expression (because of the subsumption property).

For a constructor call expression (Type-New), the types of the actual parameters are computed. If there is any class declaration that matches these types, the return type t'', which is identical for all declarations of a class (as stated by the WF-Prog rule), is taken and its self-references are substituted with the statically known type of the constructed object $C(\overline{f} : \overline{t})$.

The rules for well-formed types check (a) whether all paths exist in the given context (WF-Path) by using path normalization and bounding, and (b) whether all classes exist (WF-Class) with matching field names.[4]

A class declaration is well-formed (WF-Decl), if all type declarations are well-formed in the context of the declaration. The constructor expression must be well-typed and its type must be a subtype of the declared return type. The set of fields in the class declarations must include all fields of direct superclasses. Further, it is required that values are not used in the types of fields. This ensures the property that the bound of a normalized path is always a class type.

Finally, two conditions are imposed on a program P in order for it to be well-formed (rule WF-Prog): (a) all declarations must be well-formed, and (b) all declarations of the same class must have the same sets of fields and identical return types.

[4]The auxiliary function Fields is defined in Fig. 5.6.

Expression typing:

$$\frac{\Gamma \vdash \epsilon \text{ OK}}{\Gamma \vdash \textbf{this} : \epsilon} \ (\textsc{Type-This}) \qquad \frac{\Gamma \vdash v \text{ OK}}{\Gamma \vdash v : v} \ (\textsc{Type-Value})$$

$$\frac{\begin{array}{c}\Gamma \vdash e : t \\ \lceil f \rceil_t = t' \qquad \Gamma \vdash t' \text{ OK}\end{array}}{\Gamma \vdash e.f : t'} \ (\textsc{Type-Field})$$

$$\frac{\begin{array}{c}\Gamma \vdash \overline{e} : \overline{t} \qquad \Gamma \vdash C(\overline{f} : \overline{t}) \text{ OK} \\ \Gamma \vdash C(\overline{f} : \overline{t'}) : t'' \ldots \in \text{Match}(C(\overline{f} : \overline{t})) \\ \lceil t'' \rceil_{C(\overline{f}:\overline{t})} = t''' \qquad \Gamma \vdash t''' \text{ OK}\end{array}}{\Gamma \vdash \textbf{new } C(\overline{f} = \overline{e}) : t'''} \ (\textsc{Type-New})$$

Well-formed type:

$$\frac{\begin{array}{c}\Gamma \vdash p \rightsquigarrow p' \\ \Gamma \vdash p' \prec t \qquad \Gamma \vdash t \text{ OK}\end{array}}{\Gamma \vdash p \text{ OK}} \ (\text{WF-}\textsc{Path})$$

$$\frac{\text{Fields}(C) = \overline{f} \qquad \forall i.\ \Gamma \vdash t_i \text{ OK}}{\Gamma \vdash C(\overline{f} : \overline{t}) \text{ OK}} \ (\text{WF-}\textsc{Class})$$

$$\frac{\Gamma \vdash C(\overline{f} : \overline{v}) \text{ OK}}{\Gamma \vdash C(\overline{f} = \overline{v}) \text{ OK}} \ (\text{WF-}\textsc{Value})$$

Well-formed declaration:

$$\frac{\begin{array}{c}\Gamma = C(\overline{f} : \overline{t}) \\ \Gamma \vdash C(\overline{f} : \overline{t}) \text{ OK} \qquad \Gamma \vdash t \text{ OK} \\ \overline{t} \text{ does not contain values} \\ \Gamma \vdash e : t' \qquad \Gamma \vdash t' <: t \\ \forall i, \overline{f'}.\ \text{Fields}(C_i) = \overline{f'} \ \Rightarrow\ \overline{f'} \subseteq \overline{f}\end{array}}{C(\overline{f}, \overline{t}) : t \textbf{ extends } \overline{C}\ \{e\} \text{ OK}} \ (\text{WF-}\textsc{Decl})$$

$$\frac{\begin{array}{c}\forall D \in P.\ D \text{ OK} \\ \left[\begin{array}{l}\forall D, D' \in P. \\ \quad D = C(\overline{f} : \overline{t}) : t \ldots \ \wedge\ D' = C(\overline{f'} : \overline{t'}) : t' \ldots \ \Rightarrow\ \overline{f} = \overline{f'} \ \wedge\ t = t'\end{array}\right]\end{array}}{P \text{ OK}} \ (\text{WF-}\textsc{Prog})$$

Figure 5.8: Typing

5.2.7 Properties of vc^n

In this section, meta-theoretical properties of vc^n will be discussed: The soundness, the decidability, and expressiveness of the type system.

5.2.7.1 Soundness

We use the standard method to prove the soundness of the calculus by a progress and a preservation theorem [WF94]. The progress theorem states that every well-typed expression in a well-typed program is either a value or can be further reduced. The preservation theorem ensures that if well-typed expression e is reduced to e', then the type of e' is a subtype of the type of e.

Theorem 5.2.1 (Progress). *If* P OK *and* $\epsilon \vdash e : t$ *then* $\exists v.\ e = v$ *or* $\exists e'.\ e \rightarrow e'$

Theorem 5.2.2 (Preservation). *If* P OK *and* $\epsilon \vdash e : t$ *and* $e \rightarrow e'$
then $\exists t'.\ \epsilon \vdash e' : t' \ \wedge\ \epsilon \vdash t' <: t$

The proofs of both theorems have been verified by the Isabelle/HOL proof assistant and are available for download at [GMO06]. To understand why these theorems hold, we present a few key lemmas from the proof.

The substitution lemma of the calculus (Lemma 5.2.1) states that substitution of **this** in an expression e with a value v corresponds to the substitution of **this** with v in the type of e. The assumption $\epsilon \vdash v <: [u]_v$ should be read as: v is appropriate as value of **this** in the context where the type of **this** is u.

Lemma 5.2.1 (Substitution). *If* $u \vdash e : t$ *and* $\epsilon \vdash v <: [u]_v$ *and* $\epsilon \vdash v$ OK
then $\exists t'.\ \epsilon \vdash e_{\{\!\{\mathbf{this} \mapsto v\}\!\}} : t' \ \wedge\ \epsilon \vdash t' <: [t]_v$

The substitution lemma is necessary for ensuring type preservation when reducing a constructor call (**new** v) to $e_{\{\!\{\mathbf{this} \mapsto v\}\!\}}$ (Red-New in Fig. 5.3), where e is the implementation of the constructor. The lemma states that the type of the resulting expression $e_{\{\!\{\mathbf{this} \mapsto v\}\!\}}$ is a subtype of $[t]_v$. The type of (**new** v) is $[t']_v$, where t' is the declared return type of the constructor. The reduction preserves the type, because $[t]_v$ is a subtype of $[t']_v$, which follows from Lemma 5.2.5 and the fact that the type of the constructor implementation t is a subtype of the declared return type t' in a well-formed program.

Lemmas 5.2.2 and 5.2.4 state that the properties of static semantics are preserved at runtime. In particular, Lemma 5.2.2 states that if a declaration D matches a type t, then the match will be preserved at runtime for any possible value v of **this**. Analogously, lemma 5.2.3 states that subtype relations are preserved at runtime. Further, lemma 5.2.4 states that subtypes produce more matching declarations.

Lemma 5.2.2 (Preservation of Matching). *If* $u \vdash D \in \text{Match}(t)$ *and* $\epsilon \vdash v <: [u]_v$ *and* $\epsilon \vdash v \text{ OK}$ *, then* $\epsilon \vdash D \in \text{Match}([t]_v)$

Lemma 5.2.3 (Preservation of Subtyping). *If* $\epsilon \vdash v <: [u]_v$ *and* $\epsilon \vdash v \text{ OK}$ *and* $u \vdash t' <: t$*, then* $\epsilon \vdash [t']_v <: [t]_v$

Lemma 5.2.4 (Monotonicity of Matching). *If* $\Gamma \vdash D \in \text{Match}(t)$ *and* $\Gamma \vdash t' <: t$*, then* $\Gamma \vdash D \in \text{Match}(t')$

Preservation of matching and subtyping hold due to the properties of substitution. The definition of matching (rule Match in Fig. 5.6) substitutes the self-references in each field type with the type used for matching $C(\overline{f} : \overline{t})$. Replacing the assumed type of **this** by a subtype must not invalidate matching relations. This is the case due to an invariance property of the precise substitution, stated in lemma 5.2.5: Only equivalent types will be produced when the context type is strengthened; hence, the subtype check in (Match) cannot fail.

Lemma 5.2.5 (Invariance of the Precise Substitution). *If* $[t]_u = t'$ *and* $\Gamma \vdash t' \text{ OK}$ *and* $\Gamma \vdash u' <: u$*, then* $\exists t''.\ [t]_{u'} = t'' \ \wedge\ \Gamma \vdash t'' \simeq t'$

The imprecise substitution in types has only a weaker property as stated in lemma 5.2.6. This property is, however, sufficient for soundness because the imprecise substitution is only used for field access and return types of constructors, where a loss in precision does not influence soundness.

Lemma 5.2.6 (Covariance of the Imprecise Substitution). *If* $\lceil t \rceil_u = t'$ *and* $\Gamma \vdash t' \text{ OK}$ *and* $\Gamma \vdash u' <: u$*, then* $\exists t''.\ \lceil t \rceil_{u'} = t'' \ \wedge\ \Gamma \vdash t'' <: t'$

5.2.7.2 Completeness

In order for the type system to be sound, path normalization must have the property that equivalent paths are indistinguishable in the operational semantics. This can be expressed formally as follows:

Lemma 5.2.7 (Soundness of Path Normalization). *If* $P \text{ OK}$ *and* $t \vdash t \text{ OK}$ *and* $t \vdash p \simeq p'$*, then for all* v *with* $\epsilon \vdash v \text{ OK}$ *and* $\epsilon \vdash v <: [t]_v$ *(think:* v *is a value of* **this** *that is allowed by* t*) and* $v.p \rightarrow_* v_1$ *and* $v.p' \rightarrow_* v_2$*, we have* $v_1 = v_2$*.*

However, even a very trivial path equivalence relation such as $\Gamma \vdash p \simeq p' :\Leftrightarrow p = p'$ would have this property. In order to demonstrate the expressiveness of path normalization, we have proven a completeness property,[5] which says that the implication also holds in the

[5]The proof is available at [GMO06].

reverse direction if we also consider possible extensions of the program - a fixed program may be too limited to distinguish two paths. For this reason, we added the program that we are talking about in the formulas in the following lemma:

Lemma 5.2.8 (Completeness of Path Normalization)**.** *If* P OK *and* $P, t \vdash t$ OK , *then* $P, t \vdash p \simeq p'$ ***if and only if*** *for all extensions* $P' = P, P''$ *of* P *such that* P' OK *and all* v *with* $P', \epsilon \vdash v$ OK *and* $P', \epsilon \vdash v <: [t]_v$ *and* $P' \vdash v.p \rightarrow_* v_1$ *and* $P' \vdash v.p' \rightarrow_* v_2$*, we have* $v_1 = v_2$*.*

In other words, if two paths are operationally indistinguishable, then they will be equivalent in the type system. Since type equivalence is just path equivalence propagated to the type level (see $\simeq$-Class), the result applies to types as well.

5.2.7.3 Decidability

The definitions of most relations are syntax directed, i.e., at least one of the relation arguments in premises is a structural part of the relation arguments in the conclusion, while the other arguments remain unchanged. This applies to type equivalence, to substitution, and to expression typing. Decidability is less obvious for path normalization, for path bounding, and for subtyping. Hence, we will illustrate how these relations can be turned into terminating algorithms.

Figure 5.9 describes an algorithm for path normalization and bounding, which is equivalent to the rules in Fig. 5.4, i.e., if $\Gamma \vdash p \leadsto p'$, then $normalize(\Gamma, \varnothing, p) = p'$; if normalization is not possible, the algorithm generates an error (either raised explicitly or due to pattern matching failure).

It is easy to see that derivations of path normalization are unique. Therefore, if during normalization of p we encounter p again, then normalization of p is not possible and an error is raised. We use the second parameter Δ to keep track of the paths, the normalization of which can cause such cycles, and throw an error if we are about to normalize a path, which is already in Δ. The only place that can cause a cycle is the path normalization in the premises of rule $\leadsto$-Field2; all other premises normalize structurally smaller paths.

The algorithm always terminates, because Δ must grow with each recursive call on a path that is not structurally smaller. On the other hand, Δ cannot grow indefinitely, because it includes only paths computed by path bounding. It is easy to see that every possible path bound is a declared type in the context. Hence, the set of all path bounds in a fixed context is finite and the algorithm is guaranteed to terminate.

$$\text{normalize}(\Gamma, \Delta, \epsilon) = \epsilon$$

$$\text{normalize}(\Gamma, \Delta, p.f) = \begin{cases} \mathit{error}, & \text{if } \ p.f \in \Delta, \\ \text{normalize}(\Gamma, \Delta \cup \{p'\}, p'), & \text{if } \ t = p', \\ \text{normalize}(\Gamma, \Delta, p).f, & \text{if } \ t = C(\overline{f} : \overline{t}), \end{cases}$$
$$\text{where} \quad t = \text{bound}(\Gamma, \Delta, p.f)$$

$$\text{bound}(C(\overline{f} : \overline{t}), \Delta, \epsilon) = C(\overline{f} : \overline{t})$$
$$\text{bound}(\Gamma, \Delta, p.f) = t, \ \text{where} \ \ C(\ldots f : t \ldots) = \text{bound}(\Gamma, \Delta, \text{normalize}(\Gamma, p))$$

Figure 5.9: Path normalization algorithm

$$\text{depth}(p) = 0$$
$$\text{depth}(v) = 0$$
$$\text{depth}(C(\overline{f}, \overline{t})) = \max(\text{depth}(\overline{t})) + 1$$

$$\text{measure}(t', t) = \begin{cases} 2 \times \text{depth}(t) + 1, & \text{if } \ t' \text{ is not a class type,} \\ 2 \times \text{depth}(t), & \text{if } \ t' \text{ is a class type.} \end{cases}$$

Figure 5.10: Measure function showing decidability of subtyping

The subtyping definitions can directly be implemented by a recursive algorithm. Its termination can be proved by a measurement function that assigns a natural number to each pair of types, such that the measure of types, compared by a subtype relation in premises, is always smaller than the measure of the types, compared in the conclusion. Such a measure function is described in Fig. 5.10. It basically states that the depth of the type on the right-hand side of the subtype relation must decrease in at most two inference steps, if we do not count the intermediate inference rules for Match.

5.2.8 Dispatch

In this section, possible dispatch strategies are discussed, each answering the question as which of the class declarations matching a constructor should be selected for execution in a different way. All discussed strategies are specializations of the most general strategy defined by the function Select in Fig. 5.3. This means that our selection of a specific dispatch strategy does not compromise the soundness of the calculus, as long as it can guarantee that something will be selected from every valid matching set of declarations:

Property 5.2.1. *If* P OK *and* $\epsilon \vdash t$ OK *and* $\epsilon \vdash D \in Match(t)$ *for some declaration D, then there exists a declaration D' such that* $D' \in Select(t)$ *and* $\epsilon \vdash D' \in Match(t)$*.*

A reasonable assumption to expect from any dispatch strategy is that it implements overriding: More specific declarations should hide more general ones. Declaration D' is more specific than declaration D, if any value that matches D' also matches D:

Definition 5.2.1. *Declaration* D' *is more specific than* D*, if for all* $v = C(\overline{f} = \overline{v})$ *with* $\epsilon \vdash v$ OK *and* $\epsilon \vdash D' \in match(C(\overline{f} : \overline{v}))$ *it follows that* $\epsilon \vdash D \in match(C(\overline{f} : \overline{v}))$*.*

Definition 5.2.1 describes the desired property of overriding, but it is not constructive, because it quantifies over all possible values. A constructive way to compare two declarations could be achieved by comparing their signatures by the subtype relation. The problem, however, is that the signatures may involve paths and, thus, cannot be compared in the global (empty) context. The solution is to use the signature of the declarations to compare as contexts, as in the constructive definition of the overrides relation below:

Definition 5.2.2. *Declaration* D' *overrides* D ($D' \ll D$)*, if* $\mathrm{Sig}(D') \vdash \mathrm{Sig}(D') <: \mathrm{Sig}(D)$ *and not* $\mathrm{Sig}(D) \vdash \mathrm{Sig}(D) <: \mathrm{Sig}(D')$*.*

The overrides relation has the property that D' overrides D implies D' is more specific than D. By using it, the definition of Select can be refined so that it guarantees that the overridden declarations are hidden (SELECT-OVER in Fig. 5.11). Given a non-empty set of declarations, a declaration can be found that is not overridden by any other declaration from the set. This is because the overriding relation is transitive and asymmetric. Thus, for such a definition of Select, constructor calls in a well-formed program will always succeed.

The rule SELECT-OVER is, however, not deterministic because there can be several declarations that do not override each other. There are different methods for eliminating this non-determinism. Following the tradition of multi-dispatch, these methods can be classified into symmetric and asymmetric ones.

Symmetric dispatch requires that only the most specific declaration can be selected. A possible definition of symmetric dispatch is given by rule (SELECT-SYMM) in Fig. 5.11. It requires that from the set of the matching declarations we can select one declaration that overrides all the others.

The type checking rules of Fig. 5.8 are not sufficient to guarantee that symmetric dispatch will always succeed. The simplest constructive way to guarantee that symmetric dispatch always succeeds, is to require that the overriding relation defines a total order on

Select for overriding:

$$\frac{\epsilon \vdash D \in \text{Match}(C(\overline{f} : \overline{v})) \qquad \forall D' \in \text{Match}(C(\overline{f} : \overline{v})).\ \neg D' \ll D}{D \in \text{Select}(C(\overline{f} = \overline{v}))} \quad (\textsc{Select-Over})$$

Select for symmetric dispatch:

$$\frac{\epsilon \vdash D \in \text{Match}(C(\overline{f} : \overline{v})) \qquad \forall D' \in \text{Match}(C(\overline{f} : \overline{v})).\ D' \neq D \ \Rightarrow\ D \ll D'}{D \in \text{Select}(C(\overline{f} = \overline{v}))} \quad (\textsc{Select-Symm})$$

Select for asymmetric dispatch:

$$\frac{\begin{array}{c} X = \{D.\ \epsilon \vdash D \in \text{Match}(C(\overline{f} : \overline{v})) \ \wedge\ (\forall D' \in \text{Match}(C(\overline{f} : \overline{v})).\ \neg D' \ll D)\} \\ D \in X \qquad \forall D' \in X.\ D' \neq D \ \Rightarrow\ D <_{C(\overline{f}:\overline{v})} D' \end{array}}{D \in \text{Select}(C(\overline{f} = \overline{v}))}$$

$$(\textsc{Select-Asymm})$$

Figure 5.11: Variations of dispatch

all declarations of the same class. The problem is that such a requirement is very strict, i.e., it rejects programs that fulfill property 5.2.1. We could try to borrow less restrictive solutions that are available for symmetric dispatch for methods [CGL92, CL94, MC99]. It is, however, not straightforward, because the relations that are easy to compute in a simple type system, where types correspond to plain classes, may be difficult to compute or even undecidable in a type system with dependent types. For example, it is difficult to determine if two types are overlapping, i.e., if they have a common (valid) value in the program. It is also difficult to check if a type is an upper bound of an intersection of two other types (i.e., it is a supertype of all their common subtypes). Definition of tolerant constructive well-formedness rules that guarantee the property 5.2.1 for symmetric dispatch is a topic for future research.

The general principle of asymmetric dispatch is to define an additional ordering relation that supplements the order of the overriding relation. The ordering relation $D <_t D'$ says that D precedes D' when they are incomparable by overriding relation. In the general case, the ordering relation may be not absolute, but relative to t - the type being matched. Rule Select-Assym in Fig. 5.11 defines the general principle of asymmetric dispatch: the matching declarations are first filtered by the overriding relation, then from the declarations that are not overridden, we select the one that is the smallest by the additional ordering relation.

The supplementary ordering relation can be defined in different ways. The simplest way is to define it explicitly by assigning order numbers to declarations in the program or by having precedence declarations similar to the precedence declarations of aspects in AspectJ [KHH$^+$01]. The order can also be determined implicitly, by considering the order of class declarations, the order of field declarations and the order of parent classes in the **extends** clause. For example, one could consider extending the mixin linearization algorithm of the *vc* [EOC06] calculus for multiple fields.

In vc^n, one could also consider new variations on dispatch that would not make sense for multimethods. For example, classes that only contribute structure (such as new supertypes or fields) but not behavior (only default constructor) could be excluded from the dispatch algorithm. In vc^n, all declarations of a class must have the same set of fields, but if we would add self-initializing fields or mutable fields that do not need to be initialized via constructor parameters, then a class declaration could extend the object layout without interfering with the dispatch mechanism.

5.3 DC_C Calculus

The semantics of dependent classes, defined by the vc^n calculus, does not support abstract classes and methods and is abstracted from specific dispatch strategies. Extending this calculus with support for the new features appeared to be more difficult than we expected.

The strategy for checking method completeness and uniqueness, which will be described in Sec. 5.5.4.2, relies on the possibility to intersect types. For example, during uniqueness analysis it is important to determine if two implementations of a method can match for a particular concrete set of parameters and thus are potentially ambiguous. Such check, in principle, tests disjointness of the types corresponding to the signatures of the methods. Furthermore, we also need to compute the intersection of the two types, representing the potentially ambiguous parameter sets, in order to check if these cases are completely resolved by other, more specific implementations of the method.

The extension of vc^n with intersection types introduces new shapes of types, such as intersections of multiple classes or intersections of class types and paths. Such new shapes of types invalidate the assumptions of the algorithmic design of the calculus. In particular, intersection types introduce alternative branches for path normalization defined in Sec. 5.2.3. For example, if the bound of a path p is the type $p' \wedge p''$, where $\wedge$ denotes type intersection, normalization of p can proceed both to p' and to p'', and thus is not unique. Moreover, according to the transitivity of type equivalence, since p is equivalent to both p' and p'', we must also conclude the equivalence of p' and p''. The problem is that this equivalence would not be derived through normalization of p' and

p''. These problems also affect subtyping relationship in vc^n, because it also relies on uniqueness of path bounds and their normal forms.

The search for a cannonical representation of types in the presence of type intersection has lead to the representation of types as intersections of the corresponding sets of primitive constraints. For the types of dependent classes defined in vc^n, two kinds of primitive constraints were sufficient: one for specifying a class of a certain path, and another for specifying equivalence between two paths.

We found that the representation of types as sets of constraints leads to a simpler and more intuitive calculus. In this section we present a new DC_C calculus, encoding dependent classes by means of a constraint system. The constraints are intensively used both in the static and the dynamic semantics. The pivotal relation in the calculus is constraint entailment, which replaces the main relations of vc^n: type equivalence, subtyping, static and dynamic dispatch. Specific type substitution relationships of the vc^n calculus, can be expressed by simple variable renaming and union of constraint sets.

The primitive constraints and constraint entailment axioms of DC_C were selected so that they are sufficient to encode the types of vc^n. Although the core machinery of the calculus is more general and could be relatively easily extended with new kinds of constraints and entailment rules, the major obstacle for adding new features is ensuring decidability of the necessary operations on constraints. The current decidability proof, given in Sec. 5.5, is written specifically for the selected constraint entailment axioms and well-formedness rules of the calculus.

DC_C extends the semantics of vc^n with support for abstract methods and classes, and symmetric method dispatch. In DC_C we gave up the unification of methods and classes of vc^n, because method and class declarations play a totally different role in uniqueness and completeness checking: uniqueness and completeness is checked for implementations of methods, while class constructors determine the set of available concrete parameters that must be considered during these checks.

DC_C also has a different runtime structure than vc^n. In vc^n runtime execution was defined as a context-free normalization of terms to values, described as nested record structures. In DC_C we introduce a runtime structure closer to the one of object-oriented systems with explicit object identities and relationships between objects based on these identities. The graph of interconnected objects can be represented as a heap structure. In DC_C an expression evaluates to an identifier pointing to an object in the heap.

The runtime structure of DC_C is analogous to the one used for νObj [OCRZ03], a calculus with nominal dependent types. A heap structure was also used in definition of vc calculus [EOC06]. The advantage of a heap structure is that it preserves object identities and enables shared references to objects. Such structure is more natural for

$$
\begin{array}{lll}
P & \in \text{Program} & ::= \overline{D} \\
D & \in \text{Decl} & ::= C(x.\ \overline{a}) \mid \forall x.\ \overline{a} \Rightarrow a \\
 & & \quad \mid m(x.\ \overline{a}) : t \mid m(x.\ \overline{a}) : t := e \\
t & \in \text{Type} & ::= [x.\ \overline{a}] \\
a, b, c & \in \text{Constr} & ::= p \equiv p \mid p :: C \mid p.\mathbf{cls} \equiv C \\
p, q & \in \text{Path} & ::= x \mid p.f \\
e & \in \text{Expr} & ::= x \mid e.f \mid \mathbf{new}\ C(\overline{f} \equiv \overline{e}) \mid m(e)
\end{array}
\qquad
\begin{array}{lll}
x, y, z & - & \text{variable names} \\
f & - & \text{field names} \\
C & - & \text{class names} \\
m & - & \text{method names}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{MType}(m, x, y) & = \left\{ \langle \overline{a}; \overline{b} \rangle \mid (m(x.\overline{a}) : [y.\ \overline{b}] \ldots) \in P \right\} \\
\mathit{MImpl}(m, x) & = \left\{ \langle \overline{a}; e \rangle \mid (m(x.\overline{a}) : [y.\ \overline{b}] := e) \in P \right\}
\end{array}
$$

Figure 5.12: Syntax

nominal dependent types, because it gives a direct interpretation of equivalent paths: two paths are equivalent if they point to *the same* object at runtime. In vc^n we had to approximate object identity by structural equality of values: in vc^n two paths are equivalent if they evaluate to *structurally equal* values at runtime.

In addition to that, a heap structure is more convenient when working with constraints. As we will see in Sec. 5.2.2, each heap can be easily translated to a set of constraints completely describing its objects and their relationships. This enables using the constraint system for dynamic dispatch and expression typing.

The remainder of the section gives a definition of the DC_C calculus. Sec. 5.3.1 defines the syntax of the calculus, and explains it relation to the syntax of the DepJ language. Sec. 5.3.2 presents the constraint system of DC_C, which is the central piece of DC_C used both in its dynamic semantics, which is defined in Sec. 5.3.3, and its type system, which is the subject of Sec. 5.3.4

5.3.1 Syntax

The syntax of DC_C is defined in Fig. 5.12. Types in DC_C are basically lists of constraints to be satisfied by their instances. A type (t) is of the form $[x.\ \overline{a}]$, where x is a bound variable, and $\overline{a}$ is a list of constraints on x. An object belongs to a type, if it fulfills all its constraints. Constraints can only use path expressions, (p), i.e., variables and navigation over fields starting from variables.

The types of dependent classes can be encoded by constraints expressing equivalence between paths $p \equiv q$ and constraints requiring that a path refers to an object of a certain class. The order of p and q is ignored. Two paths are considered equivalent if it is known that they always refer to the same object at runtime. Constraints of the form $p :: C$ specify that p refers to an instance of C. A stronger constraint is $p.\mathbf{cls} \equiv C$, which requires that p refers to an object instantiated by a constructor of C, thus excluding indirect instances of C inferred by inheritance rules. The latter type of constraints is introduced for rather internal purposes: as we will see in type checking rules, they are necessary in the analysis of completeness and uniqueness of methods.

A program (P) consists of a list of declarations (D) that can be constructor declarations, abstract method declarations, method implementations and constraint entailment rules. The constructor declarations constrain the objects that can be constructed: a constructor $C(x.\ \overline{a})$ allows creating instances of C that satisfy constraints $\overline{a}$. The constraint entailment rules are used in the derivation of constrained entailment as will be explained later in this section.

A declaration of the form $m(x.\ \overline{a}) : t := e$ describes a method that expects an argument x satisfying constraints $\overline{a}$ and returns an object that is computed by the expression e of type t. The return type $t = [y.\ \overline{b}]$ can depend on the type of the argument, because the argument x can appear in the constraints $\overline{b}$ that must be fulfilled by the returned object. Methods declared without implementing expressions are considered as abstract and are used only for type checking. DC_C supports only methods with one parameter, because methods with multiple parameters can be easily encoded as methods accepting objects with multiple fields.

DC_C supports expressions of four types: variable access, field access, object construction, and method call. The calculus is functional, and thus does not support field assignment. The lower section of Fig. 5.12 defines two auxiliary functions for more convenient representation of the information about method typing and implementation.

In the definitions, we will implicitly assume equivalence over α-renaming of bound variables: the variable of a method argument, the bound variable of a constructor, the bound variable of a type, and the bound variable of a constraint entailment rule.

The main difference between the syntax of the DepJ language and the syntax of DC_C is in the representation of types. A DepJ type t is translated to $[x.\ [\![x : t]\!]]$ in DC_C, where $[\![p : t]\!]$ is a set of constraints encoding the fact that path p is of type t. The encoding is defined by the following rules[6]:

$$[\![p : p']\!] = (p' \equiv p)$$
$$[\![p : C(\overline{f} : \overline{t})]\!] = (p :: C, [\![p.f_1 : t_1]\!], \ldots, [\![p.f_{|\overline{f}|} : t_{|\overline{f}|}]\!])$$

Also unlike DepJ, the DC_C calculus does not provide class declarations. Hence classes must be decomposed into a set of more primitive declarations:

[6] If p' is of type p then the paths are equivalent. If p is of type $C(\overline{f} : \overline{t})$ then p is an instance of C, and for each field f_i path $p.f_i$ is of type t_i.

```
abstract class Nat {
    abstract Nat prev();
}
class Zero extends Nat {
    Nat prev() { return new Zero(); }
}
class Succ(Nat p) extends Nat {
    Nat prev() { return this.p; }
}
```

$$
\begin{array}{l}
\texttt{Zero}(\texttt{x}.\ \epsilon) \\
\forall \texttt{x}.\ \texttt{x} :: \texttt{Zero} \Rightarrow \texttt{x} :: \texttt{Nat} \\
\texttt{Succ}(\texttt{x}.\ \texttt{x.p} :: \texttt{Nat}) \\
\forall \texttt{x}.\ \texttt{x} :: \texttt{Succ}, \texttt{x.p} :: \texttt{Nat} \Rightarrow \texttt{x} :: \texttt{Nat} \\
\texttt{prev}(\texttt{x}.\ \texttt{x} :: \texttt{Nat}) : [\texttt{y}.\ \texttt{y} :: \texttt{Nat}] \\
\texttt{prev}(\texttt{x}.\ \texttt{x} :: \texttt{Zero}) : [\texttt{y}.\ \texttt{y} :: \texttt{Nat}] := \textbf{new}\ \texttt{Zero}() \\
\texttt{prev}(\texttt{x}.\ \texttt{x} :: \texttt{Succ}, \texttt{x.p} :: \texttt{Nat}) : [\texttt{y}.\ \texttt{y} :: \texttt{Nat}] := \texttt{x.p}
\end{array}
$$

Figure 5.13: Defining natural numbers in DepJ and in DC_C

- Each non-abstract class declaration corresponds to a *constructor declaration* with constraints corresponding to the types of the class parameters.
- The *inheritance relations* can be encoded as constraint entailment rules. The fact that class C($\overline{f}$:$\overline{t}$) inherits from C' can be encoded as the constraint entailment rule $\forall x.\ \overline{a} \Rightarrow x :: C'$, where $\overline{a} = [\![x : C(\overline{f} : \overline{t})]\!]$.
- All *method declarations* can be externalized by introducing an extra parameter for the receiver object and using the name of this parameter instead of the reference to **this**.

Figure 5.13 demonstrates translation of the DepJ classes for natural numbers to the DC_C calculus. We have constructors for the two concrete classes Zero and Succ(p: Nat). The inheritance of Zero and Succ(p: Nat) from Nat is encoded by the corresponding constraint entailment rules. The method prev is externalized and takes the enclosing object as a parameter[7]. Types Nat and Zero are translated to types $[\texttt{x}.\ \texttt{x} :: \texttt{Nat}]$ and $[\texttt{x}.\ \texttt{x} :: \texttt{Zero}]$ correspondingly. The types of fields are translated to additional constraints. For example, Succ(p: Nat) is encoded as $[\texttt{x}.\ \texttt{x} :: \texttt{Succ}, \texttt{x.p} :: \texttt{Nat}]$.

5.3.2 Constraint System

The constraint system of DC_C is defined in the style of the sequent calculus [GTL89]: the sequent $\overline{a} \vdash a$ is interpreted as constraint entailment: constraints $\overline{a}$ entail constraint a. Figure 5.14 specifies the rules for derivation of the sequents. Rules C-Ident and C-Cut are standard rules of the intuitionistic sequent calculus, the rest are specific to the programming language. We also implicitly assume the standard structural rules of sequent calculus that allow permutations, weakening, and contraction of the context. Usually we will refer to the constraints on the left-hand side of $\vdash$ as the context, and the

[7]In the case when the method already has other parameters, the original parameters must be encoded as fields of the single method parameter.

$$a \vdash a \quad (\text{C-IDENT}) \qquad \frac{\overline{a} \vdash c \qquad \overline{a'}, c \vdash b}{\overline{a}, \overline{a'} \vdash b} \quad (\text{C-CUT})$$

$$\epsilon \vdash p \equiv p \quad (\text{C-REFL}) \qquad \frac{\overline{a} \vdash a_{\{x \mapsto p\}} \qquad \overline{a} \vdash p' \equiv p}{\overline{a} \vdash a_{\{x \mapsto p'\}}} \quad (\text{C-SUBST})$$

$$\frac{\overline{a} \vdash p.\mathbf{cls} \equiv C}{\overline{a} \vdash p :: C} \quad (\text{C-CLASS}) \qquad \frac{(\forall x.\ \overline{a} \Rightarrow a) \in P \qquad \overline{b} \vdash \overline{a}_{\{x \mapsto p\}}}{\overline{b} \vdash a_{\{x \mapsto p\}}} \quad (\text{C-PROG})$$

Figure 5.14: Constraint entailment

constraints on the right-hand side as constraints entailed by the context. Note, that we use notation $\overline{a} \vdash \overline{b}$ differently than in the sequent calculus: in our case it is a shortcut for a list of judgments $\overline{a} \vdash b_1 \ \ldots \ \overline{a} \vdash b_n$ and, thus, means that all b_i are entailed by $\overline{a}$, while in the sequent calculus this would mean that at least one of b_i is entailed by $\overline{a}$.

Rules C-REFL and C-SUBST specify the properties of path equivalence, while the other typical rules of equivalence such as symmetry and transitivity can be derived from these rules. The rule C-REFL establishes reflexivity of path equivalence. The rule C-SUBST tells that a path can be substituted for an equivalent path at any position of any other constraint. Using C-SUBT, we can e.g., derive $y.f' :: C$ from $x.f.f' :: C$ and $y \equiv x.f$, because $(y.f' :: C) = (x.f' :: C_{\{x \mapsto y\}})$ and $(x.f.f' :: C) = (x.f' :: C_{\{x \mapsto x.f\}})$.

The rule C-PROG makes it possible to specify new axioms for the constraint system in programs. The universally quantified variable x in a rule declaration in the program can be instantiated with any path. Such declarations are similar to Horn clauses of Prolog and unrestricted usage of them can quickly lead to an undecidable constraint system, and thus undecidable type checking. Therefore, DC_C supports only a very restricted form of the rule, which has enough expressive power to express inheritance declarations between dependent classes: the constraint at the right side of implication must be $x :: C$, where x is the bound variable of the rule. The restrictions are specified by the well-formedness rule WF-RD in Fig. 5.17.

The last rule C-CLASS tells that $p :: C$ is weaker than $p.\mathbf{cls} \equiv C$, i.e., a direct instance of a class is an instance of a class.

The constraint entailment relation plays a pivotal role in DC_C. For example, we do not need any additional relationship for expressing subtyping, because t is a subtype of t', if constraints of t entail the constraints of t', i.e., given $t = [x.\ \overline{a}]$ and $t' = [x.\ \overline{a'}]$, t is a subtype of t' iff $\overline{a} \vdash \overline{a'}$.

$$\begin{array}{lll} o & ::= & \langle C; \overline{f} \equiv \overline{x} \rangle \\ h & ::= & \overline{x} \mapsto \overline{o} \quad (x_i \text{ distinct}) \end{array} \qquad \begin{array}{lll} OC(x,o) & = (x.\mathbf{cls} \equiv C, x.\overline{f} \equiv \overline{x}) & \text{where } \ o = \langle C; \overline{f} \equiv \overline{x} \rangle \\ HC(h) & = \bigcup\limits_i OC(x_i, o_i) & \text{where } \ h = \overline{x} \mapsto \overline{o} \end{array}$$

$$\frac{\begin{array}{cc} x \notin dom(h) & o = \langle C; \overline{f} \equiv \overline{x} \rangle \\ C(x.\ \overline{b}) \in P & HC(h), OC(x,o) \vdash \overline{b} \end{array}}{\langle h; \mathbf{new}\ C(\overline{f} \equiv \overline{x}) \rangle \to \langle h, x \mapsto o; x \rangle} \text{(R-New)} \qquad \frac{(x.f \equiv y) \in HC(h)}{\langle h; x.f \rangle \to \langle h; y \rangle} \text{(R-Field)}$$

$$\frac{\begin{array}{c} S = \{ \ \langle \overline{a}; e \rangle \mid \langle \overline{a}; e \rangle \in MImpl(m,x) \wedge HC(h) \vdash \overline{a} \ \} \\ \langle \overline{a}; e \rangle \in S \qquad \forall \langle \overline{a'}; e' \rangle \in S.\ (e' \neq e) \longrightarrow \ (\overline{a'} \vdash \overline{a}) \wedge \neg(\overline{a} \vdash \overline{a'}) \end{array}}{\langle h; m(x) \rangle \to \langle h; e \rangle} \text{(R-Call)}$$

$$\frac{\langle h; e \rangle \to \langle h'; e' \rangle}{\langle h; e.f \rangle \to \langle h'; e'.f \rangle} \text{(RC-Field)} \qquad \frac{\langle h; e \rangle \to \langle h'; e' \rangle}{\langle h; m(e) \rangle \to \langle h'; m(e') \rangle} \text{(RC-Call)}$$

$$\frac{\langle h; e \rangle \to \langle h'; e' \rangle}{\langle h; \mathbf{new}\ C(\overline{f} \equiv \overline{x}, f \equiv e, \overline{f'} \equiv \overline{e'}) \rangle \to \langle h'; \mathbf{new}\ C(\overline{f} \equiv \overline{x}, f \equiv e', \overline{f'} \equiv \overline{e'}) \rangle} \text{(RC-New)}$$

Figure 5.15: Operational semantics

5.3.3 Operational Semantics

The operational semantics is defined by a small-step reduction relation on a pair of a heap and an expression. The structure of heaps is described on the top left corner of Fig. 5.15. A heap (h) is a list of mappings from variables to objects (o), and each object is specified by a class and a list of the values of its fields, which are again considered as references in the heap. Since we require the heap to be functional, $h(x)$ denotes the object of h referenced by x.

To employ the constraint system in the operational semantics, we define a function HC (top right corner of Fig 5.15), which takes a heap and gives the constraints satisfied by all variables of the heap. The function OC converts a definition of an object to a list of constraints on a given variable. Now, using the constraint system we can tell which constraints are satisfied by a heap: a is satisfied by a heap h, if $HC(h) \vdash a$ holds.

Since values represent references in the heap, the latter are considered as the values of DC_C, and the result of evaluation must be a pair of a heap and a variable. In fact, evaluation can be seen as a process of moving information from the expression part to the heap part. The heap is monotonic in the sense that during reduction only new objects can be added to the heap, while existing objects remain unchanged.

The reduction rules are defined in Fig. 5.15. Following the small-step style, the congruence rules RC-FIELD, RC-NEW and RC-CALL propagate reduction to subexpressions, and only when all subexpressions of an expression are reduced to normal forms, which in our case are variables, one of the computation rules can be applied. There is one computation rule for each of three possible types of redexes: R-FIELD for $x.f$, R-NEW for **new** $C(\overline{f} \equiv \overline{x})$, and R-CALL for $m(x)$.

Field access $x.f$ is reduced to the value of $x.f$ in the heap, i.e., heap constraints must include $x.f \equiv y$. Otherwise, x does not have field f in the heap.

Object construction **new** $C(\overline{f} \equiv \overline{x})$ is reduced by introducing a fresh variable, representing the new object, and extending the heap with an object with C as it class and $\overline{x}$ as the values of its fields. Further, it is checked that a program contains a constructor of C with constraints satisfied by the new object. This is necessary for constraining the set of possible objects. We can ensure that certain classes are abstract, i.e., they do not have direct instances, while for the instances of concrete classes we can ensure presence of certain fields and impose constraints on them.

A method call $m(x)$ is reduced to the expression of the most specific applicable method implementation. Both applicability of the implementations of a method and selection of the most specific of them are determined by constraints. A method declaration is applicable if its argument constraints are entailed by the constraints of the context[8]. A method declaration D is more specific than D', if the argument constraints of D entail the argument constraints of D', but not the other way around.

5.3.4 Type Checking

Type assignment is defined in Fig. 5.16. The context of type assignment is described by a list of constraints, which provide information about variables that occur in the expression. The typing rules ensure that all free variables of the assigned types also appear in the context. An expression type can be seen as a collection of constraints that will hold for all possible values of the expression at runtime, if the runtime environment satisfies the constraints of the context. Since there can be different constraints satisfied by an expression, the relation does not guarantee unique type assignment and the rule T-SUB explicitly enables weakening the type of an expression.

The type of a variable x (rule T-VAR) asserts equivalence to that variable. Further, we check that the variable is available in the context by checking that $x :: C$ is derivable

[8]Note that we implicitly employ α-renaming of the method to unify the the formal argument with x.

$$\frac{\begin{array}{c}\overline{c} \vdash e : [x.\ \overline{a}] \qquad \overline{c}, \overline{a} \vdash x.f :: C \\ \overline{c}, \overline{a}, x.f \equiv y \vdash \overline{b} \qquad x \notin FV(\overline{b})\end{array}}{\overline{c} \vdash e.f : [y.\ \overline{b}]} \ (\text{T-Field})$$

$$\frac{\overline{c} \vdash x :: C}{\overline{c} \vdash x : [y.\ y \equiv x]} \ (\text{T-Var})$$

$$\frac{\overline{c} \vdash e : [x.\ \overline{a'}] \qquad \overline{c}, \overline{a'} \vdash \overline{a}}{\overline{c} \vdash e : [x.\ \overline{a}]} \ (\text{T-Sub})$$

$$\frac{\begin{array}{c}\overline{c} \vdash e : [x.\ \overline{a}] \\ \langle \overline{a'}; \overline{b} \rangle \in \mathit{MType}(m, x, y) \qquad \overline{c}, \overline{a} \vdash \overline{a'} \\ \overline{c}, \overline{a}, \overline{b} \vdash \overline{b'} \qquad x \notin FV(\overline{b'})\end{array}}{\overline{c} \vdash m(e) : [y.\ \overline{b'}]} \ (\text{T-Call})$$

$$\frac{\begin{array}{c}\forall i.\ \overline{c} \vdash e_i : [x_i.\ \overline{a_i}] \\ \overline{b} = (x.\mathbf{cls} \equiv C), \bigcup_i \overline{a_i}_{\{x_i \mapsto x.f_i\}} \\ C(x.\ \overline{b'}) \in P \qquad \overline{c}, \overline{b} \vdash \overline{b'}\end{array}}{\overline{c} \vdash \mathbf{new}\ C(\overline{f} \equiv \overline{e}) : [x.\ \overline{b}]} \ (\text{T-New})$$

Figure 5.16: Type assignment

from some C. Checking for $p :: C$ is a reliable way of ensuring that path p will be valid at runtime, because during evaluation $p :: C$ can be derived only from a heap constraint $x.\mathbf{cls} \equiv C'$ for some x equivalent to p, which ensures that p evaluates to x.

The constraints of a field access $e.f$ (rule T-Field) are the constraints on $x.f$ entailed by the type of e, where x is the bound variable of the type of e. To eliminate x we take constraints entailed by the type of e and $y \equiv x.f$ that are free of x, and y is then used as the bound variable of the type of $e.f$. Availability of field f for e at runtime is ensured by checking whether $x.f :: C$ is derivable from the type of e.

Typing the method call $m(e)$ (rule T-Call) checks that there is a declaration of m, whose parameter constraints are entailed by the constraints of the type of e. The rule shows dependent typing of method calls: the type constraints of $m(e)$ are derived both from the declared return type of m and the type of the argument e. Since the argument variable x does not appear in the context it must be eliminated, so the entailed constraints of return types and the argument types free of x are taken.

The type of an object construction **new** $C(\overline{f} \equiv \overline{e})$ (rule T-New) consists of a constraint declaring that C is the class of the object and the constraints of its fields, which are taken from the types of the expressions assigned to fields. Further, it is checked that the new object fulfills the constraints of at least one of the constructors of class C.

Type checking a program and its declarations is defined in Fig. 5.17. For a declaration to be well-formed, all its used variables must be bound. Rule WF-MI additionally ensures that an expression implementing a method satisfies the constraints of the declared return type. Rule WF-RD limits the constraint derivation rules that can be specified in a program: only constraints of the form $x :: C$ can be derived if some other class of x is

$$\frac{FV(\overline{a}) = \{x\}}{\text{wf } C(x.\ \overline{a})}\ (\text{WF-CD}) \qquad \frac{FV(\overline{a}) = \{x\} \qquad FV(\overline{b}) = \{x, y\}}{\text{wf } (m(x.\ \overline{a}) : [y.\ \overline{b}])}\ (\text{WF-MS})$$

$$\frac{FV(\overline{a}) = \{x\} \qquad x :: C' \in \overline{a}}{\text{wf } (\forall x.\ \overline{a} \Rightarrow x :: C)}\ (\text{WF-RD}) \qquad \frac{FV(\overline{a}) = \{x\} \qquad FV(\overline{b}) = \{x, y\} \qquad \overline{a} \vdash e : [y.\ \overline{b}]}{\text{wf } (m(x.\ \overline{a}) : [y.\ \overline{b}] := e)}\ (\text{WF-MI})$$

$$\frac{\begin{array}{c}\forall D \in P.\ \text{wf } D \\ \forall m.\ \forall \langle \overline{a}; \overline{b} \rangle, \langle \overline{a'}; \overline{b'} \rangle \in \mathit{MType}(m, x, y).\ \ \overline{b} = \overline{b'} \\ \forall m.\ \forall \langle \overline{a}; \overline{b} \rangle \in \mathit{MType}(m, x, y).\ \ \text{complete}(m, [x.\ \overline{a}]) \\ \forall m.\ \text{unique}(m)\end{array}}{\text{wf } P}\ (\text{WF-P}\textsc{rog})$$

Figure 5.17: Type checking

known. Such rules are sufficient for encoding inheritance between dependent classes. This strict restriction limits the expresive power of the calculus, but we need it for proving the decidability of the contraint system in Sec. 5.5.

Well-formedness of a program requires well-formedness of all its declarations, completeness and uniqueness of method implementation, and that all declarations of a method have the same return types.

"Concrete arguments" in DC_C are variables of well-formed heaps, i.e., heaps that can be created during evaluation. More precisely, a well-formed heap must satisfy the following requirements (upper part of Fig. 5.18): (1) all objects in the heap must comply to at least one of the constructors in the program, (2) all variables referenced by fields are defined in the heap. Now we can define completeness and uniqueness of method implementations in terms of well-formed heaps (lower part of Fig. 5.18). For a method m to be complete on an argument type $[x.\ \overline{a}]$, an applicable implementation of m must exist for each well-formed heap that satisfies $\overline{a}$ (rule Complete). For a method m to be unique, the unique most specific implementation of m must exist in every set of implementations of m applicable on a well-formed heap (rule Unique). These rules define the meaning of method uniqueness and completeness in the calculus, but cannot be directly implemented, because we cannot generate all possible heaps. In Sec. 5.5.4, we will design constructive algorithms for method checking that satisfy these definitions.

$$\frac{\begin{array}{c}\forall x \in \mathit{dom}(h).\ \ \exists C(x.\ \overline{a}) \in P.\ \ \mathit{HC}(h) \vdash \overline{a} \\ \forall\ (x.f \equiv y)\ \in\ \mathit{HC}(h).\ \ y \in \mathit{dom}(h)\end{array}}{\text{heap } h} \quad \text{(WF-Heap)}$$

$$\frac{\forall h.\ \ \text{heap } h\ \wedge\ \mathit{HC}(h) \vdash \overline{a}\ \longrightarrow\ (\exists\langle\overline{b}; e\rangle \in \mathit{MImpl}(m, x).\ \ \mathit{HC}(h) \vdash \overline{b})}{\text{complete}(m, [x.\ \overline{a}])} \quad \text{(Complete)}$$

$$\frac{\left[\begin{array}{l}\forall h, S.\ \ \text{heap } h\ \wedge\ S = \left\{\ \langle\overline{b}; e\rangle \mid \langle\overline{b}; e\rangle \in \mathit{MImpl}(m, x)\ \wedge\ \mathit{HC}(h) \vdash \overline{b}\ \right\}\ \wedge\ S \neq \varnothing \\ \quad\longrightarrow\ (\exists\langle\overline{b}; e\rangle \in S.\ \ \forall\langle\overline{b'}; e'\rangle \in S.\ \ (e \neq e')\ \longrightarrow\ (\overline{b} \vdash \overline{b'})\ \wedge\ \neg(\overline{b'} \vdash \overline{b}))\end{array}\right]}{\text{unique}(m)}$$

(Unique)

Figure 5.18: Well-formed heaps. Completeness and uniqueness

5.4 Soundness of DC_C

Soundness of DC_C calculus is defined in terms of progress and preservation properties: reduction of a well-formed pair of a heap and an expression does not get stuck (progress), and the type of the expression is preserved (preservation). In DC_C calculus the preservation property also implies that the resulting heap is well-formed and it preserves the constraints of the original heap.

Theorem 5.4.1 (Soundness)**.** *If* wf P *and* heap h *and* $\mathit{HC}(h) \vdash e : t$*, then either reduction of* e *does not terminate, or there exist* x *and* h'*, such that* $\langle h; e\rangle \rightarrow \langle h'; x\rangle$ *and* $\mathit{HC}(h') \vdash\ \mathit{HC}(h)$ *and* heap h' *and* $\mathit{HC}(h') \vdash x : t$*.*

Proof. Follows directly from lemmas 5.4.7, 5.4.11, 5.4.5 and 5.4.6. □

The remainder of the section is divided in two parts. In Sec. 5.4.1 we prove preservation of expression type and heap constraints in reduction. Then in Sec. 5.4.2 we prove the progress property.

5.4.1 Type Preservation

Since the constraint entailment relationship provides subtyping in the DC_C calculus, it must have the properties that we would otherwise expect from a subtyping relationship: transitivity, context weakening, variable substitution. Context weakening does not need to be proved, because it is directly supported by the structural rules of the constraint entailment relationship.

Transitivity of constraint entailment is ensured by the C-Cut rule, but it is not very convenient for a direct use, so we will start by proving a more convenient lemma for transitivity.

Lemma 5.4.1. *If* $\overline{c}, \overline{a} \vdash \overline{a'}$ *and* $\overline{c}, \overline{a'} \vdash \overline{a''}$ *then* $\overline{c}, \overline{a} \vdash \overline{a''}$

Proof. By induction on the length of $\overline{a'}$ for any $\overline{c}$.

Case $\overline{a'} = \epsilon$*:* By structural rules.

Case $\overline{a'} = \overline{b}, b$*:* Assuming (1): $\overline{c}, \overline{a} \vdash \overline{b}, b$ and (2): $\overline{c}, \overline{b}, b \vdash \overline{a''}$ and the induction hypothesis show $\overline{c}, \overline{a} \vdash \overline{a''}$.

The induction hypothesis states that for all $\overline{c'}$ such that $\overline{c'}, \overline{a} \vdash \overline{b}$ and $\overline{c'}, \overline{b} \vdash \overline{a''}$ holds $\overline{c'}, \overline{a} \vdash \overline{a''}$.

From (1) and (2) we have $\overline{c}, b, \overline{a} \vdash \overline{b}$ and $\overline{c}, b, \overline{b} \vdash \overline{a''}$.

Now we can apply the induction hypothesis with $\overline{c'} = \overline{c}, b$ to obtain $\overline{c}, b, \overline{a} \vdash \overline{a''}$. From (1) we also have $\overline{c}, \overline{a} \vdash b$. Then the result follows by C-Cut and the structural rules.

□

Now we will prove that constraint entailment is preserved if all occurrences of a variable are substituted by an arbitrary path. This is the main property of the constraint system for ensuring type preservation.

Lemma 5.4.2. *If* wf P *and* $\overline{c} \vdash a$ *then* $\overline{c}_{\{x \mapsto p\}} \vdash a_{\{x \mapsto p\}}$

Proof. By induction on derivation of $\overline{c} \vdash a$.

Case C-Subst*:* Assuming (1): $\overline{c}_{\{x \mapsto p\}} \vdash (a_{\{y \mapsto p'\}})_{\{x \mapsto p\}}$ and (2): $\overline{c}_{\{x \mapsto p\}} \vdash p''_{\{x \mapsto p\}} \equiv p'_{\{x \mapsto p\}}$ show $\overline{c}_{\{x \mapsto p\}} \vdash (a_{\{y \mapsto p''\}})_{\{x \mapsto p\}}$.

Let's take a fresh variable z. In particular we ensure $z \neq x$ and $z \notin FV(p, p', p'')$.

Now, if $b = a_{\{y \mapsto z\}}$, then $a_{\{y \mapsto p'\}} = b_{\{z \mapsto p'\}}$, which gives (3): $(a_{\{y \mapsto p'\}})_{\{x \mapsto p\}} = (b_{\{z \mapsto p'\}})_{\{x \mapsto p\}}$.

Our next goal is to exchange the order of replacing variables z and x. For this consider how various occurrences of x and z are replaced in $(b_{\{z \mapsto p'\}})_{\{x \mapsto p\}}$. Variable x can occur in b, p' and p. Since $z \neq x$, the occurences of x in b and p' are replaced by p, and the occurences of x in p remain unchanged. Variable z can occur only in

b and these occurences are replaced by $p'_{\{x\mapsto p\}}$. Equivalent replacements take place in $(b_{\{x\mapsto p\}})_{\{z\mapsto p'_{\{x\mapsto p\}}\}}$.

Thus $(b_{\{z\mapsto p'\}})_{\{x\mapsto p\}} = (b_{\{x\mapsto p\}})_{\{z\mapsto p'_{\{x\mapsto p\}}\}}$, which with (3) gives (4): $(a_{\{y\mapsto p'\}})_{\{x\mapsto p\}} = (b_{\{x\mapsto p\}})_{\{z\mapsto p'_{\{x\mapsto p\}}\}}$.

Analogously, we can obtain (5): $(a_{\{y\mapsto p''\}})_{\{x\mapsto p\}} = (b_{\{x\mapsto p\}})_{\{z\mapsto p''_{\{x\mapsto p\}}\}}$.

From (1) and (4) we have $\overline{c}_{\{x\mapsto p\}} \vdash (b_{\{x\mapsto p\}})_{\{z\mapsto p'_{\{x\mapsto p\}}\}}$. Then C-Subst with (2) gives $\overline{c}_{\{x\mapsto p\}} \vdash (b_{\{x\mapsto p\}})_{\{z\mapsto p''_{\{x\mapsto p\}}\}}$, which proves the case because of (5).

Case C-Prog*:* Assuming $\overline{c}_{\{x\mapsto p\}} \vdash (\overline{a}_{\{y\mapsto p'\}})_{\{x\mapsto p\}}$ and $(\forall y.\ \overline{a} \Rightarrow a) \in P$ show $\overline{c}_{\{x\mapsto p\}} \vdash (a_{\{y\mapsto p'\}})_{\{x\mapsto p\}}$.

Because of wf P we know $FV(\overline{a}, a) = \{y\}$ by WF-RD.

Thus $x = y$, or x cannot occur in $\overline{a}$ and a. In both cases we have $(\overline{a}_{\{y\mapsto p'\}})_{\{x\mapsto p\}} = \overline{a}_{\{y\mapsto p'_{\{x\mapsto p\}}\}}$ and $(a_{\{y\mapsto p'\}})_{\{x\mapsto p\}} = a_{\{y\mapsto p'_{\{x\mapsto p\}}\}}$.

Now the result follows from by taking $p'_{\{x\mapsto p\}}$ as p in C-Prog.

Other cases: Proof is straightforward.

□

Next, we need two simple properties of expression typing: one to allow context weakening, and defining the property of variable type. Note that we don't need to prove the usual substitution lemma for expression typing, which tells that expression type is preserved after substituting a variable with an expression of a correct type. It is very likely that some form of this property can be proved for the calculus, but is not necessary for proving type preservation in reduction, because none of the reduction rules substitute variables.

Lemma 5.4.3 (Weaken Typing Context)**.** *If* $\overline{c} \vdash e : t$ *and* $\overline{c'} \vdash \overline{c}$ *then* $\overline{c'} \vdash e : t$

Proof. Straightforward by induction on typing derivation using the lemma 5.4.1 to weaken the context of constraint entailment. □

Lemma 5.4.4 (Variable Typing)**.** *If* $\overline{c} \vdash x : [x'.\ \overline{a}]$ *then* $\overline{c} \vdash \overline{a}_{\{x'\mapsto x\}}$

Proof. Since variable typing can be derived only by T-Var and T-Sub, we get $\overline{c} \vdash x : [x'.\ x' \equiv x]$ and (1): $\overline{c}, x' \equiv x \vdash \overline{a}$. By lemma 5.4.2 we can rename all occurrences of x' to x in (1). Since we can assume that x' is not in $\overline{c}$, we get $\overline{c}, (x' \equiv x') \vdash \overline{a}_{\{x'\mapsto x\}}$, and thus $\overline{c} \vdash \overline{a}_{\{x'\mapsto x\}}$ □

Now we will prove three preservation properties for reduction. Besides the typical expression preservation property, we will prove two preservation properties for heaps: preservation of heap constraints, and preservation of heap well-formedness.

Lemma 5.4.5 (Preservation of Heap Constraints)**.** *If* $\langle h; e\rangle \rightarrow \langle h'; e'\rangle$ *then* $HC(h') \vdash HC(h)$

Proof. By induction on reduction. Only rule R-New changes the heap, and this change extends the heap, so $HC(h')$ is always a superset of $HC(h)$. □

Lemma 5.4.6 (Preservation of Heap Well-Formedness)**.** *If* $\langle h; e\rangle \rightarrow \langle h'; e'\rangle$ *and* heap h *and* $HC(h) \vdash e : t$ *then* heap h'

Proof. By induction on reduction. Only the case R-New needs to be considered because it changes the heap. Then (1): $h' = (h, x \mapsto o)$ and (2): $o = \langle C; \overline{f} \equiv \overline{x}\rangle$ and (3): $C(x.\ \overline{b}) \in P$ and (4): $HC(h') \vdash \overline{b}$.

According to WF-Heap, each object in the heap must match some constructor, and its fields must point to variables defined in the heap. Since h' is an extension of h, if a contructor matches in $HC(h)$ it will also match $HC(h')$, and the variables of the fields of the objects of h remain defined in h'. So the objects of h are valid in h' too. So it remains to show validity of the new object o.

The fields of o point to variables $\overline{x}$. Since **new** $C(\overline{f} \equiv \overline{x})$ is well-typed, according to T-New the variables are well-typed too. According to T-Var, we know that $HC(h') \vdash x_i :: C_i$ for all x_i, and this is possible only if x_i is defined h'. So the variables $\overline{x}$ are defined in h', and according to (3) and (4) the object o matches a constructor. So the new object o is valid in h', and h' is well-formed. □

Lemma 5.4.7 (Type Preservation)**.** *If* wf P *and* $\langle h; e\rangle \rightarrow \langle h'; e'\rangle$ *and* $HC(h) \vdash e : t$ *then* $HC(h') \vdash e' : t$

Proof. By induction on derivation of reduction.

Case R-Field*:* Assuming $\langle h; x.f\rangle \rightarrow \langle h; y\rangle$ and $HC(h) \vdash x.f \equiv y$ and $HC(h) \vdash x.f : t$ show $HC(h) \vdash y : t$, where $t = [z.\ \overline{b'}]$.

Since field typing can be derived only by applying rules T-Field and T-Sub, we know that (1): $HC(h) \vdash x : [x'.\ \overline{a'}]$ and (2): $HC(h), \overline{a'}, x'.f \equiv z \vdash \overline{b}$ and (3): $x' \notin FV(\overline{b})$ and (4): $HC(h), \overline{a'} \vdash x'.f :: C$ and (5): $HC(h), \overline{b} \vdash \overline{b'}$.

By lemma 5.4.4 on (1) follows (1a): $HC(h) \vdash \overline{a'}_{\{x' \mapsto x\}}$. Using lemma 5.4.2 we rename x' to x in (2) and (4), having in mind that x' does not appear in $\overline{b}$ and

$HC(h)$. Then after applying lemma 5.4.1 with (1a), we get (2a): $HC(h), x.f \equiv z \vdash \overline{b}$ and (4a): $HC(h) \vdash x.f :: C$. C-Subst with (2a) and $HC(h) \vdash x.f \equiv y$ gives (2b): $HC(h), z \equiv y \vdash \overline{b}$. T-Var on (4a) gives $HC(h) \vdash y : [z.\ z \equiv y]$. T-Sub with (2b) and (5) yields $HC(h) \vdash y : [z.\ \overline{b'}]$, which had to be shown.

Case R-New*:* Assuming $\langle h; \textbf{new}\ C(\overline{f} \equiv \overline{x})\rangle \to \langle h, x \mapsto o; x\rangle$ and $o = \langle C; \overline{f} \equiv \overline{x}\rangle$ and $HC(h) \vdash \textbf{new}\ C(\overline{f} \equiv \overline{x}) : t$ show $HC(h, x \mapsto o) \vdash x : t$ where $t = [x'.\ \overline{a}]$.

Since object typing can be derived only by rules T-New and T-Sub, we know that (1): $HC(h), \overline{b} \vdash \overline{a}$ and (2): $\overline{b} = (x'.\textbf{cls} \equiv C, \bigcup_i \overline{a'_i}_{\{x_i \mapsto x'.f_i\}})$ and (3): $\forall i.\ HC(h) \vdash x_i : [x_i.\ \overline{a'_i}]$.

Lemma 5.4.4 on (3) gives $HC(h) \vdash \overline{a'_i}$. Hence (3a): $HC(h), x'.f_i \equiv x_i \vdash \overline{a'_i}_{\{x_i \mapsto x'.f_i\}}$ by C-Subst. By definition, $HC(h, x \mapsto o) = HC(h), x.\textbf{cls} \equiv C, x.\overline{f} \equiv \overline{x}$. So with (3a) and (2) we know that $HC(h, x \mapsto o), x' \equiv x \vdash HC(h), \overline{b}$. With lemma 5.4.1 and (1) we get (4): $HC(h, x \mapsto o), x' \equiv x \vdash \overline{a}$. Moreover, $HC(h, x \mapsto o) \vdash x :: C$, because $x.\textbf{cls} \equiv C \in HC(h, x \mapsto o)$. Thus $HC(h, x \mapsto o) \vdash x : [x'.\ x' \equiv x]$ by T-Var. Finally with (4) and T-Sub, we have $HC(h, x \mapsto o) \vdash x : [x'.\ \overline{a}]$, which had to be shown.

Case R-Call*:* Assuming $\langle h; m(x)\rangle \to \langle h; e\rangle$ and $\langle \overline{a}; e\rangle \in \mathit{MImpl}(m, x)$ and $HC(h) \vdash \overline{a}$ and $HC(h) \vdash m(x) : t$ show $HC(h) \vdash e : t$ where $t = [y.\ \overline{b'}]$.

Since method typing can be derived only by rules T-Call and T-Sub, we know that (1): $HC(h) \vdash x : [x'.\ \overline{a'}]$ and (2): $\langle \overline{a''}; \overline{b}\rangle \in \mathit{MType}(m, x, y)$ and (3): $HC(h), \overline{b}, \overline{a'} \vdash \overline{b'}$ and (4): $x' \notin FV(\overline{b'})$.

In a well-formed P the return types of all declarations of a method are equal, and method implementation expressions are of the declared types. Therefore, from $\langle \overline{a}; e\rangle \in \mathit{MImpl}(m, x)$ and $\langle \overline{a''}; \overline{b}\rangle \in \mathit{MType}(m, x, y)$ we conclude $\overline{a} \vdash e : [y.\ \overline{b}]$. Then with $HC(h) \vdash \overline{a}$ and lemma 5.4.3 we have (5): $HC(h) \vdash e : [y.\ \overline{b}]$. Lemma 5.4.4 on (1) gives (1a): $HC(h) \vdash \overline{a'}_{\{x' \mapsto x\}}$. Using lemma 5.4.2 we rename x' to x in (3). Since x' can be taken so that it does not appear in $HC(h)$ and $\overline{b}$, and by (4) it also does not appear in $\overline{b'}$, the renaming gives us $HC(h), \overline{b}, \overline{a'}_{\{x' \mapsto x\}} \vdash \overline{b'}$ By lemma 5.4.1 with (1a), we simplify to $HC(h), \overline{b} \vdash \overline{b'}$. With T-Sub and (5) we get $HC(h) \vdash e : [y.\ \overline{b'}]$, which we had to show.

Case RC-Field*:* Assuming $\langle HC(h); e.f\rangle \to \langle HC(h'); e'.f\rangle$ and $\overline{c} \vdash e.f : t$ and the induction hypothesis $\forall t'.\ HC(h) \vdash e : t' \longrightarrow HC(h') \vdash e : t'$ show $HC(h') \vdash e'.f : t$, where $t = [y.\ \overline{b'}]$.

By field typing over rules T-Field and T-Sub, we know that (1): $HC(h) \vdash e : [x'.\ \overline{a'}]$ and (2): $HC(h), \overline{a'}, z \equiv x'.f \vdash \overline{b}$ and (3): $x' \notin FV(\overline{b})$ and (4): $HC(h), \overline{a'} \vdash x'.f :: C$ and (5): $HC(h), \overline{b} \vdash \overline{b'}$.

By the induction hypothesis and (1) we have (1a): $HC(h') \vdash e : [x'.\ \overline{a'}]$. By lemma 5.4.5 $HC(h') \vdash HC(h)$. Hence with lemma 5.4.1 from (2), (3), and (5) we can get (2a), (3a), and (5a) in which h is replaced with h'. Then from (1a), (2a), (3a), (4) and (5a) using T-Field and T-Sub we get $HC(h') \vdash e'.f : [y.\ \overline{b'}]$, which had to be shown.

Other Cases: Proofs of cases RC-New and RC-Call are analogous to the proof of RC-Field, with the difference that we use rules T-New and T-Call instead of T-Field.

□

5.4.2 Progress

First, we need to prove one property of the constraint system that ensures validity of checking if the path p will point to some object at runtime by checking entailment of the constraint $p :: C$. We used this check in the rule T-Var for ensuring validity of variable access, and in the rule C-Field for field access. To ensure validity of such check, we need to prove that $p :: C$ can be derived only if p is equivalent to some path in the context with a constraint on its class. Then in the context of heap constraints the path must be equivalent to some variable of the heap, and thus points to the object of that variable.

Lemma 5.4.8. *If* $\mathrm{wf}\, P$ *and* $\overline{c} \vdash p.\mathbf{cls} \equiv C$ *then exists* p' *such that* $\overline{c} \vdash p \equiv p'$ *and* $(p'.\mathbf{cls} \equiv C) \in \overline{c}$

Proof. By induction on derivation of $\overline{c} \vdash p.\mathbf{cls} \equiv C$. For case C-Prog the well-formedness rule WF-RD is exploited, which does not allow derivation of $p.\mathbf{cls} \equiv C$ through program rules. □

Lemma 5.4.9. *If* $\mathrm{wf}\, P$ *and* $\overline{c} \vdash p :: C$
then exist p' *and* C' *such that* $\overline{c} \vdash p \equiv p'$ *and either* $(p'.\mathbf{cls} \equiv C') \in \overline{c}$ *or* $(p' :: C') \in \overline{c}$

Proof. By induction on derivation of $\overline{c} \vdash p :: C$. For case C-Prog the well-formedness rule WF-RD is exploited, which requires that $x :: C'$ must be in the premises of a constraint entailment rule of the program. The case C-Class follows by lemma 5.4.8. □

Now the progress property is proved in the usual way:

- If an expression is not a value then it contains a subexpression of the redex form.
- If an expression is well-typed then its subexpressions are well-typed too.

- A well-typed redex can be reduced by the corresponding reduction rule, because expression typing prevents the potential stuck condition of that rule:
  - Field access typing guarantees availability of the field.
  - Method call typing guarantees availablity of the most specific method implementation matching the arguments.
  - Instantiation expression typing gurantees availability of a matching constructor.

Lemma 5.4.10 (Subexpression Typing)**.** *If* $\overline{c} \vdash e : t$ *and* e' *is a subexpression of* e *then* e' *is well-typed too, i.e.,* $\exists t'.\ \overline{c} \vdash e' : t'$

Proof. The property holds for immediate subexpressions by induction on derivation of typing. Then by induction on the transive closure, the property holds for all subexpressions. □

Lemma 5.4.11 (Progress)**.** *If* wf P *and* heap h_s *and* $\langle h_s; e_s\rangle \rightarrow_* \langle h; e\rangle$ *and* $HC(h_s) \vdash e_s : t$ *then* e *is a variable or* $\exists e', h'.\ \langle h; e\rangle \rightarrow \langle h'; e'\rangle$

Proof. By lemma 5.4.7 follows that $HC(h) \vdash e : t$. Since the only expressions without further subexpressions are variables or new expressions without fields **new** $C(\epsilon \equiv \epsilon)$, e must be a variable or contain a subexpression e' of redex form, i.e., $e' = m(x)$, $e' = x.f$, or $e' =$ **new** $C(\overline{f} \equiv \overline{x})$. if e has many redexes, then e' selected to be the first that is found by deep-first traversal over structure of e. In other words, $e \in E$, where
$E\ ::=\ e'\ \mid\ m(E)\ \mid\ E.f\ \mid\ \textbf{new}\ C(\overline{f} \equiv \overline{x}, E, \overline{f} \equiv \overline{e})$
By induction over E follows that if e' is reducable then e is reducable too, because at each step one of the congruence rules RC-Field, RC-New or RC-Call is applicable. Further, by lemma 5.4.10, there exists t' such that $HC(h) \vdash e' : t'$. So it remains to show that every well-typed redex is reducable.

Case ($e' = x.f$): Since field access is typable only by rules T-Field and T-Sub, we know that (1): $HC(h) \vdash x : [x'.\ \overline{a}]$ and (2): $HC(h), \overline{a} \vdash x'.f :: C$.

By lemma 5.4.4 on (1) we have $HC(h) \vdash \overline{a}_{\{x' \mapsto x\}}$. By renaming x' to x in (2) with lemma 5.4.2, and then simplifying with lemma 5.4.1 we get $HC(h) \vdash x.f :: C$. By lemma 5.4.9 there are a path p and class name C' such that $p :: C' \in HC(h)$ or $p.\mathbf{cls} \equiv C' \in HC(h)$, and $HC(h) \vdash x.f \equiv p$. By definition of HC, p must be variable y. So $HC(h) \vdash x.f \equiv y$, and $x.f$ is reducible by rule R-Field.

Case ($e' = m(x)$)*:* Hence $HC(h) \vdash m(x) : t'$. Since method call is typable only by rules T-Call and T-Sub, we know that (1): $HC(h) \vdash x : [x'.\ \overline{a'}]$ and (2): $HC(h), \overline{a'} \vdash \overline{a''}_{\{x \mapsto x'\}}$ and (3): $\langle \overline{a''}; \overline{b} \rangle \in \mathit{MType}(m, x, y)$.

By lemma 5.4.4 on (1) follows $HC(h) \vdash \overline{a'}_{\{x' \mapsto x\}}$. By renaming x' to x in (2) with lemma 5.4.2 and lemma 5.4.1 we get (4): $HC(h) \vdash \overline{a''}$. From heap h_s and lemma 5.4.6 we have (5): heap h. From wf P we know that method m is completely implemented. According to Complete with (4) and (5), exists $\langle \overline{b'}; e \rangle \in \mathit{MImpl}(m, x)$ such that $HC(h) \vdash \overline{b'}$. So the set $S = \{\ \langle \overline{a}; e \rangle \mid \langle \overline{a}; e \rangle \in \mathit{MImpl}(m, x)\ \wedge\ HC(h) \vdash \overline{a}\ \}$ is not empty. Now according to Unique there is a unique most specific element of S, which allows us to apply R-Call to reduce $m(x)$.

Case ($e' = \textbf{new}\ C(\overline{f} \equiv \overline{x})$)*:* Since new object can be typed only by rules T-New and T-Sub, we know that (1): $\forall i.\ HC(h) \vdash x_i : [x'_i.\ \overline{a_i}]$. (2): $\overline{b} = (x'.\textbf{cls} \equiv C, \bigcup_i \overline{a_i}_{\{x'_i \mapsto x'.f_i\}})$ and (3): $C(x'.\ \overline{b'}) \in P$ and (4): $HC(h), \overline{b} \vdash \overline{b'}$.

Let $o = \langle C; \overline{f} \equiv \overline{x_i} \rangle$ and $h' = (h, x' \mapsto o)$. To show that $\textbf{new}\ C(\overline{f} \equiv \overline{x})$ is reducible with R-New it is sufficient to show $HC(h') \vdash \overline{b'}$, because then (3) gives a matching constructor.

Lemma 5.4.4 on (1) yields $HC(h) \vdash \overline{a_i}_{\{x'_i \mapsto x_i\}}$. Then by C-Subst we obtain (1a): $HC(h), x'.\overline{f} \equiv \overline{x} \vdash \overline{a_i}_{\{x'_i \mapsto x'.\overline{f}\}}$. By definition of o and h' we know that $HC(h') = (HC(h), x'.\textbf{cls} \equiv C, x'.\overline{f} \equiv \overline{x})$. So with (1a) and (2) we get $HC(h') \vdash \overline{b}$. Finally, lemma 5.4.1 with (4) yields $HC(h') \vdash \overline{b'}$, which had to be shown.

□

5.5 Decidability of DC_C

The vc^n calculus was defined in an algorithmic style in order to ensure its decidability from the very beginning. In the case of the DC_C calculus we have a different situation, because the calculus is defined in a declarative way and contains a number of not constructive steps that cannot be directly implemented.

In this section we prove the decidability of the two relations used in the static semantics: constraint entailment and expression typing. The decidability of the constraint entailment is handled in Sec. 5.5.1, where we first define algorithmic constraint entailment and then show that the declarative and algorithmic entailment relations are equivalent. Section 5.5.2 is completely devoted to the decidability of variable elimination in constraints, which is the most problematic step in expression typing. The remainder of the decidability proof for expression typing is given in Sec. 5.5.3.

$$\overline{a} \models_A p \equiv p \quad (\text{CA-}\textsc{Refl}) \qquad\qquad \overline{a} \models_A a_i \quad (\text{CA-}\textsc{Ident})$$

$$\frac{\overline{a} \models_A p.\mathbf{cls} \equiv C}{\overline{a} \models_A p :: C} \ (\text{CA-}\textsc{Class}) \qquad\qquad \frac{\overline{a} \models_A \overline{b}_{\{x \mapsto p\}} \qquad p \sqsubset \overline{a} \qquad (\forall x.\ \overline{b} \Rightarrow x :: C) \in P}{\overline{a} \models_A p :: C} \ (\text{CA-}\textsc{Prog})$$

$$\frac{p \sqsubset \overline{a} \qquad \overline{a} \models_A p.\mathbf{cls} \equiv C \qquad \overline{a} \models_A p \equiv p'}{\overline{a} \models_A p'.\mathbf{cls} \equiv C} \ (\text{CA-}\textsc{Subst1}) \qquad\qquad \frac{p \sqsubset \overline{a} \qquad \overline{a} \models_A p \equiv p'' \qquad \overline{a} \models_A p' \equiv p}{\overline{a} \models_A p' \equiv p''} \ (\text{CA-}\textsc{Subst3})$$

$$\frac{p \sqsubset \overline{a} \qquad \overline{a} \models_A p :: C \qquad \overline{a} \models_A p \equiv p'}{\overline{a} \models_A p' :: C} \ (\text{CA-}\textsc{Subst2}) \qquad\qquad \frac{\overline{a} \models_A p \equiv p'}{\overline{a} \models_A p.f \equiv p'.f} \ (\text{CA-}\textsc{Subst4})$$

Figure 5.19: Algorithmic constraint entailment rules

The last part of the section (Sec. 5.5.4) deals with the decidability of method completeness and uniqueness checking. At first we show that in the presence of recursive dependent classes, uniqueness and completeness checking is at least as hard as solving arbirary equations on natural numbers, and thus is undecidable. Then we construct an iterative algorithm for these checks, which gives a sound approximation of method completeness and uniqueness checking at each iteration. Furthermore, the algorithm always terminates if recursive classes are forbidden.

5.5.1 Decidability of Constraint Entailment

The definition of constraint entailment by the rules of Fig. 5.14 is not constructive and cannot be directly implemented. The rule C-Cut cannot be directly implemented, because constraint c does not appear in the conclusion, and must be guessed from an infinite set of possible constraints. Analogously, the problem with rule C-Subst is that it requires guessing path p. The third problematic rule is C-Prog. It does not introduce any new variables that do not appear in the conclusion, but the problem is that it enables deriving $p :: C$ from class constraints on paths longer than p. Thus, an algorithm based on this rule may not terminate because paths may grow indefinitely.

These problems are eliminated in the alternative set of rules presented in Fig. 5.19, which define relation $\overline{a} \models_A a$. The rules CA-Refl, CA-Class, and CA-Ident are analogous to C-Refl, C-Class, and C-Ident; the rule C-Cut and the structural rules C-Weaken, C-Contract and C-Perm are not necessary, because they can be proved as lemmas; the rule CA-Prog is a constrained version of its counterpart; and the rules CA-Subst1, CA-Subst2, CA-Subst3 and CA-Subst4 are special cases of C-Subst.

Lemma 5.5.1. *$\overline{a} \vdash_A a$ is decidable.*

Proof. We will show decidability of derivation of $\overline{a} \vdash_A a$ by showing that there is only a finite number of judgements that can appear in the derivation tree.

Let S be the set of all paths that appear on the left side of the constraint entailment rules of the program:
$S = \{\ p_{\{x' \mapsto x\}} \mid p \sqsubset \overline{a}\ \wedge\ (\forall x'.\ \overline{a} \Rightarrow x' :: C) \in P\ \}$.

Then let S' be the set of all paths that can appear in the premises of the rule CA-Prog in the derivation tree of $\overline{a} \vdash_A a$:
$S'_{\overline{a}} = \{\ p_{\{x \mapsto p'\}} \mid p \in S\ \wedge\ p' \sqsubset \overline{a}\ \}$.

Now, by induction on derivation of $\overline{a} \vdash_A a$, we can prove that all judgements in the derivation contain only paths from the set $S''_{\overline{a};a} = S'_{\overline{a}} \cup \{\ p \mid p \sqsubset (\overline{a}, a)\ \}$:

Case CA-Prog*:* Then $a = (p :: C)$ and (1): $(\forall x.\ \overline{b} \Rightarrow x :: C) \in P$ and (2): $p \sqsubset \overline{a}$ and $\overline{a} \vdash_A \overline{b}_{\{x \mapsto p\}}$.

By induction hypothesis the judgements in the derivation of $\overline{a} \vdash_A b_{i\{x \mapsto p\}}$ can contain only paths from $S''_{\overline{a};b_{i\{x \mapsto p\}}}$. To prove the case we need to show $S''_{\overline{a};b_{i\{x \mapsto p\}}} \subseteq S''_{\overline{a};p :: C}$.

Let's assume $p' \sqsubset b_{i\{x \mapsto p\}}$. Then there exist $q \sqsubset b_i$ and $p' \sqsubset q_{\{x \mapsto p\}}$. By definition of S and (1), we know that $q \in S$. Then with (2) and definition of $S'_{\overline{a}}$, we have $p' \in S'_{\overline{a}}$. Hence $S'_{\overline{a}} \cup \{\ p \mid p \sqsubset (\overline{a}, b_{i\{x \mapsto p\}})\ \} \subseteq S'_{\overline{a}} \cup \{\ p \mid p \sqsubset (\overline{a}, a)\ \}$, which is equivalent to $S''_{\overline{a};b_{i\{x \mapsto p\}}} \subseteq S''_{\overline{a};a}$.

Other Cases:: The premises of the rules with a conclusion $\overline{a} \vdash_A a$ contain only paths from $\{\ p \mid p \sqsubset (\overline{a}, a)\ \}$.

Since the set $S''_{\overline{a};a}$ is finite, and so is the set of class names in the program, there is only a finite number of judgements that can be used for derivation of $\overline{a} \vdash_A a$, thus the algorithm that traverses all possible paths of derivation would terminate. □

Now, since we have proved that $\overline{a} \vdash_A a$ is decidable, the decidability of $\overline{a} \vdash a$ can be showed, by proving equivalence of these relationships for well-formed programs. For this we will show that all rules of $\overline{a} \vdash a$ are also valid in $\overline{a} \vdash_A a$.

First, we will prove that we can replace the context with a larger set of constraints. This property subsumes all the structural rules: C-Weaken, C-Contract and C-Perm.

Lemma 5.5.2. *If* $\overline{a} \vdash_A a$ *and* $\overline{a} \subseteq \overline{a'}$ *then* $\overline{a'} \vdash_A a$

Proof. By induction on $\overline{a} \vdash_A a$. All cases, except CA-Ident follow directly from induction hypothesis. For the case CA-Ident we have to prove that for each a_i exists j such that $a'_j = a_i$, which is true because of $\overline{a} \subseteq \overline{a'}$. □

Now we will prove the C-Cut rule as a lemma of the algorithmic constraint entailment.

Lemma 5.5.3. *if* $\overline{a} \vdash_A c$ *and* $\overline{a}, c \vdash_A a$ *then* $\overline{a} \vdash_A a$

Proof. By induction on $\overline{a}, c \vdash_A a$.

All cases, except CA-Ident follow directly from induction hypothesis.

Case CA-Ident*:* Assuming (1): $a \in (\overline{a}, c)$ and (2): $\overline{a} \vdash_A c$ show $\overline{a} \vdash_A a$.

From (1) we know that either (i) $a = c$ or (ii) $a \in \overline{a}$. The case (i) is proved by (2) and the case (ii) is proved by CA-Ident.

□

Lemma 5.5.4. *if* $\overline{a} \vdash_A c$ *and* $\overline{a'}, c \vdash_A a$ *then* $\overline{a}, \overline{a'} \vdash_A a$

Proof. Follows from lemmas 5.5.3 and 5.5.2. □

The next goal is to prove that the C-Subst rule as a lemma of the algorithmic constraint entailment. For this we need to show that in well-formed programs the rules CA-Subst1, CA-Subst2 and CA-Subst3, can be relaxed by removing the requirement $p \sqsubset \overline{a}$ (Lemmas 5.5.7, 5.5.9, 5.5.11 and 5.5.11). Then we need prove that the relaxed rules and CA-Subst4 cover all possible cases of C-Subst (Lemma 5.5.12).

Lemma 5.5.5. *If* $\text{wf}\, P$ *and* $\overline{a} \vdash_A p \equiv p'$
then $p = p'$ *or exist* q, q', q'', $\overline{f}$ *such that* $p = q.\overline{f}$ *and* $p' = q'.\overline{f}$ *and* $q'' \sqsubset \overline{a}$ *and* $\overline{a} \vdash_A q \equiv q''$ *and* $\overline{a} \vdash_A q' \equiv q''$

Proof. Proof by induction on derivation of $\overline{a} \vdash_A p.f \equiv p'.f'$.

Case CA-Refl*:* Gives $p = p'$.

Case CA-Subst4*:* Follows directly from the inductive hypothesis.

Case CA-Subst3*:* Then $\overline{a} \vdash_A p \equiv p''$ and $\overline{a} \vdash_A p' \equiv p''$ and $p'' \sqsubset \overline{a}$, which proves the case by taking $\overline{f} = \epsilon$.

Case CA-IDENT*:* Then $(p \equiv p') \in \overline{a}$, which again proves the case by taking $\overline{f} = \epsilon$.

□

Lemma 5.5.6. *If* wf P *and* $\overline{a} \vdash_A p \equiv p'$*, and* $p \neq p'$ *and* $p = q.\overline{f}$ *and* $\overline{a} \vdash_A q \equiv q''$ *and* $q'' \sqsubset \overline{a}$ *and* $\forall q_0 \sqsubseteq p.\ \overline{a} \vdash_A q \equiv q'' \ \wedge\ q'' \sqsubset \overline{a} \ \longrightarrow\ q_0 \sqsubseteq q$
then $p' = q'.\overline{f}$ *and* $\overline{a} \vdash_A q' \equiv q''$ *for some* q'*.*

Proof. Since $p \neq p'$, from $\overline{a} \vdash_A p \equiv p'$ and lemma 5.5.5 we know that there exist q_0, q_0' and q_0'' and $\overline{f'}$ such that (1): $p = q_0.\overline{f'}$ and (2): $p' = q_0'.\overline{f'}$ and (3): $q_0'' \sqsubset \overline{a}$ and (4): $\overline{a} \vdash q_0 \equiv q_0''$ and (5): $\overline{a} \vdash q_0' \equiv q_0''$.

Since q is the largest path with properties of (1), (3) and (4), we have $q = q_0.\overline{f''}$ and $\overline{f'} = \overline{f''}.\overline{f}$. Now let's take $q' = q_0'.\overline{f''}$. Then $p' = q'.\overline{f}$, and $\overline{a} \vdash q' \equiv q''$ follows from (5) and CA-SUBST4. □

Lemma 5.5.7. *If* wf P *and* $\overline{a} \vdash_A p \equiv p'$ *and* $\overline{a} \vdash_A p \equiv p''$ *then* $\overline{a} \vdash_A p' \equiv p''$

Proof. If $p = p'$ then the thesis follows from $\overline{a} \vdash_A p \equiv p''$.

If $p \neq p'$, then by lemma 5.5.5 we can take q such that $p = q.\overline{f}$ and and $\overline{a} \vdash_A q \equiv q_0$ and (1): $q_0 \sqsubset \overline{a}$ for some q_0 and $\overline{f}$. If there are multiple paths that satisfy these conditions, let q be the largest of them.

Then by lemma 5.5.6 we can take q' and q'' such that $p' = q'.\overline{f}$ and $p'' = q''.\overline{f}$ and $\overline{a} \vdash_A q' \equiv q_0$ and $\overline{a} \vdash_A q'' \equiv q_0$.

Then $\overline{a} \vdash_A q' \equiv q''$ by rule CA-SUBST3 with (1), and the thesis follows by rule CA-SUBST4. □

Lemma 5.5.8. *If* $\overline{a} \vdash_A p.\mathbf{cls} \equiv C$ *then exists* p' *such that* $p' \sqsubset \overline{a}$ *and* $\overline{a} \vdash_A p \equiv p'$ *and* $\overline{a} \vdash_A p'.\mathbf{cls} \equiv C$

Proof. By cases on derivation of $\overline{a} \vdash_A p.\mathbf{cls} \equiv C$. □

Lemma 5.5.9. *If* wf P *and* $\overline{a} \vdash_A p.\mathbf{cls} \equiv C$ *and* $\overline{a} \vdash_A p \equiv p'$ *then* $\overline{a} \vdash_A p'.\mathbf{cls} \equiv C$

Proof. By lemma 5.5.8 we can take p'' such that (1): $p'' \sqsubset \overline{a}$ and $\overline{a} \vdash_A p \equiv p''$ and (2): $\overline{a} \vdash_A p''.\mathbf{cls} \equiv C$. By lemma 5.5.7 we get (3): $\overline{a} \vdash_A p' \equiv p''$. Then CA-SUBST1 on (1), (2) and (3) gives $\overline{a} \vdash_A p'.\mathbf{cls} \equiv C$. □

Lemma 5.5.10. *If* $\overline{a} \vdash_A p :: C$ *then exists* p' *such that* $p' \sqsubset \overline{a}$ *and* $\overline{a} \vdash_A p \equiv p'$ *and* $\overline{a} \vdash_A p' :: C$

Proof. By cases on derivation of $\overline{a} \vdash_A p :: C$ and lemma 5.5.8. □

Lemma 5.5.11. *If* wf P *and* $\overline{a} \vdash_A p :: C$ *and* $\overline{a} \vdash_A p \equiv p'$ *then* $\overline{a} \vdash_A p' :: C$

Proof. By lemma 5.5.10 we can take p'' such that (1): $p'' \sqsubset \overline{a}$ and $\overline{a} \vdash_A p \equiv p''$ and (2): $\overline{a} \vdash_A p'' :: C$. By lemma 5.5.7 we get (3): $\overline{a} \vdash_A p' \equiv p''$. And CA-Subst2 on (1), (2) and (3) gives $\overline{a} \vdash_A p' :: C$. □

Lemma 5.5.12. *If* wf P *and* $\overline{a} \vdash_A \overline{a}_{\{x \mapsto p\}}$ *and* $\overline{a} \vdash_A p \equiv p'$ *then* $\overline{a} \vdash_A \overline{a}_{\{x \mapsto p'\}}$.

Proof. Proof by cases of $\overline{a}$:

Case $a = (x.\overline{f} :: C)$*:* The case follows from the rule CA-Subst4 and the lemma 5.5.11.

Case $a = (x.\overline{f}.\mathbf{cls} \equiv C)$*:* The case follows from the rule CA-Subst4 and the lemma 5.5.9.

Case $a = (x.\overline{f} \equiv x.\overline{f'})$*:* Assuming $\overline{a} \vdash_A p.\overline{f} \equiv p.\overline{f'}$ show $\overline{a} \vdash_A p'.\overline{f} \equiv p'.\overline{f'}$.

By applying CA-Subst4 on $\overline{a} \vdash_A p \equiv p'$ have $\overline{a} \vdash_A p.\overline{f} \equiv p'.\overline{f}$ and $\overline{a} \vdash_A p.\overline{f'} \equiv p'.\overline{f'}$. The case follows from the assumption and lemma 5.5.7.

Case $a = (x.\overline{f} \equiv y.\overline{f'})$*:* Assuming $\overline{a} \vdash_A p.\overline{f} \equiv y.\overline{f'}$ show $\overline{a} \vdash_A p'.\overline{f} \equiv y.\overline{f'}$.

The case follows from CA-Subst4 and lemma 5.5.7.

□

Now it remains to prove the rule C-Prog for the algorithmic constraint entailment, by showing that the conditions of CA-Prog are ensured by program well-formedness.

Lemma 5.5.13. *If* wf P *and* $\overline{a} \vdash_A \overline{b}_{\{x \mapsto p\}}$ *and* $(\forall x.\ \overline{b} \Rightarrow a) \in P$ *then* $\overline{a} \vdash_A a_{\{x \mapsto p\}}$.

Proof. By well-formedness of P we know that $a = x :: C$ and $x :: C' \in \overline{b}$ for some C and C'.

By lemma 5.5.10 we can take p' such that $p' \sqsubset \overline{a}$ and $\overline{a} \vdash_A p \equiv p'$.

By lemma 5.5.12 we have $\overline{a} \vdash_A \overline{b}_{\{x \mapsto p'\}}$. Now with CA-Prog we have $\overline{a} \vdash_A p' :: C$.

Finally, we obtain $\overline{a} \vdash_A p :: C$ from $\overline{a} \vdash_A p \equiv p'$ and lemma 5.5.11. □

Now we can prove soundness (lemma 5.5.14) and completeness (lemma 5.5.15) of algorithmic constraint derivation. The soundness proof is straightforward, and the completeness proof is based on the lemmas that we have proved.

Lemma 5.5.14. *If* $\overline{a} \vdash_A a$ *then* $\overline{a} \vdash a$.

Proof. By induction on derivation of $\overline{a} \vdash_{\!\!A} a$. Cases CA-Ident, CA-Class, CA-Refl and CA-Prog follow from correspondingly C-Ident, C-Class, C-Refl and C-Prog. Cases CA-Subst1, CA-Subst2 and CA-Subst3 follow from C-Subst. Finally, the case CA-Subst4 follows from C-Subst and C-Refl: From $\overline{a} \vdash p.\overline{f} \equiv p.\overline{f}$ (by C-Refl) and $\overline{a} \vdash p \equiv q$ we have $\overline{a} \vdash p.\overline{f} \equiv q.\overline{f}$ (by C-Subst). □

Lemma 5.5.15. *If* wf P *and* $\overline{a} \vdash a$ *then* $\overline{a} \vdash_{\!\!A} a$.

Proof. By induction on derivation of $\overline{a} \vdash a$. The cases C-Ident, C-Class and C-Refl follow from correspondingly CA-Ident, CA-Class and CA-Refl. The case C-Prog is proved by lemma 5.5.13, the case C-Subst by lemma 5.5.12, the case C-Cut by lemma 5.5.4, and the cases of the structural rules by lemma 5.5.2. □

Lemma 5.5.16. *If* wf P *then* $\overline{a} \vdash a$ *iff* $\overline{a} \vdash_{\!\!A} a$.

Proof. Follows directly from the lemmas 5.5.14 and 5.5.15. □

Now when we proved that our algorithmic rules are sound and complete, we know that constraint entailment is decidable in well-formed programs.

Theorem 5.5.1. *If* wf P *then derivation of* $\overline{a} \vdash a$ *is decidable.*

Proof. Follows directly from the lemmas 5.5.16 and 5.5.1 □

5.5.2 Decidability of Variable Elimination

Expression typing rules T-Field and T-Call as part of their definition eliminate a variable from a set of constraints. This elimination is defined declaratively, and in order to show decidability of these rules we need to show that variable elimination is decidable. Moreover, for transforming T-Field and T-Call to rules assigning minimal types, we will need optimal elimination of variables. Thus, in this section we will show that for any set of constraints $\overline{a}$ and any variable x, we can compute a new set of constraints $\overline{b}$ that are entailed by $\overline{a}$, free of x and optimal, i.e., (i) $x \notin FV(\overline{b})$ and (ii) $\overline{a} \vdash \overline{b}$ and (iii) $\forall \overline{b'}.\ \overline{a} \vdash \overline{b'} \ \wedge\ x \notin FV(\overline{b}) \ \longrightarrow\ \overline{a'} \vdash \overline{b'}$.

The most difficult is to eliminate a variable from path equivalence constraints. For this we need to define a special heap structure that encodes these equivalences, then remove x from the heap, and translate the reduced heap back to a set of constraints. In previous section we showed that $a \vdash a$ is equivalent to $a \vdash_{\!\!A} a$ in well-formed programs (see lemma 5.5.16). Since induction over $a \vdash_{\!\!A} a$ is easier, we prove all properties of abstract heaps and variable elimination for $a \vdash_{\!\!A} a$.

5.5.2.1 Abstract Heap and Its Construction

An abstract heap consists of a labeled graph defined on a finite set of references, and a mapping from variables to references. The edges of this graph are labeled by fields and specify navigation from reference to reference over fields. In such a heap a path is interpreted as the navigation starting from the reference that corresponds to the variable of the path and following the edges that are labeled by the fields of the path.

Definition 5.5.1. *An* abstract heap *is a tuple $\langle A; \sigma; \rho \rangle$, where A is a finite set of references, $\sigma : \mathrm{Var} \to A$ is a mapping from variable names to references, and the function $\rho : A \times \mathrm{Field} \to A$ defines navigation from references to references over fields.*

We will use equivalence classes of paths in a constraint set to represent the references of a corresponding abstract heap.

Definition 5.5.2. *Let $EqSets(\overline{a}) = X$ be equivalence sets of paths contained by $\overline{a}$:*

(i) $\bigcup\limits_{P \in X} P = S$

(ii) $\forall P \in X.\ P \neq \varnothing$

(iii) $\forall p, q \in S.\ \ \overline{a} \vdash_A p \equiv q \ \longleftrightarrow (\exists P \in X).\ p, q \in P)$

where $S = \{\ p \mid p \sqsubset \overline{a}\ \}$.

Definition 5.5.3. *For each set of constraints $\overline{a}$ we can define a corresponding abstract heap $CsToHeap(\overline{a})$.*

1. *Let $A = EqSets(\overline{a})$.*
2. *Let σ be the function mapping a variable x to the reference containing the path x: $\sigma(x) = r$, such that $r \in A$ and $x \in r$.*
   There is at most one a that satisfies these conditions, because equivalence sets are disjoint, and x can belong to only one of them.
3. *Let $\rho(r, f)$ point to the equivalence set of the paths of r extended with the field f: $\rho(r, f) = r'$, if exist p and p' such that (i) $r' \in A$ and (ii) $p \in r$ and (iii) $p' \in r'$ and (iv) $\overline{a} \vdash_A p.f \equiv p'$.*
   There is at most one r' that satisfies these conditions, because $p.f$ and $p'.f$ are equivalent for all $p, p' \in r$, and thus can be equivalent to the paths of one equivalence class only.
4. *Then $CsToHeap(\overline{a}) = \langle A; \sigma; \rho \rangle$.*

Lemma 5.5.17. *If* $\mathrm{wf}\, P$ *then $CsToHeap(\overline{a})$ is computable.*

Proof. $\mathit{EqSets}(\overline{a})$ is constructed by checking $\overline{a} \vdash_A p \equiv p'$ for all $p, p' \sqsubset \overline{a}$. So computability of $\mathit{EqSets}(\overline{a})$ follows from decidability of constraint entailment in a well-formed program (theorem 5.5.1).

The computation of σ is straightforward, and computability of ρ again relies on the decidability of constraint entailment, because for all fields in program f and all $r, r' \in A$, we must check $\overline{a} \vdash_A p.f \equiv p'$ for some $p \in r$ and $p' \in r'$. □

Definition 5.5.4. *Function pref evaluates a path to a reference in a heap $h = \langle A; \sigma; \rho\rangle$:*

$\mathit{pref}(h, x) = \sigma(x)$
$\mathit{pref}(h, p.f) = \rho(\mathit{pref}(h, p), f)$

Lemma 5.5.18. *If $\mathit{pref}(h, p) = \mathit{pref}(h, q)$ and $\mathit{pref}(h, p.\overline{f}) = r$ then $\mathit{pref}(h, q.\overline{f}) = r$*

Proof. Follows by induction on the length of $\overline{f}$ and the definition of *pref*. □

Lemma 5.5.19. *If $h = \mathit{CsToHeap}(\overline{a})$ and $\mathit{pref}(h, p) = r$ and $p' \in r$ then $\overline{a} \vdash_A p \equiv p'$,*

Proof. Let $h = \langle A; \sigma; \rho\rangle$.

Let's prove by induction on p.

Case x: By definition of *pref* we know that $\mathit{pref}(h, x) = \sigma(x)$, which gives $x \in r$. So x is in the same equivalence class as p', which means that they are equivalent.

Case $p.f$: By definition of *pref* we know that $\mathit{pref}(h, p.f) = \rho(r', f)$, where (1): $r' = \mathit{pref}(h, p)$. Let's take (2): $q \in r'$. From $p' \in r$ and by the definition of ρ we get $\overline{a} \vdash_A q.f \equiv p'$. Induction hypothesis with (1) and (2) gives $\overline{a} \vdash_A p \equiv q$. Hence $\overline{a} \vdash_A p.f \equiv p'$ by lemma 5.5.7.

□

Lemma 5.5.20. *If $h = \mathit{CsToHeap}(\overline{a})$ and $\mathit{pref}(h, p) = r$ and $\mathit{pref}(h, p') = r'$, then $r = r'$ iff $\overline{a} \vdash_A p \equiv p'$.*

Proof. Let $h = \langle A; \sigma; \rho\rangle$.

Since equivalence sets are not empty, we can take some $q \in r$ and $q' \in r'$. By lemma 5.5.19 $\overline{a} \vdash_A p \equiv q$ and $\overline{a} \vdash_A p' \equiv q'$. Because of lemma 5.5.7, $\overline{a} \vdash_A p \equiv p'$ is derivable iff $\overline{a} \vdash_A q \equiv q'$ is derivable, and by definition of the equivalence sets, $\overline{a} \vdash_A q \equiv q'$ is derivable iff they belong to the same equivalence class, i.e., iff $r = r'$. □

Lemma 5.5.21. *If* wf P *and $\overline{a} \vdash_A p.f \equiv q$ and $q \sqsubset \overline{a}$ then exists q' such that $\overline{a} \vdash_A p \equiv q'$ and $q' \sqsubset \overline{a}$*

Proof. By induction on derivation of $\overline{a} \vdash_A p.f \equiv q$ for any q. In wf P we have to consider only following cases:

Case CA-REFL*:* Then $p.f = q$. Hence $p \sqsubset \overline{a}$ and $\overline{a} \vdash p \equiv p$.

Case CA-IDENT*:* Then $(p.f \equiv q) \in \overline{a}$. Then $p \sqsubset \overline{a}$ and $\overline{a} \vdash p \equiv p$.

Case CA-SUBST3*:* By induction hypothesis.

Case CA-SUBST4*:* Then $p.f = p'.\overline{f}$ and $q = q'.\overline{f}$ and (1): $\overline{a} \vdash p' \equiv q'$. If $\overline{f} = \epsilon$ then $p' = p.f$ and the case follows by the induction hypothesis.

Otherwise $\overline{f} = \overline{f'}.f$ and $p = p'.\overline{f'}$. Hence $\overline{a} \vdash p \equiv q'.\overline{f'}$ by CA-SUBST4 with (1). This proves the case, because $q'.\overline{f'} \sqsubset q \sqsubset \overline{a}$.

□

Lemma 5.5.22. *If* wf P *and* $\overline{a} \vdash a$ *then* $FV(a) \subseteq FV(\overline{a})$ *or* $\exists p.\ a = (p \equiv p)$

Proof. By induction on derivation of $\overline{a} \vdash a$. Proof of the case C-PROG relies on the fact that entailment rules in well-formed programs do not contain free variables (WF-RD). □

Lemma 5.5.23. *If* wf P *and* $h = CsToHeap(\overline{a})$ *and* $\overline{a} \vdash_A p \equiv q$ *and* $q \sqsubset \overline{a}$ *then* $pref(h, p)$ *is defined.*

Proof. Let $h = \langle A; \sigma; \rho \rangle$.

Let's prove by induction on p for any q.

Case x: Then $\overline{a} \vdash_A x \equiv q$ and $q \sqsubset \overline{a}$. According to lemma 5.5.22, $x = q$ or $x \in FV(\overline{a})$, and, since $q \sqsubset \overline{a}$, we know that in both cases $x \sqsubset \overline{a}$. Thus $\sigma(x)$ is defined, and $pref(h, x) = \sigma(x)$.

Case $p.f$: Then $\overline{a} \vdash_A p.f \equiv q$ and $q \sqsubset \overline{a}$. By lemma 5.5.21, we can take q' such that $\overline{a} \vdash_A p \equiv q'$ and $q' \sqsubset \overline{a}$. Then by induction hypothesis (1): $pref(h, p) = r'$. Since equivalence classes are not empty, let's take (2): $p' \in r'$. By lemma 5.5.19 we have $\overline{a} \vdash_A p \equiv p'$, which gives $\overline{a} \vdash_A p.f \equiv p'.f$ by CA-SUBST4, and with $\overline{a} \vdash_A p.f \equiv q$, we get (3): $\overline{a} \vdash_A p'.f \equiv q$ (lemma 5.5.7). Since $q \sqsubset \overline{a}$, there exists an equivalence class r such that (4): $q \in r$. From (2), (3), (4) and the definition of ρ we get $\rho(r', f) = r$. Now with (1) and the definition of *pref* follows $pref(h, p.f) = r$.

□

Lemma 5.5.24. *If* wf P *and* $h = CsToHeap(\overline{a})$ *then* $\overline{a} \vdash_A p \equiv p'$ *iff* $p = p'$ *or exist* q, q', $\overline{f}$ *such that* $p = q.\overline{f}$ *and* $p' = q'.\overline{f}$ *and* $pref(h, q) = pref(h, q')$.

Proof. Let's prove the statement from right to left. If $p = p'$ then $\overline{a} \models_A p \equiv p'$ by CA-Refl. Otherwise, lemma 5.5.20 with $\mathit{pref}(h,q) = \mathit{pref}(h,q')$ gives $\overline{a} \models_A q \equiv q'$, which again implies $\overline{a} \models_A p \equiv p'$ (CA-Subst4).

Now let's prove the statement from left to right. If $p = p'$ then the case is proved. Otherwise, according to lemma 5.5.5, we can take q, q' and q'' and $\overline{f}$ such that (1): $p = q.\overline{f}$ and (2): $p' = q'.\overline{f'}$ and (3): $q'' \sqsubset \overline{a}$ and (4): $\overline{a} \models_A q \equiv q''$ and (5): $\overline{a} \models_A q' \equiv q''$. From (3), (4), (5) and lemma 5.5.23 we know that $\mathit{pref}(h,q)$ and $\mathit{pref}(h,q')$ are defined. Then with $\overline{a} \models_A q \equiv q'$ and lemma 5.5.20 we have $\mathit{pref}(h,q) = \mathit{pref}(h,q')$, which we had to show. □

Now, we can check if two paths p and p' are equivalent in $\overline{a}$ by constructing an abstract heap of these constraints, removing the equal suffix from p and p', and comparing *pref* values for the remaining prefixes. According to the lemma 5.5.24, such equivalence checking procedure is sound and complete with respect to the constraint entailment relation.

5.5.2.2 Elimination of a Variable from an Abstract Heap

Now instead of eliminating a desired variable directly from the set of constraints, we will remove the variable from the heap first, and then construct a set of constraints corresponding to the reduced heap. Removing a variable from a heap structure is much easier, because we know that evaluation of $\mathit{pref}(h, x.\overline{f})$ uses only the reference $\sigma(x)$ and the references reachable from it over ρ. Thus, if we eliminate the references that are not reachable from the variables other than x, the remaining heap will be sufficient to check the equivalence between all paths that do not start with x.

Definition 5.5.5. *$\mathit{elimH}(x,h)$ eliminates variable x from heap h:*
$\mathit{elimH}(x, \langle A;\sigma;\rho\rangle) = \langle A';\sigma';\rho'\rangle$
where $A' = \{\ r \mid y \in \mathit{dom}(\sigma)\ \wedge\ y \neq x\ \wedge\ r = \mathit{pref}(h, y.\overline{f})\ \}$
and $\sigma'(y) = \sigma(y)$, *if* $y \in \mathit{dom}(\sigma) \setminus \{x\}$
and $\rho'(r,f) = \rho(r,f)$, *if* $r \in A'$

Lemma 5.5.25. *If $y \neq x$ then $\mathit{pref}(\mathit{elimH}(x,h), y.\overline{f}) = \mathit{pref}(h, y.\overline{f})$*

Proof. By induction on the length of $\overline{f}$, and the definitions of *pref* and *elimH*. □

Definition 5.5.6. *A cycle of path p in heap h is a pair $\langle q;q'\rangle$ such that $q \sqsubset q' \sqsubseteq p$ and $\mathit{pref}(h,q) = \mathit{pref}(h,q')$*

Lemma 5.5.26. *$\mathit{pref}(h,p) = r$ then exists q without cycles such that $\mathit{pref}(h,q) = r$*

Proof. Let's apply an induction on the size of p.

Case $p = x$: x cannot contain cycles. So we can take $q = x$.

Case $p = p'.f$: By induction hypothesis we can take q' without cycles such that $\mathit{pref}(h, p') = \mathit{pref}(h, q')$. Then $\mathit{pref}(h, p) = \mathit{pref}(h, q'.f)$ by definition of *pref*.

If $\forall q'' \sqsubseteq q'.\ \mathit{pref}(h, q'.f) \neq \mathit{pref}(h, q'')$ then $q'.f$ is without cycles, and we can take $q = q'.f$.

Otherwise, we can take q'' such that $q'' \sqsubseteq q'$ and $\mathit{pref}(h, q'') = \mathit{pref}(h, q'.f)$. Since q' is free of cycles, q'' is without cycles too. So we can take $q = q''$.

□

Lemma 5.5.27. *$\mathit{elimH}(x, h)$ can be computed for every x and every finite h.*

Proof. We have to compute
$A' = \{\ r \mid y \in \mathit{dom}(\sigma)\ \wedge\ y \neq x\ \wedge\ r = \mathit{pref}(h, y.\overline{f})\ \}$
$\sigma'(y) = \sigma(y)$ if $y \in \mathit{dom}(\sigma) \setminus \{x\}$
$\rho'(r, f) = \rho(r, f)$ if $r \in A'$.

The computation of A' quantifies over all paths $y.\overline{f}$, for which $\mathit{pref}(h, y.\overline{f})$ is defined. There can be an infinite number of such paths, but because of lemma 5.5.26 it is sufficient to consider only paths without cycles. Since the set of references in h is finite, there is also only a finite number of paths without cycles, and A' can be computed by evaluating all these paths.

σ' and ρ' are computable, because σ and ρ are defined on finite sets. □

5.5.2.3 Construction of Abstract Heap Constraints

Definition 5.5.7. *Given a heap $h = \langle A; \sigma; \rho\rangle$, let $\mathit{Paths}(h, r)$ be the set of paths without cycles that evaluate to r, for $r \in A$. Then we define $\mathit{HeapToCs}(h)$ as the set of equivalence constraints relating the paths of the same reference and the paths of the references related by the ρ function:*

$$\begin{array}{lcl} \mathit{HeapToCs}(h) & = & \{\ p \equiv p' \mid r \in A\ \wedge\ p, p' \in \mathit{Paths}(h, r)\ \} \\ & & \cup\ \{\ p.f \equiv p' \mid \rho(r, f) = r'\ \wedge\ p \in \mathit{Paths}(h, r)\ \wedge\ p' \in \mathit{Paths}(h, r')\ \} \end{array}$$

where
$\mathit{Paths}(h, r) = \{\ p \mid \mathit{pref}(h, p) = r\ \wedge\ p$ is without cycles $\}$.

Lemma 5.5.28. *$\mathit{HeapToCs}(h)$ is computable for every finite heap $h = \langle A; \sigma; \rho\rangle$.*

Proof. The definition of $\mathit{HeapToCs}(h)$ quantifies only over finite sets and thus can be directly implemented. $\mathit{Paths}(h, r)$ is computable because there is only a finite set of paths without cycles in a given heap. □

Lemma 5.5.29. *If $\mathit{HeapToCs}(h) = \overline{a}$ and $p \sqsubset \overline{a}$, then $\mathit{pref}(h, p)$ is defined.*

Proof. Let $h = \langle A; \sigma; \rho \rangle$.

Since $\overline{a}$ contains only equivalence constraints and $p \sqsubset \overline{a}$, we can take q such that $p \sqsubseteq q$ and $(q \equiv q') \in \overline{a}$. It is sufficient to show that $\mathit{pref}(h, q)$ is defined.

According to the definition of $\mathit{HeapToCs}(h)$ we have to consider two cases:

(i) $q \in \mathit{Paths}(h, r)$. Then $\mathit{pref}(h, q) = r$.

(ii) $q = q''.f$ such that $q'' \in \mathit{Paths}(h, r)$ and $q' \in \mathit{Paths}(h, r')$ and (1): $\rho(r, f) = r'$. Then $\mathit{pref}(h, q'') = r$, and with (1) follows $\mathit{pref}(h, q''.f) = r'$.

□

Lemma 5.5.30. *If $\mathit{HeapToCs}(h) = \overline{a}$ and $p \sqsubset \overline{a}$, and $\overline{a} \vdash_A p \equiv p'$ then $\mathit{pref}(h, p) = \mathit{pref}(h, p')$.*

Proof. Let $h = \langle A; \sigma; \rho \rangle$. From $p \sqsubset \overline{a}$ and lemma 5.5.29, we know that $\mathit{pref}(h, p) = r$ for some r.

We will prove $\mathit{pref}(h, p') = r$ by induction on derivation of $\overline{a} \vdash_A p \equiv p'$.

Case CA-Refl*:* Then $p' = p$. Hence $\mathit{pref}(h, p') = r$

Case CA-Ident*:* Then $(p \equiv p') \in \overline{a}$. According to the definition of $\mathit{HeapToCs}(h)$ we have consider following cases:

(i) $p, p' \in \mathit{Paths}(h, r)$. Then $\mathit{pref}(h, p) = r = \mathit{pref}(h, p')$.

(ii) $p = q.f$ and $q \in \mathit{Paths}(h, r'')$ and $p' \in \mathit{Paths}(h, r')$ and $\rho(r'', f) = r'$. Then $\mathit{pref}(h, p') = r' = \mathit{pref}(h, q.f) = \mathit{pref}(h, p) = r$.

(iii) $p' = q.f$ and $q \in \mathit{Paths}(h, r')$ and $p \in \mathit{Paths}(h, r)$ and $\rho(r', f) = r$. Then $\mathit{pref}(h, p') = \mathit{pref}(h, q.f) = r$.

Case CA-Subst3*:* By induction hypothesis $\mathit{pref}(h, p) = \mathit{pref}(h, p'') = \mathit{pref}(h, p')$.

Case CA-SUBST4*:* Assuming (1): $pref(h,p) = pref(h,p')$ and (2): $p.\overline{f} \sqsubset \overline{a}$ show $pref(h,p.\overline{f}) = pref(h,p'.\overline{f})$.

From (2) and lemma 5.5.29 we know that $pref(h,p.\overline{f})$ is defined. Then lemma 5.5.18 with (1) gives $pref(h,p.\overline{f}) = pref(h,p'.\overline{f})$.

□

Lemma 5.5.31. *If* $HeapToCs(h) = \overline{a}$ *and* $pref(h,p) = pref(h,q)$ *then* $\overline{a} \vdash_A p \equiv q$

Proof. Let $h = \langle A; \sigma; \rho \rangle$.

Let's apply an induction on the total number of cycles in p and q (See Def. 5.5.6).

Case 0: Then p and q are free of cycles. Let $r = pref(h,p)$. Then $p, q \in Paths(h,r)$, and by definition of $HeapToCs$ we get $(p \equiv q) \in \overline{a}$. Hence $\overline{a} \vdash_A p \equiv q$.

Case (n+1): Let's take the leftmost cycle of p, i.e., we take p', $\overline{f}$, f and $\overline{f'}$ such that (1): $p = p'.\overline{f}.f.\overline{f'}$ and (2): $pref(h,p') = pref(h,p'.\overline{f}.f)$ and $p'.\overline{f}$ is free of cycles. Moreover, from (2) and the definition of $pref$ we know that $pref(h,p') = \rho(pref(h,p'.\overline{f}), f)$. Let $r = pref(h,p')$ and $r' = pref(h,p'.\overline{f})$. Since p' and $p'.\overline{f}$ are free of cycles, we have $p' \in Paths(r)$ and $p'.\overline{f} \in Paths(r')$ and $r = \rho(r', f)$. Thus by definition of $HeapToCs$ we have $(p'.\overline{f}.f \equiv p') \in \overline{a}$. Hence (3): $\overline{a} \vdash_A p'.\overline{f}.f \equiv p'$.

From (1), (2) and lemma 5.5.18 we have $pref(h,p'.\overline{f'}) = pref(h,p)$, and with the theorem assumption $pref(h,p'.\overline{f'}) = pref(h,q)$. Since $p'.\overline{f'}$ eliminates one cycle in p, by induction hypothesis we get $\overline{a} \vdash_A p'.\overline{f'} \equiv q$. Then with (3): and C-SUBST we have $\overline{a} \vdash_A p'.\overline{f}.f.\overline{f'} \equiv q$, which is equivalent to $\overline{a} \vdash_A p \equiv q$. □

5.5.2.4 Elimination of a Variable from Constraints

Now we can give a constructive definition of elimination of a variable from a set of constraints.

Definition 5.5.8. *Function* $elim(x, \overline{a})$ *eliminates variable* x *from constraints* $\overline{a}$*. From path equivalence constraints of* $\overline{a}$*, variable* x *is eliminated by eliminating it from the corresponding abstract heap, and computing equivalence constraints without* x*. Further we include all other constraints that are valid for the paths that appear in the computed equivalence constraints without* x*.*

In other constraints paths starting with x *are replaced by equivalent paths starting with other variables.*

$$\begin{aligned} \mathit{elim}(x,\overline{a}) = \;& \overline{b} \;\cup\; \{\; p :: C \mid p \sqsubset \overline{b} \;\wedge\; \overline{a} \models_A p :: C \;\} \\ & \cup\; \{\; p.\mathbf{cls} \equiv C \mid p \sqsubset \overline{b} \;\wedge\; \overline{a} \models_A p.\mathbf{cls} \equiv C \;\} \end{aligned}$$
where $\overline{b} = \mathit{HeapToCs}(\,\mathit{elimH}(x,\,\mathit{CsToHeap}(\overline{a})))$

Lemma 5.5.32. *If* wf P *and* $\mathit{elim}(x,\overline{a}) = \overline{b}$ *then* $x \notin \mathit{FV}(\overline{b})$.

Proof. Let $\mathit{elimH}(x, \mathit{CsToHeap}(\overline{a})) = \langle A;\sigma;\rho\rangle$. Then $x \notin \mathit{dom}(\sigma)$ by definition of elimH. Hence by definition of $\mathit{HeapToCs}$ none of the paths in $\overline{b'} = \mathit{HeapToCs}(\langle A;\sigma;\rho\rangle)$ can start with x. Since $\overline{b}$ can contain only paths from $\overline{b'}$ we know that $x \notin FV(\overline{b})$. □

Lemma 5.5.33. *If* wf P *and* $\mathit{elim}(x,\overline{a}) = \overline{b}$ *then* $\overline{a} \models_A \overline{b}$.

Proof. Let $\mathit{CsToHeap}(\overline{a}) = h$ and $\mathit{elimH}(x,h) = h'$ and $\mathit{HeapToCs}(h') = \overline{b'}$. Then
$$\begin{aligned} \overline{b} = \;& \overline{b'} \;\cup\; \{\; p :: C \mid p \sqsubset \overline{b'} \;\wedge\; \overline{a} \models_A p :: C \;\} \\ & \cup\; \{\; p.\mathbf{cls} \equiv C \mid p \sqsubset \overline{b'} \;\wedge\; \overline{a} \models_A p.\mathbf{cls} \equiv C \;\} \end{aligned}$$

We have to show $\overline{a} \models_A b$ for every $b \in \overline{b'}$, because the remaining constraints of $\overline{b}$ are entailed by $\overline{a}$ by their definition.

So, assuming $(p \equiv q) \in \overline{b'}$ we have to show $\overline{a} \models_A p \equiv q$. From $(p \equiv q) \in \overline{b'}$ follows $\mathit{pref}(h',p) = \mathit{pref}(h',q)$ by lemma 5.5.30. From lemma 5.5.32 we know that neither p nor q start with x. Hence $\mathit{pref}(h,p) = \mathit{pref}(h',p) = \mathit{pref}(h',q) = \mathit{pref}(h,q)$ by lemma 5.5.25. Hence $\overline{a} \models_A p \equiv q$ by lemma 5.5.24. □

Lemma 5.5.34. *If* wf P *and* $\mathit{elim}(x,\overline{a}) = \overline{b}$ *and* $\overline{a} \models_A a$ *and* $x \notin FV(a)$ *then* $\overline{b} \models_A a$

Proof. Let $h = \mathit{CsToHeap}(\overline{a})$ and $h' = \mathit{elimH}(x,h)$ and $\overline{b'} = \mathit{HeapToCs}(h')$.

Then $$\begin{aligned} \overline{b} = \;& \overline{b'} \;\cup\; \{\; p :: C \mid p \sqsubset \overline{b'} \;\wedge\; \overline{a} \models_A p :: C \;\} \\ & \cup\; \{\; p.\mathbf{cls} \equiv C \mid p \sqsubset \overline{b'} \;\wedge\; \overline{a} \models_A p.\mathbf{cls} \equiv C \;\} \end{aligned}$$

Proof by cases of a.

Case $a = (p \equiv q)$*:* Hence (1): $\overline{a} \models_A p \equiv q$ and $x \notin FV(p,q)$. By lemma 5.5.24 with (1) we can take p', q' and $\overline{f}$ such that $p = p'.\overline{f}$ and $q = q'.\overline{f}$ and $\mathit{pref}(h,p') = \mathit{pref}(h,q')$. Since $x \notin FV(p',q')$ we can apply lemma 5.5.25 to obtain $\mathit{pref}(h',p') = \mathit{pref}(h,p') = \mathit{pref}(h,q') = \mathit{pref}(h',q')$. Hence $\overline{b'} \models_A p' \equiv q'$ by lemma 5.5.31. Hence $\overline{b} \models_A p \equiv q$ by CA-Subst4 and the definition of $\overline{b}$.

Case $a = (p :: C)$*:* Hence (1): $\overline{a} \models_A p :: C$ and $x \notin FV(p)$. By lemma 5.5.10 there exists p' such that $p' \sqsubset \overline{a}$ and $\overline{a} \models_A p \equiv p'$. Then by lemma 5.5.23 we know that $\mathit{pref}(h,p) = r$ for some r. Since $x \notin FV(p)$, by lemma 5.5.25 we have (2): $\mathit{pref}(h,p) = \mathit{pref}(h',p) = r$.

By lemma 5.5.26 we can take q without cycles such that such that (3): $\mathit{pref}(h', q) = r$. Hence $q \in \mathit{Paths}(h', r)$ by definition of *Paths*. Hence (4): $q \sqsubset \overline{b'}$ by definition of *HeapToCs*. From (2), (3) and lemma 5.5.31 we get $\overline{b'} \vdash_A p \equiv q$ and (5): $\overline{b} \vdash_A p \equiv q$ by definition of $\overline{b}$. Then $\overline{a} \vdash_A p \equiv q$ by lemma 5.5.33. Hence $\overline{a} \vdash_A q :: C$ by lemma 5.5.12 with (1). Hence $q :: C \in \overline{b}$ by (4) and definition of $\overline{b}$. Hence $\overline{b} \vdash_A p :: C$ with (5) and lemma 5.5.12.

Case $a = (p.\mathbf{cls} \equiv C)$: The proof is analogous to the case $a = (p :: C)$

□

Lemma 5.5.35. *$\mathit{elim}(x, \overline{a})$ can be computed for any $\overline{a}$ and x.*

Proof. $\mathit{elim}(x, \overline{a})$ is defined as:

$$\begin{aligned}\mathit{elim}(x, \overline{a}) = \;& \overline{b} \;\cup\; \{\; p :: C \mid p \sqsubset \overline{b} \;\wedge\; \overline{a} \vdash_A p :: C \;\} \\ & \cup\; \{\; p.\mathbf{cls} \equiv C \mid p \sqsubset \overline{b} \;\wedge\; \overline{a} \vdash_A p.\mathbf{cls} \equiv C \;\}\end{aligned}$$
where $\overline{b} = \mathit{HeapToCs}(\mathit{elimH}(x, \mathit{CsToHeap}(\overline{a})))$

According to lemmas 5.5.17, 5.5.27 and 5.5.28, we can compute $\overline{b}$.

The remaining constraints of $\mathit{elim}(x, \overline{a})$ can also be computed, because there is a finite number of p such that $p \sqsubset \overline{b}$, and according to theorem 5.5.1, $\overline{a} \vdash_A a$ is decidable for any a. □

5.5.3 Decidability of Expression Typing

All expression typing rules, defined in Fig. 5.16, except the subsumption rule T-Sub, follow the structure of the expression. Thus in order to show decidability of expression typing it is necessary to eliminate the subsumption rule. This can be done by defining a relation $\overline{a} \vdash_M e : t$, which assigns minimum types to expressions. A minimum type does not need to be unique, but must have the property that it subsumes all other types of the expression.

The rules for minimal typing can be obtained from the rules of Fig. 5.16 by removing T-Sub and changing the way of eliminating variable x in rules T-Field and T-Call. For example, in T-Field, constraints $\overline{b}$ are any constraints free from x that are entailed by $(\overline{c}, \overline{a}, x.f \equiv y)$. For minimal typing we must take the most specific set of constraints $\overline{b}$, i.e., any other set of constraints $\overline{b'}$ that are free of x and entailed by $(\overline{c}, \overline{a}, x.f \equiv y)$ must also be entailed by $\overline{b}$.

$$\frac{\begin{array}{c}\overline{c} \vdash_M e : [x.\ \overline{a}] \qquad \overline{c}, \overline{a} \vdash x.f :: C \\ \overline{b} = \mathit{elim}(x, (\overline{c}, \overline{a}, x.f \equiv y))\end{array}}{\overline{c} \vdash_M e.f : [y.\ \overline{b}]}\ \text{(TM-Field)}$$

$$\frac{\begin{array}{c}\overline{c} \vdash_M e : [x.\ \overline{a}] \\ \langle \overline{a'}; \overline{b} \rangle \in \mathit{MType}(m, x, y) \qquad \overline{c}, \overline{a} \vdash \overline{a'} \\ \overline{b'} = \mathit{elim}(x, (\overline{a}, \overline{b}, \overline{c}))\end{array}}{\overline{c} \vdash_M m(e) : [y.\ \overline{b'}]}\ \text{(TM-Call)}$$

$$\frac{\overline{c} \vdash x :: C}{\overline{c} \vdash_M x : [y.\ y \equiv x]}\ \text{(TM-Var)}$$

$$\frac{\begin{array}{c}\forall i.\ \overline{c} \vdash_M e_i : [x_i.\ \overline{a_i}] \\ \overline{b} = (x.\mathbf{cls} \equiv C), \bigcup_i \overline{a_i}_{\{x_i \mapsto x.f_i\}} \\ C(x.\ \overline{b'}) \in P \qquad \overline{c}, \overline{b} \vdash \overline{b'}\end{array}}{\overline{c} \vdash_M \mathbf{new}\ C(\overline{f \equiv e}) : [x.\ \overline{b}]}\ \text{(TM-New)}$$

Figure 5.20: Minimal type assignment

Using the *elim* function we can give a constructive definition of expression typing, as shown in Fig. 5.20. To show decidability of expression typing, we need to show decidability of the $\overline{a} \vdash_M e : t$, to show that it is minimal, and to show that the minimal type can be computed for all typed expressions.

Lemma 5.5.36. *For any $\overline{a}$ and e, there is an algorithm which finds t such that $\overline{a} \vdash_M e : t$, or determines that such t does not exist.*

Proof. Definition of $\overline{a} \vdash_M e : t$ in Fig. 5.20 is syntax directed, i.e., in each rule typing of an expression is based on the types of the subexpressions. All other relationships used in the rules are decidable. The decidability of variable elimination is proved by lemma 5.5.35, and decidability of constraint entailment is proved by the theorem 5.5.1. □

Lemma 5.5.37. *If* wf P *and* $\overline{c} \vdash_M e : t$ *then* $\overline{c} \vdash e : t$

Proof. Proof by induction on derivation of $\overline{c} \vdash_M e : t$. Each rule of minimal typing is subsumed by the corresponding typing rule, because the premises of the rules are identical except rules for elimination of variable, but the required properties of *elim* are proved by lemmas 5.5.32 and 5.5.33. □

Lemma 5.5.38. *If* wf P *and* $\overline{c} \vdash e : [x.\ \overline{a}]$ *then exists* $\overline{a'}$ *such that* $\overline{c} \vdash_M e : [x.\ \overline{a'}]$ *and* $\overline{c}, \overline{a'} \vdash \overline{a}$

Proof. Proof by induction on derivation of $\overline{c} \vdash e : [x.\ \overline{a}]$.

Case T-SUB*:* Assuming $\overline{c} \vdash_M e : [x.\ \overline{a'}]$ and $\overline{c}, \overline{a'} \vdash \overline{a}$ and $\overline{c}, \overline{a} \vdash \overline{b}$ show $\exists \overline{b'}.\ \overline{c} \vdash_M e : [x.\ \overline{b'}]$ and $\overline{c}, \overline{b'} \vdash \overline{b}$.

We will take $\overline{b'} = \overline{a'}$. Then by lemma 5.4.1 and the case assumptions, $\overline{c}, \overline{b'} \vdash_M \overline{b}$.

Case T-FIELD*:* Assuming (1): $\overline{c} \vdash_M e : [x.\ \overline{a'}]$ and (2): $\overline{c}, \overline{a'} \vdash \overline{a}$ and (3): $\overline{c}, \overline{a} \vdash x.f :: C$ and (4): $\overline{c}, \overline{a}, y \equiv x.f \vdash \overline{b}$ and (5): $x \notin FV(\overline{b})$ show $\exists \overline{b'}.\ \overline{c} \vdash_M e.f : [y.\ \overline{b'}]$ and $\overline{c}, \overline{b'} \vdash \overline{b}$.

Lemma 5.4.1 with (2) allows us to replace a with a' in (3) and (4). So we have (3a): $\overline{c}, \overline{a'} \vdash x.f :: C$ and (4a): $\overline{c}, \overline{a'}, y \equiv x.f \vdash \overline{b}$. According to lemma 5.5.35 we can take $\overline{b'}$ such that (6): $elim(x, (\overline{c}, \overline{a'}, y \equiv x.f)) = \overline{b'}$. Now we can apply TM-FIELD to (1), (3a), (4a) and (6), which gives $\overline{c} \vdash_M e.f : [y.\ \overline{b'}]$. Finally $\overline{b'} \vdash \overline{b}$ follows by lemma 5.5.34 with (4a) and (5).

Other Cases: The proofs of the cases T-VAR and T-CALL are straightforward, and the proof of the case T-CALL is analogous to the proof of T-FIELD.

□

5.5.4 Checking Method Completeness and Uniqueness

5.5.4.1 Undecidability of Method Completeness Checking

We will show that completeness checking in the presence of recursive dependent classes and path dependent types is at least as difficult as providing a solver of an arbitrary Diophantine equation[9]. This means that a precise completeness checking is not possible and only approximate conservative algorithms can be defined. Sec. 5.5.4 presents such an algorithm.

Fig. 5.21 presents an encoding of Diophantine equations in DEPJ. Natural numbers are encoded by two constructors, Zero and Succ. Classes Add, Mult, and Power encode operations on natural numbers as relations $Add(a, b, c) = (a + b = c)$, $Mult(a, b, c) = (a * b = c)$, and $Power(a, n, c) = (a^n = c)$. More precisely, if x is an instance of Add, then x.a, x.b and x.c are encodings of numbers a, b and c, such that $a + b = c$. This is true, because every object must be created by the constructors declared in the program, so an instance of Add can be created only by the constructors of Add, which require that only numbers that are related by addition are given as values of fields a, b and c. The first constructor corresponds to the equation $a + 0 = a$, and the second constructor to the equations $a + succ(b) = succ(d)$ and $a + b = d$. Analogous statements hold for the classes Mult and Power.

[9]A Diophantine equation is a polynomial equation on integers.

```
1  abstract class Nat { }
2  class Zero extends Nat { }
3  class Succ(Nat p) extends Nat { }
4
5  class Add(Nat a, Zero b, a c) { }
6  class Add(Nat a, Succ(p: Nat) b, Succ(p:interm.c) c, Add(a: a, b: b.p, c: Nat) interm) { }
7
8  class Mult(Nat a, Zero b, Zero c) { }
9  class Mult(Nat a, Succ(p: Nat) b, interm2.c c,
10    Mult(a: a, b: b.p, c: Nat) interm1, Add(a: interm1.c, b: a, c: Nat) interm2) { }
11
12 class Power(Nat a, Zero b, Succ(p:Zero) c) { }
13 class Power(Nat a, Succ(p: Nat) b, interm2.c c,
14    Power(a: a, b: b.p, c: Nat) interm1, Mult(a: interm1.c, b: a, c: Nat) interm2) { }
15
16 class Fermat(Succ(p:Nat) a, Succ(p:Nat) b, Succ(p:Nat) c, Succ(p:Succ(p:Succ(p:Nat))) n,
17    Power(a : a, b : n, c : Nat) an, Power(a : b, b : n, c : Nat) bn, Power(a : c, b : n, c : Nat) cn,
18    Add(a : an.c, b : bn.c, c : cn.c) eq)
19 { abstract void solveFermat(); }
```

Figure 5.21: Encoding of Fermat's Last Theorem

Now we can, for example, encode the equation of Fermat's Last Theorem[10]: Class `Fermat` (line 16) encodes the relation $Fermat(a, b, c, n) = (a^n * b^n = c^n \ \wedge \ n \geq 3 \ \wedge \ a, b, c \geq 1)$. That is, if x is an instance of Fermat, then x.a, x.b, x.c and x.n encode solutions to the Fermat equation. The method solveFermat is abstract and must be implemented for possible instances of Fermat. Now, checking completeness of solveFermat is equivalent to checking whether the equation has solutions. Even if the Fermat's Last Theorem has recently been proven, all Diophantine equations can be encoded by using the same scheme based on the encoding of the natural numbers and the encodings of addition, multiplication and exponentiation. Hence, the completeness checker could be used as a universal solver of all these equations, which would be a solution to the Hilbert's tenth problem. The latter was, however, proved to be unsolvable.

5.5.4.2 Algorithm for Completeness and Uniqueness Checking

As shown in Sec. 5.5.4.1, a precise algorithm for checking completeness and uniqueness is not possible. Fortunately, we can define a conservative algorithm, which is sound and works for most useful cases. Our approach is to replace the quantification over all heaps with a variable matching the argument type of a method with a quantification on a finite a set of types that approximate these heaps. A set of types S is considered an

[10]The theorem states that an equation $a^n + b^n = c^n$ does not have solutions for positive numbers and $n > 2$.

approximation of a type t (approx(S, t)), if in every well-formed heap with a variable x of type t, the variable is also of one of the types in S (rule Approx-Def).

$$\frac{\forall h.\ \ \text{heap}\ \ h\ \wedge\ HC(h) \vdash \overline{a}\ \ \longrightarrow\ \ (\exists[x.\ \overline{a'}] \in S.\ \ HC(h) \vdash \overline{a'})}{\text{approx}(S, [x.\ \overline{a}])}\ (\textsc{Approx-Def})$$

Theorem 5.5.2 below states that given an approximation set for the argument type of an abstract method, its completeness can be ensured by checking that it is implemented for every element of the approximation set.

Theorem 5.5.2. *If* $\text{approx}(S, t)$ *and* $\forall[x.\ \overline{a}] \in S.\ \ (\exists\langle\overline{b}; e\rangle \in \mathit{MImpl}(m, x).\ \ a \vdash \overline{b})$ *then* $\text{complete}(m, t)$.

Proof. Follows directly from the rules Approx-Def and Complete, and transitivity of constraint entailment. □

For the uniqueness of a method it is enough to check that no ambiguities can occur between every pair of implementations of the method. Obviously, a choice between two implementations of a method m, MI and MI', is always unambiguous, if one method implementation is more specific than the other. If none of MI and MI' is more specific than the other, a third implementation MI'' must exist, which is more specific than MI and MI' and matches whenever MI and MI' match. This is obviously the case when for each pair of implementations MI and MI' in a program, a MI'' exists whose argument type is the intersection of the argument types of MI and MI'. Such requirement is, however, much too strong, because the argument types of MI and MI' are often disjoint, i.e., no well-formed heap matches both. For example, if we have an implementation of a method `draw` for the types $[x.\ x :: \mathit{Rectangle}]$ and $[x.\ x :: \mathit{Circle}]$, it does not make sense to require an implementation of `draw` for the type $[x.\ x :: \mathit{Rectangle}, x :: \mathit{Circle}]$, if there are no constructors in the program that could be used to construct objects that are both circles and rectangles.

For a better solution ensuring that two method implementations MI and MI' are not ambiguous for every well-formed heap matching both, we employ again type approximation sets. Theorem 5.5.3 states that if $[x.\ \overline{a}]$ and $[x.\ \overline{a'}]$ are the argument types of respectively MI and MI' and S is an approximation set of the type $[x.\ \overline{a}, \overline{a'}]$, then it is enough to check that MI and MI' are unambiguous for every type in S.

Theorem 5.5.3. *If for all* $\langle\overline{a}; e\rangle, \langle\overline{a'}; e'\rangle \in \mathit{MImpl}(m, x)$ *and* $e \neq e'$

1. *$\overline{a}$ and $\overline{a'}$ are not equivalent*

2. *exists S such that* $\text{approx}(S, [x.\ \overline{a}, \overline{a'}])$ *and*
   $\forall [x.\ \overline{b}] \in S.\ \ \exists \langle \overline{a''}; e'' \rangle \in \mathit{MImpl}(m, x).\ \ \overline{b} \vdash \overline{a''}\ \wedge\ \overline{a''} \vdash \overline{a}, \overline{a'}$

Then $\text{unique}(m)$.

Proof. Let's assume the opposite: $\neg\,\text{unique}(m)$. Then the premise of Unique does not hold for method m, i.e., we can find a well-formed heap h for which the set of matching implementations of m is not empty ($S \neq \varnothing$) and none of the elements of S is more specific than all other, i.e. (1): $\forall \langle \overline{b}; e \rangle \in S.\ \exists \langle \overline{b'}; e' \rangle \in S.\ (e \neq e') \wedge (\neg(\overline{b} \vdash \overline{b'}) \vee (\overline{b'} \vdash \overline{b}))$. Since by the assumption of the theorem, $\overline{b'}$ and $\overline{b}$ cannot be equivalent, holds $\overline{b'} \vdash \overline{b} \longrightarrow \neg(\overline{b} \vdash \overline{b'})$ and (1) can be simplified to (1a): $\forall \langle \overline{b}; e \rangle \in S.\ \exists \langle \overline{b'}; e' \rangle \in S.\ \neg(\overline{b} \vdash \overline{b'})$

Now, let's take any two pairs $\langle \overline{b}; e \rangle, \langle \overline{b'}; e' \rangle \in S$ such that $e \neq e'$. According to the second assumption of the theorem, exists a set R such that $\text{approx}(R, [x.\ \overline{b}, \overline{b'}])$ and (2): $\forall [x.\ \overline{c}] \in R.\ \ \exists \langle \overline{b''}; e'' \rangle \in \mathit{MImpl}(m, x).\ \ \overline{c} \vdash \overline{b''}\ \wedge\ \overline{b''} \vdash \overline{b}, \overline{b'}$.

Since $\mathit{HC}(h) \vdash \overline{b}, \overline{b'}$, then by the definition of approximation sets, there is a type $[x.\ \overline{c}] \in R$ approximating the heap, i.e., $\mathit{HC}(h) \vdash \overline{c}$. According to (2) we can take $\langle \overline{b''}; e'' \rangle \in \mathit{MImpl}(m, x)$ such that $\mathit{HC}(h) \vdash \overline{b''}$ (which means that $\langle \overline{b''}; e'' \rangle \in S$), and $\overline{b''} \vdash \overline{b}, \overline{b'}$.

We have showed that $\forall \langle \overline{b}; e \rangle, \langle \overline{b'}; e' \rangle \in S.\ \exists \langle \overline{b''}; e'' \rangle \in S.\ \overline{b''} \vdash \overline{b}, \overline{b'}$. Since S is finite and constraint entailment is transitive, $\exists \langle \overline{b}; e \rangle \in S.\ \forall \langle \overline{b'}; e' \rangle \in S.\ \overline{b} \vdash \overline{b'}$ must hold. Since this is a contradiction to (1a), our assumption $\neg\,\text{unique}(m)$ was wrong. □

Assuming that we have an algorithm that constructs approximation sets for given types, theorems 5.5.2 and 5.5.3 prescribe constructive algorithms for checking method completeness and uniqueness, because all universal quantifiers refer to finite sets: sets of method declarations in a program and type approximation sets. So, it remains to show how we construct approximation sets for types.

Figure 5.22 defines inductive rules for construction of approximation sets. These rules are derived from the properties of well-formed heaps and from the rules of the constraint system. Rule Appr-Refl defines a degenerate approximation of a type by itself. Further, we know that each object is a direct instance of one class only. Thus a type requiring some object to be a direct instance of multiple classes cannot be satisfied by any heap, thus, can be approximated by an empty set (Appr-Empty).

We also know that if an object is a direct instance of C, then there is a constructor of C in the program to which this object complies. If it is an indirect instance of C, then this classification is derived by one of the constraint entailment rules of the program. Thus $p :: C$ can be approximated by the types built from $p.\mathbf{cls} \equiv C$ and the constraints of the constructors of C, and types built from the constraints on the left side of the entailment rules (Appr-Class).

$$\text{approx}_I(\{t\}, t) \qquad (\textsc{Appr-Refl})$$

$$\frac{(p.\mathbf{cls} \equiv C) \in \overline{a} \qquad (p'.\mathbf{cls} \equiv C') \in \overline{a} \qquad \overline{a} \vdash p \equiv p' \qquad C \neq C'}{\text{approx}_I(\varnothing, [x.\ \overline{a}])} \quad (\textsc{Appr-Empty})$$

$$\frac{\begin{array}{c}(p :: C) \in \overline{a} \\ S = \left\{\ [x.\ \overline{a}, p.\mathbf{cls} \equiv C, \overline{b}_{\{\!\{y\mapsto p\}\!\}}] \mid C(y.\ \overline{b}) \in P\ \right\} \\ S' = \left\{\ [x.\ \overline{a}, \overline{b}_{\{\!\{y\mapsto p\}\!\}}] \mid (\forall y.\ \overline{b} \Rightarrow y :: C) \in P\ \right\}\end{array}}{\text{approx}_I(S \cup S', [x.\ \overline{a}])} \quad (\textsc{Appr-Class})$$

$$\frac{\text{approx}_I(S, t) \qquad t' \in S \qquad \text{approx}_I(S', t')}{\text{approx}_I((S \backslash \{t'\}) \cup S', t)} \quad (\textsc{Appr-Refine})$$

Figure 5.22: Inductive rules for building type approximation sets

Appr-Refine combines the other inductive rules and enables an iterative refinement of the approximation sets: we can take any type from an approximation set and replace it with an approximation set of that type (Appr-Refine).

Theorem 5.5.4. *The inductive construction of type approximation sets is sound:*
If $\text{approx}_I(S, t)$ *then* $\text{approx}(S, t)$.

Proof. Let's take any well-formed heap h. We need to prove that for all S and $\overline{a}$, if $\text{approx}_I(S, [x.\ \overline{a}])$ and $HC(h) \vdash \overline{a}$ then $\exists [x.\ \overline{a'}] \in S.\ HC(h) \vdash \overline{a'}$.

Let's apply induction on derivation of $\text{approx}_I(S, [x.\ \overline{a}])$.

Case Appr-Refl*:* Proved by case assumption.

Case Appr-Empty*:* By case assumptions $HC(h) \vdash \overline{a}$ and $(p.\mathbf{cls} \equiv C) \in \overline{a}$ and $(p'.\mathbf{cls} \equiv C') \in \overline{a}$ and $\overline{a} \vdash p \equiv p'$ and $C \neq C'$. By C-Subst and transitivity of entailment $HC(h) \vdash p.\mathbf{cls} \equiv C$ and $HC(h) \vdash p.\mathbf{cls} \equiv C'$, which is not possible by definition of heaps and HC.

Case Appr-Class*: Assuming* (1): $HC(h) \vdash \overline{a}$ *and* (2): $(p :: C) \in \overline{a}$
and $S = \left\{\ [x.\ \overline{a}, p.\mathbf{cls} \equiv C, \overline{b}_{\{\!\{y\mapsto p\}\!\}}] \mid C(y.\ \overline{b}) \in P\ \right\}$
and $S' = \left\{\ [x.\ \overline{a}, \overline{b}_{\{\!\{y\mapsto p\}\!\}}] \mid (\forall y.\ \overline{b} \Rightarrow y :: C) \in P\ \right\}$
show $\exists [x.\ \overline{a'}] \in (S \cup S').\ HC(h) \vdash \overline{a'}$.

From (1) and (2) we have $HC(h) \vdash p :: C$. By induction on derivation of constraint entailment follows that in derivation of $p :: C$ appears one of the rules C-Class and C-Prog with conclusion $p' :: C$ where $HC(h) \vdash p \equiv p'$. (Derivation of $p :: C$ by C-Ident is not possible, because $HC(h)$ by definition does not contain constraints of this form.)

Let's assume that $p :: C$ is derived over C-Class. Then $HC(h) \vdash p.\mathbf{cls} \equiv C$. Then by induction on derivation $(p'.\mathbf{cls} \equiv C) \in HC(h)$ for some p' equivalent to p. By definition of HC we know that p' must be some variable x, i.e., (3): $HC(h) \vdash p \equiv x$ and $h(x) = \langle C; \overline{f} = \overline{x}\rangle$. Since heap is well-formed, the object must match some constructor, i.e., $C(x.\ \overline{b}) \in P$ and $HC(h) \vdash \overline{b}$. By C-Subst with (3) we have $HC(h) \vdash \overline{b}_{\{x \mapsto p\}}$. Let $\overline{a'} = (\overline{a}, p.\mathbf{cls} \equiv C, \overline{b}_{\{x \mapsto p\}})$. Then $HC(h) \vdash \overline{a'}$ and $[x.\ \overline{a'}] \in S$, which we needed to show.

Now let's assume that $p :: C$ is derived over C-Prog. Then $(\forall x.\ \overline{b} \Rightarrow x :: C) \in P$ and $HC(h) \vdash \overline{b}_{\{x \mapsto p\}}$, but then $HC(h)$ matches a type from S', which we needed to show.

Case Appr-Refine*: Assuming* (1): $\exists [x.\ \overline{a'}] \in S.\ HC(h) \vdash \overline{a'}$ *and* (2): $[x.\ \overline{b}] \in S$ *and* (3): $HC(h) \vdash \overline{b} \longrightarrow \exists [x.\ \overline{a'}] \in S'.\ HC(h) \vdash \overline{a'}$
show $\exists [x.\ \overline{a'}] \in ((S \backslash \{[x.\ \overline{b}]\}) \cup S').\ HC(h) \vdash \overline{a'}$.

Using (1) take $[x.\ \overline{a'}] \in S$ such that $HC(h) \vdash \overline{a'}$.

If $[x.\ \overline{a'}] \neq [x.\ \overline{b}]$ then $[x.\ \overline{a'}] \in (S \backslash \{[x.\ \overline{b}]\})$ and the case is proved.

If $[x.\ \overline{a'}] = [x.\ \overline{b}]$ then $HC(h) \vdash \overline{b}$ and according to (3) we can take $[x.\ \overline{a''}] \in S'$ such that $HC(h) \vdash \overline{a''}$ and the case is proved again. □

The relation $\text{approx}_I(S, t)$ suggests an algorithm for constructing an approximation set S for t, which starts with a degenerate approximation $\{t\}$ and refines it by iteratively applying Appr-Refine in combination with one of the rules Appr-Empty and Appr-Class. However, it does not determine the order in which to apply the rules. Since Appr-Empty removes types without instances from S, it can be applied eagerly. Appr-Class is applicable on every occurrence of $p :: C$ in the types of S. By expanding a type with a constraint $p.\mathbf{cls} \equiv C$, new constraints of the form $p' :: C'$ can appear in the refined approximation set. The well-formedness rule WF-RD (Fig. 5.17) tells that paths in the new constraints are either equal to p or are extensions of it with new fields. Thus, if constraints on p are expanded after expanding constraints on all prefixes of p, no new constraints on p will be created during the further refinement of the approximation set.

Figure. 5.23 outlines the algorithm that follows the strategy to start with expanding types with constraints on the bare variable x, then with constraints on the fields of x, then with constraints on their fields and so on. Expansion of $p :: C$ can create new constraints of the form $p :: C'$, therefore this process must be repeated iteratively until no new constraints of that form are created. Since there is a finite number of class names this iteration will terminate. This is, however, not always the case for the outer

```
$Approx(t) =$
  $S := \{t\}; P := \{x\}$
  **while** $P \neq \varnothing$ **do**
    **for each** $p \in P$ **do**
      $R := \varnothing$
      **while** $\exists C.\ C \notin R \ \wedge\ p :: C$ appears in $S$ **do**
        $R := R \cup \{C\}$
        refine $S$ by rule Appr-Class on all types in $S$ containing $p :: C$
        refine $S$ by rule Appr-Empty on all types where it is applicable
    $P := \{\ p.f \mid p \in P,\ p.f$ appears in $S\ \}$
  **return** $S$
```

Figure 5.23: An algorithm computing an approximation set for a type

loop, because indefinitely long paths can be created during the refinement process. For example, for constructing the approximation of $[\texttt{x}.\ \texttt{x} :: \texttt{Nat}]$ from Fig. 5.21 the following approximation sets will be produced in each iteration:

$S_1 = \{[\texttt{x}.\ \texttt{x} :: \texttt{Nat}]\}$
$S_2 = \{[\texttt{x}.\ \texttt{x} :: \texttt{Zero}], [\texttt{x}.\ \texttt{x} :: \texttt{Succ}, \texttt{x.p} :: \texttt{Nat}]\}$
$S_3 = \{[\texttt{x}.\ \texttt{x} :: \texttt{Zero}], [\texttt{x}.\ \texttt{x} :: \texttt{Succ}, \texttt{x.p} :: \texttt{Zero}], [\texttt{x}.\ \texttt{x} :: \texttt{Succ}, \texttt{x.p} :: \texttt{Succ}, \texttt{x.p.p} :: \texttt{Nat}]\}$
...

It can be seen that from each constraint $p :: \texttt{Nat}$ a longer constraint $p.\texttt{p} :: \texttt{Nat}$ is produced[11], and the refinement process will never terminate. The termination of the algorithm can be ensured by a strict stratification of classes so that classes are assigned to different layers, and the objects of a class can refer by their fields only to the objects of classes from the lower layers.

Such a stratification prohibits recursive dependent classes, i.e., disallows the example from Fig. 5.21 and the recursive dependencies in composite structures from Sec. 4.3.3 impossible. But, it is satisfied by most of useful examples of dependent classes, including the remaining examples of Chapter 4. In such cases, the algorithm is precise, because if the algorithm terminates without further conditions, in the resulting approximation set of a type all constraints of the form $p :: C$ are eliminated, and the type is approximated by all possible compositions of constructors.

To support recursive dependent classes, we must define some termination criteria for the refinement of the approximation set, for example, by limiting the maximal number of

[11]Here p is a metavariable referring to a path, while $\texttt{p}$ is a concrete field in the program.

iterations or the length of paths p in the expanded constraints. Such a limit can be a fixed number, or it can depend on the depth of the paths in the argument types of the method declarations being checked. Termination criteria of this kind are sufficient for type checking the examples from Sec. 4.3.3, as well as for type checking usual operations on natural numbers based on the encoding of Fig. 5.21.

6 Related Work

In this chapter we present related work for virtual classes and dependent classes. Since the idea of virtual classes is not new, in Sec. 6.1 we give a detailed comparison of our implementation of virtual classes to other implementations and formalizations of virtual classes and virtual types. Since virtual classes are a special case of dependent classes, and the vc^n calculus was indeed used as basis for the latest implementation of the type system in CaesarJ, we also compare other formalizations of virtual classes against the corresponding elements of the vc^n calculus.

There is a broad range of techniques that do not support family polymorphism and dependent typing, but still address the problem of modularizing functionality involving multiple objects, and in most of cases share certain ideas with virtual classes. We review such techniques and investigate their differences to virtual classes in Sec. 6.2.

Virtual classes were designed to be used in combination with other language features of the CaesarJ language in order to support flexible integration of independently developed components. The other language features of CaesarJ are left out of scope of this thesis, however. In Sec. 6.3, we give a quick overview of the work on the component integration problem, and its relation to virtual classes and dependent classes.

Dependent classes are also related to the idea of multiple dispatch. Therefore, in Sec. 6.4 we explain the relation of dependent classes to other advanced dispatch techniques. Finally, in Sec. 6.5 we discuss the relation of the type system elements presented in this thesis to other type systems.

6.1 Virtual Classes

The implementation of virtual classes in CaesarJ, which is presented in Chapter 3, is based on the ideas of previous work on virtual classes in BETA [MMP89, MMPN93], gbeta [Ern99b] and *vc* calculus [EOC06]. Alternative definitions of virtual classes are given in Tribe [CDNW07] and Deep [Hut06]. The section also covers the approaches similar to virtual classes, but not supporting all of their properties. For example, so-called virtual types supported in νObj [OCRZ03] and Scala [Ode09] consider only the typing aspect of virtual classes, but do not support late-bound instantiation and virtual inheritance. Nested Inheritance [NCM04] and Lightweight Family Polymorphism [ISV05] do not consider virtual classes as properties of objects and instead use classes to represent families of nested classes. In this section we analyze the differences between these approaches and our implementation of virtual classes. At the formal level we also compare them with our vc^n calculus, because it was used as a basis for the implementation of type relations in CaesarJ.

The key difference between dependent classes and the related work on virtual classes as presented in this section consists in the generalization of the dependency on multiple parameters and the resulting implications for the expressiveness and of the type system, such as types depending on multiple paths.

6.1.1 Beta

Virtual classes were introduced by Madsen and Möller-Pedersen in BETA [MMP89, MMPN93], as a consequence of generalizing procedures, classes and types into so-called *patterns* and enabling extension of nested patterns in subclasses. A pattern is a structure consisting of a set of attribute declarations, which can be either objects or patterns, and a list of actions defining the default procedure of the pattern. The actions are organized into three blocks, entry, do and exit, defining respectively the inputs, the computation and the outputs of the procedure. The inheritance between patterns supported in BETA can be used for extension of both classes and procedures. A subpattern can define new attributes and extend all three action blocks. Extension of methods is based on so called *inner-calls*: The do block of a subpattern is inserted in the INNER slot of the do block of the superpattern.

A pattern attribute declared as virtual can be extended in subclasses. Patterns are said to be extended rather than overridden, because inherited definitions cannot be replaced, e.g., method implementations (action blocks) cannot be overridden, but only extended according to the inner-call semantics. An extension can be declared either as a further-binding or a final-binding, depending on whether it is allowed to be extended further. This differentiation is especially important when using the patterns as types: a further-bound type attribute is a subtype of its bound, while a final-bound type attribute is equivalent to its bound.

Virtual types, supported by BETA, are formalized in [Tor98]. The calculus is limited to single inheritance, and inheritance between virtual classes is not permitted. The calculus supports family polymorphism for instantiation and typing, but a type can refer to the virtual types of this object only, because the calculus does not support explicit family expressions in types. In the BETA language itself, references to virtual classes of other objects than this are allowed, but are not safe as argued in [Ern01].

6.1.2 gbeta

The semantics of virtual classes of BETA is further enhanced by Ernst in gbeta [Ern99b]. Differently from BETA, gbeta provides a complete implementation of the static analysis ensuring type-safety of family polymorphism [Ern01]. Furthermore, gbeta extends the inheritance semantics of BETA with a possibility to declare inheritance between virtual patterns [Ern03], propagating mixin composition [Ern99c], and a form of dynamic inheritance [Ern99c].

The propagating mixin composition of gbeta is a form of multiple inheritance based on the idea of linearizing inheritance graph, which was used in several previous languages, including CLOS [Kee89] and Dylan [Sha96]. The inheritance graph sorting algorithm of gbeta is based on a relaxed version of the C3 algorithm [BCH+96], which guarantees three desired properties: (1) local precedence order, (2) consistency with the extended precedence graph, and (3) monotonicity. The original C3 algorithm was relaxed in gbeta in order to avoid composition errors, which can be caused by conflicting ordering of parents in inheritance clauses. The superpatterns collected by the sorting algorithm are composed by a specific mixin composition semantics defined for patterns, which propagates into nested patterns.

CaesarJ [MO03, AGMO06] provides the first implementation of virtual classes and related ideas of gbeta as an extension of Java. There is a set of differences between virtual classes of CaesarJ and gbeta, mostly caused by the differences of their base languages, Java and BETA. First, differently from Java, BETA does not support abstract methods or classes. All patterns can be used as types and at the same time be instantiated. Therefore, CaesarJ additionally defines semantics of abstract virtual classes and the rules for completeness checking of methods. Second, BETA patterns can be freely composed with each other by joining their action blocks, which ensures that mixin composition is always successful. This is not the case in Java because it is based on strict method signatures, which must be preserved by subtypes. As a result, mixin composition may produce inconsistencies, which must be detected by the compiler. The further differences are the presence of constructors in Java and super-calls in methods instead of BETA's inner-calls. The underlying implementation of the operational semantics is also different: while gbeta programs are executed by a dedidated virtual machine, CaesarJ programs are translated into pure Java bytecode.

Although CaesarJ is based on the ideas of gbeta, there are still certain differences in their core semantics. The current implementation of CaesarJ is based on a different mixin linearization algorithm, and its type relationships are based on vc^n calculus, which resolves various limitations on the structure of types outlined in Sec. 6.1.3. Also, CAESARJ does not implement final-binding, inheritance from dependent types, and dynamic inheritance, which are available in gbeta.

The first proposal for an implementation of virtual types in Java was formulated in [Tho97] as a way to introduce generics to the language. The proposed implementation is based on a translation into pure Java source code and subsequent type-checking with a Java compiler. Since the advanced type system with virtual types cannot be encoded by the type system of Java [1], type annotations are translated to dynamic type checks. Differently from the proposed implementation of virtual types in [Tho97], CAESARJ provides a complete implementation of the semantics of virtual classes and propagating mixin composition and ensures static type-safety of virtual types.

6.1.3 *vc* Calculus

The ideas of gbeta are formalized in the *vc* calculus [EOC06]. The calculus gives a precise definition of the propagating mixin composition and family polymorphism based on path-dependent types. The calculus is imperative and, thus, differentiates between two kinds of instance variables: immutable fields and mutable variables. Only the former can be used in types. The heap, which is primarily used to support mutation, also introduces explicit object identities in form of locations in the heap. Explicit object identities enable precise interpretation of equivalent paths as paths referencing the same object. The calculus is defined in an algorithmic style (i.e., using syntax directed rules), which makes proof of its decidability straightforward, and is also supplied with a thorough soundness proof. The calculus does not encode all of the semantics of gbeta, however. In particular it does not support final-bindings and inheritance from superclasses of the form `path.C`.

The vc^n calculus, presented in Sec. 5.2, was developed as a generalization of *vc* calculus to support dependency of classes on multiple objects. While *vc* references to classes in types and instantiation expressions are prefixed by a reference to a family, in vc^n class references are parameterized by references to multiple families. vc^n also removes various limitations of *vc*, as explained below.

While in *vc* a class can inherit only from other classes of the same family, vc^n supports *cross-family inheritance*, because a class can inherit from multiple other classes from different families. It is only required that the set of families of a subclass must be a superset of the families of its superclasses. vc^n can even encode *inheritance from a dependent type* by a superclass clause of the form `extends path.C`, which is supported

[1] At the least it was the case for the version of Java of that time, which did not support generics.

in gbeta. This is illustrated in the following example, which encodes a hypothetical version of traditional virtual classes with inheritance from dependent types (top) in vc^n (bottom). In the encoding, E has an additional constructor parameter, but since it has a singleton type (a path), there is only one possible value which can be passed. In fact, it would be relatively easy to devise an extension of vc^n, where constructor arguments with singleton type can be initialized automatically.

```
class C {
   class D { }
}
class E extends path.D { ... }
```

```
C
D(out: C)
E(..., out: path) extends D
```

vc^n removes certain limitations on the structure of types of *vc*. The types of *vc* must be of the form $p.C$, i.e., a path followed by a class name, where the path p is of the form $\textbf{this}.\overline{\textbf{out}}.\overline{f}$, i.e., a reference to **this** followed by a list of **out** references, expressing navigation to one of its enclosing objects, and then followed by a navigation over a list fields $\overline{f}$. Note that $\overline{\textbf{out}}$ and $\overline{f}$ can be empty lists in special cases. Such form of types is limited in at least two ways. First, it does not permit mixing references to fields with references to enclosing objects, e.g., $\textbf{this}.f.\textbf{out}$ is not a valid type in *vc*. Second, it does not support so-called *wildcard types*. A wildcard type abstracts from a family object by specifying its class only, e.g., type Graph.Node refers to a virtual class Node of an arbitrary instance of Graph. As can be seen, the wildcard types could be expressed by navigation over multiple class names. For example, *vc* could not encode the signature of method areConnected of the following example. The method takes a node of an arbitrary graph, i.e., of type Graph.Node, and one more node from the same graph as the first one, i.e., of type n1.out.Node.

```
class Graph { ...
   class Node { ... }
}

boolean areConnected(Graph.Node n1, n1.out.Node n2) { ... }
```

Both limitations on the structure of types are removed in vc^n. First, the references to enclosing objects (**out**) are represented just as simple fields, which allows them to be freely mixed with other fields of paths. Besides, abandoning the special treatment of out references significantly simplifies normalization and typing of paths (See Sec. 5.2.3). Second, vc^n supports wildcard types due to the possibility to use class types as types of fields. For example, the type denoting a node of any graph can be described as Node(g: Graph). To support wildcard types the subtype relation of vc^n contains a rule comparing a path with a class type, which is not available in *vc*.

vc also imposes relatively strict constraints on the structure of expressions: field access, variable access and instantiation expressions can be prefixed only by a path rather than by an arbitrary expression. In particular, we cannot directly access members of an object returned by instantiation expressions (which are also used to encode method calls) – the returned object must be first assigned to a field. There are no such limitations in vc^n: expressions can be combined in an arbitrary way.

A variant of *vc* was used as a basis for an initial implementation of virtual classes in CaesarJ, presented in Chapter 3. In a later implementation of the compiler we switched to the propagating mixin composition semantics based on the intuition of mixins as incremental descriptions of changes to a family of classes, which was presented in Sec. 3.3.2. Also, equivalence of paths and subtyping in CaesarJ is now based on vc^n semantics[2], because it is simpler and does not suffer from the limitations of *vc* discussed above.

6.1.4 Nested Inheritance

Jx [NCM04] is another proposal for implementation of a variant of virtual classes in Java with an alternative semantics, which is called *nested inheritance*. The major difference of nested inheritance from virtual classes (like they are defined in BETA, gbeta, and CaesarJ), is that classes are seen as attributes of the enclosing classes rather than attributes of their instances. Consequently, families of classes are represented by classes in nested inheritance, and not by objects like in BETA and its successors. For example, according to the semantics of nested inheritance, class Graph and its subclass ColoredGraph, would represent exactly two different families of their nested classes Node and Edge, whereas according to the semantics of virtual classes there are potentially infinite number of families in this example, because every instance of Graph and ColoredGraph would be considered as a separate family. A practical consequence is that virtual classes would disallow mixing nodes and edges of different graph instances, while nested inheritance would allow that as long as it is statically known that the nodes and the edges belong to the same graph class.

The dynamic aspect of family polymorphism – polymorphic instantiation of classes – is achieved through dynamic classes of objects. The family of an object can be dynamically retrieved by accessing its dynamic class and then taking the prefix of that class. Then the dynamically retrieved class can be used for instantiation of its nested classes. Nested inheritance also supports dependent typing based on paths to objects. The classes can be accessed as attributes of the dynamic classes of objects referenced by paths, i.e., $p.\textbf{class}.C$. Since family objects and references to them are not available, references to

[2]More precisely, path-dependent types of CaesarJ are based on a special case of the vc^n for single paths.

family classes are expressed by so-called *prefix types*. A prefix type of the form $P[T]$, where P is a fully qualified class name, and T is an arbitrary type, refers to the family class of T[3] that is a subclass of P. Dependent inheritance relationships are enabled by supporting a special type **This** in the **extends** clause of a nested class, which represents the enclosing family class.

Jx is superseded by J& [NQM06], which introduces *nested intersection*. Nested intersection is a form of multiple inheritance, which propagates into nested classes, and thus is comparable with propagating-mixin composition. The inheritance semantics of nested intersection resembles virtual inheritance of C++, i.e., superclasses are treated symmetrically, and repeatedly inherited superclasses are shared. Thus, differently from mixin-composition, method implementation conflicts resulting from multiple inheritance are resolved manually and conflicting super calls must be qualified by class names. J& also extends the type system of Jx with intersection types. Intersection types are also available as dynamic types of objects. A formal definition of J& and its soundness proof are given in the technical report version of [NQM06].

As it is noted in [EOC06], the classes-in-classes model can be trivially encoded by the classes-in-objects model using singleton instances of family classes, but the converse does not hold. On the other hand, in [NQM06], it is argued that the advantage of abandoning references to family objects enables more flexible inheritance relationships between classes, in particular it allows a class to inherit from more deeply nested classes.

Another consequence is that since the family is not available as an explicit object it cannot be directly referenced by variables and passed through method calls. Furthermore, instances of nested classes do not contain a reference to the family object, which could be used to keep shared state of the objects of a family. Since fields of the family object can be used to represent its configuration, the class-in-class model limits the possibility of describing shared variations of multiple objects.

Technically the latter limitation manifests itself through the derivation of the path equivalence. The type of the form p.**class** is considered not as a singleton type including only the object referenced by p, as it is in the classes-in-objects model, but as a type representing all instances of the dynamic class of p. This limits path aliasing possibilities, because from the fact that p' is of type p.**class**, we cannot derive that $p.f$ is of type $p'.f$.**class** for some valid field f. Consequently, if two objects belong to the family represented by p.**class**, we cannot derive that they also share the families represented by the fields of p.

The formalizations of Jx and J& resolve some of the limitations that we identified for the *vc* calculus in Sec. 6.1.3. Although the Jx inheritance graph is very much constrained by the single inheritance, J& provides much more flexibility and allows a class to inherit

[3]A class may have multiple families as a result of multiple levels of nesting.

from classes of multiple families. Note, however, that differently from *vc* and vc^n, all families of Jx are statically known and a global analysis of resulting inheritance graph is possible. Virtual inheritance in J& is still supported only relative to the path **this**. Also, differently from *vc*, Jx and J& support wildcard types.

The expression structure of Jx has analogous limitations like *vc*: most of expressions can be prefixed only by paths. Method parameters are even limited to variables and values. The limitations on the expression structure are removed in the definitions of J&, but at the same time it limits field types to static types only, which is a quite severe limitation, because it disallows describing relations between objects of the same family. It also significantly simplifies proofs related to typing of paths, field access and assignment expressions.

The formal definitions of Jx and J& are much more complicated than the vc^n and DC_C calculi presented in Chapter 5. Much of the complexity is created by the usage of prefix-types instead of fields to refer to the families, because paths and prefix-types must be then treated in a special way. Despite the complexity of the J& calculus, it intensively uses declarative rules, i.e., using variables in premises of the rules that are not bound in their conclusions, and thus does not give a clue on how the proposed semantics could be implemented algorithmically. No statement about decidability of the calculus is given.

6.1.5 Tribe

Tribe [CDNW07] is one more calculus formalizing virtual classes, which simplifies and generalizes the *vc* calculus [EOC06] in certain respects. Much of its simplification is achieved through abstraction from a concrete method dispatch semantics, i.e., in the cases when multiple method implementations match a method call, Tribe does not define how the most specific one is selected. Such simplification avoids the complexity of mixin linearization in *vc*. Further simplification is achieved by giving up the algorithmic style of *vc* and relying on declarative rules instead. In particular, path normalization algorithm is replaced by declarative equivalence rules integrated into the subtyping relation.

Explicit definition of variable substitution in dependent types is avoided by using lexicographic substitution of variables and relaxing the structure of types so that every substitution produces a valid type. As a result, the calculus resolves the limitations of the structure of types in *vc*, which were discussed in Sec. 6.1.3: in paths we can freely mix fields and out references; wildcard types are supported in form of navigation over multiple class names. Moreover, free substitutability of variables by types also produces novel types, described by classes followed by navigation over fields. For example, type Account.person, where Account is a class and person is a field of that class, defines the set of all persons referenced by at least one account object.

Tribe gives up the unification of classes and methods of *vc* and introduces local variables, which makes it closer to the conventional object-oriented languages. Consequently, paths in the language can start not only from `this` but also from method arguments and immutable local variables. Instead of immediately initializing fields by constructor parameters, as it is done in *vc*, in Tribe fields are initialized by assignment expressions. This makes it possible to assign values to the fields that are different from the ones given by a constructor call, and to extend the set of fields in furtherbindings of a virtual class. The cost of the assignment expressions is that Tribe cannot statically guarantee that all fields are initialized and are not reassigned to different values, and instead generates runtime errors in such situations.

Tribe resolves only a part of the limitations of *vc*. It resolves the limitations on the structure of types in *vc*, as was explained above, but not the limitations of the structure of expressions. Field assignment, method call, and instantiation expressions can be prefixed only by paths and can take only paths as parameters. The paper [CDNW07] also proposes an extension of Tribe with special constructs (adoption and over-the-top types) to allow inheritance from top-level classes, in this way supporting a bit more flexible inheritance graph than *vc*.

The subtyping relationship in Tribe is defined in a declarative style, which makes it difficult to prove its decidability. Despite that, the type system of Tribe of is not obviously simpler than the type system of the vc^n calculus, which is defined in an algorithmic style. Unlike vc^n, however, Tribe is an imperative calculus, which makes it closer to the typical object-oriented languages.

6.1.6 Deep

Virtual classes are also supported in the Deep calculus [Hut06], the main highlight of which is removing the distinction between terms and types, i.e., all syntactically valid terms can also be used as types. Consequently, the type system supports general dependent types, including arbitrary functions. Another consequence of the unification of terms and types is that record terms can be interpreted both as descriptions of objects and as record types. Unification of types and terms means not only that every valid term can be used in types, but also that types are available as first class values and can be used as result of computations.

Inheritance in Deep is expressed through composition of records: an identifier of a superclass, which refers to some record definition, can be composed with a record defining additional members or overriding some of the existing ones. Overriding semantics is achieved through asymmetric record composition semantics. Multiple inheritance can be expressed as composition of multiple records. Virtual classes can be described as nested records bound to labels of the enclosing record. Both further- and final-binding of record labels are supported, which in principle encodes virtual classes. The semantics of overriding and composing virtual classes is encoded through the propagation of record composition into subrecords. The composition semantics is based on a straightforward asymmetric merging of records, which is different from the mixin linearization approach described in Sec. 3.3 and does not support the desired properties of linearization.

Unlike several other calculi of virtual classes [EOC06, CDNW07, NCM04], Deep does not impose strict limitations on the expression structure, and even introduces new kinds of expressions – function abstractions and record compositions. Record composition is constrained to be static, because well-formedness rules require that every composition can be statically expanded to an equivalent record. Thus, although cross-family inheritance is supported by the calculus the families must be statically known.

Deep does not impose any restrictions on the structure of paths in types, and even supports dependency on arbitrary expressions, including function abstractions and applications. Type equivalence rules include beta-reduction of function terms. The support for more powerful dependent types is, however, the major reason of the undecidability of the calculus, which is proved in the paper.

In spite of the flexibility of the structure of types, wildcard types are not supported in Deep, e.g., a type like Graph.Node could not be used to abstract from a concrete graph family. According to the subtyping rules, $t.l$ is a subtype of $t'.l$ (where t and t' are types, and l is a record label), only if t is equivalent to t', e.g., a type g.Node for some identifier g of type Graph would not be a subtype of Graph.Node according to the rule. As explained in the paper, the reason for this limitation is potentially unsafe usage of such types in contravariant positions, i.e., in the method arguments.

Although unification of types and terms provides a lot of expressive power, while keeping the calculus relatively small, the lack of distinction between terms and types can also create certain limitations. One disadvantage of the proposed unification is that it eliminates abstract types, i.e., types without direct instances, because every type has a syntactically equivalent term as its instance. As a result, abstract method declarations do not make sense in such type system, because every method must be implemented for the immediate instances of its parameter types. The unification of objects and classes also eliminates distinction between singleton types and class types, because every object can be potentially used as a class. Because of lack of this distinction, it is not possible to treat class types and singleton types differently in contravariant positions, which is the reason of the restricted subtyping rule between $t.l$ and $t'.l$.

6.1.7 νObj calculus

The νObj calculus [OCRZ03] is the first solid formalization of virtual types based on path-dependent types. The calculus does not cover dynamic aspects of virtual classes, such as virtual inheritance, polymorphic instantiation, and composition of families. The calculus provides a mixture of nominal and structural typing. The structural typing is supported through record types, class types, and type intersection. The calculus supports virtual types as type members of objects, which can be also specified in record and class types. The record types with bounded member types can be used to encode generics in νObj. Nominal typing is achieved through names of objects and abstract type members. These names are used for declaration of explicit subtyping relationships beyond structural comparison. νObj even supports a special kind of nominal binding, which completely specifies the the interface of a type member, but still treats the type as different from structurally equivalent types. The paper does not elaborate on the usefulness of this kind of bindings, though.

The calculus defines an interesting operational semantics, which introduces object identities while keeping the calculus in a functional style, i.e., without a mutable heap of objects. The reduction rules do not substitute created objects in the expressions that use them, but keep them assigned to unique identifiers as part of the expression structure. Consequently, reduction of an expression works with a structure comparable to a heap: a list of assignments from identifiers to objects, followed by the yet unevaluated part of the expression. According to the soundness statement of the calculus, the result of evaluation is a list of fully evaluated object bindings, followed by a value. A value either is an identifier, referencing to an object, or a class. The operational semantics of νObj influenced the design of our DC_C calculus, which also uses an immutable heap structure for the purpose of having explicit object identifiers, and treats such identifiers as values.

Another peculiarity of the calculus is the representation of classes by expressions instead of a global class table. The classes, described as record structures supplied with a local self-reference and a self-type, can be used to instantiate objects and be composed, in this way providing a form of inheritance. The explicit self-type declaration of a class can be different from the one that can be implicitly inferred from the definition of the class. This difference is used to encode abstract members of the class: the members that appear in the self-type, but not in the class definition are considered as abstract, and only classes that are statically known to be concrete can be instantiated. The difference between the self-type and the class definition is also exploited for encoding methods by classes: the member encoding method argument is declared as abstract, this way requiring to bind it in method calls. Classes can be composed dynamically, but a precise self-type of the resulting composition must be statically known. Composition does not provide propagating semantics, and thus do cannot encode virtual inheritance and compositions of class families.

The expression structure of the calculus is quite constrained, because the instantiated object must be immediately assigned to a variable. The expression structure requires binding objects to variables before using them. Recursive nesting of expressions is possible only through class structures. Such limitation is comparable to the requirement to prefix expressions by paths in *vc*.

The type system supports singleton types based on paths starting either with object identifiers or local self-references of classes and records types. Explicit `out` references in paths are replaced by the possibility to refer to self-references. This, however, imposes a limitation on paths analogous to *vc*, because it is not possible to refer to the enclosing objects of fields. Wildcard types, i.e., the possibility to abstract from specific families, are supported through type selection.

The calculus is defined in a declarative style and is proved to be sound and confluent, but undecidable. The confluence proof is necessary because of undeterministic reduction rules. Undecidability proof is based on encoding System $F_{<:}$ [CMMS91] in the calculus.

6.1.8 Scala

Scala [Ode09] is a full-fledged programming language, integrating features of object-oriented and functional languages. Scala is compiled to Java bytecode, but is not defined as an extension of Java. Scala provides a lot of advanced language features, but in this context we are interested only in its support for virtual types.

The semantics of Scala integrates most of the ideas of the νObj calculus: Scala supports abstract and finally-bound type members, type intersection, path-dependent types, mixin composition, explicit self-references and self-type declarations in class declarations and types. Although it is claimed that generics can be encoded by virtual types of Scala [CGLO06], the language provides special features for parameterization by types with the capabilities comparable to Java generics and beyond it, e.g., Scala supports variant annotations on type parameters and parameterization by type constructors [MPO08].

Since νObj is an undecidable calculus, it cannot be directly implemented. The core of the actual Scala type system is described in the FS_{alg} calculus [CGLO06]. The declarations of the calculus follow the Scala syntax: it introduces explicit multiple inheritance between classes, and a syntactic distinction between classes (declared as traits), methods, fields, and type members. The major simplification of FS_{alg} with respect to νObj is replacing first-class classes by fixed class declarations in a program. There is also a set of features available in Scala, but not in FS_{alg}. Explicit self-type declarations are not supported in

FS_{alg}. The type members can be declared only either as unbound or as finally-bound. Type selection must be prefixed by paths, which makes it impossible to describe wildcard types. The main contribution of the calculus is that it is defined in an algorithmic style and is supplied with a proof of its decidability.

The strategy of path normalization and typing of FS_{alg} calculus was taken over in our vc^n calculus. It, however, appeared that the path normalization algorithm of FS_{alg} was unsound in our context, and we had to extend it with an additional rule, which propagates paths normalization into their prefixes (See rule $\rightsquigarrow$-Field1 in Fig. 5.4). We also found that in FS_{alg} (and in Scala), a non-singleton type is compared with a path by comparing the type with the bound of the path. This is less expressive than our subtyping rule <:-PathClass (Fig. 5.6), especially for paths including out references. For example, the call to areConnected in the method test of the listing below, would be not accepted, because it won't be determined that the type of the second argument in the call matches, i.e., n is a subtype of n.out.Node. The problem is less critical in Scala, because it does not support out references in paths and dependency of types on method parameters anyway.

```
boolean areConnected(Graph.Node n1, n1.out.Node n2) { ... }
boolean test(Graph.Node n) { areConnected(n, n); }
```

Like νObj, Scala implements only virtual types, but not virtual classes, i.e., it does not support polymorphic instantiation of classes and virtual inheritance. Although the mixin composition semantics of Scala is based on linearization inheritance graph like in gbeta and CaesarJ, the composition is not propagated into the nested class declarations.

6.2 Flexible Modularization Techniques

There is a number of languages that do not implement virtual classes, but still address the problem of flexible decomposition of software. Nevertheless, it is difficult to draw a clear line between approaches closer to virtual classes and the approaches closer to extensible modules. Lightweight family polymorphism [ISV05], although being a completely static approach, is largely based on ideas of virtual classes. The implementation of mixin layers based on C++ templates [SB98a] supports variations at the level of class collaborations, but their implementation in Java [BSR03] supports only variations of the entire application.

6.2.1 Lightweight Family Polymorphism

Lightweight family polymorphism [ISV05] proposes a lightweight solution to the problem of extending a group of mutually recursive classes, which avoids dependent typing. The proposed .FJ calculus considers enclosing classes as families of nested classes, and thus follows the classes-in-classes model. Nested classes of a family can refer to each other by relative types of the form $.C$, which are rebound to new versions of the classes in a subclass of the family class. Nested classes cannot have inheritance relationships other than the implicit inheritance from their furtherbounds, and a type based on a nested class does not have any sub- or supertypes except itself. Consequently, ColoredGraph.Node is not a subtype of Graph.Node even if ColoredGraph is a subclass of Graph. Such subtyping is prohibited in order to guarantee that only objects of the same family (i.e., enclosed by the same class) are used together.

Polymorphic usage of families is enabled by generic methods with type variables bounded by family classes. The arguments of such methods can be typed by the nested classes of these type variables. For example, method connect parameterized by a type variable X with the bound Graph, can take two objects of type X.Node as parameters. The method is polymorphic with respect to the graph family, because it can be used with nodes of different subclasses of Graph as long as it is ensured that the nodes are from the same graph class.

The type system of [ISV05] is very minimalistic, but it is extended with so-called variant types in ^FJ calculus [IV07], which enable differentiation between exact and inexact class references. The ^FJ calculus also extends .FJ with support for inheritance between nested classes and arbitrary depth of class nesting. The inexact references to classes enable description of wildcard types, e.g., in the extended type system Graph.Node describes a node of any graph family which is a subclass of Graph, while @Graph.Node is a node of precisely Graph family.

The expressive power of the lightweight family polymorphism is limited. It does not support polymorphic instantiation of classes. Inheritance is limited to single inheritance, which excludes a possibility of family composition. Polymorphic usage of family types is limited to the code inside the family and type variables of methods. Generic method calls must be statically instantiated with concrete family types, which means that a precise family type must be always statically known. If the precise type of a family object is lost during dynamic computations, the resulting imprecise reference to the family cannot be used with methods that are polymorphic with respect to a family. For example, if we have an imprecise reference to an instance of Graph, we cannot retrieve two nodes of the graph and use a polymorphic connect method to connect them.

[KT07] identifies the problem of coupling between the classes of a family, caused by the requirement to nest all such classes within the family class. Like in case of dependent classes, the proposed solution is based on type parameterization: the family class such as Graph is parameterized by the member classes of the family such as Node and Edge, and the other way around. The solution very much resembles modeling of virtual types by generics, with a slight innovation that enables referencing the type parameters of other type parameters and in this way reducing the number of type parameters required to express type dependencies among mutually related classes, e.g., it is sufficient to parameterize Node and Edge classes by the type of their graph, but not by the types of each other.

Nevertheless, the proposed solution of [KT07] is very verbose, thus a further paper of the same authors [KT08] proposes a more compact solution using so-called *lightweight dependent classes*. In the improved solution they switch back to the nested declarations of member types within a family class, but implementations of the member types are given by classes declared outside the family class and explicitly parameterized by the type of the family.

The latter classes are called lightweight dependent classes by analogy to our dependent classes, which also express dependency on a family through parameterization. In other respects lightweight dependent classes are very different from dependent classes, because they support neither any form of dynamic dispatch of classes, nor membership of a class in multiple families. The proposals of [KT07] and [KT08] do not completely solve the problem of coupling of member classes of a family either, because the family class still must contain a complete list of its member classes, which must be given in form of type parameters in [KT07], and in form of class members in [KT08].

6.2.2 Mixin Layers

The idea of mixin layers [SB98b, SB02] is to generalize mixin composition for group of classes, which is very close to the idea of the propagating mixin composition [Ern99c]. A concrete implementation of mixin layers using C++ templates is described in [SB98a]. A mixin, i.e., a class parameterized by its superclass, can be implemented as a C++ template class inheriting from its parameter. This technique is generalized to a group of classes by declaring them as nested classes of C++ templates (the templates are then called *layers* or *collaborations*). If a nested class is refined in a layer, it is declared as a subclass of the corresponding nested class of the template parameter. By inheriting from the classes of a template parameter rather than from static classes, the classes of a layer are made composable with the classes of other layers. The proposed implementation of mixin layers is an improvement of a previous proposal of implementing collaboration-based designs with C++ templates in [VN96a]. The innovation of mixin layers is usage of nested classes within C++ templates, which can substantially reduce the number of necessary template parameters to describe dependencies between classes.

The problem of using C++ templates is that they are not properly typed. The parameters of a template are not typed and it is not possible to check if the template uses the parameters correctly. Only concrete instances of templates are checked for consistency.

Family polymorphism, i.e., instantiation of nested clases or type references to them, is statically resolved during template instantiation.

In [SB98a] it is emphasized that mixin layers are an idea not bound to a particular language, and sees C++ templates as one possible implementation of it. An implementation of mixin layers for Java is available as the Jak component of JTS [BLS98] and AHEAD tool suites [BSR03]. Differently from the implementation of mixin layers in C++, the Jak applies composition at the level of packages. Consequently, such implementation supports variation binding only at the scope of an entire application. Again, only concrete compositions of mixin layers are statically checked in this approach. Instead of providing modular checking for mixin layers, the research of AHEAD is directed to a global consistency checking of all valid compositions of layers within the scope of a software product line [TBKC07, DCB09].

6.2.3 Classboxes

Bergel et al. [BDW03, BDN05] propose a module system supporting locally visible extensions to groups of classes, encapsulated within modules called *classboxes*. A classbox encompasses a set of classes, which are either declared within it or imported from other classboxes. An imported class can be refined within the scope of the importing classbox, which has an effect that the new version of the class is rebound to other classes of the classbox. In particular, overriding of a method in a classbox has a local effect, i.e., the new implementation of the method is executed only if the method is called in the context of the classbox. Consequently, multiple versions of a particular class with different implementations of the same method can coexist within the same application, which is the most distinguishing characteristic of classboxes with respect to other approaches of extensible modules.

The classboxes were originally proposed for Smalltalk [BDW03], and were later implemented for Java [BDN05]. Although the Java-based implementation of classboxes is emphasized as an implementation for a statically typed language, no statements on soundness of the proposed module system are given.

The classboxes are similar by their idea to the refinement of class families supported by virtual classes. Differently from family classes, classboxes are modules rather than classes, and thus do not support dynamic family polymorphism. Also, classboxes do not provide any mechanism analogous to propagating mixin composition, which would allow for composing independent extensions to classes. Instead, the classbox can import only one version each class.

6.2.4 Open Classes

MultiJava [CLCM00] is an extension of Java with *open classes* and multimethods. The idea of open classes is to enable extension of an existing class by supporting declarations of methods and fields outside the declaration of the class. The need for such extensions is motivated by the so-called expression problem [Tor04], i.e., independent extensions of programs with new data types and new operations on these data types.

Open classes are a pragmatic approach for solving the extensibility problem with a well-defined module system, supporting modular type checking and incremental compilation. However, modular checking of open classes is achieved at the cost of flexibility of decomposition: an implementation of a method must be located in the module of its class or in the module where the method was introduced. Besides, differently from propagating mixin composition [Ern99c] and mixin layers [SB98b], implementations of existing methods of classes cannot be overridden or extended.

Support for declarations of methods outside classes is in general quite common in languages supporting multidispatch, e.g in Cecil [Cha92], Dubious [MC99], Common Lisp [DG87], Dylan [Sha96], and Extensible ML [MBC04]. Extensible ML supports modular checking by imposing constraints on extensibility similar to the ones of Multijava. Dubious is a core language extended with four different type system to demonstrate the trade-offs between extensibility and modularity of type checking, ranging from a system supporting full extensibility and global type checking to a system with the constraints analogous to the ones of Multijava. Other languages – like Cecil, Common Lisp and Dylan – do not impose any restrictions on extensibility. These languages are either dynamic (Common Lisp and Dylan), or require the entire program for type checking (Cecil).

6.2.5 Keris

Keris [Zen02] is an experimental programming language extending Java with explicit support for manufacturing extensible software. It introduces modules as the basic building blocks, which are composed hierarchically, explicitly reflecting the architecture of the system. The composition is achieved by module aggregation. Modules can be extended in a non-invasive and in a statically type-safe manner. Using *virtual class fields*, classes contained in a module can be covariantly refined in any extensions of that module.

Keris uses aggregation while CAESARJ uses the inheritance mechanism to compose new modules (collaborations). A clear disadvantage of our composition by inheritance is that it does not always reflect the hierarchical structure of the software. Furthermore, aggregation can be seen as a better dependency firewall. However, beside the common semantics of code reuse and subtype generation, the inheritance in our case also serves for importing abstractions defined in a collaboration interface, which are very precise and stable. Considered from this point of view, the coupling is equivalent to a simple association. Furthermore, Keris uses module interfaces and dependency inference to automatically connect dependent modules deployed in a context. CaesarJ requires modeling the collaboration interface in order to let two modules communicate with each other. The automatic inference is not required in CaesarJ, since all dependencies are imported by inheritance as an abstraction, which can be replaced in a final combination with the desired implementation. Dependency inference is an interesting feature, however, we believe that it complicates the extension mechanism since it requires separation between specialization and refinement.

6.2.6 Aspect-Oriented Programming

Aspect-oriented programming (AOP) emerged as a result of critique on inflexibility of software decomposition in mainstream object-oriented programming languages. Although the concept of aspect-oriented programming was explicitly postulated in [KLM+97], it integrated the earlier ideas of subject-oriented programming [HO93], composition filters [AWB+93], and adaptive programming [LSLX94, Lie96]. AOP postulates the need to modularize so-called *crosscutting concerns* and to express them in a more concise way. A more concise expression of a crosscutting concern is achieved through avoiding repetition of the code implementing that concern in multiple places of a program, and by using special expressions (known as *pointcuts*) to quantify over locations in the static and dynamic structures of a program (known as *joinpoints*), where the code of the concern has to be inserted.

6.2.6.1 AspectJ

The mainstream AOP, most prominently represented by AspectJ programming language [KHH+01], follows so-called *asymmetric* approach, which differentiates between the base code, implementing the core functionality of the program, and the aspects, implementing more specific cross-cutting concerns. The control flow of the base code is extended by the *pointcut-advice mechanism*, and the static structure is extended by so-called *intertype declarations*.

Although the aspects could be used to modularize application-level variations like extensible modules, the resulting modularization suffers from a set of problems. Intertype declarations are especially problematic, because they create implicit dependencies between modules, i.e., a module can refer to the members introduced by aspects without declaring dependencies on those aspects. Modularity of the pointcut-advice mechanism is criticized in [Ald05], which identifies the problems of stability of pointcuts and unexpected changes to the behaviour of a program introduced by the advice, and proposes modules with explicit interfaces for aspects, called *open modules*. The open modules, however, severely constrain the advantages of aspects: their quantification possibilities and obliviousness of the base code.

The extensions achieved by virtual classes and propagating mixin composition avoid the problems related to the modularity of intertype declarations, because dependencies between such extensions are explicitly declared, and abstract family classes can be even used to define explicit interfaces between them. The problem of stability of pointcuts does not exist, because virtual classes do not support quantification, and thus explicitly refers to the overridden methods. Overriding possibilities in virtual classes can be constrained by declaring selected methods as final, i.e., as not overridable.

6.2.6.2 HyperJ

The HyperJ [TO99] programming language is based on the ideas of the subject-oriented programming [HO93] and repesents the symmetric AOP approach, i.e., differently from AspectJ it does not distinguish between base code and aspects. Instead, it decomposes software into uniform *hyperslices*. The hyperslices are composed by special composition specifications, which define the classes to be merged and resolves name conflicts.

Decomposition into hyperslices create a specific modularization structure, because each hyperslice must be self-consistent, i.e., completely independent from other hyperslices. Each hyperslice defines the abstractions, which it expects from other hyperslices, but these abstractions are not shared. This is different from the modularization achieved by virtual classes and propagating mixin composition, because we can declare explicit dependencies between family classes in form of inheritance relationships, as well as shared interfaces. The explicit dependencies and interfaces make the large-scale design of software more explicit and enforces contracts between its components.

6.2.7 Expanders

The problem of describing dependent object extensions is also addressed by expanders [WSM06], which for example can be used to express the dependency of adapters on their adaptees. In some sense, expanders could be seen as dependent classes of the objects that they expand. The difference is that expanders share the identity of the objects they expand, while dependent classes construct new objects, which can have a many-to-one relationship with their parameter objects. That is, by using dependent classes one can create multiple adapter instances of the same type for the same adaptee. On the other hand, expanders provide a solution to the problem of recycling of stateful adapters.

6.3 Integration of Independent Components

As explained in [MO03], crosscutting concerns can be modularized by AspectJ-like aspects, but these aspects do not have their own structure. Instead they are quite tightly coupled to the structure of the base code, and thus are not reusable. The same can also be said about other large-scale extension mechanisms, including virtual classes and propagating mixin composition, because they require that extensions are based on the same structure so that they can be automatically combined using some form of name-based merging.

Such solutions are not suitable for integration of independently developed reusable components, which can have different structure, i.e., are defined as independent class collaborations. The structural alignment of components also contradicts to the idea of adaptive programming [LSLX94] to define each piece of functionality in the most suitable abstractions, and in this way maximize its reusability. Moreover, the compositions defined by aspects or by various large-scale inheritance mechanisms are of static nature and do not support dynamic composition of components. These problems are addressed by Adaptive Plug and Play Components (APPCs) [ML98], Aspectual Components (AC) [LLM99] and Pluggable Composite Adapters (PCA) [MSL00], which propose various solutions for dynamic composition of components with independent structure.

6.3.1 CaesarJ

The CaesarJ programming language originated as a successor of the previous proposals for integration of reusable components within independent structure in APPCs [ML98], AC [LLM99] and PCA [MSL00]. In [MO02] a notion of collaboration interface was introduced to separate component implementation from its binding with other components. Virtual classes were identified as a useful technique for defining extensions of components, their bindings and collaboration interfaces. [MO03] introduced AspectJ-like pointcuts, advices and dynamic binding. It was showed how these mechanisms together help to decompose software into reusable components with independent structure, and combine them in a crosscutting way. In [MO04] it was demonstrated how to use CaesarJ to separate the features in a feature-oriented design into modules with independent internal structure.

In [AGMO06] the design of the language was generalized by introducing a full-fledged implementation of virtual classes and propagating mixin composition. The earlier specific language constructs were encoded as design patterns of virtual classes: collaboration interfaces, component implementations and bindings were uniformly implemented as abstract family classes; the specific binding relation was encoded by virtual inheritance; the specific composition of bindings and implementations was replaced by the generic propagating mixin composition. Only the specific *wrapper* construct was preserved to implement stateful adapters and their late-bound instantiation with respect to the adaptees. At the same time, the dynamic aspect deployment was generalized to support deployment on different scopes, including remote deployment. The resulting CaesarJ language presents an integration various general-purpose language features: pointcut-advice mechanism, dynamic aspect deployment, virtual class, and propagating mixin composition. These features are useful both in isolation and in combination with each other, as was demonstrated in [AGMO06]. The scope of this thesis was limited only to the virtual classes and propagating mixin composition implemented in CaesarJ and their application scenarios.

Dependent classes enable a further generalization of CaesarJ's language design, because they can encode both virtual classes and late-bound wrappers, which are necessary to define dependency of component instances on variations of an application in a non-invasive and extensible way. Consequently, generalization of virtual classes to dependent classes in CaesarJ would make it possible to abandon specific language constructs for late-bound wrappers. Furthermore, implementation of wrappers by dependent classes also gives proper types for wrappers, and in particular enables dependent typing with respect to the wrappees. In fact, the scenario of framework instantiation and dependency to application-specific variations, presented in Sec. 2.5.2 is very close to the scenario of component integration presented in [AGMO06]. The solution with dependent classes presented in Sec. 4.4.3.1 demonstrates usage of dependent classes instead of late-bound wrappers. The wrappers are defined as dependent classes dispatched by two objects: the framework (or component) instance and the wrappee. With dependent classes dependencies on both objects are expressed in a uniform way and have the same properties.

6.3.2 ObjectTeams

ObjectTeams [Her03] is another language addressing the problem of decomposing software into reusable components with independent structure, and also has the roots in APPCs [ML98], AC [LLM99] and PCA [MSL00]. Although ObjectTeams and CaesarJ are based on an analogous motivation, they follow different language design philosophy. While CaesarJ seeks to solve the component integration program by employing general-purpose language features, such as virtual classes or classical AOP, as much as possible, ObjectTeams is based on a set of specific language features supporting their design methodology.

The class collaborations implementing components are described by special *team* structures, consisting of a set of *roles*. The roles are connected to the base classes of the application, using so-called *call-in* and *call-out* constructs. The roles provide semantics analogous to the one of virtual classes, i.e., they can be overridden in subteams, which enables incremental refinement of teams and separation of generic team implementation from its concrete binding to application classes. A type system comparable to that of path-dependent types ensures that roles are used only in the context of their owner team. ObjectTeams does have a construct analogous to propagating mixin-composition, which would make team binding composable with various implementations of the team. Dynamic role selection in ObjectTeams [Her03] addresses the problem of expressing dependency of roles on the type of the classes playing by roles, and thus are analogous to the late-bound wrappers of CaesarJ [AGMO06], but are based on a different dispatch semantics. In principle they also describe dependency on multiple objects: the roles depend on both the type of their team and the object playing the role. Thus, they could also be implemented as dependent classes.

The advantage of the specific language design is that it can make the intended design patterns more concise, e.g., call-ins describe interception of the methods of a base class in a role more concisely than the pointcut-advice mechanism of CaesarJ. Also, the automatic conversion between base classes and roles makes it possible to avoid explicit navigation between wrappers and wrappees in CaesarJ.

6.4 Dynamic Dispatch

6.4.1 Multimethods

Dependent classes can be seen as an idea of multimethods [DG87, Cha92, Sha96, CLCM00] reapplied to dispatch of classes, as it is provided by virtual classes. In Sec. 4.2.5, we have demonstrated how multidispatch of methods can be encoded by dependent classes. A complete encoding in the opposite direction is not possible, because multimethods per se do not extend the type system and thus cannot encode the enhanced type system of dependent classes. Encoding of instantiation of dependent classes by means of multidispatched factory methods in combination with some form of multiple inheritance is hard too, because a factory method would need to be defined for each possible combination of the declarations of a dependent class. Even more problematic is the modeling of the subclasses of dependent classes, because a subclass implicitly inherits all variations of its superclass.

As was explained in Sec. 4.2.5, dependent classes enhance dispatch semantics of multi-methods by enabling dispatch over the types of the fields of objects. The possibility to dispatch over a deeper structure of objects is also supported by ExtensibleML [MBC04], which is expressed by dispatch over patterns based on expansion of data constructors. It is, however, not possible to express covariance between the objects given as parameters to a method by relating the values of their fields, like we have done that using path-dependent types.

6.4.2 Predicate Dispatch

Predicate dispatch [EKC98, Mil04] is a generalization of multi-dispatch. With predicate dispatch, a method can have multiple declarations with different predicate expressions. A method declaration is applicable to the argument values of a method call when the predicate expression evaluates to true. The dispatch selects the most specific declaration of the method that is applicable for the given arguments. The predicate expression of the most specific declaration must imply the predicate expressions of other declarations.

Predicate dispatch is more powerful than the dispatch that we presented for dependent classes. In particular, the dispatch of JPred [Mil04] is based on CVC Lite theorem prover, which supports propositional logic, while the constraint system of DC_C supports only conjunction of atomic predicates. The atomic predicates of DC_C and JPred share a lot of similarities, because both languages support equality of paths and classification of paths by classes. But differently from DC_C, JPred additionally supports linear arithmetic, which is mapped to the respective theory in CVC Lite.

Predicate dispatch is, however, not so strongly integrated with the other features of the host language like dependent classes. The general typing information, such as the types of fields used in paths, is ignored by the dispatch semantics – each path is considered as an independent variable. Also, the predicates are not available as types, which means that neither they can be used to describe more precise signatures of methods, nor they influence type checking of method implementation.

6.4.3 Predicate Classes

Predicate dispatch is primarily defined for dispatch of functions, but *predicate classes* [Cha93] apply predicate dispatch also to classes, i.e., their operations and state. A predicate class B is declared as a subclass of some class A, and is supplied with a condition based on the members of A. When the condition is true for a certain instance of A, this instance is automatically reclassified to the predicate class B. The predicate class can introduce new state and operations and override operations of the base class.

Differently from dependent classes, predicate classes can depend on mutable attributes of objects, which means that they can reclassify objects after they are created. Therefore, variation modularized by predicate classes can be bound dynamically during lifetime of an object.

Predicate classes allow using any Cecil expression as a predicate for dispatch, which makes them very powerful, but at the same time makes it impossible to statically guarantee completeness and uniqueness of dispatch. The predicate classes are primarily a dynamic mechanism, and use the predicates for dynamic dispatch only. Consequently, they cannot express covariant dependencies between objects and so support type-safe variations affecting multiple objects.

6.4.4 Completeness and Uniqueness Checking

The problem of checking method completeness and uniqueness has been intensively investigated for multimethods [Cha92, CL94, MBC04, CLCM00, BM97] and predicate dispatch [EKC98, Mil04]. The specifics of method checking for dependent classes are related to their polymorphic instantiation, dependent typing, dependent inheritance relations, and the possibility of recursive combination. Many of the approaches of method checking [CL94, MBC04, CLCM00, Mil04] emphasize modular type checking, which is achieved by imposing various limitations on the extent to which extensions with new method declarations are possible. While analogous limitations for dependent classes could also be considered, they are not compatible with the applications of dependent classes requiring support for extensions along multiple independent variation points.

6.5 Other Type Systems

6.5.1 Dependent Types

The path-dependent types that we used for type-safe family polymorphism are a special form of *dependent types*. In general, a dependent type is a type depending on a term. Specific languages and calculi with dependent types differ by the form of terms allowed in types, definition of equality between such terms, and supported combinations with other language features.

Research on dependent types has strong roots in works on logics and proof systems, in particular Martin Löf's Intuitionistic Type Theory [ML84], AUTOMATH languages [dB80], the Logical Framework (LF) [HHP93], and the Calculus of Constructions (CC) [CH88]. Barendregd's lambda-cube [Bar92] gives an elegant classification of type systems for the lambda calculus, including different kinds of interdependencies between terms and types. The corners of one face of the lambda-cube, are covered by the type systems supporting dependent types. LF [HHP93] corresponds to the simplest of these calculi – λP, which supports only first-order dependent types, while CC [CH88] corresponds to the most powerful calculus of the cube – $\lambda P\omega$ (also known as λC), which integrates dependent types with polymorphic types and terms, i.e., supports all kinds of interdependencies between types and terms.

The calculi of lambda-cube as well as the Martin Löf's type theory require strong normalization of terms. It is necessary for ensuring decidability of term equivalence relationship, which constitutes the core a type system with dependent types. The requirement for strong normalization cannot be guaranteed in many practical programs. Therefore, Cayenne [Aug98] – a programming language with dependent types based on Haskell – supports dependent types in a Turing-complete language by giving up the decidability of the type system and instead interrupting type checking after certain period of time. Another programming language, Epigram [MM04], follows a mixed approach leaving it for the developer to decide if a recursive function must be shown to be always termining or not.

Yes another strategy is followed in DML [XP99], a ML-based language with dependent types: it limits the form of terms that can be used in types to so-called *index objects*. Although the indexes appear in expressions, they are not evaluated and only used for typing. Typing of the index objects and their equivalence is defined by a constraint system. Decidability of the type system in DML depends on decidability of the underlying constraint solver. The language is not bound to a specific constraint solver, but the presented examples of DML usually assume a constraint solver supporting arithmetic expressions. The strategy of path-dependent types is analogous to that of DML, because they also make type-checking decidable by allowing a limited form of terms in types, while still using a Turing-complete language for expressions. From the perspective of DML, the subset of vc^n calculus dealing with paths can be seen as a specific index language, which is designed to identify family objects in a typical object-oriented setting and is proven to be decidable.

Combinations of dependent types with other language features are difficult, because they often render the type system undecidable. The work discussed so far explores dependent types in combination with the concepts of functional languages, while type systems with virtual types explore dependent types in combination of object-oriented features, most notably subtyping, nominal types, self-types, late-binding, and inheritance. Combination of dependent types with subtyping is investigated in [AC96], which gives a decidability proof for $\lambda P_{\leq}$ calculus – an extension of λP [Bar92] with subtyping. It points out that the combination of dependent types with subtyping is difficult even in a strongly normalizing calculus, because it produces a cyclical interdependency among subtyping, typing and kinding relationships. An attempt to encode virtual types using lambda-based calculi with dependent types in combination with other functional language features can be found in [IP99], where virtual types are encoded in an omega-order polymorphic lambda calculus with subtyping, dependent functions, and dependent records supporting bounded and manifest types for fields. The resulting calculus is complicated and most probably undecidable, which motivates direct formalization of dependent types in combination with object-oriented features.

6.5.2 Dynamic Dispatch with Dependent Types

[CC01] presents $\lambda\Pi^{\&}$ calculus, which combines dependent typing, dynamic overloading and subtyping, and can be seen as a combination of earlier calculi $\lambda P_{\leq}$ [AC96] and $\lambda\&$ [CGL92], the former combining dependent types with subtyping and the latter combining subtyping with dynamic overloading. The selected combination of features in $\lambda\Pi^{\&}$ makes it close to dependent classes, because it enables dynamic dispatch of functions by multiple parameters and their dependent typing.

The precursor calculus $\lambda\&$ [CGL92] presents an interesting encoding of object-oriented features using overloaded functions. An *overloaded function* is expressed as a union of simple function expressions with different argument and return types. The types of overloaded functions are then called *overloaded types* and are expressed as unions of simple function types. Dynamic dispatch is achieved through subtyping of overloaded types. The subtyping abstracts from certain cases of an overloaded type and so enables calling an overloaded function without a static knowledge of all its cases. The syntax of $\lambda\Pi^{\&}$ represents overloaded functions as expressions and so supports first-class overloaded functions, which is not possible in our calculi.

Overloaded functions can also be used to encode records and their subtyping. Since dynamic dispatch is based on the subtyping relationship, it supports dispatch over record structures and multi-dispatch as a special case of it. In combination with dependent types, the types of the arguments and the return types of functions can be covariantly related as the types from the same family.

Differently from our DC_C calculus, the $\lambda\Pi^{\&}$ does not address the problem of completeness and uniqueness checking of functions. The definitions of $\lambda\Pi^{\&}$ and its algorithmic version contain a non-constructive rule requiring unique selection for every valid type in a context. Completeness checking is not considered, because the overloaded functions are required to supply a direct implementation for every supported argument type, e.g., a function implementing painting of a Shape must contain a default implementation for Shape even if it is implemented for all concrete subclasses of Shape.

The $\lambda\Pi^{\&}$ calculus can encode nominal subtype relations between atomic types by means of bounded type variables in the typing context. It is, however, not possible to declare dependent inheritance relations supported by virtual classes and dependent classes in this way.

Type-checking of $\lambda\Pi^{\&}$ is in general not decidable, because its type equivalence is based on a possibly non-terminating β-reduction. The authors of the calculus also propose a variant of it with a strongly normalizing β-reduction relation, which makes the calculus decidable, but also constrains its computational power. In vc^n, we do not experience such problems, because we define separate reduction relationships for operational semantics and normalization of terms in types.

6.5.3 Parametric Polymorphism

Parametric polymorphism is a possibility to parameterize terms and types by other types. In object-oriented languages [Mey92, AFM97, BOSW98, BML97, CS98, OW97] parametric polymorphism is primarily used for parameterization of classes by types and is also known as *genericity* or *generics*.

Virtual classes and virtual types have been positioned as an alternative to generics since their inception in [MMP89]. The advantages of virtual classes and generics have been compared in [BOW98] and [TT99]. The main strength of virtual types with respect to generics is the possibility to express a covariance among a group of related classes, while the strength of generics lies is their support for structural subtyping. Therefore, [TT99] proposes unification of advantages of virtual types and generics by supporting so-called *structural virtual types*, i.e., the possibility to specify bounds of to type members of objects in their types. Such possibility is in fact supported by the νObj calculus [OCRZ03].

Encoding of generics by virtual types very much relies on support for final-bindings. Therefore, such encoding is not possible in our implementation of virtual classes, which supports further-bindings only. The presented formalizations of dependent classes do not support final-bindings either. Instead, in Sec. 4.3.9 we proposed an alternative unification of dependent classes with parametric polymorphism, based on the possibility to parameterize dependent classes by types. Such unification not only encodes generics with structural subtyping, but also enables modular specialization of classes for their generic parameters.

6.5.4 Constraint Systems

Constraint systems are widely used for type inference [Mit84, Rey85, OSW99] and programming in Constraint Logic Programming languages [JM94]. DML [XP99] integrates constraint system into the type system of a functional language for the purpose of typing index objects. In the context of the X10 language [NSPG08] advantages of constrained types for object-oriented languages are explored and a modular integration of constraint systems into a programming language is proposed.

Our DC_C calculus defines the type system of dependent classes in form of a concrete constraint system. Differently from DML [XP99] and X10 [NSPG08], the purpose of DC_C calculus is not to abstract from a concrete constraint system, but to design a minimal contraint system that can encode the intended semantics of dependent classes. A generalization of our approach to an arbitrary constraint system is not very practical, because constraint entailment in many constraint systems is undecidable, and, as we have seen in Sec. 5.5.4, method completeness and uniqueness checking creates even more difficult problems that require specific solutions.

7 Conclusions and Future Work

7.1 Conclusions

The main goal of this thesis was to show that the typical object-oriented techniques, such as inheritance and subtype polymorphism, can be made available at a larger scope, and in this way provide a better support for dealing with variations involving multiple objects.

For the purpose of making inheritance and polymorphism available for a group of classes we relied on existing ideas of virtual classes [MMP89] and family polymorphism [Ern01]. Propagating mixin composition [Ern99c] was used to enable the advantages of mixin-based inheritance [BC90] at the scope of the group of classes. We have explained how the intuitive semantics of inheritance and mixin-based inheritance can be generalized for a group of classes, and used it as basis to define a new semantics for propagating mixin composition. In the thesis we presented the implementation of virtual classes, propagating mixin composition and path-dependent types in CaesarJ programming language, which the first implementation of these ideas for Java.

Further, we proposed the concept of dependent classes, which enhances virtual classes in analogous way like multimethods [Cha92, DG87, Sha96] enhance single-dispatch. The multi-dispatch for classes not only enables dispatch of their functionality by multiple constructor parameters, but also generalizes family polymorphism with the possibility to express membership of an object in multiple families. The feasibility of the new concept was validated in two ways. First, we designed a concrete language with dependent classes, called DepJ, and implemented a type-checker and interpreter for it. Second, we formalized the features of dependent classes in vc^n and DC_C calculi, and verified their soundness and decidability.

For a practical validation of the expected advantages of virtual classes and dependent classes, we defined a set of variation scenarios that were derived from the designs of actually existing software. We identified the problems of implementing these scenarios using conventional object-oriented techniques, and showed that these problems are resolved by implementations with the advanced techniques. In particular, we have showed that virtual classes and dependent classes carry over the advantages of the corresponding conventional object-oriented techniques to a larger scope:

- The typical advantages of inheritance for modeling variations of individual classes are (1) the possibility to modularize variations affecting the interface of a class in a type-safe way, (2) support for unanticipated variability, and (3) minimal glue-code overhead to describe the variation. Virtual classes provide all these advantages for modeling variations of a group of classes. Virtual classes support all kinds of relationships between such classes: inheritance, instantiation, and references in types.

- Multiple inheritance enables composition of variations of a class modularized by inheritance. Analogously, the propagating mixin composition enables composition of variations of a group of classes modularized by means of virtual classes. Like mixin-based inheritance, propagating mixin-composition automates the composition as much as possible: it automatically resolves method implementation ambiguities, automatically merges classes and their relationships.

- Multimethods (1) enable modularization of method implementation with respect to multiple dimensions of variation, (2) support extensibility with respect to all these dimensions, and (3) provide a dynamic mapping from the selected variants to the implementation of the method with respect to these variants. Dependent classes provide all these advantages for modularizing multidimensional variations of a class, which encompass not only variations of its method implementations, but also variations of its interface and inheritance relationships.

- Dependent classes also combine the typical advantages of virtual classes and multi-dispatch for variation management. On the one hand, dependent classes can express type-safe variations affecting a group of objects. On the other hand, dependent classes can model multi-dimensional variation. These two capabilities can be combined in two ways. First, dependent classes enable type-safe modularization of variations of an object with respect to multiple groups of objects. Second, dependent classes can be used to modularize multiple variations affecting the same group of objects.

The evaluation of virtual classes and dependent classes, based on variation scenarios, demonstrates applicability of these language features for specific classes of problems rather than just for specific examples. We give an abstract description of each variation scenario and its design problems. Then we present the solutions of these problems with virtual classes or dependent classes and analyze the consequences of these solutions. Thus, the descriptions of the variation scenarios and their designs can also be seen as a collection of design patterns for virtual classes and dependent classes.

In CaesarJ we reused the results of the formal work on dependent classes for the implementation of virtual classes. Although we preserved the specific inheritance semantics of virtual classes and propagating mixin composition based on linearization of the inheritance graph, we almost completely reused the type relations of vc^n. The type relations of vc^n are simpler and at the same time more expressive than the ones of the vc calculus [EOC06], which were used for earlier implementations of CaesarJ. The simplification of type relations in vc^n was in large part due to a new perspective on expressing dependency between classes. By abandoning the nested style, we eliminated the distinction between the normal fields of objects and the special fields referencing the enclosing objects. This allowed to treat all fields in an uniform way and so simplify the type relations.

The DC_C calculus shows a further enhancement of the semantics of dependent classes with intersection types and abstract declarations. The most interesting aspect of the calculus is the reduction of the type system into a relatively simple constraint system. The constraint system contains only 3 kinds of constraints (one of which is introduced only for internal purposes) and describes the static semantics of the calculus by very generic and intuitive axioms. Path equivalence is encoded by two axioms based on the general properties of equivalence: one axiom declares reflexiveness of the equivalence, and the other one declares that equivalent paths can substitute each other in other constraints. Inheritance is encoded by program-specific entailment rules, resembling the Horn clauses.

Despite the declarativeness of the calculus, we managed to prove its decidability. The proof was relatively difficult, however, and relied on a strict limitation on the program-specific entailment rules, which permits only constraints of the form $x :: C$ (x is an instance of C) as consequents. Although the closeness of the calculus to the mathematical logic and the explicit exposure of its limitations clearly show possible directions for extending its expressiveness, the main challenge of such extensions is preserving the decidability of the calculus.

Although the vc^n and DC_C calculi defined the core semantics of dependent classes, design of a practical language with dependent classes raised a set of specific language design issues. In several cases we found that every alternative of a particular aspect of language design has its advantages. Therefore, in the implemented DepJ language, we provide combinations of these alternatives:

- The language supports both parametric and the nested styles for expressing dispatch of classes and methods. Although the parametric style is more expressive and does not impose limitations on extensibility, the nested style allows bundling a set of declarations depending on the same parameter and in this way make the code more concise. In the implementation of the language, the nested style is translated to the corresponding parameteric style before type-checking and interpretation.
- The language combines the position-based and name-based parameter binding styles. The former is more compact and thus more convenient to use, but the

latter is more flexible, because it allows varing the set of class parameters and supports partial binding of parameters.

- DepJ uses a symmetric dispatch strategy for methods, because it provides a clearly defined and intuitive method selection semantics. However, it does not support composition of method implementations based on super-calls, which is available in case of the mixin composition semantics. Therefore, in addition to the symmetric dispatch, the language supports so-called method refinements, which can be used to describe composable extensions to the implementation of a method.

The DepJ language also implements so-called second-order dependent classes, which combine the ideas of dependent classes with generics. Such combination has two consequences. First, second-order dependent classes support uniform parameterization of classes by objects and by types, as well as arbitrary combinations of such parameterizations. Second, they provide the typical advantages of dependent classes for managing variations of generic classes. In particular, they enable varing the functionality of a generic class with respect to its type parameters, and defining such variations in a modular way. Abstraction from such variations in client code and their dynamic binding is enabled by representing types as runtime values and supporting dynamic dispatch by such values.

7.2 Future Work

The interpreter of the DepJ language presents only a prototypical implementation of dependent classes. An efficient implementation of the proposed language features, e.g., comparable to the implementation of virtual classes in CaesarJ, is challenging, because we need to find efficient ways to implement the multi-dispatch of dependent classes. The strategy of the implementation of virtual classes in CaesarJ is based on linearizing their inheritance graph and mapping it to single-inheritance chains in Java bytecode. Such strategy is not very suitable for implementating dependent classes, because it would lead to a combinatorial explosion of generated Java classes. We envisage two possible ways for a practical implementation of dependent classes. One way would be to generate concrete instances of dependent classes on-demand and load them using a custom Java class-loader. Another way would be to implement support for efficient multi-dispatch of classes in the virtual machine.

Dependent classes were introduced as a combination of the ideas of virtual classes and multi-dispatch. Thus a natural further development of dependent classes would be their generalization for a more powerful form of dispatch, namely the predicate dispatch [EKC98, Mil04]. Differently from predicate classes [Cha93], such generalization would have an impact not only on the operational semantics but also on the type system of the language. The DC_C calculus is a step in this direction, because it defines a constraint system that encodes both the type system and the dynamic dispatch of dependent classes. Since a constraint system is also the major element of predicate dispatch, such encoding presents the semantics of dependent classes as a special case of a predicate dispatch.

The expressiveness of the dispatch can be increased by extending the constraint language and the constraint entailment axioms. Possible extensions include the introduction of further typical logical operators, such as disjunction and negation. Introduction of existential quantification would be necessary for supporting local self-references in types, analogous to the ones available in the νObj calculus [OCRZ03]. The definitions of dependent classes in vc^n and DC_C do not allow declaring default types for their fields. The support for default fields in DC_C could be achieved by allowing more versatile constraints as consequents of the program-specific entailment axioms, namely constraints on paths of the form $x.f$. However, all such extensions would break the assumptions made in the current decidability proof of DC_C, and thus may require to redesign it completely.

Two specific enhancements of to the type system are motivated by the presented variation scenarios:

First, in the variation scenario modeling group variations by multiple helper objects, we were faced with the limitation that only immutable references to family objects can be used in types (cf. Sec. 3.4.1.5). Because of that, we had to abandon direct references from master objects to their helpers, and model them by a map structure instead. Such scenario would benefit from the possibility to use mutable references to families.

Second, we were not able to completely reuse an implementation of a map structure, supporting dependencies of the types of values on the types of the keys (cf. Sec. 4.3.9.3). Such map structure was necessary for defining the mapping from master objects to their helpers (cf. Sec. 4.4.2.3), and from adaptees to their adapters in a particular framework instance (cf. Sec. 4.4.3.1).

An interesting research direction is a further development of the second-order dependent classes. Now they are only prototypically implemented in the type-checker and interpreter of DepJ. In the future we intend to formalize these features and verify of their soundness and decidability. The current implementation is limited to second-order types, but also an extension with higher-order types could be considered. Also, the current proposal is very minimal and could be extended with further features found in the implementations of generics for Java [BOSW98] or Scala [OZ05b], such as the possibility to specify lower-bounds to type parameters, and inheritance of constraints on type variables from super-classes. The main challenge is to design a decidable type system with such features, because it is also difficult to prove the decidability of the type system with generics supporting analogous features [KP06].

A more general direction of future work is further pursuing the idea of reusing object-oriented ideas at a larger scope. In this thesis we showed generalizations of the ideas of class inheritance, subtype polymorphism, mixin-based inheritance, and multi-dispatch. Various dynamic object-oriented features such as dynamic inheritance [US87] or object reclassification [Mez99, DDDCG02] could also be applied at the scope of a group of classes to model more sophisticated dynamic varations at that scope. Examples of work in this direction are the dynamic propagating mixin composition supported in gbeta [Ern99a], and the idea delegation layers [Ost02], which generalizes delegation-based inheritance for a group of classes. These works still lack a strong formal basis, which could serve as a basis for further improving the balance between the dynamic flexibility and the static guarantees of such language features.

This thesis showed how the ideas of inheritance and dynamic dispatch available for methods in object-oriented languages can be also successfully applied on classes. In an analogous way, these ideas could be applied on other typical software abstractions. We are currently working on the definition of inheritance and dynamic dispatch for other behavioral abstrations, namely events and state machines. The early results of this work can be found in [NnNG09].

Bibliography

[AC96] David Aspinall and Adriana Compagnoni. Subtyping dependent types. In *LICS '96: Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, page 86, Washington, DC, USA, 1996. IEEE Computer Society.

[AFM97] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. *SIGPLAN Not.*, 32(10):49–65, 1997.

[AG96] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. *Transactions on Aspect-Oriented Software Development*, 3880:135–173, 2006.

[Ald05] Jonathan Aldrich. Open modules: Modular reasoning about advice. In Andrew P. Black, editor, *ECOOP 2005 - Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer Berlin / Heidelberg, 2005.

[Aug98] Lennart Augustsson. Cayenne—a language with dependent types. In *ICFP '98: Proceedings of the third ACM SIGPLAN international conference on Functional programming*, pages 239–250, New York, NY, USA, 1998. ACM.

[AWB$^+$93] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting object interactions using composition filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Object-Based Distributed Programming.* Springer, 1993.

[Bar92] Hank P. Barendregt. Lambda calculi with types. *Handbook of logic in computer science (vol. 2): background: computational structures*, pages 117–309, 1992.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90*, pages 303–311, New York, NY, USA, 1990. ACM Press.

[BCH$^+$96] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for Dylan. In *Proceedings OOPSLA '96*, pages 69–82. ACM Press, 1996.

[BDN05] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: controlling the scope of change in Java. In *Proceedings OOPSLA '05*, pages 177–189, New York, NY, USA, 2005. ACM Press.

[BDW03] Alexandre Bergel, Stéphane Ducasse, and Roel Wuyts. Classboxes: A minimal module model supporting local rebinding. In *Modular Programming Languages, Joint Modular Languages Conference, JMLC 2003*, volume 2789 of *Lecture Notes in Computer Science*, pages 122–131. Springer, 2003.

[BLS98] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for implementing domain-specific languages. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*. IEEE Computer Society, 1998.

[BM97] François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 302–315, New York, NY, USA, 1997. ACM.

[BML97] Joseph A. Bank, Andrew C. Myers, and Barbara Liskov. Parameterized types for Java. In *Proceedings POPL '97*, pages 132–145, New York, NY, USA, 1997. ACM.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: adding genericity to the Java programming language. *SIGPLAN Not.*, 33(10):183–200, 1998.

[BOW98] Kim B. Bruce, Martin Odersky, and Phillip Wadler. A statically safe alternative to virtual types. In *Proceedings ECOOP '98. LNCS 1445*, pages 523–549. Springer, 1998.

[BSR03] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. In *Proceedings ICSE '03*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society.

[CC01] Giuseppe Castagna and Gang Chen. Dependent types with subtyping and late-bound overloading. *Inf. Comput.*, 168(1):1–67, 2001.

[CDNW07] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: A Simple Virtual Class Calculus. In Proceedings of AOSD'07, 2007.

[CGL92] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. In *Proceedings LFP '92: ACM conference on LISP and functional programming*, pages 182–192, New York, NY, USA, 1992. ACM Press.

[CGLO06] Vincent Cremet, François Garillot, Sergueï Lenglet, and Martin Odersky. A core calculus for Scala type checking. In *Proceedings MFCS*, Springer LNCS, September 2006.

[CH88] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, 1988.

[Cha92] Craig Chambers. Object-oriented multi-methods in Cecil. In *Proceedings ECOOP '92*, LNCS 615, pages 33–56. Springer, 1992.

[Cha93] Craig Chambers. Predicate classes. In *Proceedings ECOOP '93*, LNCS 707, pages 268–297. Springer, 1993.

[CL94] Craig Chambers and Gary T. Leavens. Typechecking and modules for multi-methods. In *Proceedings OOPSLA '94*, pages 1–15, New York, NY, USA, 1994. ACM Press.

[CLCM00] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. Multijava: modular open classes and symmetric multiple dispatch for java. In *Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '00, pages 130–145, New York, NY, USA, 2000. ACM.

[CMMS91] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of System F with subtyping. In *Information and Computation*, pages 750–770. Springer-Verlag, 1991.

[CS98] Robert Cartwright and Guy Steele. Compatible genericity with run-time types for the Java programming language. *SIGPLAN Not.*, 33(10):201–215, 1998.

[dB80] N. G. de Bruijn. A survey of the project AUTOMATH. In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, New York, 1980.

[DCB09] Benjamin Delaware, William Cook, and Don Batory. A machine-checked model of safe composition. In *FOAL '09: Proceedings of the 2009 workshop on Foundations of aspect-oriented languages*, pages 31–35, New York, NY, USA, 2009. ACM.

[DDDCG02] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. More dynamic object reclassification: Fickle&par;. *ACM Trans. Program. Lang. Syst.*, 24(2):153–191, 2002.

[DG87] Linda DeMichiel and Richard Gabriel. The Common Lisp Object System: An overview. In *Proceedings ECOOP '87*, pages 243–252, 1987.

[EKC98] Michael D. Ernst, Crag Kaplan, and Craig Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings ECOOP '98*, volume 1445 of *LNCS*, pages 186–211. Springer, 1998.

[EOC06] Erik Ernst, Klaus Ostermann, and William Cook. A virtual class calculus. In *Proceedings POPL '06*, pages 270–282. ACM Press, 2006.

[Ern99a] Erik Ernst. Dynamic inheritance in a statically typed language. *Nordic Journal of Computing*, 6(1):72–92, Spring 1999.

[Ern99b] Erik Ernst. *gbeta - a language with virtual attributes, Block Structure, and Propagating, Dynamic Inheritance.* PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.

[Ern99c] Erik Ernst. Propagating class and method combination. In *Proceedings ECOOP '99*, pages 67–91, London, UK, 1999. Springer-Verlag.

[Ern01] Erik Ernst. Family polymorphism. In *Proceedings ECOOP '01*, pages 303–326, London, UK, 2001. Springer-Verlag.

[Ern03] Erik Ernst. Higher-order hierarchies. In Luca Cardelli, editor, *Proceedings ECOOP '03*, LNCS 2743, pages 303–329. Springer-Verlag, 2003.

[ES95] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1995.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *ACM Symposium on Principles of Programming Languages (POPL) '98*, pages 171–183. ACM, 1998.

[GA07] Vaidas Gasiunas and Ivica Aracic. Dungeon: A case study of feature-oriented programming with virtual classes. In *2nd Workshop on Aspect-Oriented Product Line Engineering (AOPLE)*, 2007.

[GEF08] GEF homepage, 2008. http://www.eclipse.org/gef/.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns.* Addison Wesley, 1995.

[GMO06] Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Formal soundness proof of the vc^n calculus, 2006. http://www.st.informatik.tu-darmstadt.de/static/pages/projects/mvc/index.html.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, New York, NY, USA, 1989.

[Her03] Stephan Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 248–264, London, UK, 2003. Springer-Verlag.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, 1993.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings OOPSLA '93. ACM SIGPLAN Notices 28(10)*, pages 411–428, 1993.

[Hut06] DeLesley Hutchins. Eliminating distinctions of class: using prototypes to model virtual classes. In *Proceedings OOPSLA '06*, pages 1–20. ACM Press, 2006.

[IP99] Atsushi Igarashi and Benjamin C. Pierce. Foundations for virtual types. In *ECOOP '99: Proceedings of the 13th European Conference on Object-Oriented Programming*, pages 161–185, London, UK, 1999. Springer-Verlag.

[IPW99] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 1999.

[ISV05] Atsushi Igarashi, Chieri Saito, and Mirko Viroli. Lightweight family polymorphism. In *Programming Languages and Systems, Third Asian Symposium (APLAS'05)*, pages 161–177. Springer LNCS 3780, 2005.

[IV07] Atsushi Igarashi and Mirko Viroli. Variant path types for scalable extensibility. *SIGPLAN Not.*, 42(10):113–132, 2007.

[JF88] Ralph Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[JM94] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

[Kee89] Sonya E. Keene. *Object-Oriented Programming in Common LISP: A Programmer's Guide to CLOS.* Addison–Wesley, Reading, 1989.

[KHH$^+$01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings ECOOP '01*, pages 327–353, London, UK, 2001. Springer-Verlag.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings ECOOP '97*, LNCS 1241, pages 220–242. Springer, 1997.

[KP06] Andrew J. Kennedy and Benjamin C. Pierce. On decidability of nominal subtyping with variance, 2006. FOOL-WOOD '07.

[KT07] Tetsuo Kamina and Tetsuo Tamai. Lightweight scalable components. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 145–154, New York, NY, USA, 2007. ACM.

[KT08] Tetsuo Kamina and Tetsuo Tamai. Lightweight dependent classes. In *GPCE '08: Proceedings of the 7th international conference on Generative programming and component engineering*, pages 113–124, New York, NY, USA, 2008. ACM.

[Lie96] Karl J. Lieberherr. *Adaptative Object-Oriented Software: The Demeter Method.* PWS Publishing, 1996.

[LLM99] Karl Lieberherr, David Lorenz, and Mira Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, Northeastern University, March 1999.

[LLO03] Karl Lieberherr, David Lorenz, and Johan Ovlinger. Aspectual collaborations – combining modules and aspects. *Journal of British Computer Society*, 2003.

[LSLX94] Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Adaptive object-oriented programming using graph-based customization. *Commun. ACM*, 37(5):94–101, 1994.

[Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[MBC04] Todd Millstein, Colin Bleckner, and Craig Chambers. Modular typechecking for hierarchically extensible datatypes and functions. *ACM Trans. Program. Lang. Syst.*, 26(5):836–889, 2004.

[MC99] Todd Millstein and Craig Chambers. Modular statically typed multimethods. In *Proceedings ECOOP '99*, volume 1628 of *Lecture Notes in Computer Science*, pages 279–303. Springer Verlag, 1999.

[Mey92] Bertrand Meyer. *Eiffel: The Language.* Prentice-Hall, 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, second edition, 1997.

[Mez99] Mira Mezini. Towards variational object-oriented programming: The RONDO model. Technical report, Northeastern University, Boston, 1999.

[Mil04] Todd Millstein. Practical predicate dispatch. In *Proceedings OOPSLA '04*, pages 345–364. ACM Press, 2004.

[Mit84] John C. Mitchell. Coercion and type inference. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 175–185, New York, NY, USA, 1984. ACM.

[ML84] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory.* Bibliopolis, Naples, 1984.

[ML98] Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings OOPSLA '98*, pages 97–116. ACM Press, 1998.

[MM04] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

[MMP89] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In *Proceedings OOPSLA '89*, pages 397–406. ACM Press, 1989.

[MMPN93] Ole Lehrmann Madsen, Birger Møller-Pedersen, and Kristen Nygaard. *Object Oriented Programming in the Beta Programming Language.* Addison-Wesley, 1993.

[MO02] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings OOPSLA '02*, pages 52–67. ACM Press, 2002.

[MO03] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In *Proceedings AOSD '03*, pages 90–99. ACM Press, 2003.

[MO04] Mira Mezini and Klaus Ostermann. Variability management with feature-oriented programming and aspects. In *Proceedings SIGSOFT '04/FSE-12*, pages 127–136. ACM Press, 2004.

[MPO08] Adriaan Moors, Frank Piessens, and Martin Odersky. Generics of a higher kind. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 423–438, New York, NY, USA, 2008. ACM.

[MSL00] Mira Mezini, Linda Seiter, and Karl Lieberherr. Component integration with pluggable composite adapters. In Mehmet Aksit, editor, *Software Architectures and Component Technology: The State of the Art in Research and Practice*. Kluwer, 2000. University of Twente, The Netherlands.

[NCM04] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. *SIGPLAN Not.*, 39(10):99–115, 2004.

[NnNG09] Angel Núñez, Jacques Noyé, and Vaidas Gasiūnas. Declarative definition of contexts with polymorphic events. In *COP '09: International Workshop on Context-Oriented Programming*, pages 1–6, New York, NY, USA, 2009. ACM.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[NQM06] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. J&: nested intersection for scalable software composition. In *Proceedings OOPSLA '06*, pages 21–36. ACM Press, 2006.

[NSPG08] Nathaniel Nystrom, Vijay Saraswat, Jens Palsberg, and Christian Grothoff. Constrained types for object-oriented languages. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 457–474, New York, NY, USA, 2008. ACM.

[OCRZ03] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proceedings ECOOP '03*. Springer LNCS, 2003.

[Ode09] Martin Odersky. The Scala language specification. version 2.7, 2009. http://www.scala-lang.org/docu/files/ScalaReference.pdf.

[Ost02] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings ECOOP '02*, pages 89–110, London, UK, 2002. Springer-Verlag.

[OSW99] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theor. Pract. Object Syst.*, 5(1):35–55, 1999.

[OW97] Martin Odersky and Philip Wadler. Pizza into Java: translating theory into practice. In *Proceedings POPL '97*, pages 146–159, New York, NY, USA, 1997. ACM.

[OZ05a] Martin Odersky and Matthias Zenger. Independently extensible solutions to the expression problem. In *Proc. FOOL 12*, January 2005. `http://homepages.inf.ed.ac.uk/wadler/fool`.

[OZ05b] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proceedings OOPSLA '05*, pages 41–57, New York, NY, USA, 2005. ACM Press.

[Rey85] John C. Reynolds. Three approaches to type structure. In *Proc. of the international joint conference on theory and practice of software development (TAPSOFT) Berlin, March 25-29, 1985 on Mathematical foundations of software development, Vol. 1: Colloquium on trees in algebra and programming (CAAP'85)*, pages 97–138, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[SB98a] Yannis Smaragdakis and Don Batory. Implementing reusable object-oriented components. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse*, page 36, Washington, DC, USA, 1998. IEEE Computer Society.

[SB98b] Yannis Smaragdakis and Don S. Batory. Implementing layered designs with mixin layers. In *Proceedings ECCOP '98*, pages 550–570, London, UK, 1998. Springer-Verlag.

[SB02] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[Sha96] Andrew Shalit. *The Dylan reference manual: the definitive guide to the new object-oriented dynamic language.* Addison Wesley, Redwood, CA, USA, 1996.

[Szy98] Clemens Szyperski. *Component Software – Beyond Object-Oriented Programming.* Addison-Wesley, 1998.

[TBKC07] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *GPCE '07: Proceedings of the 6th international conference on Generative programming and component engineering*, pages 95–104, New York, NY, USA, 2007. ACM.

[Tho97] Kresten Krab Thorup. Genericity in java with virtual types. In *In Proceedings ECOOP 97*, pages 444–471. Springer-Verlag, 1997.

[TO99] Peri Tarr and Harold Ossher. Hyper/J user and installation manual, 1999. http://www.research.ibm.com/hyperspace.

[Tor98] Mads Torgersen. Virtual types are statically safe. In *In 5th Workshop on Foundations of Object-Oriented Languages*, 1998.

[Tor04] Mads Torgersen. The expression problem revisited four new solutions using generics. In *Proceedings ECOOP '04*, volume 3086 of *LNCS*, pages 123–143. Springer, 2004.

[TT99] Kresten Krab Thorup and Mads Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In *Proceedings ECOOP '99*, 1999.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. In *Proceedings OOPSLA '87. ACM SIGPLAN Notices 22(12)*, pages 227–242, 1987.

[VN96a] Michael VanHilst and David Notkin. Using C++ templates to implement role-based designs. In *ISOTAS '96: Proceedings of the Second JSSST International Symposium on Object Technologies for Advanced Software*, pages 22–37, London, UK, 1996. Springer-Verlag.

[VN96b] Michael VanHilst and David Notkin. Using role components in implement collaboration-based designs. In *Proceedings OOPSLA '96*, pages 359–369, New York, NY, USA, 1996. ACM Press.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[WSM06] Alessandro Warth, Milan Stanojevic, and Todd Millstein. Statically scoped object adaptation with expanders. In *Proceedings OOPSLA '06*, pages 37–56. ACM Press, 2006.

[XP99] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 214–227, New York, NY, USA, 1999. ACM.

[Zen02] Matthias Zenger. Evolving software with extensible modules. In *International Workshop on Unanticipated Software Evolution*, Malaga, Spain, June 2002.

Scientific Career

since 2009	Post doctorate researcher in the group of Prof. Mira Mezini, Technische Universität Darmstadt, Germany
2004 - 2009	PhD assistant in the group of Prof. Mira Mezini, Technische Universität Darmstadt, Germany Awarded Degree: Doctoral Degree in Engineering (Dr.-Ing.)
1997 - 2003	Studies of Computer Science, University of Vilnius, Lithuania Awarded Degrees: B.Sc. in Computer Science, 2001 M.Sc. in Computer Science, 2003